Functional Networking for Millions of Docker Desktops (Experience Report)

ANIL MADHAVAPEDDY, University of Cambridge, United Kingdom
DAVID J. SCOTT, Docker, Inc., United Kingdom
PATRICK FERRIS, University of Cambridge, United Kingdom
RYAN T. GIBB, University of Cambridge, United Kingdom
THOMAS GAZAGNAIRE, Tarides, France

Docker is a developer tool used by millions of developers to build, share and run software stacks. The Docker Desktop clients for Mac and Windows have long used a novel combination of virtualisation and OCaml unikernels to seamlessly run Linux containers on these non-Linux hosts. We reflect on a decade of shipping this functional OCaml code into production across hundreds of millions of developer desktops, and discuss the lessons learnt from our experiences in integrating OCaml deeply into the container architecture that now drives much of the global cloud. We conclude by observing just how good a fit for systems programming that the unikernel approach has been, particularly when combined with the OCaml module and type system.

CCS Concepts: • **Software and its engineering** → *Software system structures*; **Functional languages**; • **Computer systems organization** → **Cloud computing**.

Additional Key Words and Phrases: ocaml, containers, networking, hypervisors

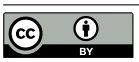
ACM Reference Format:
Anil Madhavapeddy, David J. Scott, Patrick Ferris, Ryan T. Gibb, and Thomas Gazagnaire. 2025. Functional Networking for Millions of Docker Desktops (Experience Report). *Proc. ACM Program. Lang.* 9, ICFP, Article 256 (August 2025), 19 pages. https://doi.org/10.1145/3747525

1 Introduction

Docker is a widely used developer tool designed to pull together an application stack (`docker build`), distribute the artefacts (`docker push`), and execute isolated applications on the same machine (`docker run`) while still sharing local storage and networks (`docker compose`). Developers usually compile their own Docker images via a `Dockerfile` supplied alongside their source code, and reuse other published images to share packaging efforts across programming languages. A Dockerfile to distribute an OCaml application might look like this:

```
FROM ocaml/opam:5.3                          # specify base image from the OCaml Docker org
COPY myapp.opam /app/myapp.opam              # add the local opam file to the image
RUN opam install . --deps-only               # install dependencies
COPY . /app                                  # copy the source code into the image
RUN opam install .                           # install the application code
CMD ["opam", "exec", "--", "myapp"]          # specify the command line to execute the command
```

Authors' Contact Information: Anil Madhavapeddy, University of Cambridge, Cambridge, United Kingdom, avsm2@cam.ac.uk; David J. Scott, Docker, Inc., Cambridge, United Kingdom, scott.dj@gmail.com; Patrick Ferris, University of Cambridge, Cambridge, United Kingdom, pf341@cam.ac.uk; Ryan T. Gibb, University of Cambridge, Cambridge, United Kingdom, rtg24@cam.ac.uk; Thomas Gazagnaire, Tarides, Paris, France, thomas@tarides.com.



ACM 2475-1421/2025/8-ART256
https://doi.org/10.1145/3747525

The Dockerfile above is a meta shell script that specifies a previously published filesystem image to use as a base, copies in files from the build host into the container, and executes commands to install the application. The results of each line are cached; in the example the external dependencies are only recompiled if the package metadata in the opam file changes. The resulting filesystem image can be run on another host with Docker installed, and will execute the `myapp` binary in a container that has access to the host's network and storage resources.

Docker has seen rapid adoption [11] since its first release in 2013, and is used by the majority of professional software developers [12, 20, 36]. The Docker Hub contains over 14 million filesystems and serves over 11 billion image pulls per month, and is just one of many registries to help developers share content. Docker has become a de-facto standard in the domain of managing cloud-native applications [9], and has also set a higher (but by no means perfect) standard for reproducible scientific research [8]. Software stacks increasingly combine diverse programming languages [35] and Docker makes development easier for emerging languages with less mature distribution.

The Docker ecosystem mostly uses Go, but there is also much functional programming used under the hood. There is a large OCaml system embedded within the macOS and Windows clients to dynamically translate calls between the Docker container environment (Linux) to the native operating system (macOS or Windows). In this report, we will first trace the origins of this OCaml code starting from Xen virtualisation (§2.1) through to the emergence of containerisation (§2.2). We then motivate the need for a Docker for Desktop application (§2.3) and describe the library virtual machine monitor (VMM) (§3.1) and networking components (§3.2). Our use of OCaml is somewhat unusual in that it is primarily used to build a library linked to an application that is itself written in C, ObjC, Swift and Go (§3.3). We reflect on how we built such a library stack, with discussion on the successful approaches (§4.1), challenges faced (§4.2) and underappreciated aspects of package management essential to shipping a complex system that is used by millions of users daily (§4.3).

2 The origins of functional programming in Docker

We will first give a brief origin story behind how some of the OCaml code we use dates back two decades, then introduce containerisation concepts for readers unfamiliar with Docker.

2.1 The 2000s: From Physical Machines to Operating System Virtualisation

At the turn of the century, it was common practise to manually install a Linux distribution or Windows machine and hand compile software to run on it [17]. Operating system virtualisation platforms such as Xen [4] or VMWare then enabled a larger number of "virtual" machines (VMs) to run on the same physical hardware. Applications adapted to support dynamic resource configurations [19] and in response, cloud computing tools such as OpenStack [48] and Vagrant [37] sprung up to manage clusters of VMs. These tools managed the lifecycle of deploying OS images, the software installed within them, and the configurations required to get a cluster running [47].

The toolstack for the Xen hypervisor was rewritten in OCaml [46] in 2005, becoming an early example of a functional programming language being used to manage the complexity of virtualisation. This project remains actively developed two decades on [41] and sparked a diaspora of OCaml systems library code that has a direct lineage to its eventual use in Docker. A key connecting project in 2009 was MirageOS, which restructured operating systems as libraries via a technique known as unikernels [31]. Instead of running general-purpose kernels with millions of lines of unused code for a particular application, unikernels could be disaggregated and selectively linked to the application, just as other userspace libraries such as `libc` are. MirageOS could run applications as highly specialised and memory-safe VM images [30], all written in OCaml down to the device drivers [42]. However, unikernels then lacked compatibility with existing applications, which opened up a window for an alternative approach known as "containers" to gain popularity.

2.2 The 2010s: The Rise of Containers for Application Distribution

The downside of using OS virtualisation to manage applications revolves around operational costs; wrapping an entire software stack ends up being large, hard to share and prone to bitrot. Docker was built in 2012 to provide developers with a few commands to get up and running quickly via lighterweight application containers. The key restriction was that the underlying kernel—initially Linux—had to be shared across all the running containers, and isolation between the services was not as strong as when using VMs. For most users who deployed all their software on the cloud, this was not a consequential barrier to adoption [52].

2.2.1 How Docker Runs Linux containers. Docker containers are, from the operating system level, just normal Linux processes. The Linux kernel is responsible for preventing individual processes from violating the memory integrity of other processes, but also for permitting safe sharing to facilitate useful work [18]. This is necessary, for example, to build a web frontend communicating with a backend database on the same host. These shared channels are also what lead to undesired interference if there are applications competing for global state, for example in the choice of TCP/IP network ports used. The role of Docker is to provide a convenient way to remap networking and filesystem needs to avoid such interference, via an easy-to-use interface.

