CausalRPC

A TRACEABLE DISTRIBUTED
COMPUTATION FRAMEWORK

Computer Science Tripos, Part II

Craig Ferguson, Churchill College

2018–2019

A dissertation submitted to the University of Cambridge in partial fulfillment of the requirements for the degree of Bachelor of Arts in Computer Science

Declaration of originality

I, Craig Ferguson of Churchill College, being a candidate for Part II 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.

I, Craig Ferguson of Churchill College, am content for my dissertation to be made available to the students and staff of the University.

Signed ____________________

Date ____________________

Acknowledgements

Many thanks to:

- My supervisor, **Anil Madhavapeddy**, for his endless pool of good ideas and enthusiasm.
- **Thomas Gazagnaire**, and the rest of the Irmin development team, for their assistance and collaboration in extending Irmin to support the requirements of CausalRPC.
- **Matthew Ireland** and **Peter Rugg**, for their guidance in producing this dissertation.

Proforma

Candidate Number:	2350F
Project Title:	CausalRPC: a traceable distributed computation framework
Examination:	Computer Science Tripos, Part II (June 2019)
Word Count:	11984[*]
Lines of Code:	5239[†]
Project Originator:	Dr A. Madhavapeddy
Supervisor:	Dr A. Madhavapeddy

Original aims of the project

I aimed to implement an RPC framework in OCaml using the Irmin distributed database library as a network substrate. I intended to explore the trade-offs of a novel data-oriented approach to RPC in which race conditions between clients are resolved automatically by the middleware layer. The core deliverable is a demonstration of an RPC client remotely executing functions with Irmin-serialisable parameters on a server capable of handling concurrent client requests.

Work completed

The project was successful. I implemented CausalRPC, a distributed computation framework satisfying the project success criteria. This was achieved in spite of substantial original research and open-source development contributions necessary for the success of the project.

The approach of making the statefulness of RPC explicit was surprisingly effective, allowing CausalRPC to provide stronger consistency and traceability guarantees than conventional RPC systems. This broadened the scope of the project considerably, and a variety of ambitious extensions were selected to explore the inherent trade-offs of the approach. The final version of CausalRPC supports fault-tolerant worker clusters and is compatible with MirageOS.

Special difficulties

None.

[*]Computed using TeXcount. URL: `http://app.uio.no/ifi/texcount` (visited on 13th March, 2019).
[†]Computed using `cloc`. URL: `https://github.com/AlDanial/cloc` (visited on 13th March, 2019).

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

This dissertation presents a novel model of remote procedure calls (RPCs) as functional transformations over conflict-free replicated datatypes (CRDTs). This is realised in a general-purpose RPC framework, named *CausalRPC*, which demonstrates the strengths and weaknesses of a functional, data-oriented approach to RPC. I shall argue that making the underlying statefulness of RPCs explicit leads to improved traceability and scalability, while allowing the middleware to automatically resolve concurrency issues that would otherwise be a burden on the application developer.

1.1 Motivation

RPC is an old distributed system abstraction, first made popular by the use of SunRPC in the Network File System [46]. It links a client and server by their code structure: the client invokes a function, and the server executes that function. This hides the complexity of the underlying communication, but leaves several problems unsolved:

- **Traceability**: *how can the behaviour of the system be retrieved and understood?*

  Monitoring and diagnosing distributed systems is a notorious open research problem [34, 25, 42]. The behaviour of an RPC system is the result of both the middleware and application layers; neither layer has the full picture necessary to construct a behavioural trace. If each layer maintains separate logs, the resulting linearised mess will be hard to untangle. Indeed, Burckhardt et al. [11] argue that requiring distributed systems to be understandable from a linear log may be overly restrictive, leading several groups to propose graph-based 'causal' logs [42, 41].

- **Consistency**: *how can we ensure that concurrent requests preserve data consistency?*

  Another consequence of the application layer's ignorance regarding its execution context is that the application developer must implement concurrency control mechanisms to preserve data consistency (regardless of the order in which the middleware layer chooses to schedule the RPCs). This is often achieved by adding a database transaction layer behind the RPC servers, adding more system complexity and further complicating traceability.

- **Type-safety**: *how can we safely define and implement an RPC interface?*

  Traditionally, an RPC middleware layer is invoked by calling 'stub' functions representing the services offered by the RPC server. Early implementations of RPC [70, 67] provided additional tools to generate source code for these stubs given a set of function prototypes, but these tools operate outside the type system and so are inherently unsafe. Some programming languages solved this problem by providing first-class support for RPC (e.g. Java RMI, Erlang), but this introduces its own issues. I shall discuss these problems in more detail in Chapter 2.

- **Scalability**: *how can the system scale in response to increasing load?*

  RPC is complex to parallelise across many servers because the middleware layer must respect the statefulness of client sessions. The classic solution to this problem is to introduce a session-aware load balancer to demultiplex incoming requests onto a set of RPC servers [45]. This requires the load balancer to act as an application-aware middle-man, introducing a performance bottleneck.

These issues led to a series of popular critiques of RPC [71, 74]. Throughout the 2000s, RPC was gradually replaced by Representational State Transfer (REST) as the standard architectural style of distributed services [73]. However, proper use of REST style requires the underlying service to be stateless and resource-oriented. The originator of REST has argued [23] that many use-cases fail to meet these requirements, resulting in the widespread misuse of REST as effectively re-branded RPC. The constraints of REST are particularly challenging for internal service-to-service communication, and so RPC frameworks have seen a resurgence in the internals of modern microservice architectures [62].

This dissertation presents a novel model of RPCs (§ 3.1), of my own design, that solves the four issues presented above. The fundamental idea is to make the statefulness of RPCs explicit and allow the application developer to specify the semantics of this state; this allows the middleware to automatically manage conflicts between requests and construct a trace of the application behaviour. I shall show that this approach brings a number of benefits: traceability and consistency are built into the system at a fundamental level (§ 4.1), and scalability arises naturally from a framework that understands how to resolve concurrency conflicts (§ 4.3.3). The change of approach brings a number of challenges, including high memory usage (§ 4.3.2) and a lack of true isolation between RPC clients (§ 4.4).

Managing concurrent modifications to data is a central problem in Computer Science. This project will exploit the class of solutions offered by distributed version control systems (DVCSs) such as Git [12]. Since the widespread adoption of DVCSs among software developers, attempts have been made to generalise the underlying principles into concurrent programming frameworks [11, 61]. In particular, my project will make use of Irmin [49], an OCaml library that provides tools for creating branchable, mergeable distributed datastores. The Irmin API is very similar to that of Git, and the datastores that it creates are fully compatible with the Git repository format.

1.2 Previous work

This dissertation draws inspiration from existing work on mergeable data structures [22] and on the application of DVCSs to concurrent programming [11, 61]. There are several existing applications that leverage Irmin to solve concurrency problems, including a Part II project that extended the MirageOS TCP library to use Irmin for state management [65]. To the best of my knowledge, no existing work attempts to apply the theory of CRDTs to RPC frameworks.

The type theory underlying the implementation of statically type-safe RPC frameworks is well-established, though not widely implemented due to the need for advanced static type system features that are unavailable in languages such as Java and C++. In particular, my implementation draws from the theory of pickler combinators elaborated by Abadi et al. [1].

1.3 Structure of the dissertation

Chapter 2 introduces the necessary technical background, describes the libraries and tools used for development, and discusses various aspects of the project management. Chapter 3 discusses the various design decisions made during the development of CausalRPC. Chapter 4 contains a detailed performance analysis of CausalRPC relative to competing solutions and discusses the relative merits of data-oriented and conventional RPC. The overall conclusions of the project are given in Chapter 5, along with suggestions for relevant future work.

2 Preparation

This chapter discusses the technical background of the project (§ 2.1), describes how the project requirements and starting point were elaborated from the project proposal (§ 2.2–3), and finally considers the various software development practices employed during the project (§ 2.4).

2.1 Background knowledge

The client architecture of CausalRPC is shown in Figure 2.1. This section first presents the theory of conflict-free replicated data types (§ 2.1.1) and then shows how these principles can be applied to build an RPC system from the bottom-up: starting with Git as a storage substrate (§ 2.1.2), then building a communication layer over the store using Irmin (§ 2.1.3). I introduce several advanced features of the OCaml module and type systems that are required by CausalRPC (§ 2.1.4). Finally, I give a brief introduction to concurrent programming in OCaml (§ 2.1.5).

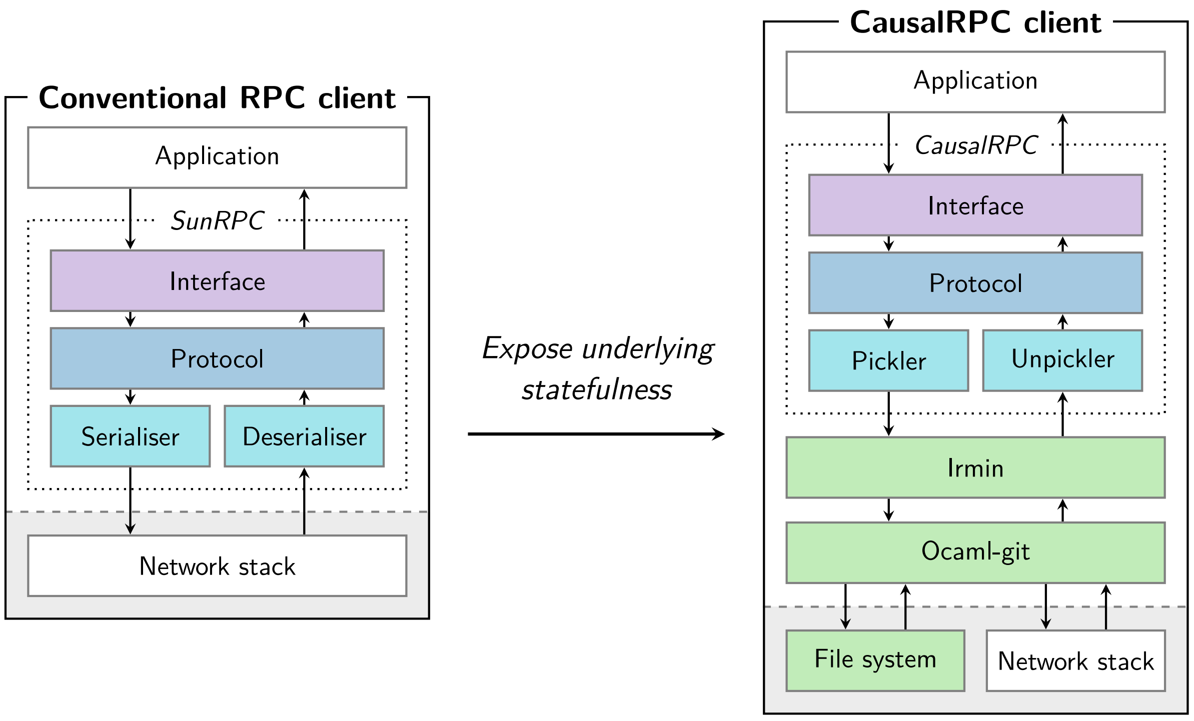


Figure 2.1: *The layered architecture of a CausalRPC client, relative to a conventional system such as SunRPC [8].*

2.1.1 Conflict-free replicated data types

Conflict-free replicated data types (CRDTs) are a formalism for achieving eventual consistency of a set of data replicas interconnected by an asynchronous network. This project will consider the *state-based* formation of CRDTs, as presented by Shapiro et al. [61]. Let $\Pi = \{p_0, \ldots, p_{n-1}\}$ be a set of non-byzantine processes. These processes each store a local replica of a set of objects, which are initially consistent. As the processes perform updates on their replicas, they periodically emit state updates such that each update eventually reaches every replica.[1] In the original presentation

[1] Any communication protocol that ensures eventual delivery of all updates to all replicas can be used; existing solutions to this problem include gossip and anti-entropy approaches [18, 59].

of CRDTs [61], these state updates contain the entire replica. CausalRPC uses *differential* updates, but the principles remain the same. See [3] for a proof of equivalence.

The processes interact with their local replicas in three ways: querying the state of an object, updating an object, and merging a remote state update into the local replica. Whenever p_i receives a state update s_j from p_j, it attempts to merge s_j with its current state s_i. Conceptually, the merge should result in the state $m(s_i, s_j)$ that is the 'most similar' to s_i whilst still containing all of the changes made to s_j since the two replicas were last consistent. Very briefly, this notion of 'similarity' is formalised by augmenting the state space S with a partial order $\sqsubseteq$ to form a join-semilattice $\langle S, \sqsubseteq \rangle$. All state changes must be monotonically non-decreasing with respect to $\sqsubseteq$. The merge function is defined to return the least state that is greater than or equal to its two input states (the *least upper bound*).

Let U_i be the set of updates either received or sent by p_i at some instant in time. Classic eventual consistency [35] requires that replicas with the same updates *eventually* reach equivalent state:

$$\forall i, j \;.\; \underbrace{\Box(U_i = U_j)}_{\substack{\textit{Updates are} \\ \textit{equal forever}}} \quad \Rightarrow \quad \underbrace{\Diamond\Box(s_i \equiv s_j)}_{\substack{\textit{Eventually, states} \\ \textit{become equivalent forever}}}$$

where the equivalence relation $\equiv$ holds for pairs of states that give the same output for all possible queries. This property allows some finite time for the processes to roll back and rewrite history in order to achieve consistency. CRDTs satisfy a stronger property, known as *strong convergence*:

$$\forall i, j \;.\; U_i = U_j \quad \Rightarrow \quad s_i \equiv s_j$$

Since all state changes are monotonically non-decreasing with respect to $\sqsubseteq$, the processes never have to rewrite their causal history when merging (a highly-desirable property for a traceable system!). For a CRDT system, the causal history of a state s is formalised as the graph of 'ancestor' states of s. When an update occurs, the new state has only a single ancestor; when a merge occurs, the ancestors are the two input states to the merge. Maintaining a record of the causal history is not necessary to achieve correctness, but CausalRPC *will* explicitly track it for two reasons:

1. In order to construct a persistent trace of the application behaviour.
2. In order to allow the use of *three-way* merges, which take as an additional argument the most-recent common ancestor of the two states to be merged. This additional context makes three-way merge functions considerably simpler to design.

2.1.2 The Git version control system

Git is a distributed version control system (DVCS) originally designed to aid the development of the Linux kernel [12]. It popularised the notion of 'branching' to allow programmers to make changes in isolation and 'merging' to reconcile those changes. While there are many DVCSs available [39], Git has particularly pure semantics that make it ideal for this project. A Git datastore (called a *repository*) is a distributed set of eventually-consistent replicas of a data structure, much like a CRDT network. The internals of Git are described in *Git from the Bottom Up* [77]; the following is a brief summary.

The distributed datatype in a Git repository is a specialised Merkle tree known as a *tree object* [12]. A tree object is an array of named pointers to children, each of which is either a nested tree object or a data block (known as a *blob*). This hierarchical structure was originally intended to describe directory trees of source code, with tree objects corresponding to directories and

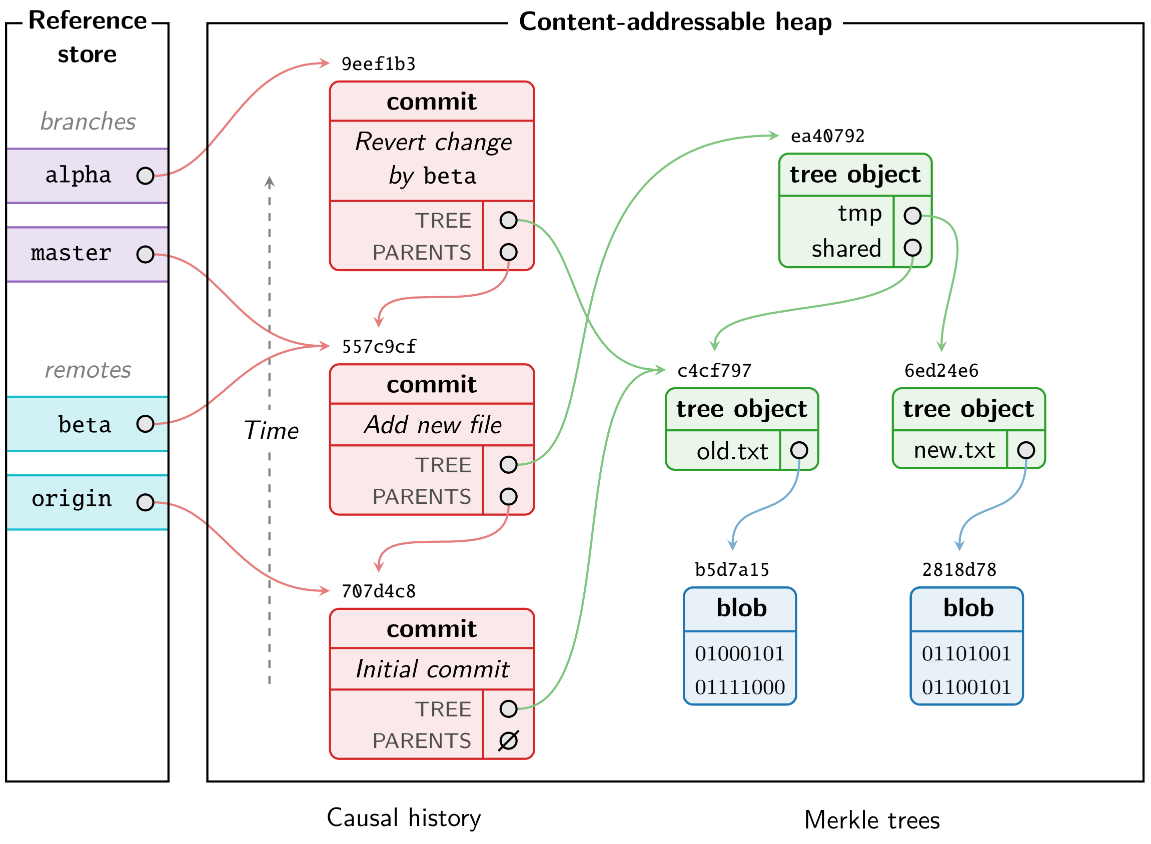


FIGURE 2.2: *The internal structure of a Git repository.*

blobs corresponding to text files, but in practice any tree-like data structure can be stored in a Git repository.

All tree nodes are immutable and are stored in a content-addressable heap, addressed by SHA-1 hashes. Thus, identical nodes have identical addresses. This design allows multiple replicas of the root tree object to co-exist efficiently in the same heap: wherever the replicas have equal sub-trees, these will only be stored once in the heap. Whenever a replica is updated, a new root tree object is created, leaving the previous one intact and potentially sharing some or all of its subtrees and data blocks. CausalRPC exploits this data-sharing property to achieve space-efficient representations of its internal state.

The causal history of a replica is a directed acyclic graph of *commits*: pointers to tree objects with associated metadata. The commits are also stored in the content-addressable heap, allowing replicas to share portions of their history as well as their data. The current state of each replica is tracked by a key-value store that maps names to commit hashes. These mutable key-value pairs are known as *references*. The structure of the store is depicted in Figure 2.2.

Each repository keeps a distinction between the replicas that it manages locally (known as *branches*) and the replicas managed by other processes (known as *remotes*). Two content-addressable heaps on different machines can be made consistent by synchronously pushing or pulling differential state updates; two replicas within the same heap are made consistent using a three-way merge algorithm.

Git provides the features needed to implement a CRDT system, but directly interfacing with the POSIX Git repository format is inefficient and unreliable for a high-throughput system. Git was intended for manual invocation, rather than automation. We shall use Irmin as an intermediate layer to solve these problems.

```
type t (* An Irmin store *)

(* Get a value from the store tree *)
val get: t
  -> string list
  -> value option Lwt.t

(* Update the store via a commit *)
val set_tree: t
  -> Info.f        (* Commit metadata *)
  -> string list  (* Path from root   *)
  -> tree          (* Tree object      *)
  -> (unit, write_error) result Lwt.t
```

(A) *The Store module*

```
type db (* A handle on a store *)

(* Sync a remote store *)
val push: db
  -> remote
  -> (unit, push_error) result Lwt.t

(* Get changes from a remote, optionally
   creating a merge commit. *)
val pull: db
  -> remote
  -> [ ‘Merge of Info.f | ‘Set ]
  -> (unit, pull_error) result Lwt.t
```

(B) *The Sync module*

LISTING 2.1: *A simplified view of the Irmin API. The return values are embedded in the* `Lwt.t` *concurrency monad, introduced in § 2.1.5.*

2.1.3 The Irmin database library

Irmin is an OCaml database library with the same data and consistency models as Git. To use the data model, the user supplies a 'backend' (which implements the reference store and the content-addressable heap) and a three-way merge function over some type B. Irmin then returns a Store module used to construct and modify datastores containing tree objects with blobs of type B. The Store module provides a simple get/set API for interacting with an Irmin store, as shown in Listing 2.1a. All store updates result in a new commit, using metadata supplied by the user, so all state changes are non-destructive and are explicitly tracked in the store's commit graph.

With the correct choice of backend, the contents of an Irmin store are persisted on a POSIX filesystem and can be inspected and modified using the `git` command line tool. Many other backends exist, including in-memory datastores on Unix and MirageOS.

The Store module also contains methods for merging multiple branches. Whenever two tree objects are merged, Irmin handles the recursive merging of subtrees and the user's merge function handles merging of blobs. In this way, we can consider Irmin as providing a 'polymorphic' tree CRDT parameterised on a leaf CRDT supplied by the user.[2]

Similarly to Git, Irmin provides a push/pull API for synchronising remote datastores: the user supplies a data-exchange implementation, and Irmin returns a corresponding Sync module (summarised in Listing 2.1b). The data-exchange format of CausalRPC will be provided by Ocaml-git [50], an implementation of the Git remote protocol written purely in OCaml. This has the advantage of maintaining compatibility with standard Git repositories without requiring non-OCaml stubs that restrict portability.

The Irmin API closely resembles the query, update and merge functions of CRDTs, but there are subtle differences. Irmin relies upon the user to ensure that all updates are safely sent to all replicas, and provides an 'unsafe' push/pull API for doing so: for example, `pull` has a `Set` option that causes Irmin to *rewrite* the causal history of the local replica to match the remote. This sort of disparity between Irmin and CRDT theory will become important in Chapter 3.

[2]Strictly speaking, the mergeable types offered by Irmin are an *extension* of CRDTs, due to the built-in tracking of causal history [22]. Since it is possible to explicitly encode causal history into any CRDT [5], the two models are equally expressive. For this reason, I will continue to refer to Irmin objects as CRDTs.

