CloudMicroHaskell

Direct-Style Distributed Haskell via Runtime Graph Serialisation

ROBERT KROOK, Chalmers University of Technology and University of Gothenburg, Sweden
LENNART AUGUSTSSON, Unaffiliated, Sweden

Cloud Haskell brings Erlang-style distributed programming to Haskell, but its treatment of mobile code exposes a difficult boundary in the source-level API. Remote processes must be expressed as static closures, messages must satisfy serialisation constraints, and participating nodes are assumed to share the relevant code.

This paper explores a different design point. We present CloudMicroHaskell, a Cloud Haskell-style library built on MicroHaskell, whose runtime represents both code and data as a combinator graph. When a process or message crosses a node boundary, CloudMicroHaskell serialises the reachable graph directly. As a result, remote spawning can be written in direct style: process bodies may capture variables from their surrounding scope, and messages may contain ordinary values, including functions, without programmer-written closure conversion.

We describe the implementation of the CloudMicroHaskell node runtime, including remote spawn, message delivery, monitors, exit propagation, and library implementations of generic servers and supervisors. We evaluate the system with process/message benchmarks, a distributed work-pool benchmark, a file-synchronisation case study, and a heterogeneous deployment on microcontrollers. The results show that runtime graph serialisation makes the Cloud Haskell programming model substantially more direct, while also making the tradeoff explicit: some guarantees enforced by Cloud Haskell's source-level types become dynamic checks, and programmers must be aware of laziness and runtime-owned resources when moving graphs between nodes.

CCS Concepts: • **Networks → Programming interfaces**; **Network properties**; • **Computer systems organization → Embedded software**; **Dependable and fault-tolerant systems and networks**; • **Theory of computation → Self-organization**; **Concurrent algorithms**.

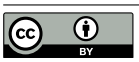
Additional Key Words and Phrases: Haskell, Distributed Haskell, Actor Model Concurrency, Concurrency, MicroHaskell, MicroHs, Serialisation, Graph Reduction, Combinators, Code Mobility

ACM Reference Format:
Robert Krook and Lennart Augustsson. 2026. CloudMicroHaskell: Direct-Style Distributed Haskell via Runtime Graph Serialisation. In *Proceedings of the 19th ACM SIGPLAN International Haskell Symposium (Haskell '26), August 24–29, 2026, Indianapolis, IN, USA*. ACM, New York, NY, USA, 31 pages. https://doi.org/10.1145/3830439.3831272

1 Introduction

Cloud Haskell [4] is a Haskell framework for writing distributed, concurrent programs in the style of Erlang. Erlang's programming model emphasises concurrency, with lightweight processes that share no memory. Instead, inter-process communication is done via message passing. This

Authors' Contact Information: Robert Krook, Chalmers University of Technology and University of Gothenburg, Department of Computer Science and Engineering, Gothenburg, Sweden, krookr@chalmers.se; Lennart Augustsson, Unaffiliated, Gothenburg, Sweden, lennart@augustsson.net.



Haskell '26, Indianapolis, IN, USA

ACM ISBN 979-8-4007-2869-3/2026/08
https://doi.org/10.1145/3830439.3831272

programming model greatly aids in writing distributed programs that execute on and communicate over a network.

The primary benefit of Cloud Haskell over Erlang is Haskell's strong type system. With it, the API of Cloud Haskell can be described in a very clear and expressive way. It is short and (mostly) elegant. Furthermore, Cloud Haskell extends Erlang's untyped message passing machinery with typed channels, so that communicating processes can exchange messages whose format is known at compile time. Erlang, being dynamically typed, cannot statically enforce a message passing protocol between processes.

These advantages come from making distribution explicit in the API, and that explicitness has practical consequences. The implementation assumes that nodes in the network run a very uniform software environment, and the machinery required to handle closures is necessarily visible to the programmer. We illustrate this with the direct-style program one might like to write

```
1  sendFunc :: Pid -> Int -> ProcessM ()
2  sendFunc p x = send p (\y -> x + y + 1)
```

The above example sends a lambda to a recipient process, which captures a free variable (x). Cloud Haskell uses the type system to enforce that values are serialised before transmission, but note that the type of the function, `Int -> Int`, says nothing about the type of the free variable. It is therefore not possible to provide an ordinary serialiser for functions from the function type alone.

To get around this, a mechanism for `Static` values was devised. Explained briefly, a value is considered static if it is defined at the top-level and refers to only bound variables or other variables defined at the top-level. Furthermore, when a static function is transmitted to a remote node, it must be explicitly closure converted, whereby the environment of the function is populated with its (serialised) free variable. The function is responsible for deserialising its environment itself. To implement the above example, the programmer must write

```
1  sendFunc :: Pid -> Int -> ProcessM ()
2  sendFunc p x = send p clo
3    where
4      -- closures can be serialised
5      clo :: Closure (Int -> Int)
6      clo = MkClosure (static sfun) (encode x)
7
8  sfun :: ByteString -> Int -> Int
9  sfun bs y = let x = decode bs
10             in x + y + 1
```

The previously intuitive and brief program has now been rewritten in a style with a significantly higher cognitive cost. Furthermore, a static value is assumed to exist on both machines (the one running the sending process and the one running the receiving process), and a closure only contains a static reference to it (together with its environment). While it seems like a function is being sent, what is actually being sent is a static reference to it and its environment.

While this is a clever solution to a tricky problem, we note that there are strict requirements placed on participating machines in the network, namely that they must run binaries that contain the same static symbols.

In this paper, we ask what Cloud Haskell might look like if the serialisation boundary were moved out of the source-level API and into the runtime representation of the program. We present CloudMicroHaskell, an exploration of the Cloud Haskell programming model using the MicroHaskell compiler [2] (MicroHs for short). MicroHs is a Haskell compiler with a much smaller and more modest runtime system than GHC, the de facto Haskell compiler. MicroHs emits a *combinator*

graph as its object code — a self-modifying byte code that represents both a program's data and code. While the cost of this decision is a significant reduction in performance, there are several benefits, which we hope to illustrate with this paper.

Using MicroHs changes the tradeoffs in several useful ways, namely

- We use a simple compiler primitive to serialise values of (almost) any kind, including functions. The first variant of `sendFunc` in the introduction is directly runnable in CloudMicroHaskell.
- Our implementation does not require every node in the network to agree beforehand on which static values should exist. Instead, nodes need only agree on a compatible CloudMicroHaskell runtime and the combinator format understood by that runtime.
- CloudMicroHaskell can serialise and transmit a message of (almost) any type, without requiring a serialisable constraint. This reduces the boilerplate required to prepare a data type for transmission over a network.

This does not remove the distribution boundary, but relocates it. The price is that this boundary is no longer enforced by Haskell's source-level types. Deserialisation is necessarily a dynamic claim that the received graph has the type expected by the receiver. If that claim is wrong, the program may fail at runtime. In that sense, CloudMicroHaskell trades some of Cloud Haskell's static safety for a much simpler programming model.

The contribution is therefore not the actor API itself, which is deliberately conventional, but the consequences of making runtime graph serialisation the mobility mechanism for that API.

2 Programming in CloudMicroHaskell

In this section we present CloudMicroHaskell. At the surface, CloudMicroHaskell looks much like Cloud Haskell. Programs create processes, identify remote nodes, and communicate by message passing. This is deliberate. The point of CloudMicroHaskell is not to propose a new process calculus, but to explore what happens when the serialisation boundary in Cloud Haskell is moved out of the source-level API and into the MicroHs runtime.

The most important differences therefore appear in the two operations that move computation and data between nodes, `spawn` and `send`. In Cloud Haskell, these operations expose distribution through `Closure` and `Serializable` constraints. In CloudMicroHaskell, these operations accept ordinary process computations and ordinary values, relying instead on MicroHs's graph serialiser (the serialiser and deserialiser are described in section 3).

We therefore focus on these two operations in the main text. The surrounding API follows the familiar Cloud Haskell and Erlang shape[1].