Docker implemented this by using a feature in Linux known as *namespaces* [53], which gives each process more control over how to resolve shared resources such as filesystem name lookups. For example, in a root filesystem containing `/alice/etc/passwd` and `/bob/etc/passwd`, two processes under different namespaces could attempt to open `/etc/passwd`, and resolve to either the version under `/alice/etc/passwd` or `/bob/etc/passwd`. The process itself never sees the indirection into the wider root filesystem, and it can never access files outside its scoped subtree. Crucially, namespacing only applies when *opening* a resource, and the resulting file descriptor operates as a normal kernel resource for subsequent operations without further overheads. This allows Linux to maintain a compatible interface to applications that might have clashing resource needs, while still providing reasonable isolation within the bounds of sharing a kernel. Docker itself is a client-server application, with a server process that runs on the host and a `docker` command-line interface (CLI) that communicates with it via an RPC socket. The daemon manages the system resources such as running containers, stored images, networks and port remapping, and dynamic storage volumes.

2.2.2 Building Docker Images. A `docker build` makes a filesystem image that contains the artefacts resulting from the execution of the input Dockerfile. The container images are stored in a layered filesystem format. The bottom layers are either bootstrapped from a distribution such as Debian or Alpine Linux or custom-built from scratch. Subsequent layers then correspond to the filesystem differences resulting from the execution of individual directives. The image format itself has been standardised since 2016 by the Open Container Initiative (OCI), with multiple independent implementations now available [5]. A typical microservice application might consist of multiple such images (e.g. a webserver, a database and a caching server) that are all built with different base Linux distributions as desired. This is where Docker differs in philosophy[1] from systems such as NixOS [14] and Guix [10] that directly tackle the problem of dependency management within a *single* filesystem, but require significant repackaging effort. Docker sidestepped this issue by allowing each package to be built independently via namespace-isolated filesystems.

2.2.3 Running Docker Containers. A `docker run` executes an OCI-compliant filesystem image by forking a namespace-isolated process with the application binary as an entrypoint. The Docker container is therefore a Linux process with namespaces programmed to prevent crosstalk via

[1] Docker, Guix and NixOS (stable) all had their first releases during 2013, making that a bumper year for packaging aficionados.

(i) process groups for resource isolation; *(ii)* network translation scripts to remap local network ports within the container to those exposed externally; *(iii)* filesystem mounts to provide a read-only root filesystem and a writable overlay filesystem for the container; *(iv)* IPC namespaces to isolate the interprocess communication channels of the container from the host; *(v)* user namespaces to map the container-local user IDs to different ones on the host so that (for example) the `ocaml` user appears as UID 1000 within the container but is actually mapped to non-interfering host UIDs 12345 or 23456 when run on different hosts. While there is some overhead involved in the construction of these namespaces, it is far lower than the spawning of a full Linux VM [28].

2.3 The Motivation for Unikernels in Docker for Desktop

While the core of Docker was written in Go when first released, our functional programming journey began when we ran into a pressing problem in 2015. In two years after its launch, Docker's userbase had expanded rapidly but hit a hard usability barrier. The majority of developers use (then and now) macOS or Windows as their primary development environment [36] but Docker requires a Linux kernel to run containers. We therefore needed to ship Docker editions for macOS and Windows that "just worked" for developers already familiar with the Linux version, and also execute the same Linux container images that were being shared widely.

Unfortunately, the state-of-the-art "Docker Toolbox" that bundled a preinstalled VirtualBox [57] and Linux VM with Docker preinstalled was unpopular with the userbase. The loose integration meant that some features weren't tightly integrated; for example, managing the container storage required knowledge of Linux and logging into the VM, or accessing a local container required navigating to a link-local IP address rather than the usual `127.0.0.1`, and corporate desktop virus scanners routinely blocked "unknown" network traffic from the Linux VM as it looked like an unknown computer on the network. Ideally, the userbase wanted a zero-installation application called "Docker for Mac" or "Docker for Windows" that worked as seamlessly as the Linux version, without ever needing to know any of the details of how the containers were executed and exposed.

Our solution lay in using virtualisation features that were just being made available in desktop operating systems, but in an unconventional way. Instead of running a separate Linux VM on the desktop OS and managing it in parallel with our application, we instead *linked* a custom hypervisor Virtual Machine Monitor (VMM) to our userspace application process on macOS or Windows, and ran Docker for Linux via that library VMM as part of the application. The Linux lifecycle is therefore fully controlled by the host application and by rerouting the Linux networking and storage calls through to the application as well, we could make Linux an implementation detail irrelevant to the user. The Docker command-line interface could then be shipped as a native macOS (or Windows) application, further increasing the integration with the native desktop environment.

This is where the MirageOS project entered. Unikernels are characterised by offering a set of functions (schedulers, storage, filesystems, etc.) that can be called directly from the application logic, but without taking control of the full system. MirageOS had ready-made libraries for TCP/IP networking, storage (some inherited from the Xen toolstack) and scheduling all written in native OCaml. While it would be complicated, we hypothesised that there are few better ways of handling the intricate state management logic needed than via the discipline of functional programming! We assembled a small team in the summer of 2015 to build a rapid proof-of-concept of this native application, first beginning with macOS and subsequently Windows. Happily for the purposes of this experience report, the sprint was successful; our first Docker for Mac demonstrator took a week to prototype, and the subsequent VMM, networking and storage unikernel code was ready for beta testing within a month (§A). While the application has been downloaded hundreds of millions of times since that first release, and has had many subsequent updates since, its core architecture remains the same as of 2025 as the one we built in 2015.

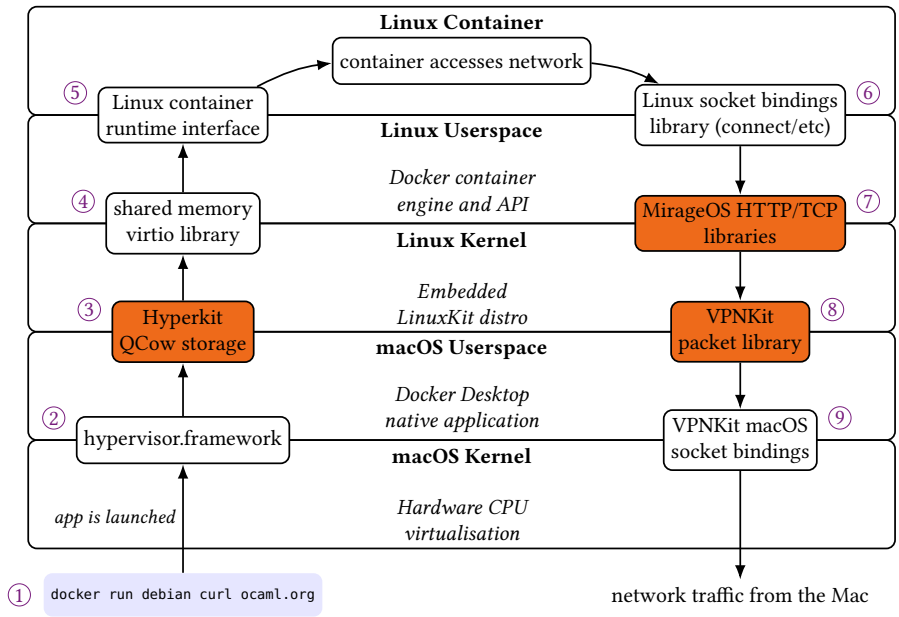


Fig. 1. Docker for Mac uses multiple virtualisation layers, whereby the macOS kernel embeds a Linux kernel within an application process, and then uses an OCaml unikernel to translate Linux network and storage I/O so that it appears to come from the Mac application. Orange nodes indicate primarily OCaml code.

3 The Architecture of Functional Programming in Docker for Desktop

We will next explain how the Docker for Desktop application works, and how functional programming was used to make it all fit together.