2.1.4 Typed RPC interfaces in OCaml

RPC interfaces typically aim to emulate the syntax of local computation as closely as possible. Early RPC systems [70, 67] achieved this by providing Unix tools to automatically generate function stubs given a set of prototypes specified in some interface description language (IDL). Later systems avoided the need these tools by integrating RPC interface generation into the compilation phase; some languages even went as far as to provide first-class support for RPC (e.g. Java RMI [56], Erlang [21]). These more integrated approaches achieve superior static type-safety and convenience at the cost of interoperability between different languages (and incompatible versions of the same language runtime).

Functional languages allow a compromise approach in which RPC interfaces are defined within the application source code, without having to embed the RPC implementation in the runtime itself. We shall see in Chapter 3 that this requires exploiting two advanced features of OCaml: higher-order functors and generalised algebraic data types.

OCaml module system

The OCaml module system is detailed in *Real World OCaml* [47]; the following is a brief summary. Type declarations and values with related functionality can be grouped into *structures*, which are named to form *modules*. The interface of a module is defined by a *signature*: a set of type declarations and nested signatures, which can be named to form *module types*. The types within a signature can be either concrete or abstract, allowing for a range of information hiding.

Modules can be composed using *functors*: modules that take other structures as parameters on construction. These can be used to parameterise a module on a set of user-supplied[3] functionality. For example, an RPC client might be provided as a MakeClient functor that requires the application developer to supply a Backend module as an argument. OCaml supports *higher-order* functors: functors that take other functors as arguments. These allow the user to supply a module that is defined in terms of functionality that they don't have access to. For instance, constructing an RPC client might require passing an interface description that is parameterised on an IDL module:

$$\mathsf{MakeClient} : \quad \mathsf{Backend} \longrightarrow (\mathsf{IDL} \longrightarrow \mathsf{Interface}) \longrightarrow \mathsf{Client}$$

Internally, the MakeClient functor supplies the IDL module necessary to build the interface. This restricts the user's use of the IDL module to defining Interface modules, potentially preventing them from using the IDL to construct unsafe methods of invoking the RPC client.

Generalised algebraic data types

In order to detect misuse of an interface at compile time, we require a method of using the IDL to encode the 'shape' of an interface in the type system. This will be achieved with *generalised algebraic data types* (GADTs). An algebraic data type (also *variant type*) is a potentially-recursive union of tagged product types. For example, one might define a binary tree as:

$$\mathsf{type}\ \underbrace{\alpha\ \mathsf{tree}}_{\textit{Type constructor}} = \underbrace{\mathsf{Branch\ of}\ \alpha\ \mathsf{tree} \times \alpha \times \alpha\ \mathsf{tree} \mid \mathsf{Leaf}}_{\textit{Data constructors}}$$

This ADT is *regular*: the type constructor is always directly applied to the polymorphic type parameter α. *Generalised* ADTs relax this constraint by allowing the user to precisely define the

[3]Throughout this dissertation, I use the term 'user' to refer to the developer using a particular library API. For example, the user of an RPC system is the application developer.

'type' of each data constructor (formally known as its *kind*). This can be used to expose extra information about a value within its type. For example, we can expose the height of a binary tree within its type, and then use this information to guarantee that the tree is always balanced. We first define type-level Peano integers:

$$\begin{aligned} &\text{type zero} &&= \text{Zero} \\ &\text{type } \eta \text{ succ} &&= \text{Succ of } \eta \end{aligned}$$

Then we extend the tree ADT to expose the height of all trees in a non-regular type parameter:

$$\begin{aligned} \text{type } (\alpha, \_) \text{ balTree} = \;& \text{Branch} : (\alpha, \eta) \text{ balTree} \times \alpha \times (\alpha, \eta) \text{ balTree} \longrightarrow (\alpha, \eta \text{ succ}) \text{ balTree} \\ \mid\;& \text{Leaf} : (\alpha, \text{zero}) \text{ balTree} \end{aligned}$$

The Branch data constructor takes a value of type α and two trees of height η and returns a tree of height η succ, ensuring that all trees are balanced. The height type parameter could also be used to implement functions that only accept trees of a certain minimum height, allowing many user errors to be caught by the type system at compile time. This example exhibits a general pattern that we will see throughout Chapter 3:

1. We want to enforce some safety property by exposing information at the type level (e.g. trees must be balanced).
2. We begin by defining a set of values that represent the information we need to enforce the safety property (e.g. Peano integers).
3. We define a GADT with data constructors that preserve the safety property for the terms they construct.
4. Functions over this GADT use the exposed type information to ensure that their input values are well-formed (e.g. a tree taken as an argument contains at least one value).

As an aside, functions over non-regular types require *polymorphic recursion* [53]: recursion where the polymorphic type variables may change at each recursive invocation. Unfortunately, type inference for such functions is known to be undecidable [37], so the implementation of CausalRPC will require heavy use of *polymorphic type annotations* to prevent the compiler from inferring a less general type than is intended.

2.1.5 Concurrent programming in OCaml

Irmin uses the Lwt concurrent programming library [75] in order to leverage the non-blocking I/O operations of Ocaml-git. This library provides cooperative threads via a concurrency monad [14]. The protocol layer of CausalRPC will process requests within Lwt threads in order to exploit request-level parallelism.

The relevant fields of the Lwt signature are shown in Listing 2.2. A thread of type α Lwt.t is initially *sleeping*, before being woken by the Lwt scheduler; after some computation, it resolves internally to a value of type α or fails with an exception. Values can be embedded into threads using the `return` operator and then supplied as inputs of other threads using the monadic `bind` operator (with a left-associative infix synonym >>=). The thread `t` >>= `f` first waits for thread `t` to return some value `x`, then computes `f(x)` and behaves as the resulting thread.

Threads can be combined to run concurrently using the `join` and `pick` operators: `join ts` resolves to unit when *all* threads in the list of threads `ts` have resolved, whereas `pick ts` resolves to the result of the *first* element of `ts` to resolve. Lwt threads are 'cooperative' in that they must explicitly `yield` control to the other threads rather than being preempted by a scheduler, giving the user very granular control over the available concurrency in the program.

```
type 'a t (* The monadic type constructor *)

val return      : 'a -> 'a t                      (* Return a value in a new thread *)
val bind (>>=)  : 'a t -> ('a -> 'b t) -> 'b t    (* Compose two threads            *)
val map  (>|=)  : 'a t -> ('a -> 'b) -> 'b t      (* Compose a thread & a function  *)
val join        : unit t list -> unit t           (* Wait for all threads to return *)
val pick        : 'a t list -> 'a t               (* Wait for only the first thread *)
val yield       : 'a t -> 'a t                    (* Allow another thread to run    *)
```

LISTING 2.2: *Functions on cooperative threads provided by Lwt.*

2.2 Requirements analysis

The original project proposal (Appendix E) emphasised traceability as an important requirement of the project. After an initial investigation into data-oriented RPC, I hypothesised that several other desirable properties could be achieved. This considerably broadened the scope of the project. In particular, CausalRPC aims to provide the four classic properties not achieved by conventional RPC systems (as highlighted in Chapter 1):

- **Traceability –** CausalRPC should construct a trace of the causal history of the user's data.
- **Consistency –** CausalRPC should ensure that all RPCs maintain the consistency of the user's data, regardless of the concurrent interleaving of requests.
- **Type-safety –** CausalRPC should leverage the OCaml module system to implement a type-safe interface description language, without requiring external source code generation.
- **Scalability –** CausalRPC should scale to support processing RPCs in parallel.

Additionally, an implicit design goal of any RPC system is interoperability with existing systems and software: the RPC abstraction seeks to make remote computation syntactically equivalent to local computation. These design goals will form the basis of the project's evaluation.

After investigating suitable project extensions, the work was segmented into work packages. This involved an assessment of the priority and risk of each task to be performed, as depicted in Figure 2.3. I scheduled these work packages according to the timeline specified in the project proposal; two extensions were left as reach goals in the event that the project ran ahead of schedule (see Figure 2.4).

2.3 Starting point

The starting point is as stated in my project proposal (Appendix E). In short, I have no prior experience with OCaml or RPC frameworks beyond the relevant courses in the Tripos.[4] I have never worked on a project in the functional programming style. I completed my project before taking the 'Concepts in Programming Languages' course, which discusses the ML module system.

2.4 Software engineering techniques

2.4.1 Development model

The project implementation can be partitioned into phases of increasing complexity, suggesting the use of an iterative development process. I adopted the *spiral model* [9], which allows a dynamic

[4]These are: Part IA Foundations of Computer Science, Part IB Concurrent and Distributed Systems, Part IB Compiler Construction and Part II Types.

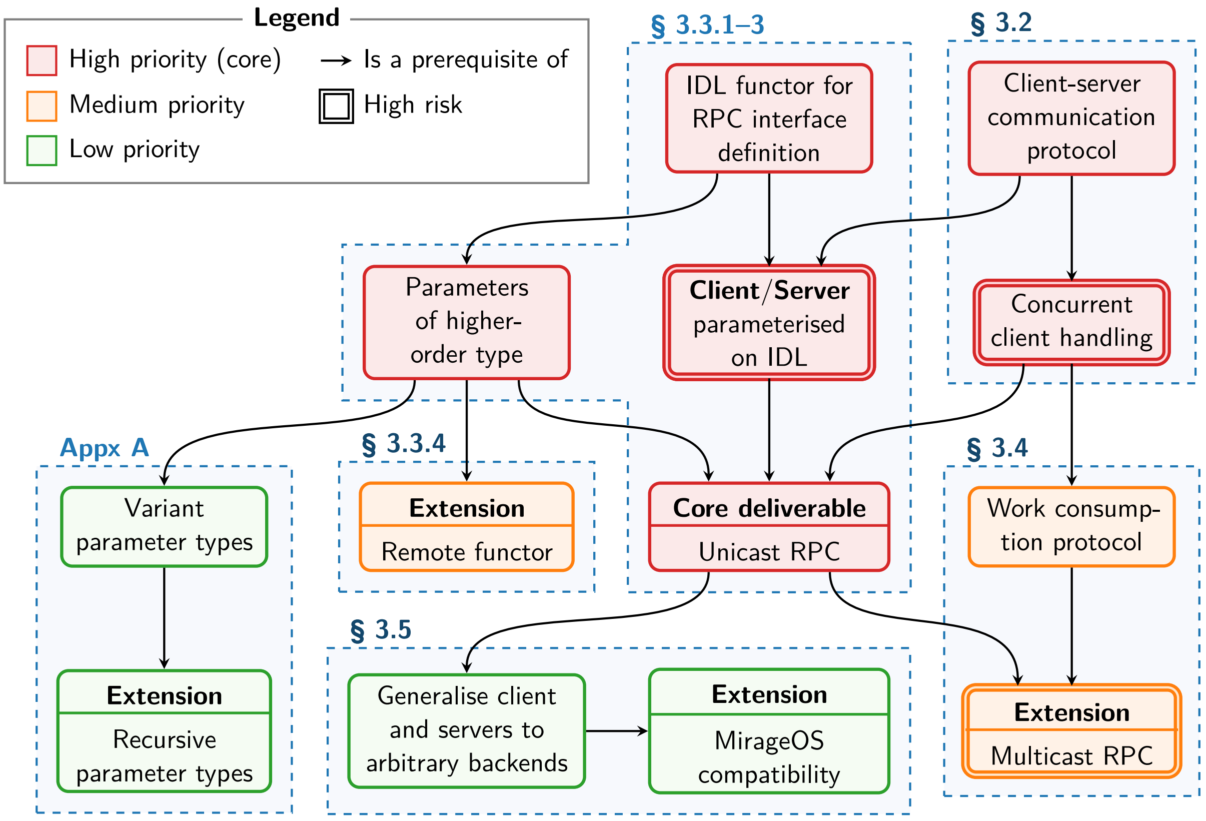

Legend
High priority (core)
Medium priority
Low priority
Is a prerequisite of
High risk
§ 3.3.1–3
IDL functor for RPC interface definition
§ 3.2
Client-server communication protocol
Parameters of higher-order type
Client/Server parameterised on IDL
Concurrent client handling
Appx A
Variant parameter types
§ 3.3.4
Extension
Remote functor
Core deliverable
Unicast RPC
§ 3.4
Work consumption protocol
Extension
Recursive parameter types
§ 3.5
Generalise client and servers to arbitrary backends
Extension
MirageOS compatibility
Extension
Multicast RPC

FIGURE 2.3: *The dependencies between the project work packages, grouped according to the stage of the iterative design process in which they were implemented. All of the planned work packages were successfully completed.*

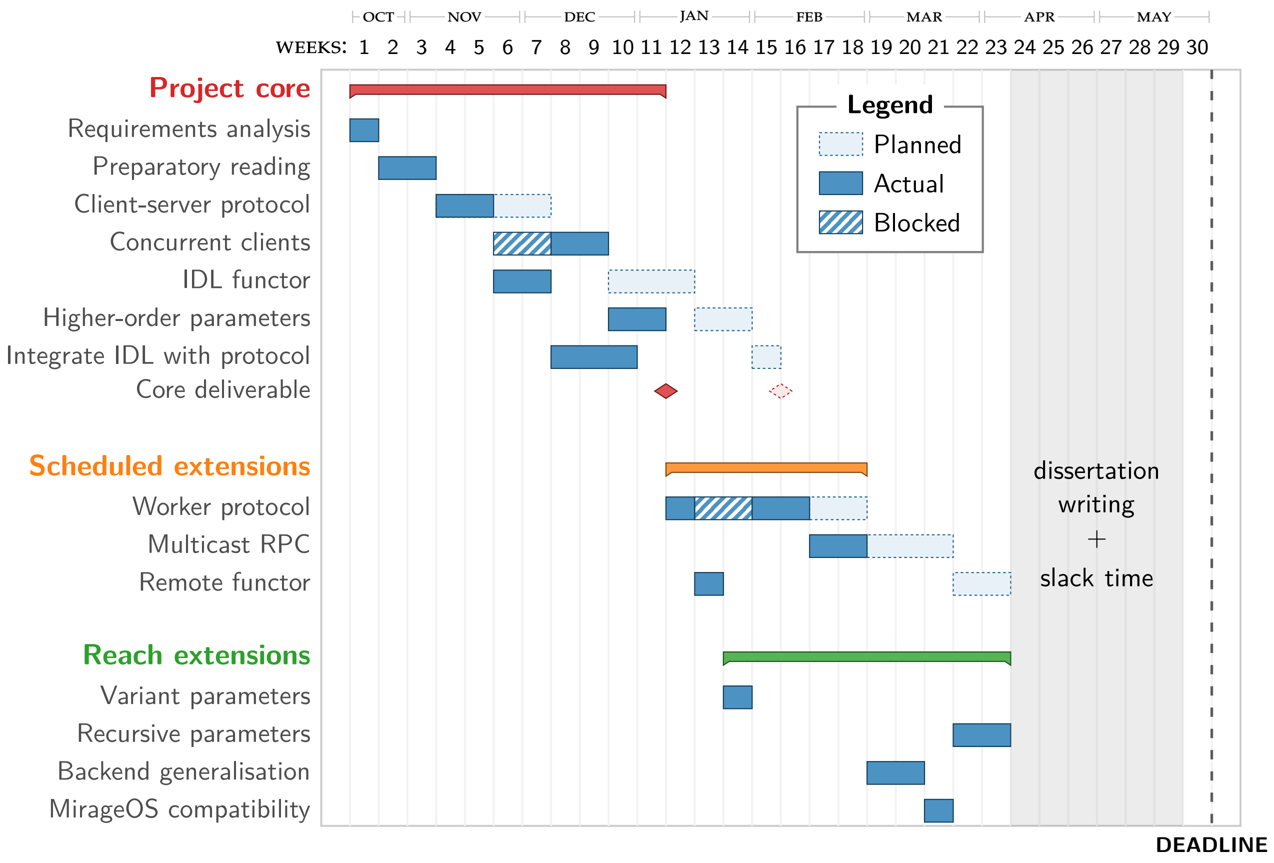


FIGURE 2.4: *Gantt chart of the project development process.*

trade-off between sequential and iterative processes according to the estimated risk of a work package. The spiral development process is constrained by six invariants [10], each designed to prioritise risk-management; I was careful to meet these invariants in order to maximise the chance of the project's success.

The core of the project was completed in two waterfall-like phases, each with non-negotiable deliverables. The more ambitious and theoretically-challenging extensions were completed using an Agile workflow, allowing the requirements to evolve rapidly throughout development.

I identified several work packages as being particularly high-risk due to their heavy reliance on advanced features of dependencies; these necessitated a more involved risk-management process. Before beginning each high-risk package, I performed a *spike* [6]: a time-boxed investigation of the problem to be solved, resulting in a time estimate for the work package. This allowed any blocking problems to be discovered as early as possible. The high-risk packages were scheduled alongside less volatile work. This proved to be an effective precaution, as the time lost to upstream blockages could be spent developing other project features (see Figure 2.4).

2.4.2 Version control, backup and testing

My project and dissertation source code were version-controlled with Git, using a feature branch workflow to allow me to work concurrently on multiple features in the event of blocking problems. I kept backups of these repositories on GitHub and Amazon S3.

The project has three levels of testing: unit testing of individual modules, integration testing of larger components, and end-to-end testing of the full system. I chose to use the Alcotest testing framework [48] to automate testing of my project as it has support for testing Lwt threads. This testing process was invoked once per commit via a continuous integration pipeline hosted by Travis CI.[5]

2.4.3 Open-source development

Many of the project's dependencies are in early stages of development. A substantial fragment of the implementation effort involved solving issues with these dependencies: either tracking down bugs or adding new features as required. This required me to collaborate with other developers and make several open-source contributions; these are documented in footnotes throughout Chapter 3 where relevant. I have included my set of changes to Irmin alongside the CausalRPC source code. The design and implementation of CausalRPC remains entirely my own.

The full list of project dependencies is given in Appendix D, along with their version numbers and licenses at the time of submission. All of my project dependencies are open source with permissive licenses (ISC, MIT, BSD-2-Clause or GPLv2), so my use of them is authorised.

2.5 Summary

CausalRPC is an OCaml RPC framework designed using the principles of CRDTs. The client and server communicate by pushing and pulling changes to a shared Irmin database via the Git remote protocol. The selection of project dependencies is briefly justified in Table 2.1. The design goals of CausalRPC seek to support the hypothesis that RPC systems can avoid the four major pitfalls of conventional RPC semantics by taking a functional, data-oriented approach.

[5]URL: https://travis-ci.com (visited on 11th January, 2019)

Tool	Justification for use
OCaml	Powerful type and module systems. Access to the MirageOS ecosystem.
Irmin	Distributed Merkle trees with CRDT-like semantics. Interchangable network and storage modules for portability.
Git	Differential state updates for Irmin. Popular tool for viewing non-linear logs.
Lwt	Cooperative threading library with minimal kernel dependencies.
Alcotest	Lightweight testing framework with support for testing Lwt threads.
Logs_lwt	Logging framework with Lwt-synchronous log commands.
Core Bench	Microbenchmarking suite used to inform low-level optimisations.
Mirage-profile	Lwt thread profiling tool; used for debugging and to inform design decisions.

TABLE 2.1: *The major dependencies of CausalRPC, along with brief justifications for their selection. Where possible, tools within the MirageOS ecosystem were selected to maintain inter-compatibility.*

3 Implementation

This chapter describes the design and implementation of CausalRPC, discussing the various challenges that were encountered and design decisions that were made. I begin by describing my stronger RPC semantics (§ 3.1). The subsequent sections describe stages of the iterative design process: starting from a client-server communication protocol (§ 3.2), then generalising on an interface description language (§ 3.3.1–3), then adding native support for RPC composition (§ 3.3.4). Section 3.4 discusses how data-oriented RPC can be smoothly transitioned into a distributed computation framework. Finally, I discuss adapting CausalRPC for usage in unikernels (§ 3.5–6).

3.1 Data-oriented RPC

The premise of data-oriented RPC is that remote procedure calls are fundamentally about performing computations over distributed data structures. The classic example of this type of distributed data structure is a network file system, the famous use-case of SunRPC [46]. The challenges with this system arise when the file system is subject to the concurrent requests of multiple clients, which may have out-of-date world-views. CausalRPC solves this problem by storing the shared data structure in a Git repository. Each RPC takes the current state of the data structure as an argument, performs some transformation, and delivers the result back to the client. For example, a move operation over the file system might have the type:

$$\textsc{move}: \underbrace{\mathsf{string\ list}}_{\textit{Source}} \rightarrow \underbrace{\mathsf{string\ list}}_{\textit{Destination}} \rightarrow \underbrace{\mathsf{filesystem} \rightarrow \mathsf{filesystem}}_{\textit{Functional transform}}$$

Clients will request RPCs by committing to the *same* repository as stores the data. The commit predecessor graph of each 'request' commit indicates the causal history of the operation, giving the context necessary to merge two RPCs with concurrent causal histories. The user (application developer) will supply a three-way merge function that encodes the logic of this merge: for example, by applying renames before moves and entering an error state if two concurrent renames are made to the same directory.

The first iteration of CausalRPC supports only a single user-defined operation with type $\nu \rightarrow \nu$, where ν is an Irmin-serialisable type. The API is a functor parameterised on three values:

- the type ν;
- a three-way merge function over values of type ν;
- a function of type $\nu \rightarrow \nu$ to be advertised by the server.

These simplifications allowed me to focus on the conflict resolution protocol that deals with race conditions between multiple clients.

As an aside, we can easily recover conventional RPC semantics from this system by considering the underlying 'data structure' to be a variant type with data constructors Input and Output that take values of the appropriate types. However, this would be against the spirit of the design of CausalRPC: the intent is that the store contains any shared mutable state, so that the user can rely on the framework to preserve the consistency of this state. The use of differential state updates at the Git layer means that the data structure can be very large without sacrificing performance.