The core difference is visible in the types

```
1  spawn :: NodeId -> ProcessM () -> ProcessM Pid
2  send  :: Identifiable process => process -> a -> ProcessM ()
```

In Cloud Haskell, remote spawning requires a `Closure` and message passing requires a `Serializable` constraint. In CloudMicroHaskell, the programmer supplies the computation or value directly. If the target process or message crosses a node boundary, the runtime will serialise the reachable MicroHs graph, capturing both the code and environment. This makes direct-style examples possible to execute, but it also means that some errors that Cloud Haskell eliminates via the type checker become runtime failures in CloudMicroHaskell.

The serialisation primitive is intentionally very broad, but not universal. Some things are owned by a particular runtime instance, such as `MVar`s and open file handles. These cannot be serialised. The exceptions are the standard handles *stdin*, *stdout*, *stderr*, and the *null pointer*. These handles

[1]The full API is described in appendix A

are recognised by the MicroHs runtime system. They can be serialised and bound to the receiver's version during deserialisation. Similarly, the null pointer is always a zero-valued pointer, and can be serialised and deserialised as such.

The example below illustrates how `spawn` is used to spawn a concurrent process that captures free variables from the outer process:

```
data PingPong = Ping | Pong

f :: ProcessM ()
f = do
  s <- self
  n <- node
  n' <- (head . filter (/= n)) <$> nodes

  child <- spawn n' $ do
    Ping <- expect
    liftIO $ putStrLn "child received ping!"
    send s Pong

  send child Ping
  Pong <- expect
  liftIO $ putStrLn "parent received pong!"
```

The parent process first fetches its own process identifier by calling `self`. It then spawns a child process on a remote node. The child refers to `s`, which is bound in the parent. When `spawn` crosses the node boundary, the child computation is serialised together with this reachable environment. Additionally, we can send the `PingPong` data type without saying how to serialise it.

To make it even clearer that code is serialised and sent to a remote node, we show yet another example:

```
f :: ProcessM ()
f = do
  s <- self
  n <- node
  n' <- (head . filter (/= n)) <$> nodes

  let interesting x = x `mod` 7 == 0 && x `mod` 11 == 0

  _ <- spawn n' $ do
    let r = length (filter interesting [1..1000000])
    r `seq` send s r

  r <- expect :: ProcessM Int
  liftIO $ print r
```

We define a local function `interesting :: Int -> Bool`, and reference it from the spawned process. `interesting` will be included in the serialised graph that is sent to the remote node, and reconstructed on that node's heap. The call to `seq` is intentional. Since CloudMicroHaskell serialises graphs, not values produced by a `Binary` encoder, sending `r` without first evaluating it could send a

thunk back to the original node. The result would still be correct, but the work would happen in the wrong place.

It is not just `spawn` that can be used to send functions to remote processes. `send` uses the same underlying serialiser, and will happily send functions. Consider the example below:

```
data Apply = Apply (Int -> Int) Int Pid

worker :: ProcessM ()
worker = do
  Apply f x replyTo <- expect
  let y = f x
  y `seq` send replyTo y

example :: ProcessM Int
example = do
  me <- self
  n  <- node
  n' <- (head . filter (/= n)) <$> nodes

  workerPid <- spawn n' worker

  let offset = 1
  send workerPid (Apply (\x -> x + offset) 41 me)

  expect
```

The message sent to `workerPid` contains a function, and that function captures `offset`. In Cloud Haskell, this is exactly the case that requires static closures and explicit environment serialisation. In CloudMicroHaskell, the function is just part of the message graph. The receiver expects an `Apply` message, extracts the function, and applies it.

The rest of the library follows the familiar process-programming structure. Nodes can be connected and queried; processes have identifiers and mailboxes; selective receive inspects messages by runtime type; monitors and exits make process failure observable; and a node-local registry lets long-running processes be found by name. These operations are not the novelty of CloudMicroHaskell, but they are necessary to make the core `spawn` and `send` design usable in larger programs. Their full API is listed in appendix A.

2.1 Process Behaviours as Libraries

The examples above use `spawn` and `send` directly. To test whether this style scales beyond our toy examples, we implement two familiar Erlang/OTP behaviours as ordinary CloudMicroHaskell libraries, generic servers and supervisors. These behaviours are not built into the runtime, but are implemented using the same process creation, message passing, monitors, and exits available to the programmer.

A generic server captures a common pattern whereby a process responds to requests and then enters a state where it is ready to respond to a subsequent request. Furthermore, the process may maintain a state which may be modified by a request, and the modified state must be used in subsequent requests. The application-specific part is described by a record of callbacks:

```
data ServerSpec st callReq castReq reply = ServerSpec
  { setup     :: ProcessM st
```

```
    , handleCall :: st -> callReq -> ProcessM (st, reply)
    , handleCast :: st -> castReq -> ProcessM st
    , tearDown   :: st -> ProcessM ()
    }

startServer :: NodeId -> ServerSpec st callReq castReq reply
            -> ProcessM (Server callReq castReq reply)
-- run the server in the *current* process
runServer :: ServerSpec st callReq castReq reply -> ProcessM ()

call :: Server callReq castReq reply -> callReq -> ProcessM reply
cast :: Server callReq castReq reply -> castReq -> ProcessM ()
```

`startServer` can start the server on a remote node, even though the server specification contains ordinary Haskell functions. Internally, a server loop is spawned together with the supplied callbacks. The programmer only implements the state transition logic, and the library handles the receive loop, reply routing, and state threading.

For example, a counter server can be described by giving its initial state and two handlers

```
data CallRequest = Get
data CastRequest = Inc | Reset

counterSpec :: ServerSpec Int CallRequest CastRequest Int
counterSpec = ServerSpec
  { setup     = return 0
  , handleCall = \st Get -> return (st, st)
  , handleCast = \st req ->
      case req of
        Inc   -> return (st + 1)
        Reset -> return 0
  , tearDown  = \_ -> return ()
  }
```

A client can then start the server remotely and interact with it through a typed handle:

```
do counter <- startServer n counterSpec
   cast counter Inc
   cast counter Inc
   i <- call counter Get
   liftIO $ print i
```

Supervisors capture a second common pattern, where a supervisor process keeps a set of child processes alive according to a restart policy. Again, the implementation is library code. A supervisor starts its children using `spawn`, monitors them, and reacts to `ProcessDied` messages by consulting its policy.

```
data ChildSpec = ChildSpec
  { childName   :: String
  , childStart  :: ProcessM ()
  , childNode   :: Maybe NodeId
  , childRestart :: Restart
  }
```

```haskell

data Restart = Permanent | Transient | Temporary
data Strategy = OneForOne | OneForAll

data SupervisorSpec = SupervisorSpec
  { strategy :: Strategy
  , intensity :: Int
  , period   :: Int
  , children :: [ChildSpec]
  }

supervise :: SupervisorSpec -> ProcessM Pid
getChild :: Pid -> String -> ProcessM (Maybe Pid)
```

The field `childStart` is an ordinary `ProcessM ()`. It may be a generic server, another supervisor, or an application-specific process. If `childNode` names a remote node, the child is spawned there. If the child dies and the restart policy says it should be restarted, the supervisor simply spawns the same computation again.

For instance, a supervised remote counter server is just a child whose start action runs the generic server

```haskell
do let child = ChildSpec
         { childName   = "counter"
         , childStart  = runServer counterSpec
         , childNode   = Just n
         , childRestart = Permanent
         }

   sup <- supervise SupervisorSpec
     { strategy = OneForOne
     , intensity = 3
     , period   = 5000
     , children = [child]
     }

   Just pid <- getChild sup "counter"
   let counter = serverFromPid pid
   cast counter Inc
```

This is the role of the Erlang-style parts of CloudMicroHaskell in this paper. They are evidence that the direct `spawn` and `send` interface is expressive enough to build structured distributed programs, while keeping the central tradeoff visible: process behaviour is easy to move because it is represented as a runtime graph, and the safety of that movement is correspondingly checked at runtime.

3 Implementation

3.1 MicroHs & Graph Reduction

The Haskell implementation used in this paper is MicroHs [2]. What matters for our purposes is not MicroHs itself, but the runtime support it provides: the serialisation primitives that make the

framework work. The design of the framework is not tied to MicroHs, provided an alternative runtime exposes equivalent graph serialisation primitives.