3.1 Hyperkit, a Library VMM to Embed Virtualisation

Figure 1 shows the flow of execution in the Docker for Mac application. It begins for the user with the macOS CLI that the application installs a native code-signed `docker` binary ① that forwards commands to a Docker server process. In our example command, we wish to use a Linux Debian base image to execute an HTTP fetch of the OCaml website. In order to do this, the client connect to the local socket server provided by Docker for Mac, and we enter a series of virtualisation layers.

Firstly, to get the basic support needed to embed a virtual kernel into a process we developed a new library VMM called HyperKit. This uses hardware virtualisation extensions [6] in Intel CPUs[2] to safely isolate virtual machines such as a Linux kernel within a macOS user process [26]. Most "type-1" hypervisors like Xen [4] boot as the very first piece of software (after the BIOS) and subsequently spawn a guest kernel. This isn't convenient for a desktop environment as it requires reboots and delicate bootloader configuration, and so macOS implemented a `Hypervisor.framework` in userspace ② that exposes just enough kernel support to expose virtual hardware traps required for running a VM as conventional function callbacks to a user-level process.

The Hyperkit application callbacks are minimal C functions that only abstract a virtual CPU, some virtual memory, and virtual interrupts with no direct support for emulated device drivers, leaving those to be implemented by the application. Our first OCaml code enters the picture to provide a programmable storage driver. We added libraries to Hyperkit to implement shared

[2]Modern ARMv8 CPUs, including Apple Silicon, also support this functionality nowadays.

memory channels ③ via the virtio protocol [44]. The bindings here were normal C FFI bindings (§4.1.3) from an OCaml perspective, using system threads and pipes to multiplex callbacks over to the Hypervisor framework. The OCaml interface to the block device was then higher level; for example using algebraic data types to express the protocol logic:

```
module Request = struct
  type t =
  | Connect of Block.Config.t * Qcow.Config.t
  | Get_info of int
  | Disconnect of int
  | Read of int * int * (Cstruct.t list)
  | Write of int * int * (Cstruct.t list)
  | Delete of int * int64 * int64
  | Flush of int
end
```

```
module Response = struct
  type ok =
  | Connect of int
  | Get_info of bool * int * int64 * bool
  | Disconnect
  | Read of int  | Write of int
  | Delete
  | Flush
  type t = (ok, Qcow.write_error) result
end
```

This request/response interface is sufficient to lift the remainder of the logic for our application storage from C into OCaml code, where higher-level logic can be more easily implemented. Having the ability to easily write custom storage management in OCaml solved one of our problems with the original Docker Toolbox, which was that the storage management was hardcoded in the VirtualBox VM and could not be customised by the user. By using OCaml to implement the storage driver, we could easily add new features such as garbage collection of unused space and integration with backup software [21].

The Hyperkit support for CPUs and memory along with this OCaml storage code was then enough to run an unmodified Linux kernel ④ within the application. This embedded Linux kernel then runs the Docker daemon, which in turn runs the containers ⑤ and exposes a normal Docker server socket that is forwarded back to the application to plant onto the host filesystem. The Linux distribution we designed is a custom one known as LinuxKit [40] to reflect its specialised nature. Instead of being a conventional standalone Linux distribution, it is only used as a component in a library VMM and can be embedded within a larger application. To minimise application startup time, we built a custom userspace that includes the absolute minimum binaries to run Docker containers, and ran every single service within a separate namespace, leaving nothing at all running in the root namespace beyond `init`.

This structure means that our application runs a single Linux VM with memory assigned to it from the application's own heap, within which a single Docker daemon runs, and Linux containers are multiplexed efficiently and share resources. The combination of Linuxkit and Hyperkit can spawn a Linux process almost as quickly as a native macOS process, with a negligible OCaml runtime startup cost. Other subsequent projects have adopted different approaches; most notably the Apple Containerization Framework introduced in June 2025 launches a separate Linux kernel for each container. While this has more isolation between the containers, it is also much slower to start up and requires more memory to run multiple containers.

3.2 VPNkit, an OCaml Unikernel for Networking Translation

While the Linux containers now ran at near-native speeds within macOS and Windows, plumbing networking traffic through to the embedded Linux instance proved surprisingly tricky. The conventional approach—used by normal desktop hypervisors such as VirtualBox—of bridging Ethernet network traffic from the desktop to the Linux VM required complex network management from the user. Even worse, network bridging in our first beta fell afoul of firewalls on corporate desktops that flagged this as malicious traffic, resulting in thousands of bug reports. Luckily, an ancient tool called SLIRP [39] came to our rescue, with an approach that was first used to connect Palmpilot

PDAs to the Internet in the mid 1990s and considered obsolete by 2015! Outgoing network traffic on a Windows or macOS host from a Linux VM triggers false positives in security scanners running on those hosts because they are configured to block all traffic that bypasses the host OS network stack. To work around this, we implemented logic in our application to intercept the network traffic from the Linux VM, and translate it in real time to the corresponding socket calls on macOS or Windows, thereby mimicking what a native macOS or Windows application might do to generate the network traffic. This sort of translation used to be done via SLIRP to connect 1990s devices to the fledgling dialup Internet, but hadn't been of much use since then—until now.

To rebuild SLIRP, we turned to more OCaml libraries from MirageOS [30] to perform the network translation required. When a container attempts a TCP[3] handshake, an ethernet frame containing the TCP SYN is sent to the host over the virtio protocol [44] over shared memory ⑥ and then fed into the user-space TCP/IP stack running on the host OS that reconstructs the traffic ⑦ into higher level protocol structures. This userspace stack, dubbed VPNkit, then calls the macOS `connect` syscall ⑧ and if successful, completes the TCP handshake. With this architecture, the application is only ever making native host socket calls ⑨ and the outgoing traffic will be perceived by the VPN policy as originating from the Docker application, rather than from a separate machine (albeit a virtual one). Deploying VPNkit in our beta tests in 2016 dropped the bug reports from corporate users by over 99%, and this approach has been a key component of Docker for Mac and Windows ever since. Our approach has subsequently seen adoption widely in the cloud world [60], bringing back an old dial-up networking trick to solve new problems in container management.

Handling incoming network traffic was also a challenge, but for different reasons. By default, when a Linux container listens on a port, it is not automatically exposed to the Internet unless requested on the CLI (e.g. `docker run -p 80:80 nginx` to expose nginx on port 80). The ideal user experience when running a container is that the exposed port appears directly on the desktop IP address, for example as `http://localhost:8080`. If we used network bridging, then another intermediate IP would be exposed instead of `localhost`, breaking our illusion of the Linux VM being invisible. Our LinuxKit kernel installs an eBPF program [33] that triggers the creation of a corresponding listening socket on the desktop host, and an OCaml port forwarder to allow the container to receive connections transparently. This allows for the perfect developer experience having container ports immediately being accessible on the Mac just as if it were a native service.

3.3 Pulling It All Together into a Native Application

All of the layers shown in Figure 1 are mostly implemented as a series of libraries — either in OCaml, Go, Swift, or C — and all linked together into a macOS application bundle that has logic to coordinate them via Swift and Objective-C. Since this is a native application, it is then code signed (essential now for modern Mac apps) and distributed as a normal drag-and-drop disk image that can just be launched by the user as any other native application is.