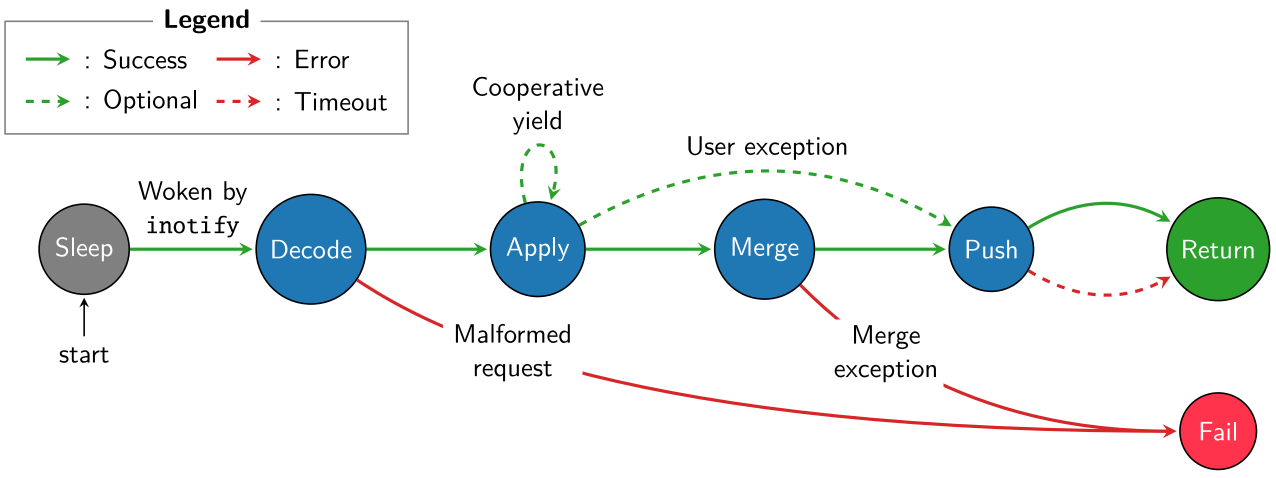


FIGURE 3.1: *Finite-state machine describing the lifecycle of a worker thread. All Lwt threads are initially sleeping, and eventually resolve to either* ***Return*** *or* ***Fail*** *after a sequence of computations inside the monad. Cooperative RPCs may yield to other worker threads to mask the latency of non-compute bound operations.*

3.2 Client-server communication protocol

At its core, the client-server communication protocol is simple: the client creates a 'request' commit, descended from a commit that points to the initial state to be transformed, and pushes it to the server. The server detects incoming requests by supplying a callback function to an Irmin 'watcher'. When using the Unix filesystem backend, the Irmin watcher uses the Inotify kernel subsystem [40] to detect changes to the store; the Irmin watcher for the memory backend is implemented using high-frequency polling. We shall see in Chapter 4 that this has important implications for performance. The Irmin watcher callback spawns a CausalRPC worker thread to handle each request, which mutates the store as requested and merges the result into a global 'server' branch containing the full causal history of the user's data structure. The lifecycle of a server thread is shown in the form of a finite-state machine in Figure 3.1. The top-level implementation of this thread is given in Appendix A.1.

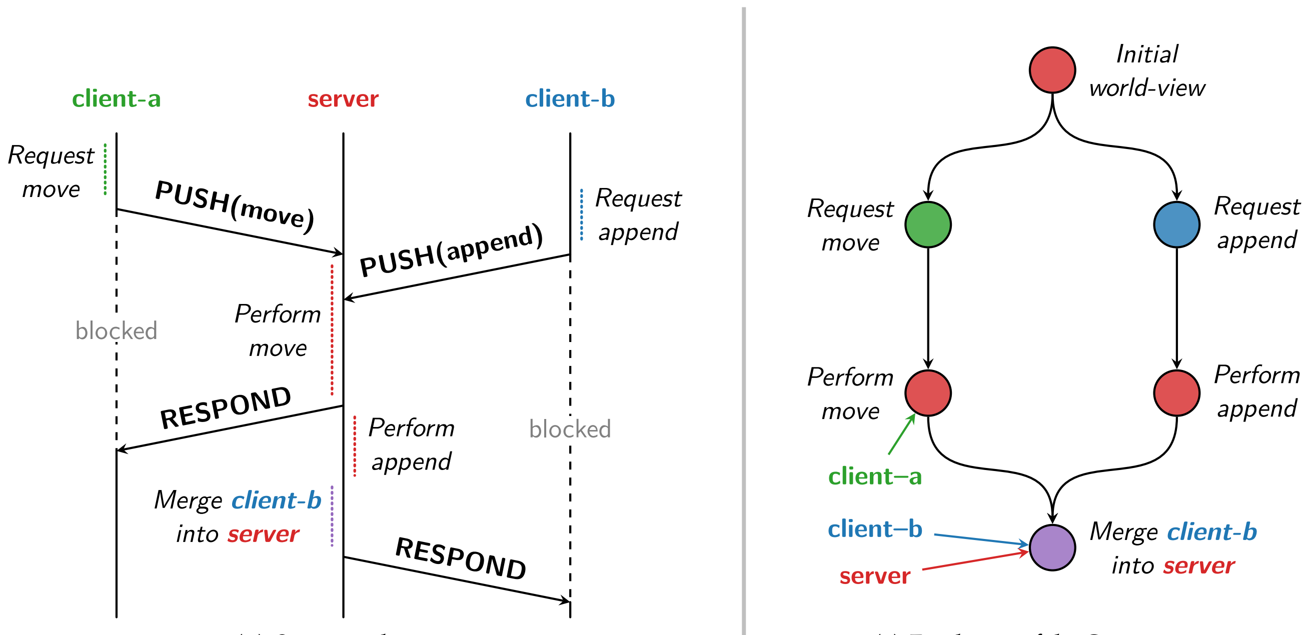
For brevity, I will avoid an in-depth discussion of the implementation details in favour of highlighting three key issues: managing the fan-in of RPC requests (§ 3.2.1), executing RPCs concurrently (§ 3.2.2), and delivering the result to the client (§ 3.2.3).

3.2.1 Job queue contention

The Irmin push/pull interface (shown in Figure 2.3b on page 6) allows conflicting changes from a remote to be automatically merged with a local branch when pulling, but not when pushing: an Irmin push updates the remote branch to point to the same commit as the local branch *even if this requires rewriting the causal history of the store*. As a result, if multiple clients were to submit concurrent requests to the same remote, the final request to be received by the Irmin layer would overwrite those of the previous clients.

This is a somewhat surprising aspect of the behaviour of Irmin, as it differs from conventional Git semantics. There are two solutions to the contention problem:

- **Client responsibility**. The server rejects any conflicting pushes, leaving the client to merge their changes locally and retry. This is the default behaviour of the '`git push`' command.
- **Server responsibility.** The server resolves the conflicting push automatically at the remote by invoking the merge function, as is done with the Irmin pull method.



(A) *Sequence diagram* (B) *Final state of the Git repository*

FIGURE 3.2: *Using a merge to resolve a race condition between two clients. Note that* **client–b** *receives the update made by* **client–a**, *but not vice versa.*

These solutions have different trade-offs: the server-side merge avoids the need to retry a push, reducing contention for the server's network stack at the cost of additional server-side compute; however, this requires an out-of-band mechanism to report merge failures to the client. After discussion with the Irmin developers, the client-side solution was adopted for general-purpose use by Irmin.[1] CausalRPC is particularly latency-sensitive, so I implemented the server-side solution for my own use.

Worker threads receive work on client-specific branches, which are guaranteed to be free of contention. These branches are explicitly merged into a central `server` branch using test-and-set operations. Figure 3.2 demonstrates the behaviour of this solution for the case of two simultaneous requests from up-to-date clients.

If the three-way merge fails, the application layer is unable to reconcile the result of the RPC with the current state of the system. The system *could* simply raise a client-side exception in this case, as is done with user-raised exceptions from within the RPC itself. However, it was decided that a better semantics would be to have the system enter an 'application error' state. The user can then conduct a post-mortem by examining the commit graph of the repository via '`git log`', hopefully simplifying the debugging process.[2]

3.2.2 Cooperative concurrent execution

Conventional RPC frameworks are typically able to execute multiple RPCs concurrently by delegating request handling to a pool of preemptive worker threads. However, the threading model used within CausalRPC is cooperative: threads must explicitly yield to one another. As a result, if the user supplies a function of the type $v \to v$, invoking this function at the server-side must be synchronous. This sacrifices the ability to exploit thread-level parallelism if not all RPCs are

[1]Irmin issue tracker: 'Support fast-forward only push operations'. URL: `https://github.com/mirage/irmin/issues/596` (visited on 21st April, 2019)

[2]The in-memory backend can be traced by examining the commit graph at runtime. However, this trace is bound to the lifetime of the CausalRPC process. To combat this problem, CausalRPC offers an option to automatically dump an in-memory store into a specified directory on the file system when an error state is detected.

compute-bound (e.g. if the user's code involves reading from disk).

CausalRPC overcomes this issue by allowing RPC operations to be expressed as Lwt threads (functions of type $\nu \rightarrow \nu$ Lwt.t), which are then scheduled within the same cooperative thread queues as the worker threads. When the user code reads from disk, for example, it may `yield` to allow another RPC to be processed in the intervening time.

3.2.3 Delivering the updated state

Once a worker thread has computed the return value of an RPC (either an updated state or an exception raised from within the user's code), this must be passed to the client. As before, the key design decision is whether this process should be client-driven or server-driven. The client could poll the server for the result, but this is network-intensive and sets a lower bound on the RPC latency (the polling interval). Alternatively, the server could explicitly push the result to the client, but this requires the server to track the location of each of the active clients and solve the issue of *Byzantine* client faults, whereby the client is intermittently unavailable to receive the results of their RPC.

This issue of dealing with so-called *orphaned* RPCs is a classic problem in traditional RPC [71, 57]. CausalRPC solves this problem by falling back on client pulls if the server push fails. The server merges all RPC results into the `server` branch, even if the client is unavailable to receive the result. If the Byzantine client later pulls from the server (or issues another RPC), they will receive the result of their previous RPC. As a result, inability to deliver an RPC result is not a failure case for a worker thread (as indicated in Figure 3.1).

3.3 User interface

The first iteration of CausalRPC supports concurrent remote execution of a single pre-defined operation, with no parameters other than the most-recent state of the store. The second iteration allows the user to define and implement a *set* of operations with one or more formal parameters, and then invoke those operations remotely as if they were local functions. This section will consider four subproblems:

§ 3.3.1 *How do we encode and decode parameters in the store?*

§ 3.3.2 *How does the user specify their RPC interface?*

§ 3.3.3 *How should the user invoke a specific RPC?*

§ 3.3.4 *How can the user efficiently compose RPCs?*

3.3.1 Synthesising user-supplied parameters

The blobs in an Irmin store must have the same type. The first iteration of CausalRPC uses a store with a finite type universe (RPC requests, branch names, user values, etc.), which can be encoded as a single variant type. However, we'd like to support remote execution of any function of the type:

$$\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \nu \rightarrow \nu \text{ (Lwt.t)}$$

where ρ_i are Irmin-serialisable types.[3] This requires serialising the auxiliary parameters ρ_i and including them in the RPC request. In order to support composite data types (lists, variants, etc.), the CausalRPC store must be capable of storing values from an infinite type universe. Worse, in

[3]Note that we still require the most deeply nested closure to have type $\nu \rightarrow \nu$ (or $\nu \rightarrow \nu$ Lwt.t) in order to preserve the correct store type.

```
module Type : sig
  type 'a t (* Type witnesses *)
  val equal: 'a t -> 'a -> 'a -> bool

  val bool: bool t
  val int: int t
      (** ... string, unit, etc. *)

  val list: 'a t -> 'a list t
  val pair: 'a t -> 'b t -> ('a * 'b) t
      (** ... option, result, etc. *)
end
```

(A) *Type witness constructors*

```
module Pickled : sig
  type t (* Pickled format *)

  val irmin_t: t Irmin.Type.t
  val test_t: t Alcotest.testable

  val pickle: 'a Type.t -> 'a -> t
  val unpickle: 'a Type.t -> t -> 'a
end
```

(B) *Corresponding pickler and unpickler*

Listing 3.1: *Summary of the modules necessary to support user-defined parameters.*

order to pass these auxilliary parameters to the user's functions at the server side, we must first recover their type information. We need one function to 'hide' the type of a parameter before storing it in the Irmin heap at the client side and one to recover the type information at the server side:

$$\begin{aligned} \text{HIDE:} &\quad \forall\alpha\ .\ \alpha \longrightarrow \mathsf{Serialised.t} \\ \text{RECOVER:} &\quad \forall\alpha\ .\ \mathsf{Serialised.t} \longrightarrow \alpha \end{aligned}$$

However, these functions are not type-safe.[4] A statically-typable solution requires some form of 'type cast', whereby the user can assert the type of α that they expect to be hidden in (or recovered from) the value of type Serialised.t. CausalRPC achieves this via a strategy known as *pickling* [36]:

$$\begin{aligned} \text{PICKLE:} &\quad \forall\alpha\ .\ \alpha\ \mathsf{Type.t} \longrightarrow \alpha \longrightarrow \mathsf{Pickled.t} \\ \text{UNPICKLE:} &\quad \forall\alpha\ .\ \alpha\ \mathsf{Type.t} \longrightarrow \mathsf{Pickled.t} \longrightarrow \alpha \end{aligned}$$

The set of values that can be 'pickled' is constrained by the set of *type witnesses* (values of type α Type.t) exposed by the Type module. A witness of type 'a Type.t' is a run-time representation of the type a with the built-in feature of being able to convert parameters of type a to and from the Pickled.t format. The Type module supplies a set of basic witnesses (int Type.t, string Type.t, etc.) and a set of *combinators* to allow the creation of witnesses of composite types, as shown in Listing 3.1a.

The basic type witnesses are simple to implement, as they need only be distinguishable from one another. Witnesses for composite types (lists, arrays, etc.) must store the witnesses of their elements, so that the corresponding pickler can be derived recursively. The challenging cases are types with user-defined constructors (such as variants) and recursively-defined types. Variant witnesses require the user to provide the corresponding data constructors and deconstructors as functions.[5] Recursive type witnesses are expressed by supplying a function f that defines the witness in terms of itself; CausalRPC then derives the actual type witness as the iterated least fixed point of f, as discussed by Abel and Matthes [2].

The signature of the Pickled module is shown in Listing 3.1b. Crucially, the module exposes an `irmin_t` value which tells the Irmin layer how to derive the byte-level representation of pickled

[4] See [19] (p. 279) for a detailed explanation.
[5] This uses the Scott encoding for implementing ADTs in the simply typed lambda calculus [33].

```
open Idl(Filesystem)

(* Define the variant type for a file *)
type file = Regular of bytes
          | Folder of (string * file) list

(* Give CausalRPC the constructors & pattern matcher *)
let file = mu (fun file -> variant
    |+ bytes                        (fun b -> Regular b)
    |+ (list (pair string file)) (fun l -> Folder l)
    |> (fun r f -> function
      | Regular bytes -> r bytes
      | Folder contents -> f contents)

let put = declare "put" (file @-> list string @-> base)
```

Listing 3.2: *Use of the IDL to define the prototype of the* PUT *operation in Equation (3.1). Describing the* `file` *type requires passing its data constructors and destructors.*

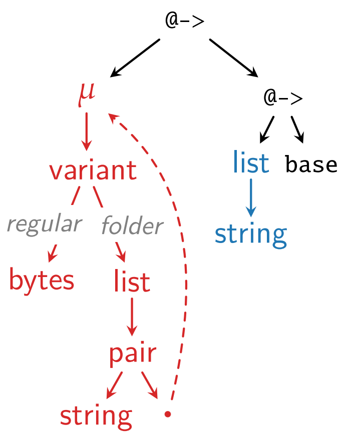


Figure 3.3: *Run-time representation of the prototype of the* PUT *operation in Equation (3.1).*

values, which in our case will be a JSON-compatible format.[6] The module also provides a `test_t` value, allowing the pickled values to be integrated into the Alcotest unit tests.

Whenever a parameter is pickled or unpickled, the corresponding type witness is required. On the client-side, this is arranged by having each RPC operation contain the type witnesses of its parameters. The server side stores a hashmap from the names of its RPC operations to the corresponding type witnesses. Using these witnesses, CausalRPC is capable of storing and retrieving parameters of any Irmin serialisable type from the store.

3.3.2 Interface description language

The user must define an RPC interface in order to construct a CausalRPC client and then implement that interface in order to construct a CausalRPC server. This section discusses the interface description language (IDL) of CausalRPC, which allows the user to construct these modules. The CausalRPC IDL operates entirely within the language runtime, unlike most RPC frameworks.

Each RPC operation takes zero-or-more auxiliary parameters, which are serialised in the RPC request that is sent to the server. We have seen that parameter types can be described by composing basic type witnesses. The same approach is used to build run-time representations of RPC prototypes by composing representations of each parameter type. The most basic prototype is a transformation over the user-supplied store value v (potentially within an Lwt thread). Parameters can be recursively added to this basic prototype using the `@->` combinator:

$$\begin{aligned}
\textsc{base}: &\quad (v \longrightarrow v) \qquad \mathsf{Proto.t}\\
\textsc{base-lwt}: &\quad (v \longrightarrow v\ \mathsf{Lwt.t})\ \mathsf{Proto.t}\\
\texttt{@->}: &\quad \forall \alpha, \beta\ .\ \alpha\ \mathsf{Type.t} \longrightarrow \beta\ \mathsf{Proto.t} \longrightarrow (\alpha \longrightarrow \beta)\ \mathsf{Proto.t}
\end{aligned}$$

Note that the v type parameter is not universally quantified, as it is explicitly specified by the user when constructing a client or server. Internally, this combinator uses GADTs to shape the type

[6] Irmin does not require that blobs be text-based, but the YoJSON format [55] was used within CausalRPC for simplicity. A more performant (but more opaque) binary representation such as that of Protocol Buffers [27] could be implemented instead.

parameter of the Proto.t value to match the type of the corresponding function. As a concrete example, consider defining an RPC operation that recursively adds a file tree to a network file system:

$$\textsc{put}: \quad \mathsf{file} \longrightarrow \mathsf{string\ list} \longrightarrow \mathsf{filesystem} \longrightarrow \mathsf{filesystem} \tag{3.1}$$

Defining the prototype for this operation involves describing the types file and string list, sequencing them together using the `@->` combinator, and finally naming the operation using a `declare` method. This process is shown in Listing 3.2; the resulting prototype is depicted in Figure 3.3. The uses passes a Filesystem module to the IDL, so the `put` value has the expected type:

$$(\mathsf{file} \longrightarrow \mathsf{string\ list} \longrightarrow \mathsf{filesystem} \longrightarrow \mathsf{filesystem})\ \mathsf{Proto.t}$$

We have seen that basic type witnesses can be combined to describe parameter types using combinators, and that these can be combined into Proto.t values using another combinator. The full IDL takes this process one step further: combining prototypes to construct interfaces. This final level ensures that a server implementation *exactly* meets the specifications of an interface description, providing the necessary functions and no more.

In order to use CausalRPC, the user must first define two functors: an Interface and an Implementation, each parameterised on an IDL module. An interface is built using a combinator over prototypes, and the corresponding implementation is built using a combinator over functions. Using GADTs, the structure of each module is exposed in a 'shape' type, allowing the type system to check that an implementation matches its specification. Having defined these modules, the user may construct client and server nodes using corresponding MakeClient and MakeServer functors. A client requires only an interface, whereas the MakeServer functor requires both an interface and a matching implementation:

```
module MakeServer
  (B: BACKEND)                           (* Unix FS / Unix Mem / MirageOS *)
  (C: CONTENTS)                          (* Base type and merge function *)
  (Iface: functor (I: IDL) -> IFACE      (* Interface over correct base type *)
            with type base = C.t)
  (Impl:  functor (I: IDL) -> IMPL       (* Matching implementation *)
            with type shape = Iface.shape): Server
```

From the perspective of the user (and the type-checker) the interface and implementation modules must have structurally equal shape types. The runtime representations of these modules are independent of the shape types: they exist purely to enforce extra safety at compile time. A type parameter that is not required at runtime is known as a *phantom type* [13]. We shall see in Chapter 4 that the use of phantom types allows CausalRPC to achieve stronger static type-safety properties than the vast majority of existing RPC systems.

3.3.3 RPC invocation as function application

Providing a convenient interface for the user to invoke RPCs is challenging. Perhaps the most intuitive mechanism is a curried 'stub' function that takes each of the parameters and issues an RPC, returning the state of the store once the RPC is complete. Implementing this interface requires a stub generator that takes prototypes and returns the corresponding stubs, as in the following examples:

$$\begin{array}{rcl}
\textit{prototype} & \Longrightarrow & \textit{stub function} \\
(\mathsf{int} \rightarrow \mathsf{int})\ \mathsf{Proto.t} & \Longrightarrow & \mathsf{int\ Store.t} \rightarrow \mathsf{int\ Store.t} \\
(\mathsf{file} \rightarrow \mathsf{string\ list} \rightarrow \mathsf{fs} \rightarrow \mathsf{fs})\ \mathsf{Proto.t} & \Longrightarrow & \mathsf{file} \rightarrow \mathsf{string\ list} \rightarrow \mathsf{fs\ Store.t} \rightarrow \mathsf{fs\ Store.t} \\
(\rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \nu \rightarrow \nu\ \mathsf{Lwt.t})\ \mathsf{Proto.t} & \Longrightarrow & \rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \nu\ \mathsf{Store.t} \rightarrow \nu\ \mathsf{Store.t}
\end{array}$$

The arity of the stub (and therefore of the stub generator itself) depends on the Proto.t object that is passed as the first argument. This complicates the implementation of the stub generator: what should its type be? As with the previous problems we have discussed, this problem is solved by using a GADT to recursively build the necessary type of the stub generator. Since the stub generator takes a prototype value, the necessary type can be built within a new type parameter of the Proto.t GADT:

$$\begin{array}{rl}
\textsc{base}: & (\nu \rightarrow \nu,\ \boldsymbol{\nu}\ \mathbf{Store.t} \rightarrow \boldsymbol{\nu}\ \mathbf{Store.t})\ \mathsf{Proto.t} \\
\textsc{base-lwt}: & (\nu \rightarrow \nu\ \mathsf{Lwt.t},\ \boldsymbol{\nu}\ \mathbf{Store.t} \rightarrow \boldsymbol{\nu}\ \mathbf{Store.t})\ \mathsf{Proto.t} \\
\texttt{@->}: & \forall \alpha, \beta, \boldsymbol{\gamma}\ .\ \alpha\ \mathsf{Type.t} \rightarrow (\beta, \boldsymbol{\gamma})\ \mathsf{Proto.t} \rightarrow (\alpha \rightarrow \beta,\ \boldsymbol{\alpha} \rightarrow \boldsymbol{\gamma})\ \mathsf{Proto.t}
\end{array}$$

The type of the stub generator can then be expressed as:

$$\textsc{stub-gen}: \quad \forall \alpha, \beta\ .\ (\alpha, \beta)\ \mathsf{Proto.t} \rightarrow \beta$$

Internally, this function is implemented by recursing over the prototype and accumulating a curried function of the correct arity and type to require the user to supply all of the auxilliary parameters before invoking the RPC.

3.3.4 Efficient RPC composition (EXTENSION)

Certain use-cases of RPC require chaining several RPCs in succession: for example, a network file system user might want to query the contents of a directory and then immediately get the attributes of its contents.[7]

One advantage of stub functions as an RPC mechanism is that they can be composed as if they were local function calls. In OCaml, a standard method of sequencing operations is via the pipe operator |>, which chains multiple transformations over a value. As an example, the pipe operator is used in Listing 3.3a to add values to the start and end of a list. This programming model is a natural fit for CausalRPC, as all RPCs are transformations over the value contained in the store. Unfortunately, chaining RPCs with the pipe operator requires invoking the client-sever protocol once for each RPC. A more efficient operator might amortize the overhead of the network (and the merge function) by aggregating consecutive RPCs into a single 'batch', then sending the RPC chain as a single request. This functionality was selected as an extension.