Serialisation & Deserialisation. MicroHs uses graph reduction with combinators [12]. This means that a running program is represented by a graph in memory that is continuously rewritten. The leaf nodes in the graph are the combinators and the interior nodes are application nodes. There is sharing (which is necessary for lazy evaluation) and cycles (because of recursion). To serialise an arbitrary value, you simply have to traverse the sub-graph reachable from the starting node, *i.e.*, the value to serialise.

What can be serialised? In general, pure computations and values can be serialised without any problem. If something cannot be serialised, the runtime will throw an exception. Here is a more detailed breakdown of what works and what does not.

Can be serialised:

- application nodes
- primitive operations, *i.e.*, combinators, basic arithmetic, etc.
- and by extension, any Haskell data type
- basic values, *i.e.*, fixed-size integers, floating point, etc.
- byte strings
- boxed arrays
- `IORef`

Cannot be serialised:

- pointers to memory allocated with `malloc()`
- foreign pointers
- C function pointers
- `MVar` values, which could be serialised, but are unlikely to work since threads are not serialised
- thread IDs, which represent threads; we have chosen not to serialise these

Serialisation API. The underlying API exposed by the runtime system to serialise and deserialise values is simple

```
1  serialize   :: a -> IO ByteString
2  deserialize :: ByteString -> IO a
```

Taken in isolation, this API is unsafe. `deserialize` makes no attempt to verify that the deserialised graph represents a value of the requested type.

CloudMicroHaskell therefore treats these operations as low-level runtime primitives, not as the user-facing serialisation API. Their use is confined to protocol points where the runtime determines the expected shape of the graph. For remote spawn, the payload is deserialised as the process function type expected by the node runtime. For message delivery, a remote message is stored together with a `TypeRep`, and selective receive compares this representation before deserialising the message payload.

These checks do not make graph deserialisation statically type safe. They rely on compatible nodes following the CloudMicroHaskell protocol and agreeing on the runtime graph format and type representations. We return to richer boundary descriptors, such as a `TypeDescribable` constraint, in the Discussion.

Laziness. Any graph can be serialised, regardless of how evaluated it is. Thus unevaluated "thunks" can be serialised. This is a feature, but it is also often not what you want. It is easy to accidentally serialise a graph that was actually intended to be evaluated first. For data types, it is

possible to control evaluation using `rnf` from the `NFData` class. For functions, the situation is more delicate. A function can only be forced to weak head normal form, which exposes a closure but does not force the values captured in its environment, nor any computation under the lambda. Sending functions therefore preserves the direct style, but it does not remove the programmer's responsibility to understand what the closure reaches. In practice, this means that values captured by mobile functions should be forced before the function is sent when their evaluation location matters.

Details of serialisation. The serialisation is implemented as a two-pass algorithm. The first pass identifies shared nodes via a depth-first traversal, whereas the second pass turns the graph into a byte string. The information from the first pass is used to guide the second pass on where to reference previously serialised nodes rather than serialising them again. Below follows an outline of the steps taken by the serialiser algorithm

- First, two bit arrays are allocated, called *visited* and *shared*, with as many elements as there are nodes in the heap. These arrays are used as mark bits during the two passes. We use two arrays rather than reserving bits in the nodes themselves for two main reasons: fewer writes to memory, and the possibility of concurrent traversals.
- When the first pass traverses the graph and sees a node, a previously unvisited node will have its *visited* bit set, whereas an already visited node will have its *shared* bit set. Any outgoing edges from the node are then recursively visited.
- The second pass flattens the graph using a post-order traversal. If a node is both *shared* and *visited*, we unset its *visited* bit, serialise the subparts, emit that node's textual representation, and finally emit a fresh label for that node. If a node is *shared* and not *visited*, it has already been serialised. In that case we emit the label that was associated with the node when it was previously serialised. If a node is not *shared*, it is serialised.
- Finally, the two temporary bit arrays are freed.

Labels can be represented in different ways. The easiest representation is the address of the node, or perhaps its offset into the heap. The representation doesn't matter as long as labels can be generated to be unique for each node.

Details of deserialisation. Deserialisation is the inverse operation of serialisation. The textual format from the serialiser is parsed, and an equivalent graph, preserving sharing and cycles, is reconstructed. Below follows an outline of the steps taken by the deserialiser algorithm

- First, we allocate an array that serves as a hash table, where labels are associated to node addresses.
- When a node is parsed, we check whether it is a label *reference* or not. If it is, we look up the label in the hash table. The parser might see a label reference before the label *definition*. In that case we allocate a placeholder indirection node and later update it to point at the parsed definition, and return the address of the indirection node.
- After a node has been parsed, if we see a label *definition*, we associate the label with the address of the just parsed node in the hash table.
- Finally, we free the memory of the hash table and return the last parsed node as the result of deserialisation.

Format. The serialisation format is textual. That makes it easier to read and debug, but it also avoids using fixed-width data for the basic types.

The format is portable across architectures in the sense that it does not expose machine byte order, pointer values, or binary code layout. Portability depends on the sender and receiver using compatible runtime versions and on primitive values being representable on the receiving runtime.

In our current system, `Int` is an architecture-dependent primitive. If a 64-bit runtime produces and serialises a value of type `Int`, the receiving runtime might not be able to represent the value if it is a 32-bit runtime. Cross-architecture protocols that must rely on numeric ranges should use fixed-width types. An optional compression/decompression pass can also be used to reduce the size of the serialised data.

Example. Consider first the function `f` below

```
f :: Int -> Int
f x = x * x + 1
```

The serialised combinator representation of this function is

```
v8.4
0
C' + @ S * @ I @@ #1 @
```

Here, `v8.4` is the format version, and `0` is the number of labels used. This is followed by some combinators, with `@` being postfix application. Another example, illustrating a recursive function, is shown below

```
fac :: Int -> Int
fac n = if n==0 then 1 else n*fac(n-1)
```

The serialised representation of this function is

```
v8.4
1
C S C == @ #0 @@ S * @ B _49382130 @ C - @ #1 @@@@@#1 @ :49382130
```

Here, `_49382130` is a label reference and `:49382130` is a label definition (because the function is recursive).

Figure 1 visualises the two serialised examples above. The purpose of the figure is simply to relate the flat textual format to the runtime object it describes: each `@` becomes an application node, while the other tokens become leaves in the graph. In the factorial example, the label reference and label definition correspond to the recursive edge in the graph. This graph, rather than a source-level closure name, is what CloudMicroHaskell serialises when values or process bodies cross node boundaries.

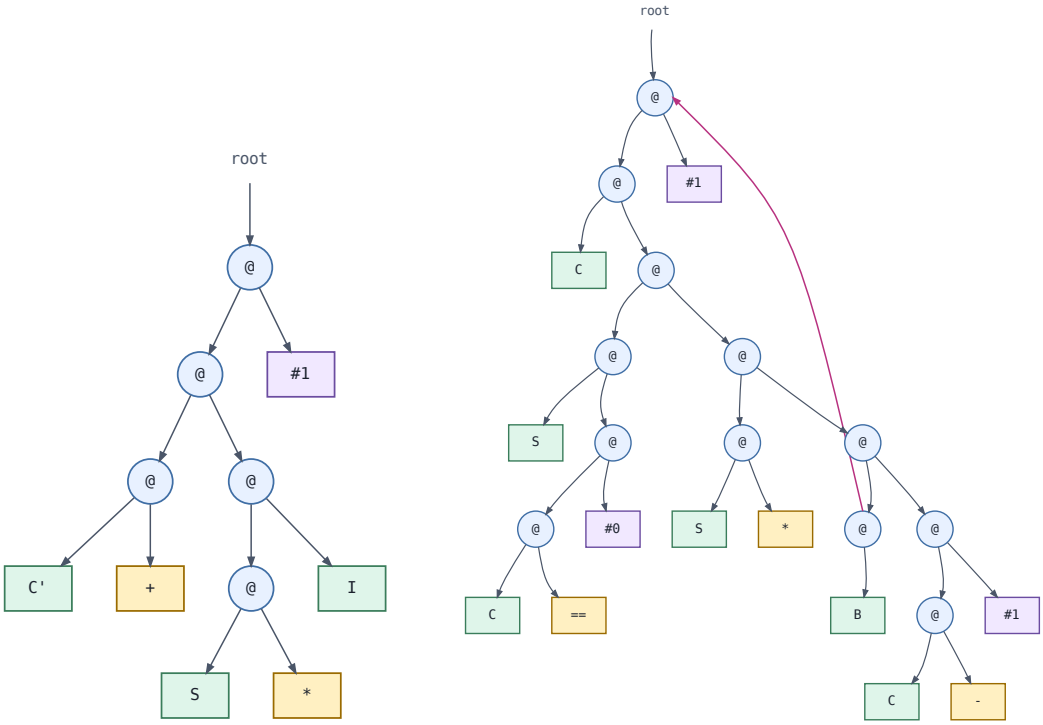
Having described the graph serialiser itself, we now describe where the distributed runtime invokes it: first in the node command loop, then in remote spawn and message delivery.