The application starts up the Hyperkit framework, initialises the OCaml runtime, allocates its storage file on the host, boots the LinuxKit kernel and the VPNkit service, and then listens on the Docker socket for the user to run the Docker CLI. The entire process is seamless to the user beyond starting the application, and the only indication that the Linux VM is running is a small whale icon in the macOS menu bar from which the service can also be configured. The approach described here has been so successful that it has been adopted by other container systems such as Podman [56], and is now a de facto way to run containers on macOS and Windows. Podman adopts a different set of choices in its libraries, of course, but the mechanisms remain similar.

[3]UDP packets are also tracked in a similar manner to stateful firewalls with port tracking

4 Reflections on Using OCaml in Docker

We will now examine some of the lessons learnt from shipping OCaml in production for a decade. Our application domain is somewhat unusual, as we are embedding the OCaml runtime deep inside an application binary alongside multiple other language runtimes (Go, Swift, C) and system libraries running on a desktop operating system. This is unlike the usual OCaml usecase of building a compiler or a standalone server or cloud service, and has thrown up some twists and turns.

4.1 The Good

The most obviously good thing about using OCaml is that our architectural tower of cards of libraries actually works at all! To quote the key feature from the OCaml manual [24, Chapter 22]:

> *The native-code compiler ocamlopt also supports the* `-output-obj` *and* `-output-complete-obj` *options, causing it to output a C object file or a shared library containing the native code for all OCaml modules on the command-line, as well as the OCaml startup code [...]The file produced by* `ocamlopt -output-complete-obj` *also contains the runtime and autolink libraries.*

The OCaml support for outputting native code libraries that can link as C libraries is a feature that often goes unsung, and was entirely key to our approach of using the Hypervisor framework combined with unikernel libraries (Figure 1) being practical. The OCaml toolchain is extremely portable, and the support for "partially linked" native code with a supported library interface to call back from foreign code is not a feature that many other languages explicitly support, and we greatly appreciate its existence. Beyond the clean toolchain, the OCaml language was also a good fit for this task for several reasons: the existence of parameterised modules to assemble the binaries, the combinator-style interfaces to facilitate concurrency, tracing and debugging, and the straightforward foreign function interface.

4.1.1 Parameterised Modules (aka OCaml Functors) for Networking. VPNKit is the OCaml service that bidirectionally translates between Linux ethernet traffic and macOS/Windows socket calls. This means it has to handle a throughput on the order of multiple gigabits per second of traffic, which can be done on a single core on most modern desktop-class CPUs.[4] On the other hand, it is important to avoid unnecessary overheads in the data plane, as this can lead to jitter and packet loss in the network traffic [27]. It is also quite difficult to test unusual network traffic patterns with real traffic, and so mock testing of the application logic is essential.

The entire assembly of VPNkit in the OCaml code was done using parameterised modules, also known as *functors* in OCaml. VPNKit has two main input interfaces: a Unix-style socket interface, and a low-level shared memory interface known as `hvsock`. The socket interface is used for the host communication, while the `hvsock` interface is used for the high-throughput data plane into the embedded Linux VM network stack. The codebase is structured around these two interfaces via OCaml signatures and functors. An example of some module signatures defined are:

```
module type FLOW_CLIENT = sig
  include Mirage_flow_combinators.SHUTDOWNABLE
  type address

  val connect: ?read_buffer_size:int -> address ->
    (flow, [`Msg of string]) result Lwt.t
end
```

```
module type Connector = sig
  include FLOW_CLIENT
  val connect: unit -> flow Lwt.t
  include READ_INTO
    with type flow := flow
    and type error := error
end
```

[4] Even with the switch to OCaml 5 (§4.2.1) we do not need multiple cores for VPNkit, just effect handlers to reduce allocation.

The module signatures above encapsulate common operations that can be performed on a network flow, such as connecting to a remote address, reading from the flow, and shutting it down. Signatures are independent, but can be structurally composed with other definitions. For example, `FLOW_CLIENT` includes the `SHUTDOWNABLE` signature from the third-party MirageOS libraries which are published separately. Similarly, `Connector` not only includes the local `FLOW_CLIENT` definition, but also another `READ_INTO` one that it further refines with type equalities to ensure that the abstract types `flow` and `error` are equal across the module signatures.

The actual implementations of these signatures are then provided by the application of combinations of parameterised modules:

```
module Bind = Bind.Make(Host.Sockets)
module Forward_unix = Forward.Make(Mclock)(Connect.Unix)(Bind)
module Forward_hvsock = Forward.Make(Mclock)(Connect.Hvsock)(Bind)
```

In the above, there is first a single-argument functor application using the `Bind` functor that accepts the host `Sockets` implementation. We then have two separate implementations of the same `Forward` interface by applying the `Forward.Make()` functor to the `Connect.Unix` and `Connect.Hvsock` modules. This allows the same code to be used for both the socket and shared memory interfaces, despite the shared memory implementation requiring a far deeper level of implementation than the relatively flat socket bindings. The test-suite provides yet another implementation of the `Connector` interface that does not require real network traffic (§B).

4.1.2 Pattern Matching and Combinators for Network Traffic. Pattern matching on algebraic data types is a classic feature of ML-style languages [32]. Their use lead to particularly elegant code in the complex packet parsing and reconstruction logic in VPNkit, which has to parse the TCP/IP and higher level (e.g. HTTP) protocol packets in real time using OCaml. The following code snippet shows how some incoming network traffic is parsed and then reconstructed in VPNkit:

```
let input_ipv4 t ipv4 = match ipv4 with
 (* UDP on port 53 -> DNS forwarder *)
 | Ipv4 {src; dst; payload = Udp { src = src_port; dst = 53; payload = Payload payload; _ }; _} ->
   let udp = t.endpoint.Endpoint.udp4 in
   !dns >>= fun t ->
   Dns_forwarder.handle_udp ~t ~udp ~src ~dst ~src_port payload >|= lift_udp_error
 (* Reconstruct HTTP proxy *)
 | Ipv4 {src; dst; payload=Tcp {src=src_port; dst=dst_port; syn; rst; raw; payload=Payload _; _}; _} ->
   let id = Stack_tcp_wire.v ~src_port:dst_port ~dst:src ~src:dst ~dst_port:src_port in
   begin match !http with
   | None -> Lwt.return_ok
   | Some http ->
       let { localhost_names; localhost_ips; _ } = t in
       let dst = dst, dst_port in
       (* HTTP proxy forwarding logic elided for clarity *)
    end
```

This code uses the previously described functor applications to implement the dataplane logic for packet reconstruction. The `input_ipv4` function is invoked by the MirageOS TCP/IP libraries (which have done bit-level parsing by this stage) when a new IPv4 packet is received, and the pattern match on the packet determines if it is a UDP packet on port 53 (DNS) or a TCP packet on port 80 (HTTP). If it is a DNS packet, the function passes the packet to the DNS forwarding logic, and if it is an HTTP packet, it reconstructs the packet using a HTTP protocol proxy module. Conventional top-down pattern matching allows for many such cases to be implemented in VPNKit quite easily, for all the layered wire protocols that are supported.

Note also the use of both error handling via `Some` and `None`, and asynchronous combinators via `Lwt.return` and the `(>>=)` bind operator, which are interleaved with the pattern matching. While this interleaving did work for many years of production use of VPNKit, it resulted in many more heap allocations than desired (due to the closure allocation from the monadic combinators). We discuss later (§4.2.1) how our shift to using the direct-style effect handlers [50] present in OCaml 5 onwards mitigates this by removing the need for monadic-style concurrent I/O.