Functional languages use the *applicative functor* design pattern to augment the pipe operator with additional context. The use of the word 'functor' here does not refer to parameterised OCaml modules, but to parameterised types T with an FMAP operation:

$$\textsc{fmap}: \quad \forall \alpha, \beta\ .\ \alpha\ T \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta\ T$$

We have already seen an example of this: Lwt's >|= operator is a pipe operator that sequences functions inside a threaded context (as demonstrated in Listing 3.3b). This concept can be applied

[7]Indeed, this was such a common use-case for NFSv2 that the revised NFSv3 contained a dedicated `READDIRPLUS` operation [68]. NFSv4 recognises the brittleness of this solution, adding the `COMPOUND` procedure for more general operation composition [30].

LOCAL

```
[2;3;4]
|> List.cons 1
|> List.append [5;6]
```

(A) *Chaining function application in OCaml*

CONCURRENT

```
let thread =
  Lwt.return [2;3;4]
  >|= List.cons 1
  >|= List.append [5;6]
in Lwt_main.run thread
```

(B) *Chaining functions inside Lwt's concurrency monad*

REMOTE

```
let rpc =
  Causal.init [2;3;4]
  <*> list_cons 1
  <*> list_append [5;6]
in Causal.issue rpc
```

(C) *Chaining RPCs inside CausalRPC's remote functor*

LISTING 3.3: *Comparison of sequential computation models in OCaml.*

to remotely-executed functions, resulting in the *remote applicative functor* [26]. The strategy is to explicitly provide an operator for sequencing RPCs within a Remote.t context, and a function that extracts the final state of the store from the Remote.t context:

$$\begin{aligned} \text{<*>} : &\quad \forall \nu\ .\ \nu\ \text{Remote.t} \longrightarrow \nu\ \text{Rpc.t} \longrightarrow \nu\ \text{Remote.t} \\ \text{ISSUE} : &\quad \forall \nu\ .\ \nu\ \text{Remote.t} \longrightarrow \nu\ \text{Store.t} \end{aligned}$$

Listing 3.3c shows these operators being used to issue a sequence of list operations within a remote store. The remote programming model offered by CausalRPC is similar to Lwt's concurrent programming model, with only two differences:

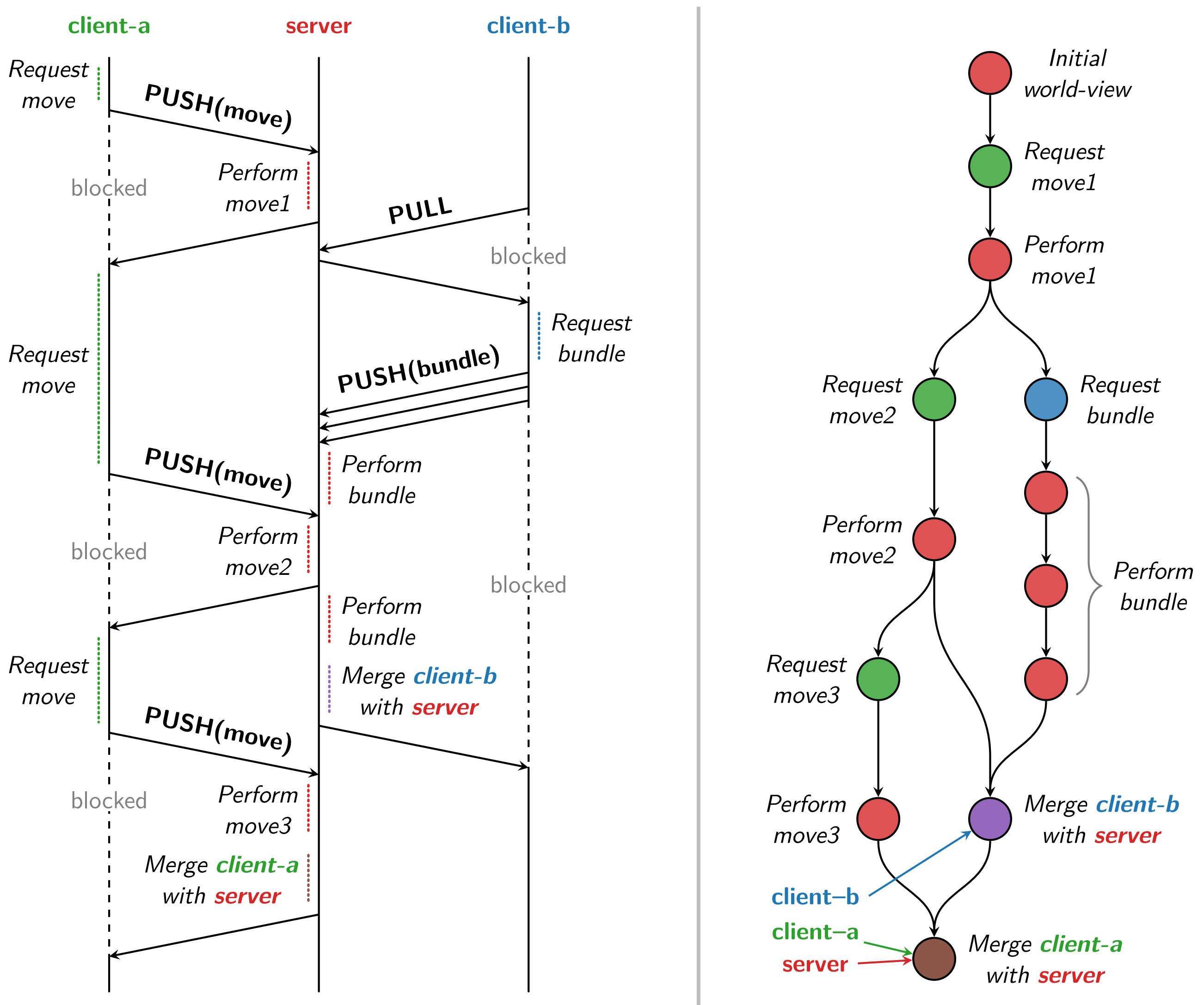
1. The chained operations must be pre-defined within the RPC interface.
2. The chain cannot branch on the values of intermediate results.[8]

Together, these two constraints ensure that any chain constructed via <*> can be executed in a single round-trip between the client and the server. As far as possible, the remote functor extension achieves the RPC design goal of matching the syntax of local computation.

Implementing the remote functor requires extending each of the three layers of CausalRPC (interface, protocol and pickler) to handle bundling of requests. There is one key design decision to be made: should clients be able to receive intermediate results of bundled operations? If yes, the clients have no method of sequencing operations in isolation; if no, the application state may diverge for longer between merges. I chose to implement the latter: bundles of operations are executed atomically within an isolated branch, ensuring isolation between clients. Each operation within the bundle gets its own commit, ensuring that the system retains its traceability properties.

Figure 3.4 shows an example of two clients issuing sequences of RPCs to a server. Client *A* uses standard function chaining to issue three operations, whereas Client *B* uses the remote functor. *A* incurs three round-trip times before the full chain is complete, but is able to perform arbitrary local computation on intermediate results. This example illustrates the design trade-off between isolation and divergence highlighted above: if intermediate results within a bundle were merged as soon as possible, *A* would have received the intermediate stages of *B*'s computation sooner (after the second move, rather than the third).

[8]Due to this constraint, the Remote.t functor does not meet the requirements of a monad. Gill et al. [26] present a generalisation of the remote applicative functor to a remote monad, but this variant is less effective at eliminating round-trip delays between the client and the server.



(A) *Sequence diagram*

(B) *Final state of the Git repository*

FIGURE 3.4: *Two clients each issuing a sequence of three of operations to the same server.* ***client-a*** *uses local function chaining via* |>, *whereas* ***client-b*** *uses the* Remote.t *functor via CausalRPC's* <*> *operator. Note that* ***client-b****'s operations occur atomically with respect to* ***client-a****'s operations, but not vice versa.*

3.4 Worker clusters (EXTENSION)

The major advantage of CausalRPC over other RPC frameworks is that it tracks the causal histories of its clients and knows how to resolve divergences. This allows the server to process requests concurrently using test-and-set operations within the underlying Irmin store. In theory, the requests could be processed fully in parallel, but unfortunately OCaml does not yet support multi-core runtimes [63]. On Unix systems, this can be bypassed by spawning multiple processes, each with its own OCaml runtime, but in order to maintain MirageOS compatibility I decided not to leverage OS scheduling in this way. Instead, this extension exploits parallelism by allowing the server to offload work to a cluster of (potentially remote) worker nodes. This demonstrates the smooth transition from data-oriented RPC to a scalable distributed computation framework such as MapReduce [17] or Hadoop [76].

In order to exploit request-level parallelism at the level of individual clients, I extended the CausalRPC store to contain a *set* of named values of type v; users can issue RPCs over specific values, or over all values in the set using a new parallel FMAP operation. Using the jargon of

MapReduce, the clients request work in units of *jobs*, each of which is a set of one-or-more *tasks* to be performed by the workers. There are two problems to be considered:

§ 3.4.1 *How are tasks allocated, and how are their results accumulated?*

§ 3.4.2 *How do the workers communicate with the serer?*

3.4.1 Task distribution and result accumulation

During any given job, the workers must distribute the tasks amongst themselves. There are many possible solutions to this problem, each with certain trade-offs: static partitioning of the key-space, explicit allocation by the server, distributed consensus among the workers, etc. A core design principle of CausalRPC is to optimistically perform operations and then later resolve any conflicts with a merge operator; this principle suggests an approach whereby workers select tasks randomly, merging their results with each other until the job is complete.[9] If there are relatively few tasks to be performed, it is likely that multiple workers will select the same task concurrently. CausalRPC mitigates this problem by allowing the workers to publicly 'consume' a task before performing it.

The remaining work in a job is described by a pair of sets of tasks: *unselected* tasks awaiting consumption, and *consumed* tasks awaiting completion. The workers each maintain a (potentially out-of-date) replica of this job data structure, from which they select tasks to consume and perform. As the work cluster makes progress on a job, the various replicas of the job data structure will diverge; these divergent states can be automatically merged by designing the work sets as a CRDT. The merge function of this CRDT is non-trivial: it must keep all work performed by the divergent workers (which may be intersecting sets), handle concurrent CONSUME and PERFORM operations of the same task, and generally preserve the *intent* of the workers where possible.

If $\mathcal{T}$ is the set of all tasks, the state-space S of the job is $(\mathcal{P}(\mathcal{T}) \times \mathcal{P}(\mathcal{T})) \cup \{\bot\}$, where $\bot$ is the state prior to initialisation by the server. The server initialises the 'unselected' set, then the workers perform transitions consisting of zero-or-more CONSUME operations of tasks in the unselected set, followed by zero-or-more PERFORM operations of tasks in the consumed set. All valid transitions in S are monotonically non-decreasing with respect to the partial order $\sqsubseteq$ defined by:

$$\frac{\overbrace{u \subseteq \mathcal{T}}^{\substack{\textit{Arbitrary initial}\\ \textit{unselected set}}}}{\bot \sqsubseteq (u, \emptyset)}\ (\textsc{init}) \qquad \frac{\overbrace{u_2 \subseteq u_1}^{\substack{\textit{Zero-or-more}\\ \textsc{perform}\ \textit{operations}}} \qquad \overbrace{c_2 \subseteq c_1 \cup (u_1 \setminus u_2)}^{\substack{\textit{Zero-or-more}\\ \textsc{consume}\ \textit{operations}}}}{(u_1, c_1) \sqsubseteq (u_2, c_2)}\ (\textsc{exec}) \qquad \frac{s_1 \sqsubseteq s_2 \qquad s_2 \sqsubseteq s_3}{s_1 \sqsubseteq s_3}\ (\textsc{trans})$$

Note that S is bounded above by the state $(\emptyset, \emptyset)$ and below by the state $\bot$, forming the lattice $\langle S, \sqsubseteq \rangle$. The merge function implements the least upper bound operator of this lattice: given two divergent states (and their most-recent common ancestor), it returns the earliest state that is not earlier than both input states (w.r.t. $\sqsubseteq$), as illustrated in Figure 3.5. This neatly captures the desire to keep all of the work performed by the workers, and gives a formal specification that can be used for unit testing the merge operation.

[9] Provided that the user's operations are pure, the workers can safely execute the same operation multiple times and receive the same result. Otherwise, CausalRPC invokes the user's merge function to resolve the conflict, potentially throwing an exception.

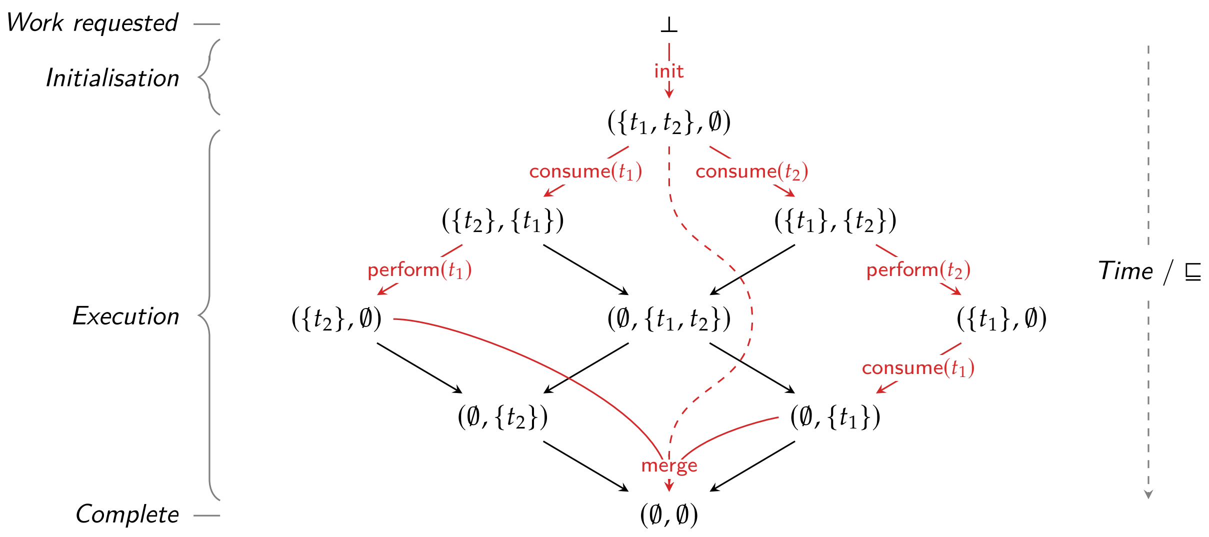


FIGURE 3.5: *Inverted Hasse diagram of S for the job $\{t_1, t_2\}$. The red arcs indicate a potential execution trace, with two workers each performing one task. All sequences of transitions converge towards $(\emptyset, \emptyset)$, which indicates termination.*

The efficiency of the merge operation is critical, as the server must invoke it once for each work submission. For this reason, I implemented a tree-based set that stores its contents as tree objects on Irmin's content-addressable heap. This allows subtrees to be checked for equality in constant time by comparing their hashes. The details will not be covered here, except to note that each task shares its data with other tasks in the set: if many tasks contain the same serialised parameters, these are stored in a single shared blob.

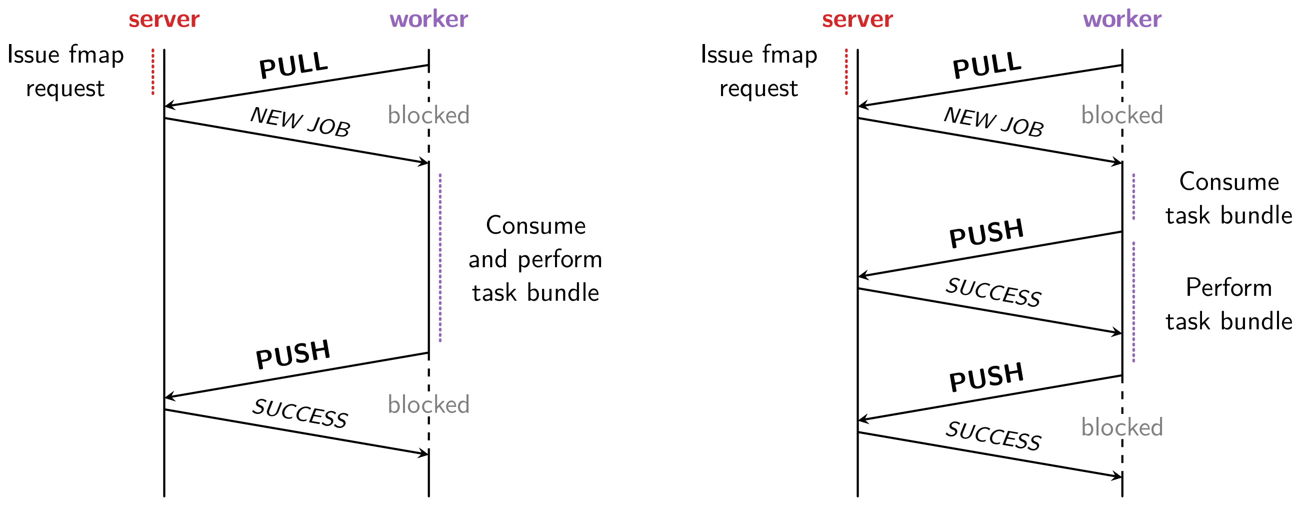
3.4.2 Worker-server communication protocol

Formalising the job data structure as a CRDT allows the workers to be very flexible: any sequence of PERFORM and CONSUME operations will result in progress being made on the job. In particular, workers may bypass pushing the intermediate 'consumed' phase without affecting correctness; this gives rise to two communication protocols, depicted in Figure 3.6. The intermediate consumption step requires additional server-side network resources, which are likely to be highly-contended due to the fan-in of work submissions from each of the workers. As a result, the server selects which of the two modes to use, switching to two-phase tasks when the network stack is highly contended.

Each worker has a fixed number of execution threads, each of which iteratively consumes bundles of tasks from a certain job until the job is complete. The lifecycle of an execution thread is depicted in Figure 3.7. Note that all network operations within the execution loop are optional. After the initial pull, the worker has all of the information necessary to complete the job; the intermediate pushes and pulls serve only to mitigate against race conditions that might cause the cluster to execute the same task multiple times.

3.5 MirageOS backend (EXTENSION)

Many of CausalRPC's dependencies are part of the MirageOS open-source movement. MirageOS [43] is a library operating system for the construction of *unikernels*: specialised, single-address-space machine images designed to run on a hypervisor or an embedded system. Unikernels contain only the libraries that are necessary for them to function, avoiding the software overhead of a traditional OS.



(A) *Two-phase work consumption* (B) *Three-phase work consumption*

FIGURE 3.6: *Sequence diagrams of two feasible work consumption protocols. The three-phase worker declares interest in a task with a consumption step; the task can be executed in parallel with this step.*

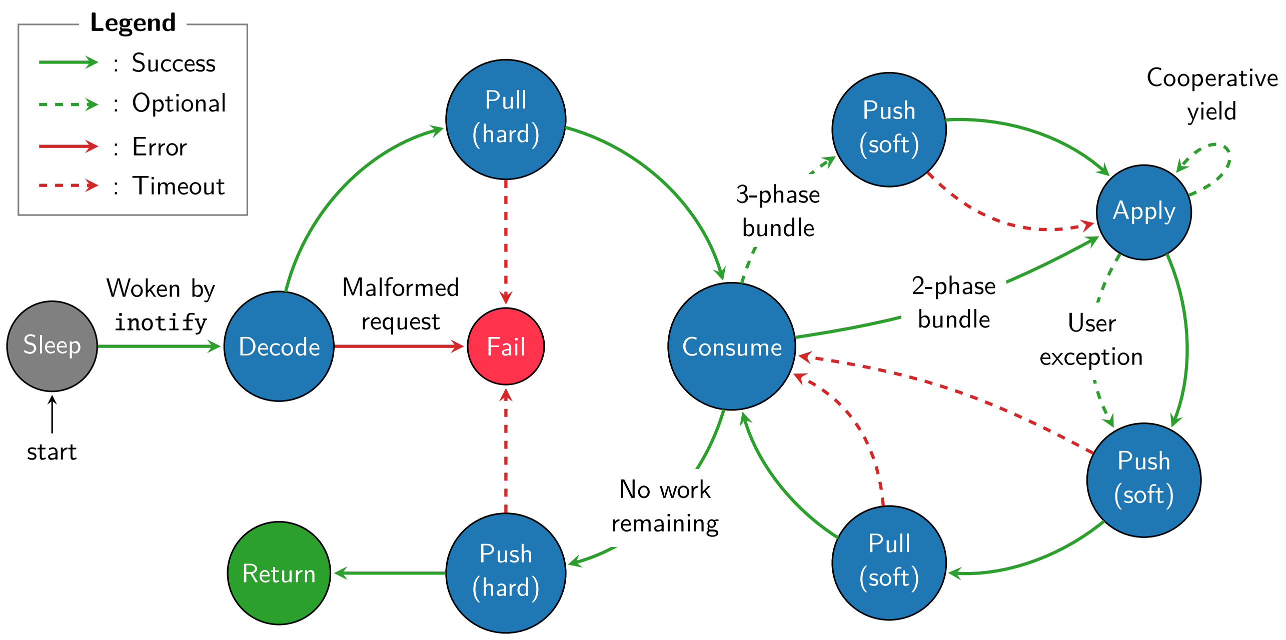


FIGURE 3.7: *Finite-state machine describing the lifecycle of a job execution thread on a worker node. Network operations within the main execution loop (right) are 'soft': they are not required for correctness, but may avoid duplication of work.*

```
(* Functor applied by the MirageOS build tool *)
module Main (Tcpip_socket: TCPIP_SOCKET) (Mclk: MCLOCK) (Pclk: PCLOCK) = struct

  (* Use Mirage building blocks to construct higher-level dependencies *)
  module Conduit = Conduit_mirage.With_tcp (Tcpip_socket)
  module Resolver = Resolver_mirage.Make_with_stack (Mclk) (Tcpip_socket)

  (* Construct a MirageOS-compatible CausalRPC backend *)
  module Backend = Causal_mirage.Make (Mclk) (Pclk) (Resolver) (Conduit)

  (* Pass the backend, store contents, interface & implementation *)
  module Unikernel = Causal.MakeWorker (Backend) (Value) (Iface) (Impl) (Config)
end
```

LISTING 3.4: *Creation of an in-memory CausalRPC worker node inside a MirageOS unikernel. Note the dependency on* `Causal_mirage`*; this module cannot be linked on a non-MirageOS architecture.*

The final extension of CausalRPC implements MirageOS-compatible client, server and worker nodes. This requires parameterising each of the node creation functors on a 'backend' module, then implementing one backend for Unix-like systems and one for MirageOS. The MirageOS backend is itself parameterised on its various kernel dependencies, which are supplied by the MirageOS build tool at compile time (see Listing 3.4).