3.2 Node Runtime Architecture

A CloudMicroHaskell program always runs a *node*. The node is invoked from `main` with a `NodeConfig`, a structure holding the node's identity (an IP address and port number). The Haskell main thread creates an initial *node state*, spawns a *TCP listener* thread that awaits incoming connection requests, and then enters the main node loop. After this initial setup, the node continuously awaits commands from either a remote peer over the network, or from a process running on the same machine.

This gives the runtime a simple shape, where processes and peer-reader threads typically do not manipulate the distributed runtime directly. Instead, they communicate with the node loop by submitting commands. The rest of this subsection explains that boundary: which requests can arrive at a node, where they come from, and how the node turns them into local runtime effects.

We first conclude that the node clearly needs to be able to process commands. Examples of commands could be *install this monitor*, *send this message*, *connect to this remote node*, to name just a few. Some of the commands the node needs to process come from processes that are running on



(a) A non-recursive value.

(b) A recursive value.

Fig. 1. Combinator graphs corresponding to the two serialised examples. Application nodes are drawn as circles; combinators, primitive operations, and literals are drawn as leaves. The recursive example requires a label in the textual serialisation because the graph contains a cycle. The figure shows multiple copies of primitive nodes for clarity; in the runtime they are allocated once and shared.

the node in question, whereas some commands will be sent from remote nodes over the network. We model the commands without much flair

```haskell
-- | A command for the node to handle
data NodeCommand
  = Local Command        -- ^ A command from a process on this node
  | NonLocal PeerCommand -- ^ A command sent from another node

-- | Commands that a local process sends to the node
data Command where
  Connect :: SockAddr -> Maybe (MVar ()) -> Command
  Deliver :: Pid -> Mail -> Command
  -- more

-- | A command that a remote node has asked the node to execute
data PeerCommand
  = PeerConnect SockAddr Socket [SockAddr]
  | PeerDeliver Pid Mail
```

```
    -- more
```

The node processes commands as they arrive in an `cmdQueue :: CommandQueue` maintained in its node state. The `CommandQueue` is, internally, a double-ended queue coupled with an `MVar` used to signal when an empty queue has been written to.

```
data Queue a = Queue [a] [a]
data CommandQueue a = CommandQueue (MVar (Queue a)) (MVar ())

newCommandQueue :: IO (CommandQueue a)
enqueueCommand :: CommandQueue a -> a -> IO ()
takeCommand :: CommandQueue a -> IO a
```

A more robust implementation might assign different priorities to commands, but our implementation treats them all as equally important.

A local process will construct a `Local Command` and write it to the command queue, whereas a remote command will be received by a green thread monitoring a socket, as a `NonLocal PeerCommand`. While a command is travelling over the network, it is represented as yet another type of command, `WireCommand`.

```
data WireCommand
  = WireConnect SockAddr [SockAddr]
  | WireDeliver Pid Int Int
  -- more
```

During transit, a command will have been turned primarily into a buffer of bytes. In the example constructors above, the `WireDeliver` variant carries a process identifier, and two sizes, denoting the length of the two following data packets (a serialised `TypeRep`, and the `Mail` itself). The TCP listener will convert a `WireCommand` to a `PeerCommand` by e.g. reading potential buffers and reconstructing values that have been serialised.

The node processes commands in the order in which they arrive. A figure that shows how the different types of commands flow through the node application is shown in fig. 2. The command interface is what lets the node remain the owner of the runtime state. Mailboxes, monitors, connections, pending receives, and the name registry are not exposed to processes directly, but are rather updated only by the node loop while handling commands. Instead of showing the entire node state definition at once, we will introduce the relevant pieces as they become necessary.

3.3 The ProcessM Monad

The `ProcessM` monad is a central part of CloudMicroHaskell, as it models a *process*. Conceptually, a process is an independent entity that performs computations regardless of what other processes do, and communicates with other processes only via message passing. It is modelled as a function that takes the node state and its own pid, and then produces an `IO` action.

```
data ProcessM a = ProcessM { unProcessM :: NodeState -> Pid -> IO a }
```

The process needs to know about the node state as the node state supplies, most notably, the `MVar` through which commands are sent. A common pattern in `ProcessM` primitives that needs to wait for a result from the node is that they allocate a fresh `MVar`, send its command to the node (which includes the `MVar`), and then block until the result is there. An example is shown below

```
-- | Register a Pid under a name
register :: Pid -> String -> ProcessM ()
register pid name = ProcessM $ \ns _ -> do
```

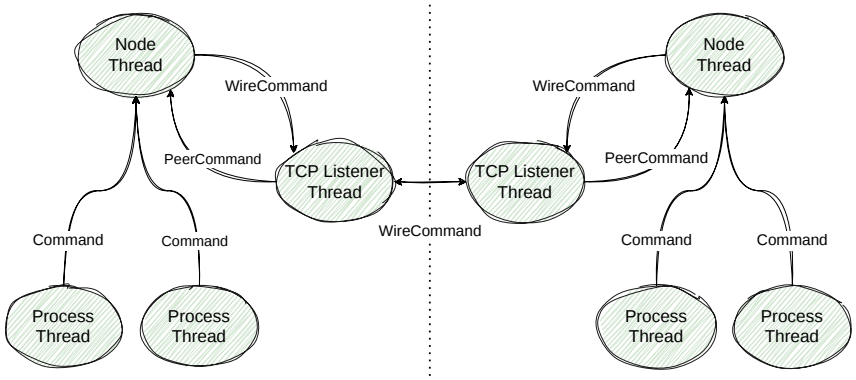


Fig. 2. The figure illustrates how the different types of commands flow through the node application. Every node in this diagram is a Haskell green thread, and the dotted line represents the network boundary. Process threads will send the node ordinary `Command` values, whereas the node thread needs to convert such a command to an appropriate `WireCommand` before the TCP listener thread can send it to a remote node. A TCP listener converts a received `WireCommand` to a `PeerCommand` before sending it to the node thread.

```
    -- create a fresh MVar to block until the node has registered the name
    mv <- newEmptyMVar

    -- send the command to the node
    enqueueCommand (cmdQueue ns) (Local (Register pid name mv))

    -- wait for the node to write to the MVar, indicating that it is done
    takeMVar mv
```

Additionally, functions that return a result fetched from the node state must force it. Consider the function `node` that returns the node a process is currently executing on.

```
node :: ProcessM NodeId
node = ProcessM $ \ns _ ->
  let nd = NodeId $ nodeAddr ns
  in rnf nd `seq` return nd
```

If we did not force the evaluation of the `NodeId`, there is a high risk that the result from `node` would be an unevaluated thunk. We illustrate this problem with the code snippet below, which shows a process that spawns another process that references the fetched `NodeId`.

```
do n <- node
   ns <- filter ((/=) n) <$> nodes
   mapM_ (\remoteNode -> spawn remoteNode (spawn n (liftIO $ putStrLn "hi"))) ns
```

The above program fetches the current process's node, and then spawns a process on every other remote node. That process in turn spawns a process on the original node again, by referencing the free variable `n`. If `n` was an unevaluated thunk that references the node state, the outer call to `spawn` would not succeed, as the serialiser would follow the graph into the node state and try to serialise its inner `MVar`s. By ensuring the result is forced, we know that the node state is no longer referenced by `n` and serialising proceeds as expected.