Also noteworthy is how the same functional programming idioms were useful to express batched operations to scan a large amount of traffic. Debugging and testing of Docker in the field is particularly difficult due to the sensitivity of the network data being handled by the users (often in corporate networks), and so we integrated local packet capture logic into VPNkit which could be debugged on-device. A snippet of the OCaml logic looks like this:

```
let connect t vnet_switch vnet_cid client_macaddr c global_arp_table =
  Filteredif.connect ~valid_subnets ~valid_sources x |> fun fif ->
  Netif.connect fif |> fun interface ->
  Dns_forwarder.set_recorder interface;
  let kib = 1024 in
  let all_traffic = Netif.add_match ~t:interface ~name:"all.pcap" ~limit:(256 * kib)
    ~snaplen:c.Configuration.pcap_snaplen ~predicate:(fun _ -> true) in
  let (_: Netif.rule) = Netif.add_match ~t:interface ~name:"dns.pcap" ~limit:(256 * kib)
    ~snaplen:1500 ~predicate:is_dns in
  let (_: Netif.rule) = Netif.add_match ~t:interface ~name:"ntp.pcap" ~limit:(64 * kib)
    ~snaplen:1500 ~predicate:is_ntp in
  or_failwith "Switch.connect" (Switch.connect interface) >>= fun switch ->
```

The `connect` function here registers various packet capture predicates. At runtime, the packets that match a particular predicate are added to an efficient shared ring buffer, and exposed via the 9P filesystem protocol [38] for debugging tools to retrieve as a pcap-format stream. This way, we can lazily retrieve the right level of debugging data needed to diagnose a problem, with the 9P protocol logging all data retrieval to ensure customer privacy needs are both audited and respected. MirageOS already provided us with the 9P filesystem protocol implementation in OCaml as a library, and Linux has a built-in implementation as well.[5]

4.1.3 The OCaml FFI Is Lightweight and Stable for Systems Interfaces. It is crucial when building systems software that the foreign function interface be as predictable as possible, since low-level bugs in this area inevitably lead to hard-to-track down heap corruption that might only manifest as seemingly unrelated undefined behaviour. The OCaml FFI is particularly good in this regard, as it exposes a C interface with a series of macros and clear guidelines about how to play well with the runtime [24, Chapter 22]. All of our foreign function uses are either directly written using this interface, or go via the C stub generation mode of the `ctypes` library [59]. Writing C code manually, as opposed to using `ctypes`, offers greater flexibility. For example, being able to handle the different runtime representation of `Unix.file_descr` on Windows when setting the time-to-live option.

The OCaml FFI has been remarkably stable for the entire decade of upgrades of compiler versions within our codebase. It has been an eventful decade of development in the upstream OCaml compiler; we began using OCaml 4.02.1 in 2015, and there have been thirty compiler releases since including a large leap from OCaml 4 to OCaml 5 that significantly rewrote the runtime to be multicore-capable [13, 49]. However, every single one of these releases have maintained backwards compatibility with existing code [49], and we have seen no bugs in production arising from compiler upgrades.

[5] https://www.kernel.org/doc/Documentation/filesystems/9p.txt

These stability guarantees maintained by the OCaml compiler development team have let us to focus on the core logic rather than on the minutiae of the compiler developments. This long-term stability let us take advantage of code written elsewhere in the fairly small community of OCaml programmers by reusing often-subtle bindings to various system interfaces developed in other projects such as the Xen hypervisor toolstack [46], Jane Street's Core suite [29], or the LiquidSoap audio streaming suite [2], and of course from the MirageOS unikernel project itself.

While Windows support is often cited as problematic for OCaml, it is surprisingly straightforward from a low-level bindings perspective. Where there were gaps for Windows-specific functionality, we found it straightforward to write direct bindings to them. However, newer systems interfaces that depart from a C interface such as Grand Central Dispatch [45] on macOS is difficult to use efficiently and correctly from OCaml; we have so far managed to find C-based alternatives on macOS. The complexity lies in GCD's callback-oriented interface that involves running OCaml functions inside C callbacks, potentially in different OS threads.

4.2 The Bad

It hasn't all been plain sailing when using OCaml at such scale, although it has been mostly good. The main issues we have encountered have been around the concurrency model, and the syntactic complexity of the OCaml module system.

4.2.1 Monadic Concurrency Composes Poorly with Other Monads. Since OCaml 4 and earlier have no built-in support for concurrency beyond the use of preemptive threading, Hyperkit and VPNkit were both first written using Lwt [55], a monadic concurrency library. Lwt solved the immediate problem of handling concurrent IO operations without requiring low-level preemptive threading and callbacks. However, the switch to monadic control flow required extreme careful with error handling, since Lwt brings in a dual notion of exceptions. Lwt-style exceptions are propagated through binds in the Lwt monad, but native OCaml exceptions (as raised by most non-Lwt interfaces) must be caught and wrapped in an Lwt exception or else risk bubbling up in an incorrect point. Other OCaml concurrency libraries such as Jane Street Async opt to use global monitors to catch exceptions instead [29, Chapter 16] to avoid having to do this exception wrapping.

Our approach to fixing this arrived recently in OCaml 5, which introduced support for writing direct-style concurrency libraries using effect handlers [50]. Our port of VPNkit to one such direct-style concurrency library called Eio [23] shows great promise, as it not only makes the control flow simpler (due to the Lwt monad not appearing in the type), but also lets us fall back to using familiar libraries such as `List` and `Seq` directly to perform maps and folds rather than having to write equivalents using Lwt.

```
module Make_packet_proxy
 (I: Mirage_flow.S) (O: Mirage_flow.S) = struct
 let run incoming outgoing =
   let rec loop () =
     I.read incoming >>= function
     | Error err -> Fmt.failwith "%a" I.pp_error err
     | Ok `Eof -> Lwt.return_unit
     | Ok (`Data buf) -> (
       O.write outgoing buf >>= function
       | Ok () -> loop ()
       | Error err ->
         Fmt.failwith "%a" O.pp_write_error err)
   in loop ()
end
```

```
module Proxy = struct
  let run incoming outgoing =
    try
      while true do
       Eio.Flow.copy incoming outgoing
      done
    with
    | End_of_file -> ()
    | Write_error err ->
       Fmt.failwith "%a" pp_write_error err
    | Read_error err ->
       Fmt.failwith "%a" pp_read_error err
end
```

The left-hand side shows the OCaml 4 VPNkit code for proxying a packet. It is functorised around the flow signatures for the input and output flows[6] with a recursive function required to loop until completed due to the monadic bind operators. Every read and write operation results in a closure allocation which introduces higher heap usage for the networking code. Errors are handled through return values, interleaving the error handling logic with the core logic.

The right-hand side listing shows the same logic ported to the direct-style concurrency library Eio that uses effect handlers instead of monadic concurrency. The `run` is now a simple `while` loop that reads from the input flow and writes to the output flow, with the blocking IO hidden behind Eio's lightweight fibers. The error handling is more explicit and less interleaved with the core logic, with normal OCaml exceptions being raised (and thus, can either be handled or allowed to bubble up higher in the callstack). Exception backtraces and callstacks are also preserved, making debugging and tracing through the OCaml code easier. It is instructive to see how the VPNkit test harness invokes the two different libraries:

```
module Lwt_proxy = Make_packet_proxy (Mirage_flow_unix.Fd) (Mirage_flow_unix.Fd)
let run_lwt () =
    Lwt_main.run @@ Lwt_proxy.run Lwt_unix.stdin Lwt_unix.stdout

let run_eio () =
    Eio_main.run @@ fun env -> Proxy.run env#stdin env#stdout
```

In the case of Lwt, we must first apply the functor and then run this through the Lwt main loop. Eio, on the other hand, must install a toplevel effect handler to intercept the effects raised by the library code. In both cases, care must be taken to initialise the libraries correctly; `Lwt_main` must not be recursively called, and the Eio handler is essential or else a toplevel unhandled effect exception will result.