Most of the implementation effort involved refactoring the module structure to keep a clean abstraction between the kernel-specific functionality and the project core.[10] The code must be partitioned such that a user can link the CausalRPC core against either the Unix backend or the MirageOS backend. This is discussed in more detail in the following section (§ 3.6).

3.6 Repository overview

CausalRPC uses the Dune build system [54]. The source code is structured according to the *de facto* standard for such projects. The repository is partitioned into five major directories: source code, unit tests, integration tests, end-to-end tests, and benchmarks (shown in Figure 3.8). All of the code in this repository was designed and implemented by me.

3.6.1 Source code

The source code directory contains a set of Dune *libraries*: packages of OCaml modules that can be pulled as dependencies by other OCaml projects. The platform-specific backends must be defined in separate Dune libraries, so that a program can be linked using only dependencies that are available for its platform. As a result, the CausalRPC project is split into three libraries: one for the core logic; one each for the Unix and MirageOS backends. Each library contains a set of top-level modules (`.ml` files), which must have acyclic dependencies. The signature of each module is supplied in a corresponding `.mli` file, which also contains documentation for that module. The structure of the core library is shown in Figure 3.9.

[10]Several inconsistencies with the APIs exposed by different Irmin backends made this initially impossible. I made several changes to these APIs. URL: https://github.com/mirage/irmin/pull/593 (visited on 1st April, 2019).

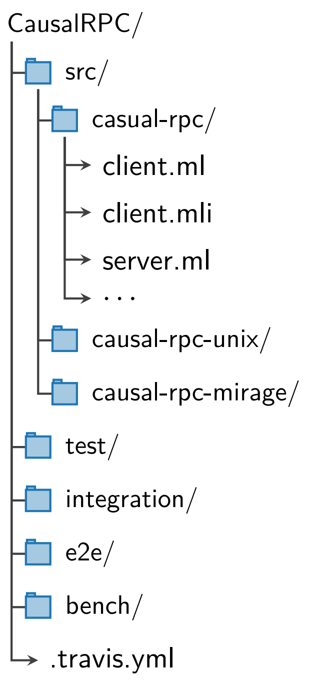


FIGURE 3.8: *Structure of the source code repository.*

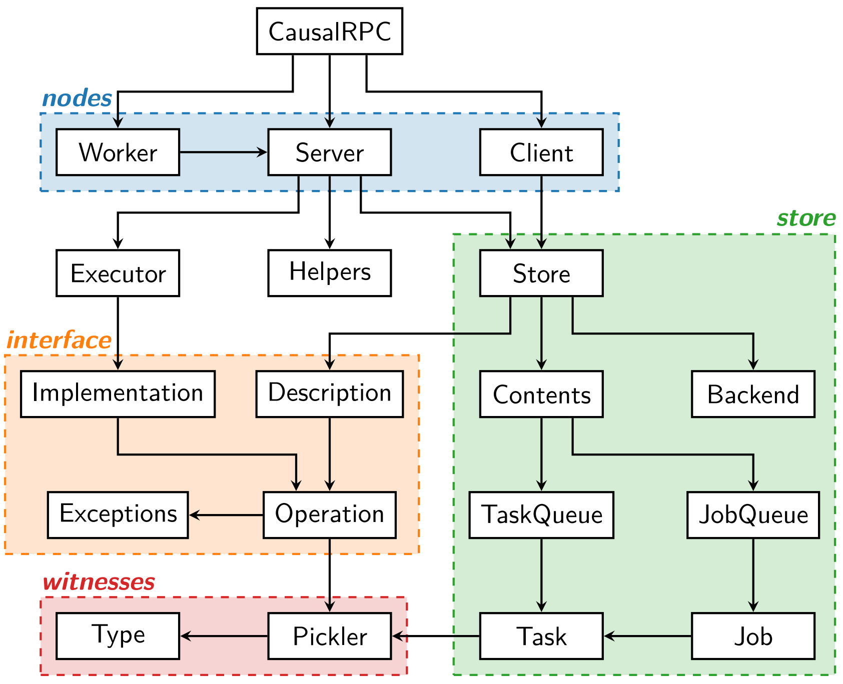


FIGURE 3.9: *DAG of* causal-rpc *module dependencies grouped into components, as reported by* `ocamldep` *[72]. Transitive dependencies not shown.*

3.6.2 Testing

In accordance with professional practice [66, 7], CausalRPC has three tiers of testing: unit testing of modules, integration testing of larger components, and end-to-end testing of the full system.

The `test` directory mirrors the source code: each top-level module of CausalRPC has a corresponding test module that verifies the behaviour of its interface. This discipline requires structuring the module hierarchy such that each module is testable. Test cases were added to prevent regression of each bug found during development.

3.7 Summary

This chapter detailed the implementation of CausalRPC. I discussed the various design choices made when encoding an RPC protocol into Git (§ 3.1–2), paying special attention to the use of CRDTs to automatically resolve race conditions. I discussed the problem of type-safety in detail (§ 3.3.1–3), and then showed how the interface could be extended to support efficient RPC composition (§ 3.3.4). In order to demonstrate the scalability of the system, I implemented worker clusters (§ 3.4) with a variety of modes of operation. Finally, I discussed the challenges of making CausalRPC compatible with MirageOS (§ 3.5–6).

4 Evaluation

This chapter evaluates the success of CausalRPC in terms of its semantics and performance. I begin by reviewing the project success criteria and demonstrating that CausalRPC meets and exceeds them (§ 4.1). I then consider the various classes of user error that are caught by the phantom-typed interface (§ 4.2). Finally, I perform a quantitative performance evaluation of the system (§ 4.3), indicating cases where CausalRPC is competitive with current RPC systems as well as more general trade-offs with the data-oriented approach (§ 4.4).

4.1 Acceptance testing

The project's success criteria are stated in the project proposal (Appendix E). Briefly, the CausalRPC server must be capable of concurrently processing RPCs from multiple remote clients. In order to demonstrate this functionality, I constructed a small NFS-like application built on top of CausalRPC. The server stores a POSIX-like directory tree (either in memory or in a persistent Git repository) and provides an RPC interface with three operations: PUT, APPEND and MOVE. Listing 4.1 shows an example interaction between two clients, each running in a sandboxed MirageOS unikernel. The sequence of events is as follows:

1. Client A connects to server S and creates a shared file inside the `/tmp` directory.
2. Client B connects to S and issues an append operation to the shared file added by A.
3. A pulls from S, sees the append issued by B, and requests that the shared file be moved to the `/log` directory. S performs the move and pushes the result to A.
4. While S is performing the move operation for A, B issues another append to the shared file, expecting it to still be within the `/tmp` directory. S performs the append on the `client-b` branch, then invokes the merge operator to resolve the conflict. The result is that the file is both appended to and moved; the merge operation has preserved the *intent* of both operations. B receives the result of the merge, including the append made by A.

The server has a persistent Unix backend, and so the application trace can be viewed by running `git log`, obtaining the output shown in Listing 4.2. Note that the trace closely corresponds

```
$ ./client-a/main.native
Initialised MirageOS client (10.0.0.1)
Connected to remote server (10.0.0.3)

Request:  Put new file at /tmp/sharedFile
Response: SUCCESS

Pulling from the server. Received updates:
  * <append> to /tmp/sharedFile

Request:  Moving /tmp/sharedFile to /log/
Response: SUCCESS
```

```
$ ./client-b/main.native
Initialised MirageOS client (10.0.0.2)
Connected to remote server (10.0.0.3)

Request:  Append to file /tmp/sharedFile
Response: SUCCESS

Pulling from the server. No updates.

Request:  Append to file /tmp/sharedFile
Response: MERGED with
  * <move> to /log/sharedFile
```

Listing 4.1: *Output logs of two CausalRPC clients after making requests to the same server. The clients are MirageOS unikernels running in Solo5 sandboxes.*

```
* server Initialised Filesystem store
* clientA Request <put>: /tmp/sharedFile
* server Perform <put> for clientA
* clientB Request <append>: /tmp/sharedFile
* server Perform <append> for clientB
|\
* | clientA Request <move>: /tmp/sharedFile -> /log/sharedFile
* | server (client--a) Perform <move> for clientA
| * clientB Request <append>: /tmp/sharedFile
| * server  Perform <append> operation for clientB
|/
*   server (server, client--b) Merge client--b into server
```

Listing 4.2: *Output of `git log` within the server file-system after resolving a race condition. The trace shows the race condition as a divergence in causal history, as expected.*

to the intended output as depicted in Figure 3.2b on page 15. Crucially, the merge function ensures that all state transitions preserve data consistency. This example also demonstrates the interoperability between the MirageOS and Unix CausalRPC backends.

4.2 Qualitative evalution

RPC interfaces for CausalRPC are defined entirely within OCaml source code. This allows the type-checker to catch a wide range of user errors when defining and using the RPC interface. For brevity, I will focus on three classes of user error:

- **Mistyped implementation (λ)**: the implementation of an RPC operation does not match the description given in the interface.
- **Subset error ($\subset$)**: the implementation fails to provide a function required by the interface.
- **Superset error ($\supset$)**: the implementation provides a function not required by the interface.

These three error classes are not-exhaustive: for instance, the user might mix up the implementations of two functions of the same type, resulting in unexpected behaviour that cannot be detected by the type-system. However, they provide a benchmark of the static type-safety of an RPC framework, in terms of the user errors that we might reasonably expect it to catch.

Listing 4.3a gives an example of user code containing a variety of such errors; Listings 4.3b-e show corresponding type errors produced by the OCaml compiler, demonstrating that all three classes of user error are detected statically. The most complex error (Listing 4.3e) indicates that the 'shapes' of the interface and implementation do not match: the compiler expects the last function to have type file $\longrightarrow$ int $\longrightarrow$ int, when in fact it has type int $\longrightarrow$ int.

To enable a comparison, I surveyed the type-safety properties of a range of RPC systems:

- *language-specific frameworks* (OCaml-rpc [51], Async [32], and Haskell-msgpack [29]);
- *multi-language frameworks* (gRPC [15] and Thrift [4]);
- *language-embedded RPC mechanisms* (Java RMI [56] and Erlang [21]).

The results of this survey are shown in Table 4.1. The systems vary widely, but some generalisations can be drawn. Broadly speaking, RPC systems use one of two mechanisms for allowing the user to implement their RPC interface:

```
let inc     = declare "increment"                base (* Increment the store     *)
let mult    = declare "multiply"         int @-> base (* Multiply by an integer  *)
let add_all = declare "add_all"     list int @-> base (* Add all values in a list *)
let load    = declare "load"            file @-> base (* Load int from file      *)

module Iface(I: IDL): INTERFACE = struct
  let api = I.(inc @ mult @ add_all @ finally load) (* Define the interface *)
end

module Impl(I: IDL): IMPLEMENTATION = struct
  let api = I.(
          (id      , fun x y -> x * y) (* Error: too many arguments              *)
        $ (mult    , fun x   -> x + 1) (* Error: too few arguments               *)
        $ (add_all , fun x y -> x + y) (* Error: incorrect argument type         *)
        f (declare "dec" base, (-) 1) (* Error: prototype not part of interface *)
      )                                (* Error: 'load' RPC not implemented      *)
end

module S = MakeServer (Memory) (Int) (Iface) (Impl) (* Try to build the server *)
```

(A) *Misuse of the CausalRPC IDL when implementing an integer RPC interface*

```
File "examples/idl_misuse.ml", line 13:
Error: This expression has type
   int -> int
but an expression was expected of type
   int -> int -> int
Type int is not compatible with type
   int -> int
```

(B) *Operation takes too few arguments* (λ)

```
File "examples/idl_misuse.ml", line 12:
Error: This function expects too many
arguments, it should have type int -> int
```

(D) *Operation takes too many arguments* (λ)

```
File "examples/idl_misuse.ml", line 14:
Error: This expression has type
   int -> int -> int
but an expression was expected of type
   int -> int list -> int
Type int is not compatible with type
   int list
```

(C) *Operation takes incorrect argument type* (λ)

```
File "examples/idl_misuse.ml", line 11:
Error: Signature mismatch:
  val api : ((int -> int) *
      ((int -> int -> int) *
        ((int list -> int -> int) *
          (int -> int)))) I.shape
is not included in
  val api : ((int -> int) *
      ((int -> int -> int) *
        ((int list -> int -> int) *
          (file -> int -> int)))) I.shape
```

(E) *Incorrect implementation shape* ($\subset$ *or* $\supset$)

Listing 4.3: *Example of an RPC interface description containing five user errors, with corresponding compilation errors produced by the type-checker.*

	Framework	Language	Embedded IDL	Type-safety[*]			Interface type	Invocation context	Size (lines)[†]
				λ	$\subset$	$\supset$			
LANGUAGE SPECIFIC	*CausalRPC*	OCaml	✓	✓	✓	✓	Implement	Functor	2.5k
	Async	OCaml	✓	✓	✗	✗	Register	Channel	54k
	OcamlRPC	OCaml	✓	✓	✗	✗	Register	Channel	4.0k
	MsgPack	Haskell	✓	✓	✗	✗	Register	Functor	3.9k
MULTI-LANGUAGE	gRPC	Python	✗	✗	✗	✗	Implement	Object	39k
	gRPC	C++	✗	✓	✓	~[‡]	Implement	Object	27k
	gRPC	Go	✗	✓	✓	✓	Implement	Channel	18k
	Thrift	Java	✗	✓	✓	~[‡]	Implement	Object	75k
	Thrift	Haskell	✗	✓	✓	✗	Implement	Channel	61k
FIRST CLASS	Erlang	Erlang	✗	✗	✗	✗	—[§]	Node	—
	Java RMI	Java	✓	✓	✓	~[‡]	Implement	Object	—

TABLE 4.1: *Qualitative comparison of a range of RPC systems.*

[*]*Static* type-safety of the implementation, in terms of the errors that it can catch. Symbols defined on page 29.
[†]Computed using `cloc`. Excluding tests and benchmarks; including any external IDL tools.
[‡]The (non-compulsory) `@Override` annotation / `override` keyword guarantees subset safety.
[§]The Erlang RPC system works by transmitting functions between nodes, so it has no concept of an interface [20].

- **IMPLEMENT**: the user implements a signature/interface that describes their RPC interface. Systems with an external IDL (gRPC, Thrift, etc.) typically generate the signature/interface in their target language to achieve subset and superset safety (where possible).

- **REGISTER**: the user passes functions to a module/object which registers them internally. Systems without an external IDL (Async, OCamlRPC, etc.) tend to implement this mechanism as it doesn't require type-level programming of the sort used in the implementation of CausalRPC. As shown in Table 4.1, this sacrifices the ability to statically detect shape-related errors ($\subset$ and $\supset$).

Of the systems I surveyed, CausalRPC is the only one to provide the type-safety properties of an **IMPLEMENT** strategy without either an external IDL or a specialised language runtime. From a purist standpoint, relying on externally-generated source code to catch user errors is not *true* static safety: the compiler provides no guarantee that the auto-generated code matches the interface description. A more pragmatic advantage of CausalRPC is that it doesn't require repeatedly re-generating source code stubs whenever the interface is changed. The inherent trade-off is that interface descriptions in CausalRPC are not compatible with other runtimes.

4.3 Performance evaluation

This section evaluates the performance of CausalRPC by: comparing RPC latency to existing RPC systems (§ 4.3.1); demonstrating the memory costs of casual traces (§ 4.3.2); and finally evaluating the scalability of the work clusters (§ 4.3.3). All experiments were performed on a dedicated 48-core machine with 8 NUMA nodes and 2 HDDs (see § C.1 for detail).

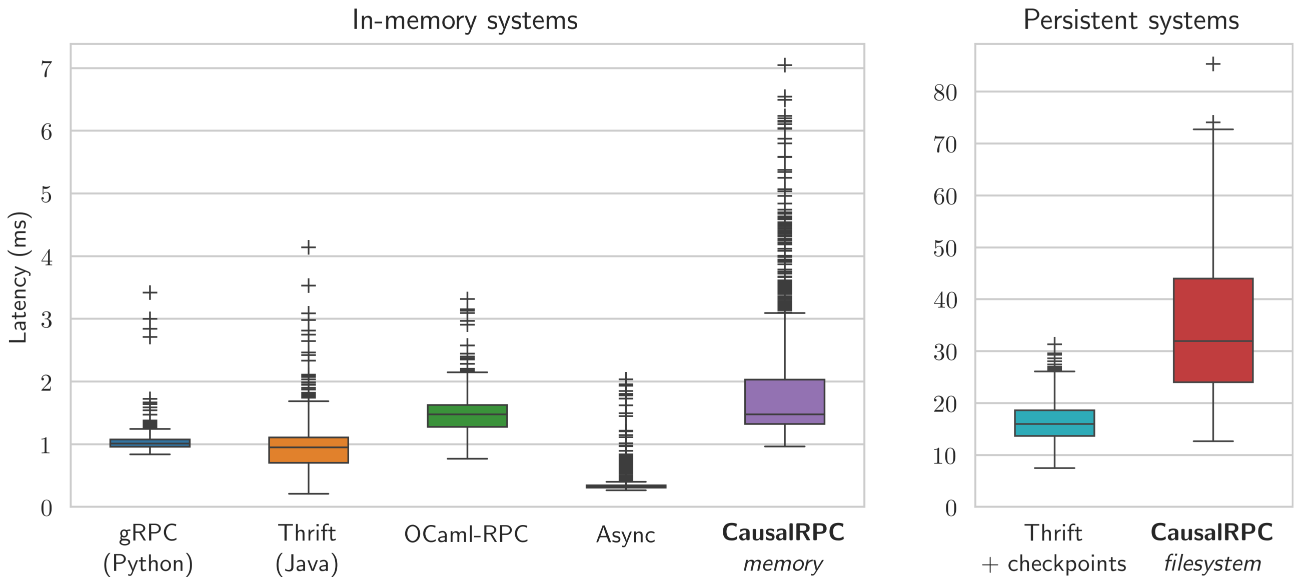


FIGURE 4.1: *Distribution of RPC latencies between a single client-server pair for a variety of RPC frameworks. Data was sampled over 1,000* INCREMENT *RPCs between independent Docker containers on the same machine.*

4.3.1 Latency

RPCs should be as seamless with respect to local computation as possible. As a result, the latency introduced by an RPC framework is particularly important. I compared the latencies of a range of RPC systems: CausalRPC (using both the in-memory and filesystem backends), OCamlRPC, Async, Thrift and gRPC. These RPC frameworks are all capable of completing an RPC in one round-trip time, so network latency is not a discriminating factor. To avoid network jitter, whilst still providing a realistic simulation of real-world performance, I ran the client/server nodes within separate Docker containers connected via a bridge network. See § C.2 for a discussion of the precautions taken to avoid systematic biases.

The benchmarking results are shown in Figure 4.1. The in-memory CausalRPC system is surprisingly competitive with optimised frameworks, given the additional run-time burdens of constructing a causal trace and supporting the embedded IDL. Figure 4.2 shows the run-time profile of the in-memory CausalRPC server for this benchmark; this gives three insights into the run-time behaviour:

1. 20% of run-time is spent performing serialisation. CausalRPC is the only system not to use a binary serialisation format, which are generally much faster to serialise than text-based representations [69]. Future work could provide a binary serialisation layer for Irmin, potentially significantly reducing the latency of CausalRPC.

2. 12% of execution time is spent polling the Irmin heap for updates; this could be eliminated altogether with an asynchronous implementation.

3. Type witness manipulation accounts for 11% of execution time. The set of possible parameter types is fixed by the user's RPC interface, and so this could be heavily optimised by using staging to generate interface-specific code at runtime [78, 16].

With these future optimisations in mind, in-memory CausalRPC has potential to be faster than both Thrift (Java) and gRPC (Python). The file-system backend has significantly larger

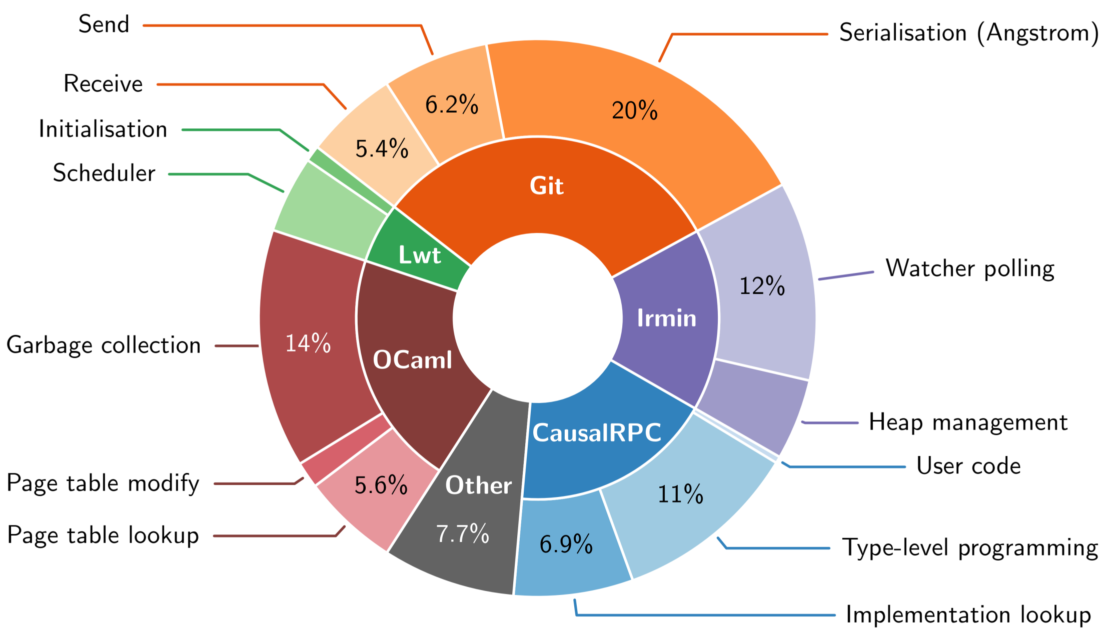


Figure 4.2: *Profile of execution time for an in-memory CausalRPC server after processing 1,000* INCREMENT *RPCs, as reported by* `gprof` *[24].*

latency (~20×), as it synchronously persists all store operations to disk. To provide a point of comparison for this type of system, I added synchronous 'checkpoint' stages to the Thrift system using Apache Spark [79]. The file-system backend is roughly twice as slow as this system; likely because it is persisting a causal trace as well as the user's data.