This is also where the conceptual process and the runtime bookkeeping part ways. A `ProcessM` computation is given a `Pid`, but the resources associated with that pid are owned by the node. The process has a mailbox, may have pending receives, may be registered under a name, and may be monitored by other processes. However, none of these are fields of the process itself. They are entries in the node state, and the process can affect them only by sending commands to the node.

This distinction is important for distribution. When a process is spawned remotely, we do not serialise its mailbox, monitors, or other local runtime resources. We serialise only the computation to run. The receiving node allocates a fresh pid, creates the corresponding local runtime entries, and then runs the computation with its own node state.

3.4 Remote Spawn

The central contribution of CloudMicroHaskell is that `spawn` can take an ordinary `ProcessM` computation and run it on another node. We now describe how a call to `spawn` moves through the node runtime, crosses the network, and results in a fresh green thread running the supplied computation on a remote machine. The example program we begin with is shown below

```
do n <- node
   anotherNode <- (head . filter ((/=) n)) <$> nodes

   let x = 7

   _ <- spawn anotherNode $ do
         liftIO $ putStrLn $ show x
         let x = 10
         liftIO $ putStrLn $ show x

   return ()
```

We note that the spawned process references the outer `x`, a free variable.

The implementation of `spawn` has a "fast path" for node-local spawns, to achieve better performance. The rest of this section, however, describes how a remote spawn is implemented (the "slow path"). The fast path differs from the slow path in that the running process spawns the new process itself, bypassing the node loop/command queue. Additionally, the process body of a locally spawned process is not serialised, as the process body does not need to traverse the network. The slow path presented below describes a superset of the actions that the fast path takes.

As `spawn` returns the `Pid` of the spawned process, it blocks until the remote node has created the process and acknowledged its `Pid`. A fresh `MVar` is allocated where the node will write the `Pid` once it receives it together with a spawn confirmation from the remote node. Additionally, a unique token is generated and paired with the spawn request, so that an acknowledgement of another spawn request is not confused with the current request. The result `MVar` is associated with the unique token in a mutable map reachable from the node state.

```
spawn :: NodeId -> ProcessM () -> ProcessM Pid
spawn (NodeId targetAddr) action = ProcessM $ \ns _myPid -> do
  -- generate token and create MVar, to record in the node state
  reqId   <- newUniqueToken
  replyVar <- newEmptyMVar
  modifyIORef' (spawnRequests ns) (Map.insert reqId replyVar)

  -- send the spawn command to the node loop
```

```
  enqueueCommand (cmdQueue ns) (SpawnOn targetAddr reqId (unProcessM action))

  -- block here until the node loop publishes the process identifier of the
  -- spawned process
  pid <- takeMVar replyVar
  modifyIORef' (spawnRequests ns) (Map.delete reqId)

  return pid
```

When the node processes the `SpawnOn` command, it fetches the socket associated with the remote node and initiates the network communication.

```
-- this function pattern matches on many commands, but we just show SpawnOn here
handleLocalCommand :: NodeState -> Command -> IO ()
handleLocalCommand ns (SpawnOn targetAddr reqId f) = do
  c <- getConn ns targetAddr
  sendSpawn c (nodeAddr ns) reqId f

-- remember that the ProcessM monad wrapped a function of type
-- NodeState -> Pid -> IO a
sendSpawn :: Socket -> SockAddr -> Int -> (NodeState -> Pid -> IO ()) -> IO ()
sendSpawn conn spawnerAddr reqId f = do
  (buf, len) <- enc f -- serialise the ProcessM function

  -- send first the serialised WireSpawn command, and then the serialised graph
  sendLine conn (show (WireSpawn spawnerAddr reqId len))
  sendAll conn buf len

  free buf
```

Here is where we invoke our magic serialisation function. The serialised graph will reference everything that was reachable from the expression, including the outer free variable. This is now a serialised function that contains both its code and environment.

After serialisation, we get a pointer to a memory buffer holding our ready-to-send bytes. The `WireCommand` that we send to the remote node includes the unique token that we have associated with the spawn request, as well as the length of the following data buffer. The remote node's TCP listener will first receive the `WireCommand`, look at it to figure out how many bytes to read for the subsequent graph, and then decode it when it turns it into a `PeerCommand`. We omit most of the listener code, but show how the received `WireSpawn` message is treated

```
handleWireCommand (WireSpawn spawnerAddr reqId payloadLen) = do
  buf <- mallocBytes payloadLen
  ok  <- recvExact conn buf payloadLen

  if not ok
    then do
      -- error handling code, omitted for brevity
    else do
      f <- dec buf payloadLen :: IO (NodeState -> Pid -> IO ())
      free buf
```

```
11          enqueueCommand (cmdQueue ns) (NonLocal (PeerSpawn spawnerAddr reqId f))
```

After the `WireSpawn` command has already been parsed, we know that we should read a subsequent buffer of data, the graph. We allocate a buffer of the appropriate length and read exactly that many bytes into it. We then invoke our magic deserialiser, which reconstructs the graph on the heap of the remote CloudMicroHaskell node. Here we annotate the deserialiser with the type of the function that `ProcessM` wraps. There is no verification that the parsed graph represents a value of that specific type, so we must be very careful to properly line up the types of the values we serialise, and the values we deserialise. We then enqueue the `PeerSpawn` command with the node loop.

The node loop on the remote node must invoke its `PeerCommand` handler to deal with the command. It starts the local process, and sends the `Pid` together with the unique token back to the original node. We omit the code for `startLocalProcess`, but mention that it uses `forkIO` to spawn a green thread that runs the `ProcessM` function we just deserialised. The process's mailbox and associated metadata are then added to the node state.

```
handleNonLocalCommand ns (PeerSpawn spawnerAddr reqId f) = do
  pid <- startLocalProcess ns f
  c   <- getConn ns spawnerAddr
  sendLine c (show (WireSpawnAck reqId pid))
```

At the wire level the original node receives the `WireSpawnAck`, which the listener promptly converts into a `PeerSpawnAck`. The spawn request is then completed by invoking the `completeSpawnRequest` function that was briefly shown in the code that described how the node handled the initial spawn request. `completeSpawnRequest` publishes the `Pid` into the `MVar` that is associated with the unique token. This unblocks the process that initiated the spawn request, which now has access to the process identifier of the new process. This concludes the handling of `spawn`.

3.5 Message Passing

The underlying mechanism for transmitting messages between communicating processes is the same. There is a fast and slow path, differing in exactly the same way that the fast and slow path did for `spawn`. As values are, like code, just subgraphs in the combinator graph, the serialiser and deserialiser work here as well. If a message has to traverse the network it will be encoded together with a runtime representation of its type. If a message does not have to traverse the network, and is being sent to a process running on the same node, the message is passed by its pointer into the graph. We capture these two kinds of messages in the following data type

```
data Mail where
  HeapMail    :: a -> Mail
  EncodedMail :: TypeRep -> Ptr Word8 -> Int -> Mail
```

We use `TypeRep` from the `Typeable` type class to observe the types of values at runtime. Both GHC and MicroHs derive `Typeable` instances for every type, so the programmer does not need to implement them.

Every process has a mailbox where its messages go, but the mailbox is owned by the node loop. In the fast path, the message will be delivered directly to the receiving process, whereas in the slow path, the action of sending the message will be delegated to the node. We omit the code for `deliverMailLocal`, which is responsible for doing the actions just described. The code for `send` itself is fairly brief

```
send :: Identifiable process => process -> a -> ProcessM ()
send targetPid x = ProcessM $ \ns _ -> do
  targetPid' <- toPid targetPid ns
```

```
    let Pid (NodeId targetAddr) _ = targetPid'
    if targetAddr == nodeAddr ns
      then deliverMailLocal Nothing ns targetPid' (HeapMail x)
      else
        enqueueCommand (cmdQueue ns) (Local (Deliver targetPid' (HeapMail x)))
```

There are two types that are `Identifiable`, `Pid` and `String`. `toPid` does a lookup in an `IORef` fetched from the node state if the instantiated type is `String`.