Whilst we saw many benefits there were also some more difficult sections to convert from monadic to direct-style concurrency. Eio uses structured concurrency with explicit scoping (an `Eio.Switch.t`) to group logically concurrent tasks together. Structured concurrency helps programmers not leak resources, but it made parts of the port less straight-forward as VPNKit was not written with this approach in mind. Also, in a monadic concurrency library like `Lwt`, every value of type `'a Lwt.t` is treated notionally as a promise. `Lwt` promises can either be translated to direct-style code or to `Eio` promises and knowing when to use which is not immediately obvious.

The Eio library is still in its early stages, but we are optimistic that it will be a good fit for the VPNkit codebase as it not only exhibits lower heap memory usage than the Lwt equivalent, but also has support for more advanced new Linux features for parallel IO such as `io_uring` instead of the older `epoll` interface for IO multiplexing.

4.2.2 Functors Are Useful in the Large, but a Pain in the Small. While the use of functors has been key to structuring the VPNkit codebase, they also bring with them a high cognitive load to the programmer. The OCaml module system is its own sub-language integrated into the familiar ML family, with different syntax and expressive power from the core language. As such, using a library that exposes its functionality as a parameterised module requires first understanding the module signature, then hunting for implementations that satisfy that signature (which may themselves be parameterised), and then ensuring that the type equalities are satisfied such that downstream users of the code will not get over-abstracted type signatures. While all of this is extremely useful to modularise a complex codebase like ours, it is a barrier to new contributors due to the lack of specialised tooling support to manipulate OCaml modules.

[6]Recall from §4.1.1 that VPNkit needs to proxy to and from both sockets and shared memory channels.

It would be extremely beneficial to be able to search for compatible modules at a type level, and in the future to have more type-level inference or dispatch for modules. There are efforts afoot within the OCaml community to address just this, such as via modular explicits [54] and modular implicits [58], as well as better support for module-level search within the new OCaml `odoc` documentation generator.

In the meanwhile, however, we have adopted a fairly simple rule of thumb also observed by FoxNet [7]: interfaces should only be functorised when they are going to be used in multiple places within the same codebase. Any single-use abstraction can be replaced with a conventional parameterically polymorphic record of functions instead or first class modules [15], which are easier to understand and maintain. This "defunctorisation" is occurring not only within VPNkit, but also within the wider MirageOS community. It does not affect the overall architecture of the codebase, which still greatly benefits from being able to cleanly abstract over systems interfaces to facilitate easier testing, debugging and code coverage.

4.3 The Ugly

Any language ecosystem, and particularly one that has been around for as long as OCaml, will have its share of warts. The following is not intended as a criticism of the hard work that has gone into many of these individual projects, but reflects the reality of having to manage the integration of a large codebase *across* ecosystems such as Go, OCaml and Swift.

Build and packaging systems, and especially their interoperability [34], were easily the biggest source of struggle when it came to managing our industrial codebase. For package management, we had to combine packages from Go, OCaml and Swift, which each have their own packaging formats. We used the Go modules system, opam for OCaml, and the Swift Package Manager, with a custom continuous integration (CI) system and monorepo-based workflow that integrated all the code in a single code repository. The CI systems we depended on from third party providers were also perpetually changing under our feet, since our need for macOS and Windows native builds were off the beaten path for most services that only supported Linux. Windows packaging for OCaml was problematic as there was no direct support in opam[7] except via community-maintained forks.

The opam support for maintaining our own custom repositories was extremely useful for our workflow, as we could maintain an internal opam repository that pinned our forks of all the relevant libraries we needed for Hyperkit and VPNkit without needing to maintain a package proxy as with Go. This was important to generate the licensing information from the third-party code we used (included in every distribution of Docker to ensure we attribute credit). We also discovered security vulnerabilities that were fixed by Apple and Microsoft (§A), and so needed careful control over our dependency tree to ensure we followed responsible vulnerability disclosure practises [22].

Our particular usecase might be slightly unusual, since we ultimately need to output binaries using the native toolchains on macOS and Windows. The CI problems might have been easier if we could cross-compile reliably from any host to Windows and macOS, but this seemed overly optimistic given the number of language toolchains involved. However, an exciting avenue we are exploring is to use the Zig toolchain as a cross-compilation base. Zig supplies a C/C++ compiler and linker wrapper that can target hundreds of CPU, OS and libc variants all from one binary (which bundles an LLVM toolchain within itself, including system headers). Once we can get the cross-walk of hosts and targets working for this, we will still need to figure out how to get Language Server Protocol support in these cross-toolchains so that typing and code completion works in the editor.

[7]This has recently been rectified by the release of opam 2.2, although we have not yet integrated this support into VPNkit.

Another area that took a lot of low-level debugging was in linking multiple language runtimes into a single process. While we had great success with linking OCaml (§4.1.3) and C, we had less with adding Go into the mix. Go began to use the `SIGURG` signal from Go 1.14 to support non-cooperative preemption of Goroutines, and the weight of POSIX signal delivery, multiple threads with masks, Grand Central Dispatch queues, and language runtimes was just too much technical debt to take on. The Go code therefore currently runs in a separate address space, but we hope to find solutions for this in the future, perhaps through the use of WebAssembly runtimes to give us a cleaner separation of concerns [51].

5 Conclusions

Our use of library-oriented programming to deliver Docker for Desktop is, we believe, a very useful way to build the "invisible systems glue" code that is pervasively needed in many systems programming tasks. There are an ever-growing number of hardware and software interfaces to access the outside world, most obviously with GPUs for machine learning workloads [1] but also FPGAs [43] and new storage and persistent memory devices [3]. These usually require significant retrofitting to work with existing codebases, and so building translation adapters like VPNkit and using library VMMs like Hyperkit will become more common in the future.

The library operating system architecture is surprisingly simple if every one of these new interfaces can be exposed as a library to link against (where applications invoke functions and manage state) instead of requiring a wrapper (handing off control to a separate process). Examples of this include the use of the Linux `liburing` library for parallel IO, the Vulkan graphics API, the OpenCL compute API, and the CUDA GPU programming API, all of which are exposed as libraries that can be linked against.

When it comes to using these libraries to assemble application logic, OCaml has been a superb choice of programming language. The sophisticated module system for organising implementations and signatures, and the higher-order programming to express their composition in an elegant yet performant way, has been a great fit for our needs. The OCaml runtime has been resolutely stable even across major upgrades, and the foreign function interface has been lightweight and predictable. The recent addition of effect handlers in OCaml 5 has addressed one of our biggest concerns, as it has allowed us to reduce the heap allocation of our networking code significantly.

We look forward to adopting effect handlers into our production codebase, as well as forthcoming developments such as modular implicits for succinctness [54, 58] and modal types [25] and data-race freedom extensions for performance through parallelism [16]. Above all, we thank the friendly and helpful OCaml community for their support over the years, and look forward to many more years of productive use of the language.