4.3.2 Memory usage

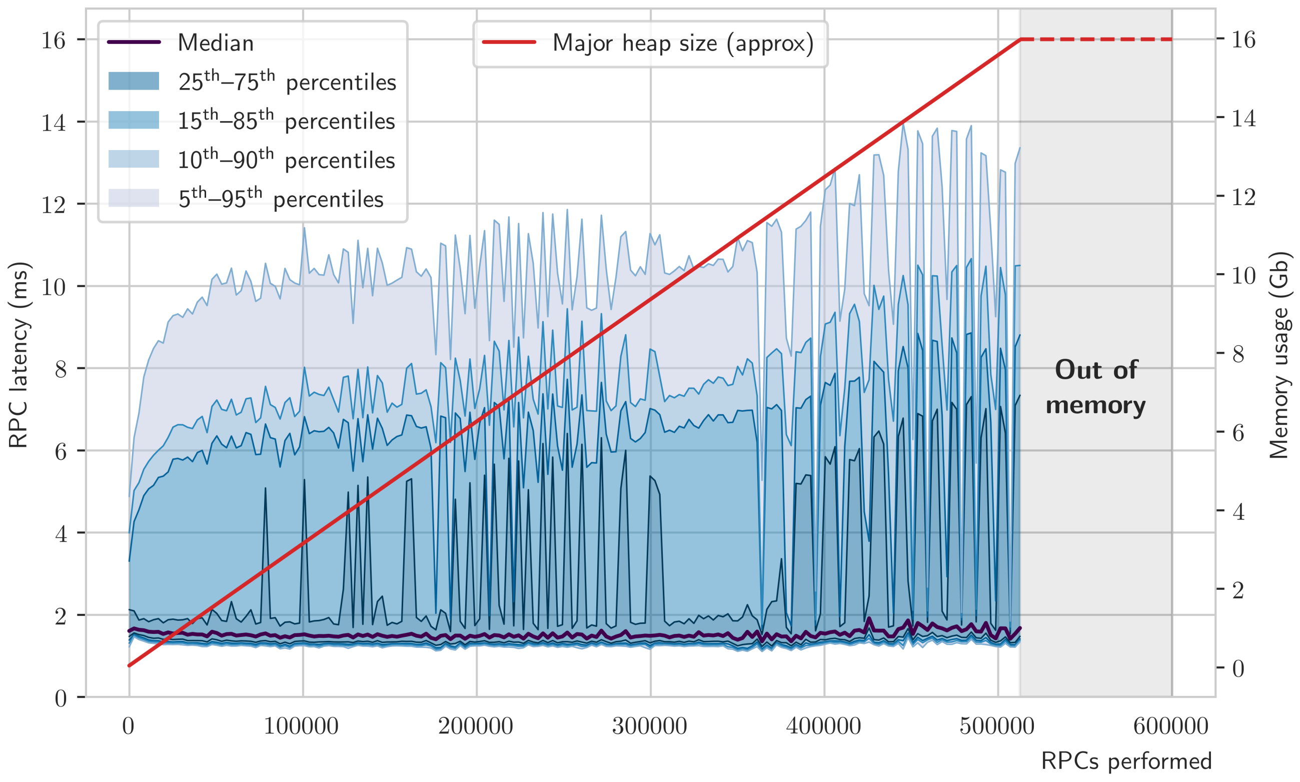
One of the major advantages of CausalRPC is its inherent traceability, but the causal trace has a corresponding memory requirement. The system has no mechanism for truncating the causal trace of the store, and so performing many operations over the same store will cause the trace to grow without bound. This is particularly problematic for the in-memory CausalRPC backend, as the OCaml heap size is limited by the memory available to the OCaml runtime.

In order to demonstrate this behaviour, I ran an experiment whereby one client repeatedly issues RPCs to the same server, using both the in-memory and file-system backends. Figure 4.3a demonstrates the linear memory growth of the store, and the resulting effect on RPC latencies. As the OCaml heap expands, the runtime spends a greater proportion of time running mark-and-sweep and heap compaction phases in order to recover space. Eventually, the system reaches the maximum heap size and crashes. The file-system backend does not have this problem (Figure 4.3b), as it can repeatedly flush the in-memory data cache to disk. An ideal solution would combine the speed of the in-memory backend with the fault-tolerance of the file-system backend, perhaps by asynchronously streaming an in-memory log to disk.

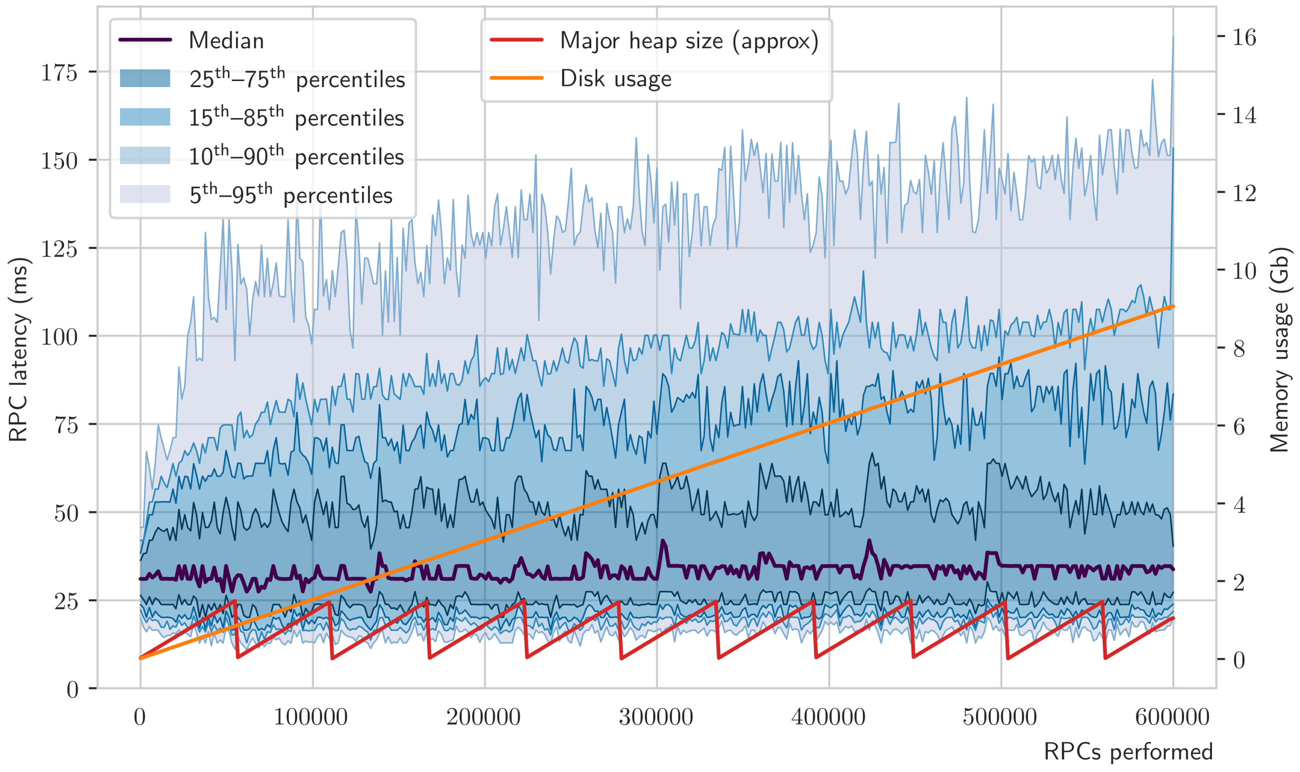
4.3.3 Work clustering

This section evaluates the scalability of CausalRPC's worker clusters. Consider the topology with a single client repeatedly issuing jobs to a cluster of workers. There are many factors affecting the performance of this system:

1. the number of workers in the cluster w;

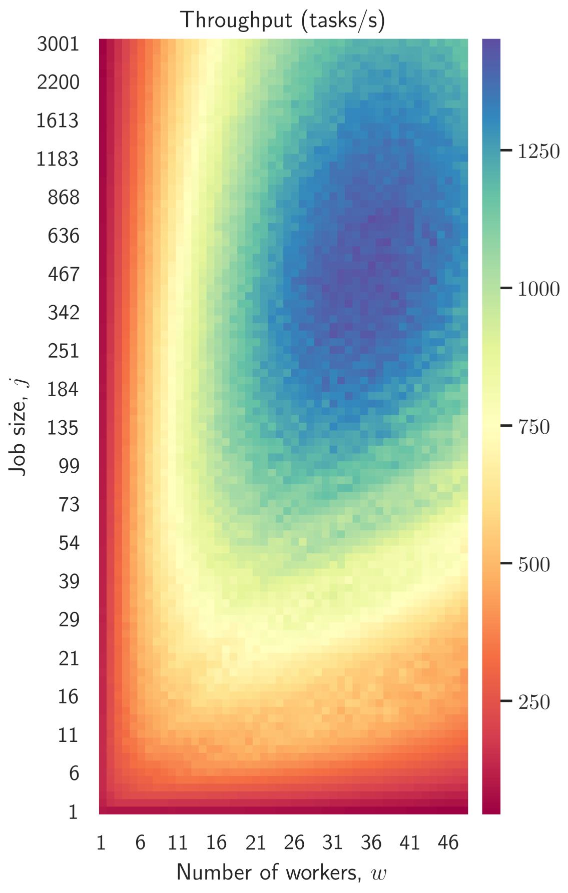


(A) *Memory backend*

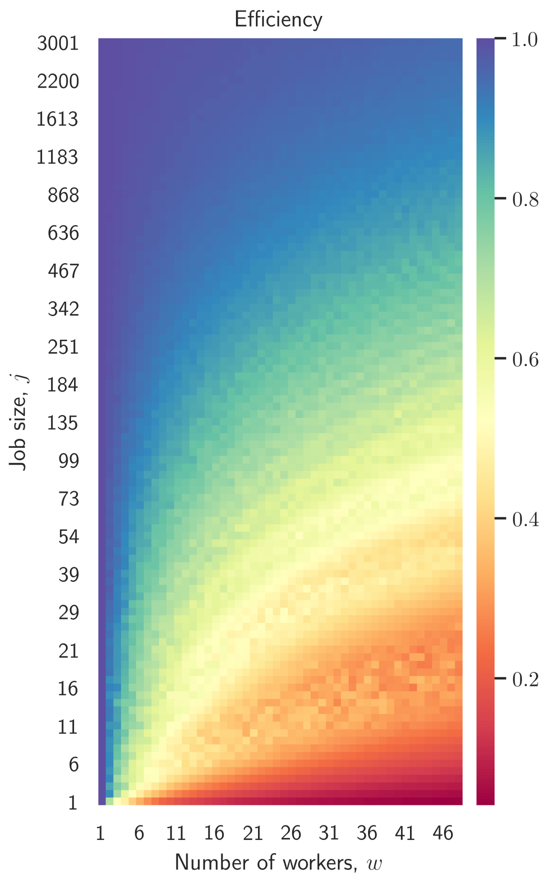


(B) *File-system backend*

FIGURE 4.3: *Latency of a single RPC during a stress test of 600,000 consecutive RPCs, for both in-memory and filesystem Irmin backends. The in-memory backend reaches the maximum heap size of* 16 GB *after ~515,000 RPCs.*



(A) *Task throughput*



(B) *Efficiency*

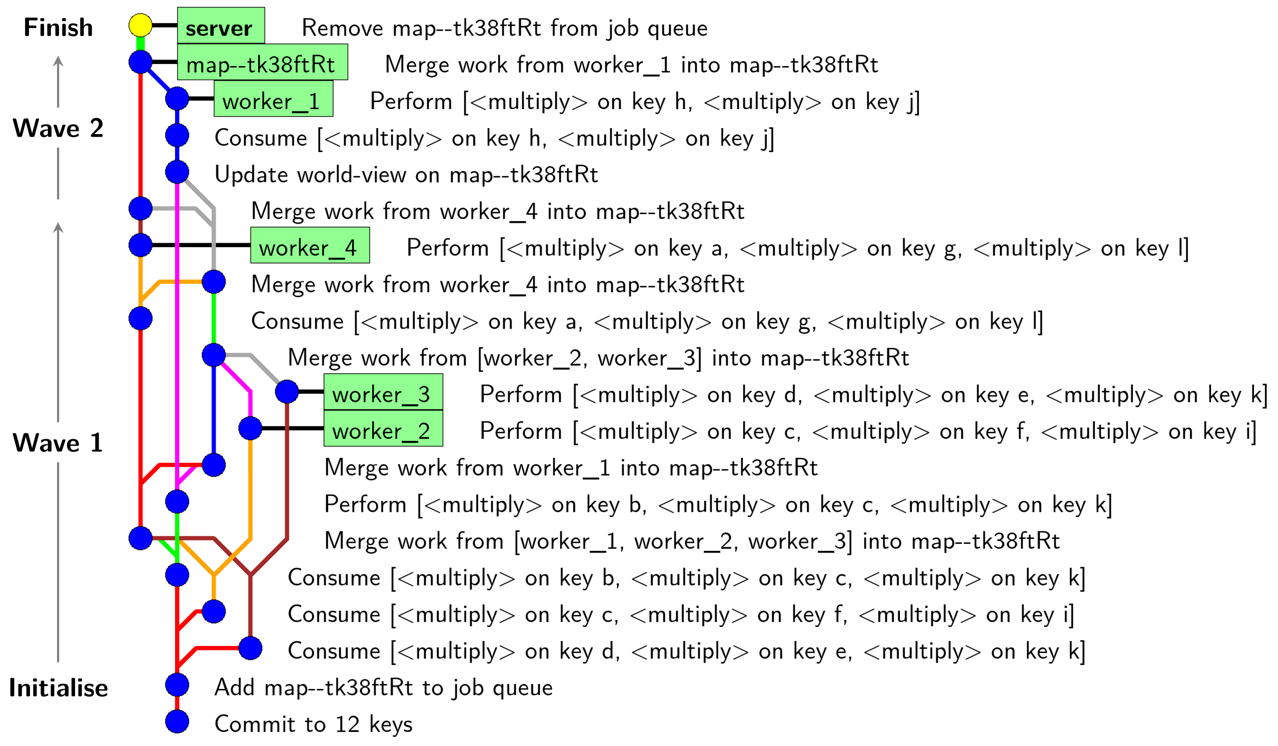
FIGURE 4.4: *Throughput and efficiency of differently-sized worker clusters executing* 30 ms *tasks, for a range of different job sizes. Each data-point is the average of 10 repeats. Efficiency is defined as the ratio of requested tasks to executed tasks.*

2. the workload parallelism, given by the number of tasks per job j and the task duration t;
3. the time complexity of selecting from the work set, and resolving contention over it;
4. the efficiency of the user's merge operator (required to merge the results of impure tasks);
5. two-phase vs. three-phase work consumption, and batched work consumption.

For brevity, we shall consider only 1–3. In the following experiments, the workers and server run on separate cores of a 48-core CPU, eliminating any network-based effects. Each task increments an integer and then spins for a time t, so all tasks are pure. Workers consume tasks one-by-one, without aggregating tasks into batches.

Figure 4.4a shows the task throughput of this system for a range of values of job size j and cluster size w, setting the task duration $t = 30$ ms. In the case $j \approx w$, increasing the number of workers generally *reduces* throughput. This can be explained with the following reasoning: for small j, the workers are likely to randomly select the same tasks as each other. These redundant tasks must still be submitted to the server, and so they limit the overall throughput of the system. Increasing j makes it more likely that workers will select different tasks, allowing them to exploit more parallelism.

This hypothesis can be tested by measuring the 'efficiency' of the work selection strategy, defined as the ratio of requested tasks j to executed tasks. The efficiency of the system is shown



```
Remove map--tk38ftRt from job queue
Merge work from worker_1 into map--tk38ftRt
Perform [<multiply> on key h, <multiply> on key j]
Consume [<multiply> on key h, <multiply> on key j]
Update world-view on map--tk38ftRt
Merge work from worker_4 into map--tk38ftRt
Perform [<multiply> on key a, <multiply> on key g, <multiply> on key l]
Merge work from worker_4 into map--tk38ftRt
Consume [<multiply> on key a, <multiply> on key g, <multiply> on key l]
Merge work from [worker_2, worker_3] into map--tk38ftRt
Perform [<multiply> on key d, <multiply> on key e, <multiply> on key k]
Perform [<multiply> on key c, <multiply> on key f, <multiply> on key i]
Merge work from worker_1 into map--tk38ftRt
Perform [<multiply> on key b, <multiply> on key c, <multiply> on key k]
Merge work from [worker_1, worker_2, worker_3] into map--tk38ftRt
Consume [<multiply> on key b, <multiply> on key c, <multiply> on key k]
Consume [<multiply> on key c, <multiply> on key f, <multiply> on key i]
Consume [<multiply> on key d, <multiply> on key e, <multiply> on key k]
Add map--tk38ftRt to job queue
Commit to 12 keys
```

FIGURE 4.5: *Output of* `gitk` *[58] after completing 12 tasks using a cluster of four workers. The workers consume tasks in batches of three, where possible. Ideally, each worker would perform exactly 3 tasks, but some tasks are duplicated due to the random selection algorithm (here: '*`c`*' and '*`k`*').* `worker_1` *completes the final two tasks while the other three workers are idle.*

in Figure 4.4b. Note the similarity between the two heatmaps in the region $j \approx w$; this suggests that inefficiency is the dominant limiting factor for throughput in this region. The efficiency of the system at small j is characterised by 'bands' at contours of j/w. These can be explained by the fact that workers tend to submit work in synchronised waves; efficiency and throughput increase significantly when all workers perform useful work on the final wave. Figure 4.5 shows the trace of an inefficient job where only one worker submits useful work on the final wave.

The job size cannot be increased arbitrarily, as this causes the server to become overwhelmed with merging large work sets.[1] This suggests that the throughput of a given workload will be maximised at certain values of j and w. This is exactly what we observe; in this case, throughput is maximised at approximately $j = 500$ and $w = 35$.

The workload parallelism is also affected by the task duration t. Each work submission requires a roughly constant processing time s at the server, plus the execution time t at the worker. For a given job size j, the overall workload consists of a serial fraction $j \cdot s$ and a parallel fraction $j \cdot t$ distributed over w workers. We can apply Amdahl's law [60] to compute the theoretical latency speedup achievable by a cluster of size w:

$$\textsc{speedup}(w) = \left(\frac{s}{s+t} + \frac{t}{w(s+t)} \right)^{-1} \tag{4.1}$$

The serial execution time s can be approximated as the mean latency of completing a single no-op task (such that $j = 1$ and $t \approx 0$), measured as $s = (1.50 \pm 0.21)$ ms over 10 repeats. This

[1] In practice, j is also bounded by the data-level parallelism available in the user's workload.

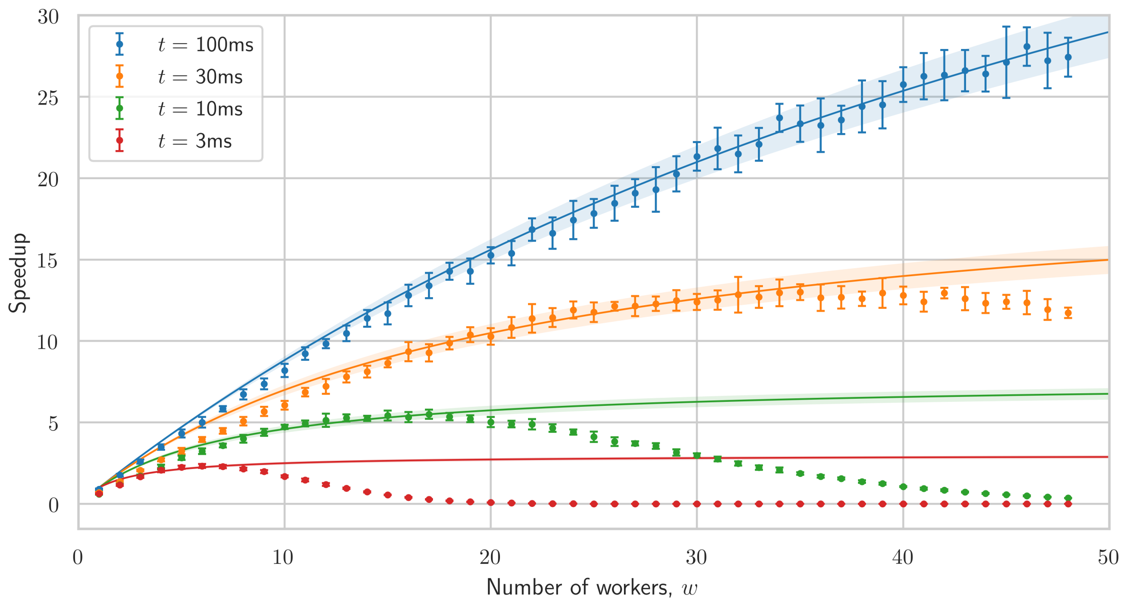


FIGURE 4.6: *Speedup of 500 tasks as a function of cluster size, plotted for a range of task execution times. Error bars show σ over 10 repeats. Each series is fitted against the maximum achievable speedup predicted by Amdahl's law, shown with uncertainty $\pm\sigma$ computed by propagating the measured uncertainty in s (see § C.3 for detail).*

measurement accounts for the round-trip time between the server and the worker, but not the inherent costs of the CausalRPC architecture (random task selection, scalability of the store, merge operations, etc.).

Figure 4.6 shows how the latency speedup achieved by the workers compares to the theoretical predictions of Amdahl's law for a range of values of t. These results show where the design decisions of CausalRPC are effective, and where they break down. For small w, the optimistic work selection routine allows the cluster to achieve near-optimal speedup. However, adding too many workers causes them to conflict with each other, putting more strain on the server to resolve those conflicts. As t increases, the rate of work submissions decreases, causing the optimal cluster size to increase.

4.4 Trade-offs of data-oriented RPC

This evaluation has compared CausalRPC to existing RPC systems along a number of dimensions. These comparisons revealed several underlying design trade-offs:

- **Type-safety versus portability**. CausalRPC demonstrates that embedding an IDL within the source language of an RPC framework allows for stronger static type-safety than is achieved by many existing RPC systems (§ 4.2). However, this approach sacrifices the inter-language portability of systems with external IDLs. On the other hand, CausalRPC is considerably more portable than runtime-dependent RPC mechanisms such as Java RMI.

- **Traceability versus memory usage**. Any behavioural trace has a corresponding memory requirement. This project explored one extreme of this spectrum: perpetual traceability at the cost of unbounded memory use (§ 4.3.2). The majority of existing systems provide comparatively sparse linear logs containing only request-level information, with no information about the application behaviour [64, 28]. Future data-oriented systems might allow the user to choose a compromise between these two extremes, perhaps using Git's 'rebase' mechanism to periodically forget portions of causal history in a controlled manner.