When the node handles the remote send, it will fetch the associated socket, encode both the message and the `TypeRep`, and then send everything through the socket.

```
handleLocalCommand ns (Deliver targetPid mail) = do
  let Pid (NodeId targetAddr) _ = targetPid
  deliverMailRemote ns targetPid mail

deliverMailRemote :: NodeState -> Pid -> Mail -> IO ()
deliverMailRemote ns targetPid (HeapMail x) = do
  let Pid (NodeId targetAddr) _ = targetPid
  c <- getConn ns targetAddr

  (valueBuf, valueLen) <- enc x
  (trBuf, trLen)       <- encodeTypeRep (typeOf x)

  sendDeliver c targetPid trLen trBuf valueLen valueBuf

  free valueBuf
  free trBuf
deliverMailRemote _ _ (EncodedMail _ _ _) = -- should not happen
```

When a message is delivered, either to a local process or a remote one, the thread responsible for doing the delivery will check whether the receiving process is already blocked waiting for a message. If the receiving process is already blocked waiting for a message, it will be delivered immediately if the message can be delivered, and only placed in the mailbox if not.

Forcing values before sending. Something to keep in mind when sending messages between processes is that messages will be sent as they are, then and there. The serialiser walks the combinator graph, and if a value is still an unevaluated thunk, the unevaluated thunk will be serialised and transmitted. This might be the intended behaviour, but it will most likely be the intent of the programmer that an evaluated value should be sent. The problems with sending unintended unevaluated thunks are twofold; firstly, a seemingly small expression might make a large part of the sender's heap be serialised and sent to the remote node. Secondly, if we are using processes and message passing to achieve speedups by distributing our computation over a network, work might not be performed on those nodes where the programmer expects it to be done. We note that in Cloud Haskell, this concern does not arise. The `Binary` instance required by Cloud Haskell's `send` requires a value to be evaluated as it is serialised.

To give an example of misplacement of work, consider `countChunk` below. It takes a list of numbers as input and sends a message to the process identifier indicating how many of those numbers were prime. Since `n` is not forced, its thunk will point to `xs`, `filter`, `length`, and `isPrime`. Serialising `n` will serialise those functions as well, and when `n` is required at the parent, the work will be done there.

```
countChunk :: Pid -> [Int] -> ProcessM ()
```

```
countChunk parent xs = do
  let n = length (filter isPrime xs)
  send parent n
```

The fix is to ensure that `n` is forced before it is given to `send`.

```
countChunkStrict :: Pid -> [Int] -> ProcessM ()
countChunkStrict parent xs = do
  let n = length (filter isPrime xs)
  n `seq` send parent n
```

In this case we can use `seq` to force evaluation, since `n` is an `Int`, but more structured data would require `rnf` or `deepseq`.

This is the same discipline familiar to programmers of parallel Haskell. A spark created by `par` only buys parallelism if the sparked expression is itself evaluated by another core. In CloudMicroHaskell, the analogous failure mode is not lost parallelism, but misplaced computation, and possibly a much larger message than the programmer intended. The remedy is the same: ensure the message is in normal form before handing it to `send`, using `seq`, bang patterns, or a deep-force such as `deepseq`.

3.6 Failure Detection and Exit Propagation

The fault-tolerance primitives in CloudMicroHaskell are built around process death detection. The runtime does not try to recover a failed process by itself, but instead makes failure observable to other processes. Application-level code, such as supervisors, can be programmed to decide whether a process should restart or not.

One process can observe the death of another by, as described previously, installing a monitor. As the node owns all the state, it maintains a table which maps each watched process to the processes that watch it, and the monitor action each watcher has requested.

```
monitors :: IORef (Map.Map Pid [(Pid, MonitorAction)])
```

Unsurprisingly, a monitor is registered by sending a command to the node loop. If the process that should be watched is local, the node records the monitor in its `monitors` table. Instead, if the process lives on another node, the request is sent to the remote node as a `WireMonitor` command. The remote node will record the monitor in its own `monitors` table. The short summary is that a node is responsible for detecting process death and recording which watchers must be notified.

Every CloudMicroHaskell process is implemented by a Haskell green thread. When a process is started, the runtime forks the thread with a finaliser. This finaliser runs regardless of whether the process returns normally, calls `terminate`, dies because of an exception, or is killed by another process via `exit`.

```
startLocalProcess :: NodeState -> String -> (NodeState -> Pid -> IO ()) -> IO Pid
startLocalProcess ns label f = do
  pid <- allocatePid ns
  tid <- forkFinally (f ns pid) $ \res -> do
    let reason = reasonFromResult res
    cleanupProcess ns pid
    notifyMonitors ns pid reason
  modifyIORef' (threadIds ns) (Map.insert pid tid)
  return pid
```

The finaliser turns the result of the thread into an `ExitReason`. Normal return and `terminate` become `ExitNormal`, whereas termination by `exit` preserves the `ExitReason` supplied by the caller.

An uncaught exception becomes `ExitOther`. The runtime will then remove the process-local state such as its mailbox, pending receives, registered name, and thread identifier, before finally notifying the monitors.

Notification is also routed through the node loop. For each registered watcher the node loop checks whether it is a local process or not. A local watcher is notified immediately, whereas a remote watcher is notified by sending a `WireProcessDied` command to the watcher's node.

```
applyMonitorAction ns watcherPid watcheePid TrapExit reason =
  deliverMailLocal ns watcherPid (HeapMail (ProcessDied watcheePid reason))

applyMonitorAction ns watcherPid watcheePid Succumb _ = do
  tids <- readIORef (threadIds ns)
  case Map.lookup watcherPid tids of
    Just tid -> throwTo tid (MonitoredProcessDied watcheePid)
    Nothing -> return ()
```

A monitor registered with `TrapExit` will receive the notification via a `ProcessDied` message in its mailbox. The monitor process can then observe the termination and decide what to do. Supervisor processes use this behaviour to figure out whether a process should be restarted or not. The `Succumb` monitor action will make the monitor receive an asynchronous exception, making it die together with the watched process. This is useful to tie together the lifetimes of processes that are meant to exist at the same time.

Node disconnection is treated as process failure. If an open socket is closed the surviving nodes will assume that any monitored process on that node has terminated. The surviving nodes will consider any monitored process as terminated with an `ExitOther` reason. This makes the running processes able to detect network failure as if it were ordinary process failure.

Explicit `exit` signals use the same path. If the target process is running on the current node, the node raises a `ProcessExit` exception in its thread, whereas if the process is remote, the node instead sends a `WireExit` command to the node that owns the process. In either case, the target process eventually runs the same finaliser.

4 Evaluation

We evaluate CloudMicroHaskell as a prototype implementation of a design point, rather than as a mature high-performance actor runtime. The evaluation therefore asks four questions. First, what are the basic costs of process creation, message passing, and runtime graph serialisation? Second, can a distributed CloudMicroHaskell program obtain speedup on a parallel workload despite those costs? Third, is the programming model expressive enough to structure a nontrivial distributed application? Finally, does the small MicroHs runtime make the same model usable on platforms outside the reach of a conventional GHC-based system?

The ring benchmark addresses the first question, while the distributed work-pool benchmark addresses the second. The MicroSync case study addresses expressiveness by exercising named processes, dynamic process creation, generic servers, supervisors, monitors, and cross-node messaging in one application. The heterogeneous-node case study addresses portability by running CloudMicroHaskell nodes across microcontrollers and an ordinary laptop.

4.1 Benchmarks

Ring Benchmark. We take inspiration from Erlang with our first evaluation. A classic Erlang example is to spawn a *ring* of processes, where a message is sent from one process to another, through the entire ring, until the message arrives again at the original process. We use the ring

of processes to measure how long it takes, on average, to spawn a single worker, and how long it takes to send a message from one process to another. We build a ring of 3000 processes.