Acknowledgments

We would like to thank all the contributors over the years to VPNkit including Milas Bowman, Emmanuel Briney, Ian Campbell, Mathieu Champlon, Justin Cormack, Frédéric Dalleau, Akim Demaille, Ilya Dmitrichenko, Simon Ferquel, Pierre Gayvallet, Riyaz Faizullabhoy, Christiano Haesbaert, Anca Iordache, Thomas Leonard, Richard Mortier, Terry Moschou, Rolf Neugebauer, Mindy Preston, Michael Roitzsch, Guillaume Rose, Akihiro Suda, Balraj Singh, Magnus Skjegstad, David Sheets, Sebastiaan van Stijn, Tibor Vass, Gaetan de Villele, Ryuichi Watanabe, YAMAMOTO Takashi, Jeremy Yallop, and the original Docker project founder Solomon Hykes. We are also grateful to the MirageOS and the OCaml development teams for their support and assistance. The anonymous reviewers at ICFP also gave us substantial feedback that greatly improved the presentation of this paper.

A Timeline of the Early Days of VPNKit

This is a brief timeline of the significant developments in the early days of VPNKit, from its inception in 2015 to its release as a separate project in 2017. The timeline is based on the commit history of the VPNKit repository, and the release notes of the Docker for Mac beta program.

- **01-12-2015**: Prototype sprint to build Hyperkit and VPNkit demonstrators.
- **20-01-2016**: first VPNkit commit into monorepo
- **27-01-2016**: `networkType=vpnkit` added to Docker for Mac beta in response to bug reports, as optional and experimental.
- **30-01-2016**: Add HTTP support, which needed us to fix TCP half-close in MirageOS stack because default Mirage types only had `close` and not `shutdown`.
- **30-01-2016**: Added 9P control interface for creating port forwards
- **15-02-2016**: Use VPNkit network by default for internal comms.
- **07-03-2016**: Offer "VPN safe networking" as an option to the user in the shipping Docker for Desktop.
- **13-04-2016**: Use a privileged helper tool to bind privileged ports on macOS so that (e.g.) a web server can run on port 80.
- **14-04-2016**: Use `AF_VSOCK` for socket forwarding, removing some internal functors needed to forward over IP.
- **16-04-2016**: Add dual NICs, one for VPNkit, and another for `Vmnet.framework`. Vpnkit is slower but reliable, and the kernel is faster but doesn't always work over VPNs or due to macOS resource limits or if Internet Connection Sharing is used.
- **27-04-2016**: Support ICMP (ping) forwarding.
- **01-05-2016**: Functorize the whole stack to support Windows IO
- **01-07-2016**: Use libuv instead of the default Lwt engine, needed for scalability especially on Windows where `WaitForMultipleObjects` is very slow.
- **19-07-2016**: Add rate limiting code to avoid exhausting file descriptors on macOS, where the limits are much lower than Linux
- **20-09-2016**: macOS Sierra 10.12 includes a security fix for a `vmnet.framework` security vulnerability we discovered[8]
- **03-10-2016**: Allow internal state to be queried over 9P, which is essential for on-site diagnostics.
- **23-02-2017**: Rename to VPNkit and release as an open-source project.

The timeline continues after this to the present day, with the most recent development as of 2025 being the port to OCaml 5 and the use of the Eio library for concurrency (§4.2.1).

B Supplemental Code Listings

These are some code listings that are instructive extensions from the explanations in the main text.

The following snippet shows how the VPNkit network stack is constructed from the MirageOS libraries a series of functor applications. In the case of ICMPv4, the code below illustrates how we extend the original module definition with custom logic specific to our use in Docker. The original ICMPv4 logic is included in a wrapper module, and then some of the functions are redefined subsequently to override the input parsing logic.

```
module Netif = VMNET
module Ethif1 = Ethernet.Make(Netif)
module Arpv41 = Arp.Make(Ethif1)(Host.Time)
module Dhcp_client = Dhcp_client_mirage.Make(Mirage_random_stdlib)(Host.Time)(Netif)
```

[8] See https://support.apple.com/en-us/103424

```
module Ipv41 = Dhcp_ipv4.Make(Mirage_random_stdlib)(Mclock)(Host.Time)(Netif)(Ethif1)(Arpv41)
module Icmpv41 = struct
  include Icmpv4.Make(Ipv41)
  let packets = Queue.create ()
  let input _ ~src ~dst buf =
    match Icmpv4_packet.Unmarshal.of_cstruct buf with
    | Error msg ->
      Log.err (fun f -> f "Error unmarshalling ICMP message: %s" msg);
      Lwt.return_unit
    | Ok (reply, _) ->
      let open Icmpv4_packet in
      begin match reply.subheader with
        | Next_hop_mtu _ | Pointer _ | Address _ | Unused ->
          Log.err (fun f -> f "received an ICMP message which wasn't an echo-request or reply");
          Lwt.return_unit
        | Id_and_seq (id, _) ->
          Log.info (fun f ->
            f "ICMP src:%a dst:%a id:%d" Ipaddr.V4.pp src Ipaddr.V4.pp dst id);
            Queue.push (src, dst, id) packets;
            Lwt.return_unit
      end
end
module Udp1 = Udp.Make(Ipv41)(Mirage_random_stdlib)
module Tcp1 = Tcp.Flow.Make(Ipv41)(Host.Time)(Mclock)(Mirage_random_stdlib)
include Tcpip_stack_direct.Make(Host.Time)
    (Mirage_random_stdlib)(Netif)(Ethif1)(Arpv41)(Ipv41)(Icmpv41)(Udp1)(Tcp1)
```

The following snippet shows how the DHCP server in VPNkit is implemented using the MirageOS DHCP server library. The DHCP server is a stateful server that maintains a database of leases, and updates this database as clients request and release leases. The server also sends replies to clients with the appropriate lease information, and logs boot requests from clients. The server is implemented as a function that takes the current database, a configuration, a network interface, and a buffer containing the DHCP packet, and returns the updated database.

```
let input net config database buf =
  let open Dhcp_server in
  match Dhcp_wire.pkt_of_buf buf (Cstruct.len buf) with
  | Error e -> Lwt.return database
  | Ok pkt ->
    let tm = Clock.elapsed_ns () |> Duration.to_sec |> Int32.of_int in
    match Input.input_pkt config database pkt tm with
    | Input.Silence -> Lwt.return database
    | Input.Update database -> Lwt.return database
    | Input.Reply (reply, database) ->
      let open Dhcp_wire in
      if pkt.op <> Dhcp_wire.BOOTREQUEST || not !logged_bootrequest
      then Log.info (fun f -> f "%s from %s" <...>)
```

References

[1] Martín Abadi. 2016. TensorFlow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 1. doi:10.1145/2951913.2976746

[2] David Baelde, Romain Beauxis, and Samuel Mimram. 2011. Liquidsoap: A high-level programming language for multimedia streaming. In *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 99–110. doi:10.1007/978-3-642-18381-2_8

[3] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. 2021. Persistent memory: A survey of programming support and implementations. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–37. doi:10.1145/3465402

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 164–177. doi:10.1145/945445.945462

[5] Vincent Batts. 2016. Open Containers Initiative Image Specification. https://github.com/opencontainers/image-spec

[6] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association.

[7] Edoardo Biagioni, Robert Harper, and Peter Lee. 2001. A Network Protocol Stack in Standard ML. *Higher Order Symbol. Comput.* 14, 4 (Dec. 2001), 309–356. doi:10.1023/A:1014403914699

[8] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 71–79. doi:10.1145/2723872.2723882

[9] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. 2022. *Kubernetes: up and running*. OReilly Media.

[10] Ludovic Courtès and Ricardo Wurmus. 2015. *Reproducible and User-Controlled Software Environments in HPC with Guix*. Springer International Publishing, 579–591. doi:10.1007/978-3-319-27308-2_47

[11] DataDog. 2015. Surprising Facts about Real Docker Adoption. https://www.datadoghq.com/docker-adoption/

[12] Docker. 2024. Docker Stack Overflow Survey: Thank You. https://www.docker.com/blog/docker-stack-overflow-survey-thank-you-2024

[13] Stephen Dolan, Leo White, and Anil Madhavapeddy. 2014. Multicore OCaml. In *the 4th ACM OCaml Users and Developers Workshop*.