- **Optimism versus efficiency**. CausalRPC workers are inherently optimistic: they perform tasks without reaching any prior consensus among the cluster, relying on the merge operator to handle any race conditions that might arise. This means that tasks may be performed more than once. However, by explicitly tracking the causal history of each state, CausalRPC ensures that each store transformation occurs at most once. The CRDT theory that underlies the job data structure allows the workers to be very flexible in their interactions with the server: they can choose to be more optimistic by bypassing the intermediate consumption phase or increasing their batch size.

5 Conclusions

This dissertation has presented CausalRPC, a distributed computation framework that demonstrates the semantics and performance trade-offs of a novel approach to RPCs in which statefulness is made explicit.

5.1 Achievements

In the introduction to this dissertation, I highlighted four properties not provided by conventional RPC systems: consistency, traceability, type-safety and scalability. In Chapter 2, I hypothesised that these properties could be recovered by taking a functional, data-oriented approach to RPC. This section considers the evidence that CausalRPC has provided for this hypothesis.

CausalRPC automatically resolves data conflicts between concurrent requests, and maintains a trace of its causal history that is fully introspectable with the standard `git` command line tool (§ 4.1). I demonstrated that this can be achieved while retaining a request latency that is competitive with high-performance RPC frameworks (§ 4.3.1), although the long-term memory cost of constructing such a verbose trace is considerable (§ 4.3.2). I explored the trade-offs of a persistent and fault-tolerant design, using the Irmin file-system backend, and found that this persistance costs a factor of 20× in performance.

I took an ambitious approach to the design of the interface description language: CausalRPC implements a set of pickler combinators in order to support RPCs with higher-order, mutually-recursive parameter types (Appendix B). I leveraged the advanced type system of OCaml to catch a wide range of user errors at compile time (§ 4.2), and to provide an RPC interface that is syntactically equivalent to local computation (§ 3.3.3). This was achieved without the use of external tools or syntax-level code generation, demonstrating the power of embedding domain-specific languages within a functional host language.

I extended CausalRPC to be a multicast distributed computation framework by adding support for dynamically-scalable work clusters (§ 3.4). This was achieved without weakening the consistency semantics presented to the user. I demonstrated that these work clusters are capable of achieving latency speedups of the order predicted by Amdahl's law, but that server contention is a limiting factor on the cluster size (§ 4.3.3). Finally, I adapted CausalRPC to be deployable on MirageOS (§ 3.5), making my project a contribution to the growing ecosystem of unikernel-deployable software.

5.2 Future work

The project uncovered many interesting avenues which could not be explored within the time constraints of the project. For example:

- **Remote threads**. The remote functor implemented in this project generalises conventional RPC semantics for more efficient composition of RPCs. This functor could be extended to exploit per-client thread-level parallelism via remote analogues of Lwt.pick and Lwt.choose, perhaps using long-lived HTTP/2 streams to achieve gRPC-like performance [52].

- **Hybrid backends**. This project provided two classes of backend: fully persistant and in-memory. A production-ready system would likely use a hybrid of these two approaches, using write-back memory caches to achieve competitive performance while retaining persistance. This would require implementing either a Linux FUSE backend or a Mirage filesystem backend for Irmin, and so is an ambitious project in its own right.

- **Other traceable systems.** This project considered the application of CRDTs to RPC, but there are many other applications that could be considered. For example, a similar method could be applied to the problem of data aggregation in IoT-style sensor networks [44]. Irmin would be a suitable candidate for deployment in embedded devices, due to its compatibility with MirageOS.

5.3 Lessons learned

This project was my first experience with functional programming outside of the Tripos. I challenged myself to embrace the idioms of advanced functional programming: designing type-safe structures using GADTs, sequencing operations over those structures using functors, and structuring the code into composable units using the module system. The end result is an RPC framework with many qualitative advantages over existing systems.

This project has been an insight into the intersection between distributed systems research and open-source software development. Despite OCaml being a relatively old language, OCaml development has recently been revitalised by open-source initiatives such as OCaml Labs and Oscigen. I have diagnosed, reported and fixed bugs; added features and documentation where necessary; and generally improved the dependencies that I worked with. I enjoyed the collaborative nature of open-source development, and have chosen to continue working with the Irmin development team full-time; after I graduate, I will be employed by Tarides[1] to work on improving Irmin and other open-source OCaml projects.

[1]See https://tarides.com (visited on 20th April, 2019).

Bibliography

[1] M. Abadi, L. Cardelli, B. Pierce and G. Plotkin. 'Dynamic Typing in a Statically-typed Language'. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 213–227. isbn: 0-89791-294-2. doi: 10.1145/75277.75296.

[2] A. Abel and R. Matthes. 'Fixed points of type constructors and primitive recursion'. In: *International Workshop on Computer Science Logic*. Springer. 2004, pp. 190–204.

[3] P. S. Almeida, A. Shoker and C. Baquero. 'Delta State Replicated Data Types'. In: *CoRR* abs/1603.01529 (2016). arXiv: 1603.01529.

[4] Apache. *thrift: a lightweight, language-independent software stack with an associated code generation mechanism for point-to-point RPC*. url: https://github.com/apache/thrift (visited on 10/01/2018).

[5] A. Baboulevitch. 'Data Laced with History: Causal Trees & Operational CRDTs'. url: http://archagon.net/blog/2018/03/24/data-laced-with-history/ (visited on 17/04/2019).

[6] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004. isbn: 0321278658.

[7] A. Beresford. *Lecture notes for Software and Security Engineering*. 2019.

[8] A. D. Birrell and B. J. Nelson. 'Implementing Remote Procedure Calls'. In: *ACM Trans. Comput. Syst.* 2.1 (February 1984), pp. 39–59. issn: 0734-2071. doi: 10.1145/2080.357392.

[9] B. W. Boehm. 'A Spiral Model of Software Development and Enhancement'. In: *Computer* 21.5 (May 1988), pp. 61–72. issn: 0018-9162. doi: 10.1109/2.59. url: http://dx.doi.org/10.1109/2.59.

[10] B. Boehm and W. J. Hansen. *Spiral development: Experience, principles, and refinements*. Tech. rep. Carnegie-Mellon University, 2000.

[11] S. Burckhardt, A. Baldassion and D. Leijen. 'Concurrent Programming with Revisions and Isolation Types'. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM SIGPLAN, October 2010.

[12] S. Chacon and B. Straub. *Pro Git*. 2nd. Berkely, CA, USA: Apress, 2014. isbn: 1484200772, 9781484200773.

[13] J. Cheney and R. Hinze. *First-class phantom types*. Tech. rep. Cornell University, 2003.

[14] K. Claessen. 'A Poor Man's Concurrency Monad'. In: *J. Funct. Program.* 9.3 (May 1999), pp. 313–323. issn: 0956-7968. doi: 10.1017/S0956796899003342.

[15] Cloud Native Computing Foundation. *grpc: A high performance, open-source universal RPC framework*. url: https://grpc.io (visited on 10/01/2018).

[16] K. Czarnecki, J. T. O'Donnell, J. Striegnitz and W. Taha. 'DSL Implementation in MetaOCaml, Template Haskell, and C++'. In: *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*. 2003, pp. 51–72. doi: 10.1007/978-3-540-25935-0_4.

[17] J. Dean and S. Ghemawat. 'MapReduce: Simplified Data Processing on Large Clusters'. In: *Commun. ACM* 51.1 (January 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[18] A. Demers et al. 'Epidemic Algorithms for Replicated Database Maintenance'. In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC '87. Vancouver, British Columbia, Canada: ACM, 1987, pp. 1–12. ISBN: 0-89791-239-X. DOI: 10.1145/41840.41841.

[19] D. Duggan. 'A Type-Based Semantics for User-Defined Marshalling in Polymorphic Languages'. In: *Proceedings of the Second International Workshop on Types in Compilation*. TIC '98. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 273–297. ISBN: 3-540-64925-5.

[20] Ericsson. *Erlang External Term Format documentation*. URL: http://erlang.org/doc/apps/erts/erl_ext_dist.html (visited on 14/04/2019).

[21] Ericsson. *Erlang RPC module summary*. URL: http://erlang.org/doc/man/rpc.html (visited on 14/04/2019).

[22] B. Farinier, T. Gazagnaire and A. Madhavapeddy. 'Mergeable persistent data structures'. In: *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*. 2015.

[23] R. T. Fielding. 'REST APIs must be hypertext-driven'. In: *Untangled musings of Roy T. Fielding* (2008), p. 24.

[24] Free Software Foundation. *gprof(1) – Linux man page*. URL: https://linux.die.net/man/1/gprof (visited on 12/04/2019).

[25] T. Gazagnaire and L. Hélouët. 'Event Correlation with Boxed Pomsets'. In: *Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*. FORTE '07. Tallinn, Estonia: Springer-Verlag, 2007, pp. 160–176. ISBN: 978-3-540-73195-5. DOI: 10.1007/978-3-540-73196-2_11.

[26] A. Gill et al. 'The Remote Monad Design Pattern'. In: *SIGPLAN Not.* 50.12 (August 2015), pp. 59–70. ISSN: 0362-1340. DOI: 10.1145/2887747.2804311.

[27] Google. *Developer Guide: Protocol Buffers encoding format*. 2019. URL: https://developers.google.com/protocol-buffers/docs/encoding (visited on 29/03/2019).

[28] gRPC development team. *grpc-java issue tracker: solve logging*. 2016. URL: https://github.com/grpc/grpc-java/issues/1577 (visited on 12/03/2019).

[29] Haskell Community. *Haskell implementation of MessagePack*. URL: https://github.com/msgpack/msgpack-haskell (visited on 14/04/2010).

[30] T. Haynes and D. Noveck. *Network File System (NFS) Version 4 Protocol*. RFC 7530. March 2015. DOI: 10.17487/RFC7530.

[31] A. Ivanov. *Optimizing web servers for high throughput and low latency*. 6th September 2017. URL: https://blogs.dropbox.com/tech/2017/09/optimizing-web-servers-for-high-throughput-and-low-latency/ (visited on 29/03/2019).

[32] Jane Street. *async: Jane Street Capital's asynchronous execution library*. URL: https://github.com/janestreet/async (visited on 09/01/2018).

[33] J. M. Jansen, R. Plasmeijer and P. Koopman. 'Functional Pearl: Comprehensive Encoding of Data Types and Algorithms in the λ-Calculus'. In: *Journal of Functional Programming* (January 2010).

[34] J. Joyce, G. Lomow, K. Slind and B. Unger. 'Monitoring distributed systems'. In: *ACM Transactions on Computer Systems (TOCS)* 5.2 (1987), pp. 121–150.

[35] L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie and I. Greif. 'Replicated Document Management in a Group Communication System'. In: *Proceedings of the 1988 ACM Conference on Computer-supported Cooperative Work*. CSCW '88. Portland, Oregon, USA: ACM, 1988, pp. 395–. isbn: 0-89791-282-9. doi: 10.1145/62266.1024798.

[36] A. J. Kennedy. 'FUNCTIONAL PEARL Pickler Combinators'. In: *J. Funct. Program.* 14.6 (November 2004), pp. 727–739. issn: 0956-7968. doi: 10.1017/S0956796804005209.

[37] A. J. Kfoury, J. Tiuryn and P. Urzyczyn. 'The Undecidability of the Semi-unification Problem'. In: *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: ACM, 1990, pp. 468–476. isbn: 0-89791-361-2. doi: 10.1145/100216.100279.

[38] H. Ku. 'Notes on the use of propagation of error formulas'. In: *Journal of Research of the National Bureau of Standards* 70.4 (1966).

[39] D. M. B. D. Kuhn. *Distributed Version Control Systems*. 2010.

[40] R. Love. 'Kernel Korner: Intro to Inotify'. In: *Linux J.* 2005.139 (November 2005), pp. 8–. issn: 1075-3583.

[41] S. Ma, X. Zhang and D. Xu. 'ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting'. In: January 2016. doi: 10.14722/ndss.2016.23350.

[42] J. Mace, R. Roelke and R. Fonseca. 'Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems'. In: *ACM Trans. Comput. Syst.* 35.4 (December 2018), 11:1–11:28. issn: 0734-2071. doi: 10.1145/3208104.

[43] A. Madhavapeddy and D. J. Scott. 'Unikernels: The Rise of the Virtual Library Operating System'. In: *Commun. ACM* 57.1 (January 2014), pp. 61–69. issn: 0001-0782. doi: 10.1145/2541883.2541895.

[44] A. Madhavapeddy, K. C. Sivaramakrishnan, G. Gordon and T. Gazagnaire. 'An architecture for interspatial communication'. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2018, Honolulu, HI, USA, April 15-19, 2018*. 2018, pp. 716–723. doi: 10.1109/INFCOMW.2018.8406931.

[45] makdharma. *gRPC Load Balancing*. 2017. url: https://grpc.io/blog/loadbalancing (visited on 13/03/2019).

[46] S. Microsystems. *NFS: Network File System Protocol Specification*. United States, 1989.

[47] Y. Minsky, A. Madhavapeddy and J. Hickey. *Real World OCaml: Functional programming for the masses*. O'Reilly Media, 2013. isbn: 9781449324759.

[48] Mirage. *alcotest: A lightweight and colourful test framework*. url: https://github.com/mirage/alcotest (visited on 09/01/2018).

[49] Mirage. *irmin: a distributed database that follows the same design principles as Git*. url: https://github.com/mirage/irmin (visited on 09/01/2018).

[50] Mirage. *ocaml-git: Pure OCaml Git format and protocol*. url: https://github.com/mirage/ocaml-git (visited on 26/01/2018).

[51] Mirage. *ocaml-rpc: Light library to deal with RPCs in OCaml*. url: https://github.com/mirage/ocaml-rpc (visited on 09/01/2018).

[52] A. N. Monteiro. *An HTTP/2 implementation written in pure OCaml*. URL: https://github.com/anmonteiro/http2af (visited on 27/03/2019).

[53] A. Mycroft. 'Polymorphic Type Schemes and Recursive Definitions'. In: *Proceedings of the 6th Colloquium on International Symposium on Programming*. London, UK, UK: Springer-Verlag, 1984, pp. 217–228. ISBN: 3-540-12925-1.

[54] OCaml community. *A composable build system for OCaml*. URL: https://github.com/ocaml/dune (visited on 27/03/2019).

[55] OCaml community. *Low-level JSON parsing and pretty-printing library for OCaml*. URL: https://github.com/ocaml-community/yojson (visited on 29/03/2019).

[56] Oracle. *Java Remote Method Invocation API*. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html (visited on 14/04/2019).

[57] F. Panzieri and S. K. Shrivastava. 'Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing'. In: *IEEE Trans. Softw. Eng.* 14.1 (January 1988), pp. 30–37. ISSN: 0098-5589. DOI: 10.1109/32.4620.

[58] Paul Mackerras. *`gitk(1)` – Linux man page*. URL: https://git-scm.com/docs/gitk (visited on 12/04/2019).

[59] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer and A. J. Demers. 'Flexible Update Propagation for Weakly Consistent Replication'. In: *SIGOPS Oper. Syst. Rev.* 31.5 (October 1997), pp. 288–301. ISSN: 0163-5980. DOI: 10.1145/269005.266711.

[60] D. P. Rodgers. 'Improvements in Multiprocessor System Design'. In: *SIGARCH Comput. Archit. News* 13.3 (June 1985), pp. 225–231. ISSN: 0163-5964. DOI: 10.1145/327070.327215.

[61] M. Shapiro, N. Preguiça, C. Baquero and M. Zawirski. 'Conflict-Free Replicated Data Types'. In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by X. Défago, F. Petit and V. Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3.

[62] A. Shreve. 'Intro to gRPC: A Modern Toolkit for Microservice Communication'. Twilio Signal conference. 2017.

[63] K. Sivaramakrishnan. 'State of Multicore OCaml'. OCaml development meeting, Paris. 2018. URL: https://discuss.ocaml.org/t/ocaml-multicore-report-on-a-june-2018-development-meeting-in-paris/2202 (visited on 19/03/2019).

[64] S. Skelton. *Developer Friendly Thrift Request Logging*. 2013. URL: http://stevenskelton.ca/developer-friendly-thrift-request-logging (visited on 12/03/2019).

[65] G. Sklenárová. *Functional Network Stacks with MirageOS and Irmin*. May 2016.

[66] I. C. Society, P. Bourque and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014. ISBN: 0769551661, 9780769551661.

[67] R. Srinivasan. *Binding protocols for ONC RPC version 2*. Tech. rep. 1995.

[68] P. Staubach, B. Pawlowski and B. Callaghan. *NFS Version 3 Protocol Specification*. RFC 1813. June 1995. DOI: 10.17487/RFC1813.

[69] A. Sumaray and S. K. Makki. 'A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform'. In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '12. Kuala Lumpur, Malaysia: ACM, 2012, 48:1–48:6. ISBN: 978-1-4503-1172-4. DOI: 10.1145/2184751.2184810.

[70] D. C. Swinehart, P. T. Zellweger and R. B. Hagmann. 'The Structure of Cedar'. In: *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*. SLIPE '85. Seattle, Washington, USA: ACM, 1985, pp. 230–244. ISBN: 0-89791-165-2. DOI: 10.1145/800225.806844.

[71] A. S. Tanenbaum and R. van Renesse. 'A Critique of the Remote Procedure Call Paradigm'. In: 1988.

[72] *The OCaml system release 4.07: Documentation and user's manual*. Tech. rep. July 2018, pp. 1–752. URL: http://caml.inria.fr/pub/docs/manual-ocaml/ (visited on 14/03/2019).

[73] S. Vinoski. 'Convenience Over Correctness'. In: *IEEE Internet Computing* 12.4 (July 2008), pp. 89–92. ISSN: 1089-7801. DOI: 10.1109/MIC.2008.75.

[74] S. Vinoski. 'It's Just a Mapping Problem'. In: *IEEE Internet Computing* 7.3 (May 2003), pp. 88–90. ISSN: 1089-7801. DOI: 10.1109/MIC.2003.1200306.

[75] J. Vouillon. 'Lwt: A Cooperative Thread Library'. In: *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*. ML '08. Victoria, BC, Canada: ACM, 2008, pp. 3–12. ISBN: 978-1-60558-062-3. DOI: 10.1145/1411304.1411307.

[76] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[77] J. Wiegley. *Git from the Bottom Up*. 2017. URL: https://jwiegley.github.io/git-from-the-bottom-up/ (visited on 14/04/2019).

[78] J. Yallop. 'Staged generic programming'. In: *PACMPL* 1.ICFP (2017), 29:1–29:29. DOI: 10.1145/3110273.

[79] M. Zaharia et al. 'Apache Spark: A Unified Engine for Big Data Processing'. In: *Commun. ACM* 59.11 (October 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664.

A Excerpts from the protocol layer

The protocol layer of CausalRPC builds a communication protocol on top of the type-safe store interface provided by the pickler layer, using Lwt threads (§ 2.1.5) to exploit request-level concurrency. This appendix demonstrates this programming style with two excerpts from the implementation of CausalRPC.

A.1 Server RPC processing

The finite-state machine (FSM) of an RPC request is depicted in Figure 3.1 (page 14). The following code implements this FSM:

```
let server_request_thread ~branch_name ~local ~inital_tree req =
  (* Perform [rpcs] on [key], then return result to [remote] *)
  decode req >>= fun (remote, key, rpcs) ->

  (* Get the initial tree object *)
  Store.get local key >>= fun initial_val ->

  (* Helper function to execute a single RPC *)
  let rpcexec (old_val, tree) rpc =

    (* Lookup the requested operation in the interface *)
    Impl.get_operation rpc.operation_name Impl.api
    >>= (function | Some operation -> Lwt.return operation
                  | None -> Lwt.fail Malformed_request)

    (* Apply the operation using the serialised params & previous state *)
    >>= Executor.apply rpc.params old_val

    (* Commit value to store (each RPC gets its own commit) *)
    >>= fun new_val -> Tree.add tree key (Value new_val)
    >>= fun new_tree -> Store.set_tree
      ~info:(commit_info "%a" pp_rpc rpc) local [] tree

    >|= fun _ -> (new_val, new_tree)

  (* Fold the RPCs to produce a final state, passing the initial state *)
  in Lwt_list.fold_left_s rpcexec (initial_val, initial_tree) rpcs

  (* Merge working branch into master branch *)
  >>= fun _ -> Store.merge_with_branch
    ~info:(merge_info "Merge %s into server" branch_name) branch_name server_master

  (* Push back to the client *)
  >>= fun () -> Sync.push_soft local remote
```

A.2 Worker job processing

The FSM of a job processing thread is depicted in Figure 3.7 (page 25). The following code implements this FSM:

```
let worker_request_thread ~batch_size ~three_phase ~worker_name req =

  (* We pull remote work into req_branch, and perform the work on work_branch. The server
     merges work from origin/work_branch into origin/req_branch, completing the cycle. *)
  decode req >>= fun (request_name, req_branch, work_branch, req_remote, work_remote) ->

  Sync.pull_hard req_branch req_remote ‘Set >>= fun () ->

  let rec task_exection_loop () =
    Sync.pull_soft req_branch req_remote (‘Merge fast_forward_only)

    (* Consume a task from the queue *)
    >>= fun _ -> task_consume ~batch_size work_branch req_remote request_name
    >>= fun tasks -> if (not tasks_remaining tasks) then
      Lwt.return () (* Drop out of the loop *)
    else
      (* In three-phase mode, we first declare intent to perform these tasks *)
      let three_phase_thread = if three_phase
        then Sync.push_soft work_branch work_remote
        else Lwt.return () in (* in two-phase mode, jump straight to apply *)

      let apply_thread = Store.get_tree work_branch ["/"]
        >>= fun initial_tree -> apply tasks initial_tree (* Perform tasks *)
        >>= fun final_tree -> Store.set_tree             (* Update the store *)
          ~allow_empty:true
          ~info:(result_info request_name tasks)
          work_branch ["/"] result_tree in

      Lwt.join [three_phase_thread; apply_thread] (* Push and apply concurrently *)
      >>= fun () -> Sync.push_soft work_branch work_remote

      (* Pull only if push succeeded *)
      >>= fun success -> if success then Sync.pull_soft req_branch req_remote ‘Set
                                    else Lwt.return ()

      >>= fun () -> Store.merge_with_branch work_branch
        ~info:(merge_info "Updating world-view on %s" request_name) request_name

      >>= fun () -> log_info "Changes pushed to branch %s" request_name
      >>= Backend.yield      (* Allow another request to be processed *)
      >>= task_exection_loop (* Loop to execute more tasks *)

  in task_exection_loop ()
  >>= Sync.push_hard work_branch work_remote
```

B Higher-order picklers

CausalRPC implements a library of *type witnesses*: values that can be used to generate type-level serialisers and deserialisers in a statically-typed language (known as *picklers*). These allow the user to describe parameter types within an RPC interface.