All processes in the ring are spawned on the same node. Node-local `spawn` and `send` have fast-paths whereby the actions are performed directly by the process rather than the node, avoiding the need to hand over to the node thread. We can, however, instrument CloudMicroHaskell to take the long path through the node loop, simulating what would happen if the `spawn` or `send` targeted another node. Additionally, even for node-local `spawn` or `send`, we can force serialisation and deserialisation, to measure how long those operations take. We measure the performance both with and without serialisation enabled, to gain some insight into how much time encoding and decoding the graph takes. The serialised process is roughly 32 kilobytes of data, whereas the serialised message sent through the ring is roughly 300 bytes.

We run this experiment on a laptop with an Intel i5-8365U CPU, and 16 GB of RAM.

fig. 3 and fig. 4 presents the elapsed time when spawning a process on the same node, using either the fast path or the slow path. The fast path takes roughly 60 µs, whereas the slow path took, for this example, 550 µs. The elapsed time is reported as segments, to try and measure what precisely it is that is taking time. The reported segments are

- **dispatch**: The running process handing off to the thread responsible for doing the `spawn`. In the fast path this is roughly figuring out that the spawning process itself should do it, whereas in the slow path it is a context switch to the node thread.
- **register**: The time it takes to create and initialise process resources, such as mailboxes and the process identifier, and to register them in the node state.
- **fork**: The time it takes to actually spawn the green thread that will execute the new process. Additionally, the node state is updated to include the fresh thread ID of the spawned thread.
- **wakeup**: The time it takes to hand control back to the initial process, which is done by writing the fresh process identifier to an `MVar` that the initial process reads from.
- **enc**: The time the runtime takes to serialise the reachable subgraph representing the process body to be spawned.
- **dec**: The time it takes for the runtime to deserialise and reconstruct the graph from the serialised representation.

It takes CloudMicroHaskell 371,347 µs to send a message through the entire ring.

For comparison, a state of the art actor model concurrency system, Erlang, takes on average 1.27 µs to spawn a single process. Sending a message through a ring of 3000 processes takes on average 1456 µs.

Distributed Work Pool Benchmark. This benchmark measures whether adding CloudMicroHaskell nodes yields proportional speedup on an embarrassingly parallel workload. We count the primes in a fixed integer interval, dividing it into equally sized segments that a work pool distributes as units of work across remote workers. The work pool is implemented as a CloudMicroHaskell application, where a master process takes N units of work, spawns M worker processes, and distributes the units of work to the workers one at a time. A unit of work is modelled as a `ProcessM a` computation.

We distribute the workload over 6 remote Google Cloud E2-standard-2 machines, each equipped with 8 GB of RAM and 2 vCPUs. Each instance runs a CloudMicroHaskell node; these nodes are connected together. We run the workload using between one and six nodes, averaging over 10 results. The results can be seen in fig. 5.

4.2 Case Study 1: MicroSync

To illustrate the programming model beyond small code snippets and toy examples, we implement a file syncing application, MicroSync. The case study is modest in scope, but deliberately nontrivial.

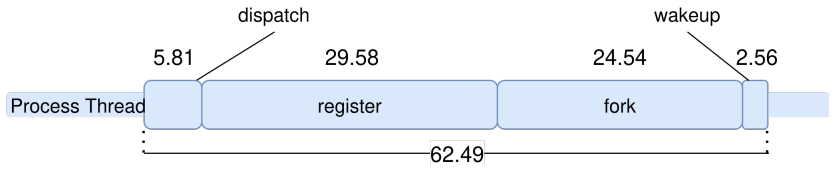


Fig. 3. Breakdown of node-local process spawn time on the fast path, where the spawning process creates the child directly without routing through the node loop.

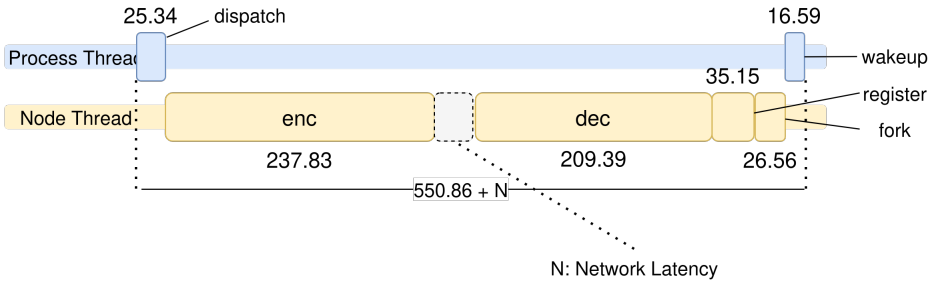


Fig. 4. Breakdown of process spawn time on the slow path, where the process body is serialised, routed through the node loop, deserialised, and then started locally. A remote spawn on comparable hardware would add the network round-trip and transmission time.

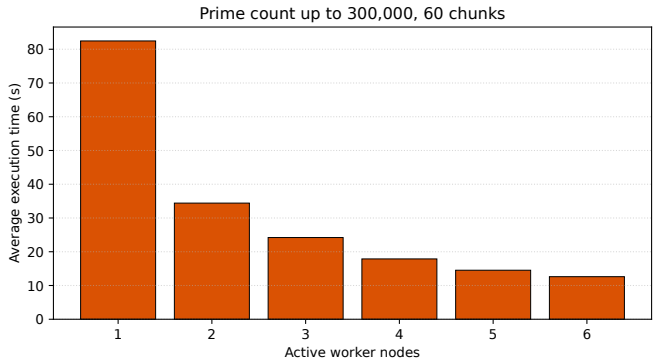


Fig. 5. Distributed work-pool benchmark for counting primes up to 300,000, split into 60 work units. Bars show mean execution time over 10 runs on one to six Google Cloud E2-standard-2 worker nodes; runtime drops from 82.5 s to 12.6 s, a 6.5× speedup over the one-node baseline.

Several machines work together to keep the contents of a directory in sync, discover each other, broadcast local edits, exchange metadata, exchange partial chunks of files, and recover from failing helper processes. The case study shows that this application is expressed fairly naturally as a collection of communicating processes.

We dive straight in, and illustrate how an edit of a local file on machine A is advertised and applied to a remote copy of the same file on machine B; fig. 6 shows the corresponding message flow.

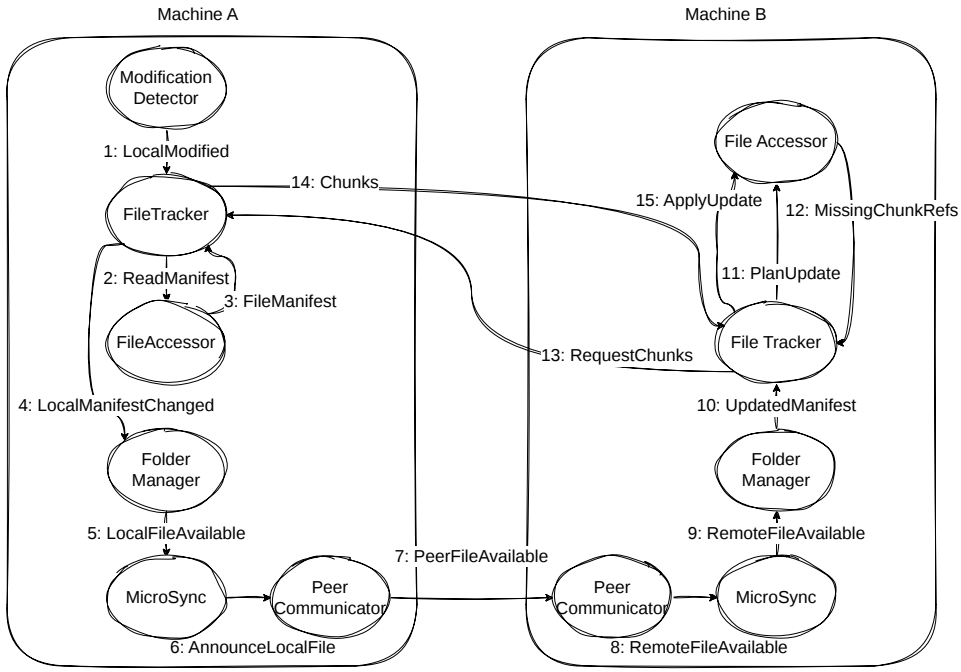


Fig. 6. Message flow for synchronising a local edit from machine A to machine B. The numbered arrows show the messages described in the walkthrough: the update is first advertised through the peer processes, after which the receiving file tracker requests missing chunks directly from the file tracker that advertised the version.