[14] Eelco Dolstra and Andres Löh. 2008. NixOS: A purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 367–378. doi:10.1145/1411204.1411255

[15] Jacques Garrigue and Alain Frisch. 2010. First-class modules and composable signatures in Objective Caml 3.12. In *ACM SIGPLAN Workshop on ML*.

[16] Aïna Linn Georges, Benjamin Peters, Laila Elbeheiry, Leo White, Stephen Dolan, Richard A. Eisenberg, Chris Casinghino, François Pottier, and Derek Dreyer. 2025. Data Race Freedom à la Mode. *Proc. ACM Program. Lang.* 9, POPL, Article 23 (Jan. 2025), 31 pages. doi:10.1145/3704859

[17] Bob Glickstein and K. Hodgson. 1998. GNU Stow—Managing the Installation of Software Packages. https://www.gnu.org/software/stow/

[18] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. 1976. Protection in operating systems. *Commun. ACM* 19, 8 (Aug. 1976), 461–471. doi:10.1145/360303.360333

[19] Brian Hayes. 2008. Cloud computing. *Commun. ACM* 51, 7 (July 2008), 9–11. doi:10.1145/1364782.1364786

[20] Solomon Hykes. 2013. The future of Linux containers. https://www.youtube.com/watch?v=wW9CAH9nSLs

[21] Docker Inc and the MirageOS team. 2025. OCaml QCow. https://github.com/mirage/ocaml-qcow/tree/main/lib

[22] Nobutaka Kawaguchi, Charles Hart, and Hiroki Uchiyama. 2024. Understanding the Effectiveness of SBOM Generation Tools for Manually Installed Packages in Docker Containers. *Journal of Internet Services and Information Security* 14, 3 (2024), 191–212.

[23] Thomas Leonard, Patrick Ferris, Christiano Haesbaert, Lucas Pluvinage, Vesa Karvonen, Sudha Parimala, K Sivaramakrishnan, Vincent Balat, and Anil Madhavapeddy. 2023. Eio 1.0-effects-based IO for OCaml 5. In *OCaml Workshop*, Vol. 36.

[24] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, KC Sivaramakrishnan, and Jérôme Vouillon. 2023. *The OCaml system release 5.1: Documentation and user's manual*. Ph. D. Dissertation. Inria.

[25] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP, Article 253 (Aug. 2024), 30 pages. doi:10.1145/3674642

[26] Anil Madhavapeddy. 2016. Improving Docker with Unikernels: Introducing HyperKit, VPNKit and DataKit. https://www.docker.com/blog/docker-unikernels-open-source/

[27] Anil Madhavapeddy, Alex Ho, Tim Deegan, Dave Scott, and Ripduman Sohan. 2007. Melange: creating a "functional" internet. *ACM SIGOPS Operating Systems Review* 41, 3 (jun 2007), 101–114. doi:10.1145/1272998.1273009

[28] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 559–573. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[29] Anil Madhavapeddy and Yaron Minsky. 2022. *Real World OCaml: Functional Programming for the Masses*. Cambridge University Press. doi:10.1017/9781009129220

[30] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, Dave Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. ACM, Houston Texas USA, 461–472. doi:10.1145/2451116.2451167

[31] Anil Madhavapeddy and Dave Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. *ACM Queue* 11, 11 (nov 2013), 30–44. doi:10.1145/2557963.2566628

[32] Luc Maranget. 2008. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. 35–46.

[33] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 1–8.

[34] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proc. ACM Program. Lang.* 2, ICFP, Article 79 (July 2018), 29 pages. doi:10.1145/3236774

[35] Sam Newman. 2021. *Building microservices*. OReilly Media.

[36] Stack Overflow. 2024. Stack Overflow Developer Survey 2024. https://survey.stackoverflow.co/2024/technology

[37] Jay Palat. 2012. Introducing vagrant. *Linux Journal* 2012, 220 (2012), 2.

[38] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. 1990. Plan 9 from bell labs. In *Proceedings of the summer 1990 UKUUG Conference*. London, UK, 1–9.

[39] Kelly Price, Tom May, and Chris Moustakis. 1996. Slirp, the PPP/SLIP-on-terminal emulator. https://slirp.sourceforge.net

[40] The Linuxkit Project. 2025. LinuxKit, a toolkit for building custom minimal, immutable Linux distributions. https://github.com/linuxkit/linuxkit

[41] The XenServer Project. 2025. XenAPI. https://github.com/xapi-project/

[42] Gabriel Radanne, Thomas Gazagnaire, Anil Madhavapeddy, Jeremy Yallop, Richard Mortier, Hannes Mehnert, Mindy Preston, and Dave Scott. 2019. Programming Unikernels in the Large via Functor Driven Development. doi:10.48550/arXiv.1905.02529

[43] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. 2023. Hardcaml: An OCaml hardware domain-specific language for efficient and robust design. *arXiv preprint arXiv:2312.15035* (2023). doi:10.48550/arXiv.2312.15035

[44] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[45] Kazuki Sakamoto, Tomohiko Furumoto, Kazuki Sakamoto, and Tomohiko Furumoto. 2012. Grand central dispatch. *Pro Multithreading and Memory Management for iOS and OS X* (2012), 139–145.

[46] Dave Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. 2010. Using functional programming within an industrial product group: perspectives and perceptions. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ACM, Baltimore Maryland USA, 87–92. doi:10.1145/1863543.1863557

[47] Dave Scott, Richard Sharp, and Anil Madhavapeddy. 2012. Programming the Xen cloud using OCaml. In *the 1st ACM OCaml Users and Developers Workshop*. ACM.

[48] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. 2012. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications* 55, 3 (2012).

[49] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (aug 2020), 1–30. doi:10.1145/3408995

[50] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 206–221. doi:10.1145/3453483.3454039

[51] Benedikt Spies and Markus Mock. 2021. An evaluation of webassembly in non-web environments. In *2021 XLVII Latin American Computing Conference (CLEI)*. IEEE, 1–10.

[52] James Turnbull. 2014. *The Docker Book: Containerization is the new virtualization*. James Turnbull. https://dockerbook.com/

[53] Alexander Viro. 2001. Per-process namespaces for Linux. https://lore.kernel.org/all/Pine.GSO.4.21.0102242253460.24312-100000@weyl.math.psu.edu/

[54] Samuel Vivien, Didier Rémy, Thomas Réfis, and Gabriel Scherer. 2024. On the design and implementation of Modular Explicits. (Sept. 2024). https://cambium.inria.fr/~remy/ocamod/implicits.html Presented at the OCaml 2024 workshop, available electronically.

[55] Jérôme Vouillon. 2008. Lwt: a cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. 3–12.

[56] Daniel Walsh. 2023. *Podman in Action: Secure, rootless containers for Kubernetes, microservices, and more*. Simon and Schuster.

[57] Jon Watson. 2008. Virtualbox: bits and bytes masquerading as machines. *Linux Journal* 2008, 166 (2008), 1.
[58] Leo White, Frédéric Bour, and Jeremy Yallop. 2015. Modular implicits. *arXiv preprint arXiv:1512.01895* (2015).
[59] Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2018. A modular foreign function interface. *Science of Computer Programming* 164 (oct 2018), 82–97. doi:10.1016/j.scico.2017.04.002
[60] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/young

Received 2025-02-27; accepted 2025-06-27