B.1 Parameter type universe

The set of parameters that can be used within CausalRPC interfaces is as follows:

BASIC

```
val bool:   bool t
val bytes:  bytes t
val char:   char t
val float:  float t
val int:    int t
val int32:  int32 t
val int64:  int64 t
val string: string t
val unit:   unit t
```

COMBINATORS

```
val pair:    'a t -> 'b t -> ('a * 'b) t
val proj2_1: ('a * 'b) t -> 'a t
val proj2_2: ('a * 'b) t -> 'b t
val triple:  'a t -> 'b t -> 'c t -> ('a * 'b * 'c) t
val array:   'a t -> 'a array t
val list:    'a t -> 'a list t
val option:  'a t -> 'a option t
val result:  'a t -> 'b t -> ('a, 'b) result t
val mu:      ('a t -> 'a t) -> 'a t
```

VARIANTS

```
val variant: string -> 'b -> ('a, 'b, 'b) open_v
val case0:   string -> 'a ->                ('a,       'a case_v) case
val case1:   string -> 'b t -> ('b -> 'a) -> ('a, 'b -> 'a case_v) case
val (|~):    ('a, 'b, 'c -> 'd) open_v -> ('a, 'c) case -> ('a, 'b, 'd) open_v
val (|>):    ('a, 'b, 'a -> 'a case_v) open_v -> 'a t
```

B.2 Implementation details

Most of the type witnesses are simple to implement: basic witnesses need only be differentiable from each other, and many of the higher-order witnesses need only be tagged values containing their constituent witnesses. There are two particularly interesting cases: variant types (§ B.2.1) and type-level recursion (§ B.2.2).

B.2.1 Variant types

The implementation of variant types was heavily inspired by a similar construct within Irmin.[1] The constructors and deconstructors for a variant type are dependent on the user-defined tags for that type. For example, consider the following variant:

```
type maybe = Left of string | Right of int
```

The tags Left and Right have two uses in OCaml: for constructing values of type maybe and for pattern-matching on the contents of a maybe value. The user-supplied description of this type requires passing functions that implement this behaviour. The user begins by naming the variant and specifying its deconstructor using `variant`:

```
let maybe_open = variant "maybe" (fun left right -> function
        | Left s -> left s
        | Right i -> right i)
```

[1]URL: https://github.com/mirage/irmin/blob/master/src/irmin/type.ml (visited on 2019th May, 10)

This gives an 'open' variant representation that allows data constructors to be passed to it using the `|~` combinator, for example:

```
let maybe_open = maybe_open
  |~ case1 "Left"  string (fun s -> Left s)
  |~ case1 "Right" int    (fun i -> Right i)
```

As the cases are passed to the open variant, the internal structure accumulates them in a list. The variant must be 'sealed' via the `|>` operator to indicate that all cases have been specified, at which point the cases are stored in an array for efficient run-time access.

B.2.2 Recursive types

The user is able to define a the witness of a recursive type by supplying a function that defines the type in terms of itself:

$$\mu : \quad \forall \alpha \,.\, (\alpha \ \text{Type.t} \longrightarrow \alpha \ \text{Type.t}) \longrightarrow \alpha \ \text{Type.t}$$

This was implemented using *cyclic types* in OCaml:

```
(* Cyclic type definition *)
type 'a recur = { mutable recur: 'a Type.t }

(* Fix-point operator over type witnesses *)
let mu: type a. (a Type.t -> a Type.t) -> a Type.t = fun f ->
  let rec x = { recur = Recursive x } in (* Construct a base type *)

  (* Apply the function to the cyclic value *)
  let x' = f (Recursive x) in
  x.recur <- x'; x'
```

For example, this allows a tree datatype to be defined as follows:

```
let tree = mu (fun (x: tree t)  ->
  variant "tree" (fun leaf branch -> function
      | Leaf -> leaf
      | Branch (l, v, r) -> branch (l, v, r))
  |~ case0 "Leaf" Leaf
  |~ case1 "Branch" (triple x int x) (fun (l, v, r) -> Branch (l, v, r))
  |>)
```

Mutually-recursive sets of type witnesses can similarly be expressed as the least fixed points of functions over n-tuples of type witnesses, given suitable projection functions:

$$P_1^2 : \quad \forall \, \alpha, \beta \,.\, (\alpha \times \beta) \ \text{Type.t} \longrightarrow \alpha \ \text{Type.t}$$
$$P_2^2 : \quad \forall \, \alpha, \beta \,.\, (\alpha \times \beta) \ \text{Type.t} \longrightarrow \beta \ \text{Type.t}$$

C Evaluation methodology

C.1 Machine specifications

The machine used for evaluation has the following specifications:

- 4 AMD Opteron 6168 CPUs, for a total of 48 cores.
- 64 GB of system memory (1333 MT s^{-1}) over 8 non-uniform memory access (NUMA) nodes.
- 2 Seagate Constellation 1 TB SATA drives (7200 rpm).
- Linux kernel version 4.15.0 on the Ubuntu 18.04 LTS operating system.

C.2 Avoidance of systematic bias

The quantitative evaluation of CausalRPC (§ 4.3) involved comparing the latency of single client-server communication against a number of other RPC frameworks. It was decided to use Docker to ensure a clean abstraction between the client and server nodes running on the same machine. In order to conduct a fair comparison, I optimised the performance of each web server in accordance with professional practice [31] by:

- stopping non-essential processes, and migrating existing ones to a CPU-dedicated shield;
- disabling swap devices and increasing the maximum number of huge pages;
- pinning the client and server nodes to different CPUs and NUMA nodes.

I wrote a bash script to automate this process:

```bash
#!/usr/bin/env bash
set -euxo pipefail

# Stop unnecessary services / docker containers
for s in beanstalkd cron dbus rsyslog unattended-upgrades; do service $s stop; done
docker kill $(docker ps -q)

# Migrate system threads to cpu0 shield
cset set --cpu=0 systasks
cset proc --move / systasks --kthread --force

# Huge pages. No swapping
echo 512 > /proc/sys/vm/nr_hugepages
swapoff -a

# Run client and server in separate NUMA nodes
docker run --cpuset 0-5 --cpuset-mems 1 \
           --volume /mnt/client --name 'thrift-client' \
           --log-opt mode=non-blocking \ # less invasive result collection
           --network bench \
           --detach thrift bench-client.sh

docker run --cpuset 6-11 --cpuset-mems 2 \
           --volume /mnt/server --name 'thrift-server' \
           --log-driver none \
           --network bench \
           --detach thrift bench-server.sh
```

The script used to optimise the work cluster benchmark is similar.

C.3 Propagation of uncertainty within Amdahl's law

§ 4.3.3 analyses the performance of the worker clusters with respect to three parameters: work cluster size w, number of tasks per job j, and the duration of the tasks t. Under the assumption that each task requires some fixed execution time f at the server, the speedup S of executing a fixed batch size is given by Amdahl's law[1]:

$$S(w, f, t) = \left(\frac{f}{f+t} + \frac{t}{w(f+t)} \right)^{-1} \tag{C.1}$$

Since the variables w, f, and t are mutually-independent, the variance of speedup σ_S^2 can be approximated by truncating the multivariate Taylor expansion of (C.1) to first-order terms [38]:

$$\sigma_S^2 = \left| \frac{\partial S}{\partial w} \right|^2 \cdot \sigma_w^2 + \left| \frac{\partial S}{\partial f} \right|^2 \cdot \sigma_f^2 + \left| \frac{\partial S}{\partial t} \right|^2 \cdot \sigma_t^2$$

However, w and t can be set arbitrarily, giving $\sigma_w^2 = \sigma_t^2 = 0$. Thus:

$$\sigma_S = \left| \frac{\partial S}{\partial f} \right| \cdot \sigma_f = \frac{t \cdot w \cdot (w-1)}{(fw+t)^2} \cdot \sigma_f$$

For the experimental setup described in § 4.3.3, I measured $f = 1.50\,\mathrm{ms}$ and $\sigma_f = 0.21\,\mathrm{ms}$. This allows σ_s to be computed for a given t and w, as shown in Figure 4.6 on page 37.

[1]This analysis ignores any initial start-up time for the job, which is assumed to be negligible for large job sizes j. For the purposes of evaluation, I set $j = 500$.

D Software versions

Table D.1 lists the versions and licenses of all software libraries used for the development and evaluation of CausalRPC. The listed homepages were current as of 10th January, 2019.

Name	Version	License	Homepage
Alcotest	0.8.4	ISC*	`https://github.com/mirage/alcotest`
Alcotest-Lwt	0.8.0	ISC*	`https://github.com/mirage/alcotest`
Core_bench	0.11.0	MIT†	`https://github.com/janestreet/core_bench`
Dune	1.5.1	MIT†	`https://github.com/ocaml/dune`
Fmt	0.8.5	ISC*	`http://erratique.ch/software/fmt`
`git`	2.17.1	GPLv2§	`https://git-scm.com`
Irmin	`c1098efd`	ISC*	`https://github.com/mirage/irmin`
Irmin-Git	1.3.0	ISC*	`https://github.com/mirage/irmin`
Irmin-Unix	1.3.3	ISC*	`https://github.com/mirage/irmin`
Irmin-Mirage	1.3.0	ISC*	`https://github.com/mirage/irmin`
Logs	0.6.2	ISC*	`https://github.com/dbuenzli/logs`
Lwt	4.1.0	MIT†	`https://ocsigen.org/lwt`
Mirage-profile	0.8.2	BSD-2C‡	`https://github.com/mirage/mirage-profile`
OCaml	4.05.0	GPLv2§	`https://ocaml.org`
OCamldot	0.16.0	MIT†	`https://github.com/facebook/infer`
OCamlgraph	1.8.8	GPLv2§	`http://ocamlgraph.lri.fr/index.en.html`
OCaml-Git	2.0.0	ISC*	`https://github.com/dinosaure/ocaml-git`

Table D.1: *Versions of software used for development and evaluation.*

*URL: `https://opensource.org/licenses/ISC` (visited on 24th April, 2019)
†URL: `https://opensource.org/licenses/MIT` (visited on 24th April, 2019)
‡URL: `https://opensource.org/licenses/BSD-2-Clause` (visited on 24th April, 2019)
§URL: `https://www.gnu.org/licenses/old-licenses/gpl-2.0.txt` (visited on 24th April, 2019)

E Project proposal

My project proposal follows on the next page.

The rest of this page is left intentionally blank.

Computer Science Tripos – Part II Project Proposal

TraceRPC: Traceable RPC over Irmin

Candidate 2350F

October 19, 2018

Project supervisor and originator:	Dr A. Madhavapeddy (avsm2)
Director of studies:	Dr J. K. Fawcett (jkf21)
Project overseers:	Dr R. Mullins (rdm34) & Prof. P. Lio' (pl219)

1 Description of the work to be undertaken

Remote procedure calls (RPCs) provide a useful abstraction in the design of distributed systems: the calling process simply invokes a local procedure with serialisable arguments and the RPC framework handles the rest. An important cost of this abstraction is that it becomes much harder to construct a trace of the behaviour of the system when things go wrong.

Irmin is a distributed database library built on the design principles of Git[1]. This library allows the construction of distributed databases with user-defined consistency models. A key advantage of the Git-like approach is that traceability is inherent in the system: recovering the runtime history is as simple as running `git log` on any of the nodes in the system. The goal of this project is to construct an RPC framework that uses an Irmin datastore for data transfer and logging. One possible design of the datastore is shown in Figure 1. The project will be written in OCaml, the language in which Irmin is implemented.

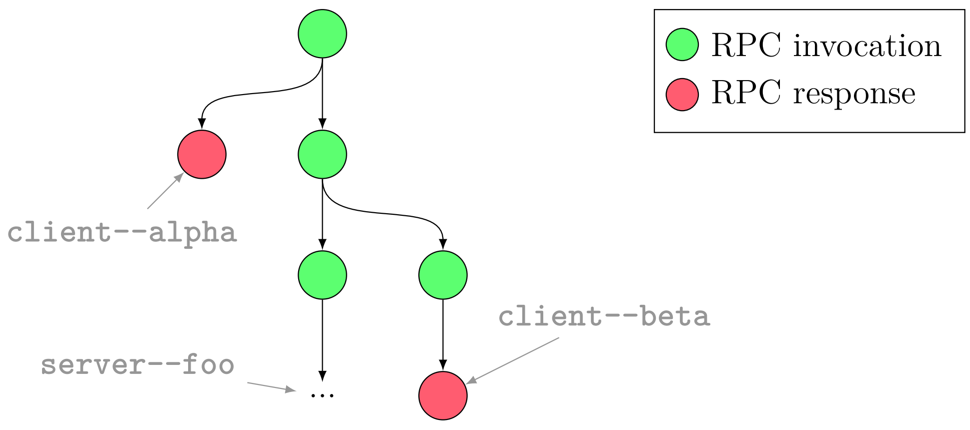


Figure 1: Diagram showing a possible mapping of RPC semantics onto Git branches. The client invokes an RPC by committing parameters to the Irmin store and pushing. The server fetches the new branch, invokes the local procedure implementation as requested, then commits the result of the RPC to Irmin and pushes. The server performs fast-forward merge operations on the client branches to assert a total order of operations.

[1] `https://github.com/mirage/irmin`

2 Description of the starting point

OCaml I have no prior experience with OCaml except for its usage in the Part IB Compiler Construction course. I have never worked on a major project in the functional programming style; however, I am familiar with basic functional programming principles from various courses in the Computer Science Tripos. I have provisioned time to work through *Real World OCaml*[2], an introductory text on OCaml development.

Irmin I have never worked with Irmin. My project plan includes time to familiarise myself with the inner workings of the library. I have used the `git` command line tool for several years, so I am familiar with the concepts of branching and merging that will form the basis of the RPC implementation.

RPC frameworks I am familiar with the general concepts of remote procedure calls from the Part IB Concurrent and Distributed Systems course. I have not built any RPC frameworks or applications using existing RPC frameworks before.

Prior Work There are several RPC libraries available for OCaml, including an implementation for the MirageOS unikernel[2] and a remote extension of the Irmin API using the Cap'n Proto data interchange format[3]. My project will implement a novel RPC mechanism and will not rely on any source code from these libraries.

3 Success criteria for the core deliverable

The project will be considered successful if the RPC implementation satisfies the following conditions:

- The RPC client (an OCaml application) is capable of remotely executing one of a set of exposed functions with Irmin serialisable parameters using a synchronous RPC.
- The RPC server supports communication with multiple clients concurrently.

The project will be evaluated by creating a small application to demonstrate a simple use-case of the RPC framework, likely a messaging application. This application will then be subjected to a simulated workload in order to conduct performance benchmarks of the RPC framework compared to other implementations. Irmin's lack of a garbage collector is a known issue[1], so analysing the memory usage of the RPC framework as a function of the number of messages sent would likely yield interesting results. The evaluation could also show how the performance of the framework degrades under load by examining latency as a function of the number of active RPC clients.

As OCaml code can be compiled to run natively, a wide range of performance benchmarks could be performed using conventional profiling tools such as `perf` on Linux[4].

[2] `https://github.com/mirage/ocaml-rpc`
[3] `https://github.com/zshipko/irmin-rpc`
[4] `https://perf.wiki.kernel.org`

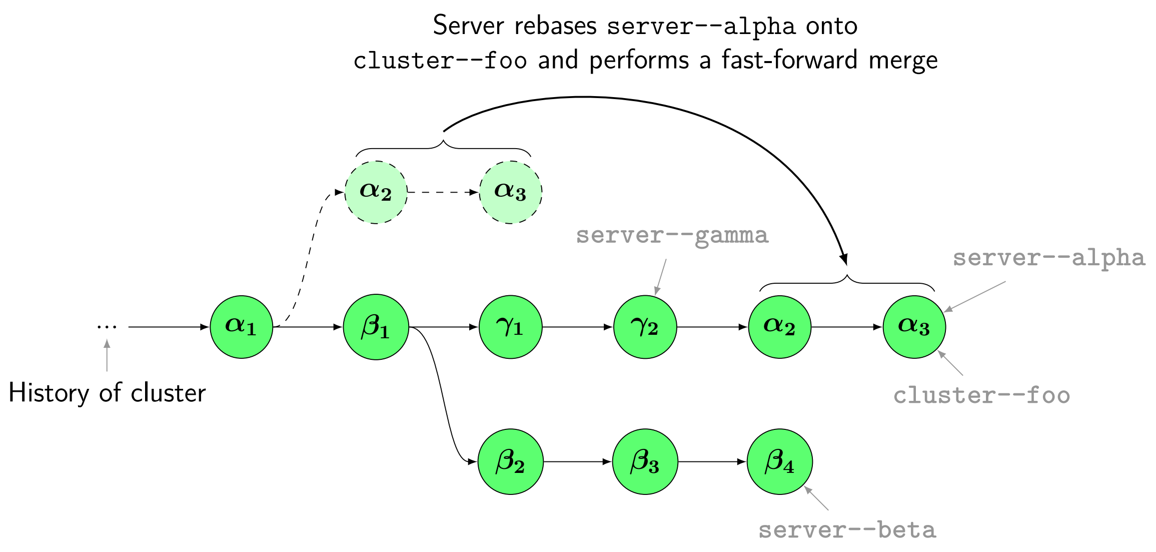


Figure 2: Diagram showing a possible design for exploiting parallelism using branches in Irmin. The multiple servers receive RPC invocations from clients and rebase onto `server` branches as before. When strong consistency is necessary, one of the servers rebases onto a global `cluster` branch that defines a total order on all RPCs.

4 Possible extensions

Asynchronous RPC The conventional method of synchronising data between multiple Irmin stores is via the `Irmin.Sync` module[5], which provides push/pull semantics very similar to the `git` command line tool. Using this module, the RPC server will have to poll in order to detect new clients and the client will have to poll in order to receive the results of the RPC. A viable extension of the core implementation could use the notification subsystem of Irmin to implement asynchronous RPCs.

Thread/server level parallelism The RPC framework could be extended to support multiple threads or servers operating on a single database, as shown in Figure 2.

Optimisations The project is amenable to a large variety of optimisations. For example, a binary serialisation format could be used for reduced latency compared to a text-based format such as JSON.

As each of these extensions is relatively performance-focused, a suitable success criterion for each of them would be a demonstration of measurable performance improvements with respect to some metric. For instance, an asynchronous RPC implementation would be expected to exhibit reduced network usage compared to a polling implementation.

5 Work plan

My work plan consists of 14 2-week work packages. I have reserved three of these for exam revision and contingency; the rest are associated with at least one milestone each.

18th Oct – 31st Oct ***19th Oct**: Proposal deadline*

[5] `https://mirage.github.io/irmin/irmin/Irmin/Sync/index.html`

Read relevant chapters of *Real World OCaml.* Study Irmin documentation and experiment with transferring data structures via the push/pull interface of `Irmin.Sync`. Investigate possible underlying data representations for encoding remote procedure calls in Irmin.

Milestone 1: *Sent supervisor a brief document detailing initial data representation ideas.*

1st Nov – 14th Nov

Plan the implementation of the project, including the software development strategy to be used. Produce a short document detailing the proposed project structure.

Milestone 2: *Implementation plan sent to supervisor.*

15th Nov – 28th Nov ***28th Nov****: Last lecture of Michaelmas term*

Produce a minimal working implementation of an RPC over Irmin, supporting only a single client and synchronous RPC calls.

Milestone 3: *Demonstration of a single client invoking RPC over Irmin.*

29th Nov – 12th Dec

Extend the implementation to support multiple users, using Git branches for isolation.

Milestone 4: *Demonstration of the RPC server handling multiple clients concurrently.*

13th Dec – 26th Dec *Exam revision and contingency.*

27th Dec – 9th Jan

Produce a document describing the work achieved so far and what remains to be done. Plan the evaluation of the project and document the proposed strategy.

Milestone 5: *Progress report complete.*
Milestone 6: *Evaluation plan sent to supervisor.*

10th Jan – 23th Jan ***17th Jan****: First lecture of Lent term*

Implement a simple application that uses the RPC framework, to be used for performance benchmarking. Produce slides for the progress presentation and rehearse delivery. If time permits, begin work on the extension tasks.

Milestone 7: *Application to be used for performance benchmarking is functional.*
Milestone 8: *Progress report presentation complete.*

24th Jan – 6th Feb ***1st Feb****: Progress report deadline*

Evaluate the core deliverables, according to the plan agreed with my supervisor. Write initial drafts of the introduction and preparation chapters.

Milestone 9: *Evaluation of core deliverable complete.*
Milestone 10: *Draft of introduction and preparation chapters written.*

7th Feb – 20th Feb

Write initial drafts of the implementation and evaluation chapters. If time permits, continue work on the project extensions.

Milestone 11: *Draft of implementation and evaluation chapters written.*

21st Feb – 6th March

Carry out evaluation of any extensions completed so far. Write an initial draft of the conclusion chapter and submit the first draft of the dissertation for review.

Milestone 12: *First draft of the dissertation complete.*

7th March – 20th March *Exam revision* ***13th March**: Last lecture of Lent term*

21st March – 3rd April

Improve the dissertation based on collected feedback. Finish evaluating any extensions.

Milestone 13: *Evaluation of extensions complete.*
Milestone 14: *Second draft of the dissertation complete.*

4th April – 17th April

Revise the dissertation using the second round of feedback from my supervisor.

Milestone 15: *Final draft of the dissertation complete.*

18th April – 1st May ***25th April**: First lecture of Easter term*

Carry out any final alterations to the dissertation.

Milestone 16: *Dissertation has been submitted.*

2nd May – 15th May *Exam revision* ***17th May**: Dissertation deadline*

6 Resource declaration

My personal laptop (Lenovo Thinkpad T460p)

Specifications: Intel i7-6700HQ (x86, 3.5GHz, 8 cores), 16GB RAM, Ubuntu 18.04 LTS

For writing and compiling project code, executing unit tests and writing the dissertation. In the event of hardware failure, I can use the MCS workstations in the CL for development; these are pre-installed with the tools I will need (`git`, `ocaml`, `opam`, `vim` etc.).

My project source code and dissertation will be version controlled using separate private Git repositories hosted by GitHub. For redundancy, I will use TravisCI to store backups of each repository in Amazon S3. I keep daily backups of my laptop's `/home` directory (containing my development repositories) on a personal RAID-1 storage array.

References

[1] B. Farinier, T. Gazagnaire, and A. Madhavapeddy. "Mergeable persistent data structures". In: *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*. 2015.

[2] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml: Functional programming for the masses.* O'Reilly Media, 2013. isbn: 9781449324759.