On machine A, a user modifies a local file. The *modify detector* process assigned to monitor that particular file detects this edit by observing the last modified timestamp, and reports this fact to its parent *file tracker* process. The file tracker process is MicroSync's designated owner of one tracked file. It receives the message from the modification detector that the file has likely been modified, and asks the *file accessor* for the file manifest. A file accessor process is the only process that is allowed to modify a file on the disk, and is implemented as a server.

A file manifest is a compact description of a file version. It records, among other things, the path to the file, the file size, and most importantly, a list of chunk references. Each chunk reference contains an offset into the file, the size of the chunk, and a hash of the chunk contents. MicroSync uses *Content-Defined Chunking* to divide a file into chunks. The chunks are chosen based on the contents of the file, rather than using fixed byte offsets. The effect of this is that a small edit tends to change only chunks close to where the edit happened, allowing a remote MicroSync instance to request only the chunks that it does not already have.

Once the file accessor has supplied the file manifest, the file tracker sends it to the *folder manager* process. The folder manager process is the designated owner of the entire folder being synchronised. For this local-modification case, its main role is to turn the tracker's manifest into a file-version announcement that is sent upwards, to the top-level *MicroSync* process. The top-level MicroSync process routes the announcement to its *peer communicator* process. The peer communicator process informs the remote peer communicator on machine B what machine A's version of the file looks like. On machine B, the information flows in a similar way from the peer to the MicroSync top-level

process, from there to the folder manager, which will in turn inform the responsible file tracker process that there exists a potentially new version of the file.

The file tracker on machine B will ask its file accessor for the local file manifest, figure out that the version on machine A is newer, and then compute which chunks of the file it should request from machine A. The broadcast message from machine A included the process identifier of the specific file tracker on machine A, so the equivalent file tracker on machine B will now request the missing chunks directly from the file tracker on machine A. This process identifier is only assumed valid for this one exchange, and is not retained on machine B for subsequent use. Once the chunks have been delivered, the file tracker on machine B instructs its file accessor to rebuild the file according to the new file manifest. It will supply the new chunks, and together with the chunks that already exist on machine B, the file is updated to have the same contents as the version on machine A.

fig. 6 shows two machines for clarity. However, in general, a MicroSync instance may be connected to several remote peers. The peer process broadcasts file-version announcements to all of them, and each receiving instance independently decides how to react to the announcement. The described application is built entirely from CloudMicroHaskell primitives such as named processes for discovery, message passing for routing, generic servers for stateful services, supervisors for recovery, etc. The following paragraph gives some details on the implementation, but naturally omits most of the application code.

The Process Tree. We have already touched upon several processes that are part of a MicroSync instance, but here we give a complete description of the different kinds of processes. A MicroSync instance is implemented as a process tree, with the top level process being the MicroSync process. It registers itself under the name `"microsync"` and starts a supervisor for the long-running services of the application. The supervisor uses a one-for-one strategy to permanently supervise four child processes. The child processes in question are the *Event Logger* process, the *Folder Manager* process, the *Peer Communication* process, and finally the *Peer Connector* process.

The event logger is a generic server, registered under the name `"eventlogger"`, whose service is to accept events and log them to a persistent file on disk. It gives the rest of the application a single named process which events can be sent to for logging.

The peer connector process discovers remote MicroSync instances by querying connected CloudMicroHaskell nodes for a registered `"peer"` process. This peer process is the control-plane process for remote MicroSync instances: it tracks connected peers, monitors them for disconnection, and forwards file-version announcements between machines.

The folder manager process owns the state of the synchronised folder. It manages the set of files being tracked at a point in time, and starts or stops file tracker processes as necessary. Additionally, it manages a folder scanner process, which scans the folder for new and deleted files, while each tracked file is represented by a dynamically added file tracker process.

A file tracker process is created dynamically for each tracked file. It owns the synchronisation protocol for that file and presents the rest of the application with a single process to address. The tracker itself is implemented as a generic server, registered under the file path, and it starts a private supervision tree for the helper processes associated with the file. Its modify detector reports suspected local changes, while its file accessor serialises all reads and writes to the underlying file.

fig. 7 illustrates how the processes are structured, and who supervises whom.

4.3 Case Study 2: Heterogeneous Nodes

The second case study exercises a different motivation for building CloudMicroHaskell on MicroHs: portability rather than performance. We run a CloudMicroHaskell node on each of two

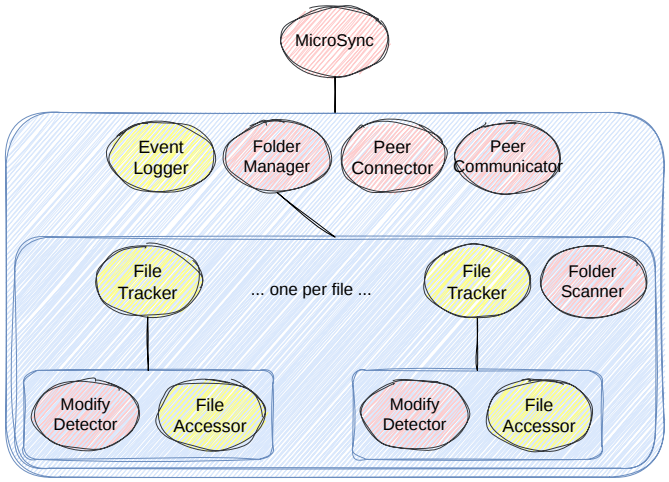


Fig. 7. Process tree of one MicroSync instance. Blue containers are supervisors, yellow processes are generic servers, and red processes are ordinary processes. File trackers are created dynamically, one per tracked file, and each tracker supervises its own modify detector and file accessor.

STM32H563ZI microcontrollers running Zephyr RTOS, and a third node on the laptop used for the ring benchmark. The microcontrollers have 2 MB of flash memory, 640 KB of SRAM, and run at up to 250 MHz. All three devices are connected by Ethernet through a network switch.

This setup is deliberately heterogeneous. The laptop node runs a 64-bit MicroHs runtime, whereas the microcontroller nodes run a 32-bit runtime built for Zephyr. The nodes are therefore not executing the same binary, but they do share a compatible CloudMicroHaskell protocol, graph format, and runtime representation. This exercises the claim that CloudMicroHaskell nodes need not agree on a common set of static symbols in the style of Cloud Haskell. Instead, they must agree on the runtime format used to serialise and deserialise graphs. The messages used in this case study contain primitive values that are representable on both the 64-bit and 32-bit runtimes. The case study therefore demonstrates portability of the runtime format and programming model across heterogeneous nodes, not that every possible value of every architecture-dependent primitive can be transported unchanged.

The application is small but end-to-end. One microcontroller runs a generic server that controls LEDs. The second microcontroller runs a process that reacts to button presses and sends requests to the LED server. The laptop runs a third CloudMicroHaskell node that accepts keyboard commands and sends the same kind of requests. Thus, both an embedded node and a laptop node communicate with a server process running on another embedded node, using the ordinary CloudMicroHaskell message-passing interface.

This case study is not a performance benchmark. Its purpose is to show that the implementation is not tied to GHC-sized machines or to a homogeneous cluster of identical binaries. The same process abstraction, message format, and generic-server library used in the earlier examples can also be deployed across resource-constrained devices.

5 Discussion

Making Serialisation Safer. The implementation of CloudMicroHaskell uses the unsafe primitives for serialisation and deserialisation. This should not be read as the authors claiming that static checks are undesirable. The goal of this project is to explore a programming model where the static checks by Cloud Haskell are turned into runtime checks.

A safer API can be built on top of the unsafe primitives, eliminating some of the risk. A natural refinement is to add a lightweight type-description constraint, for example

```
1 serializeHL  :: TypeDescribable a => a -> IO ByteString
2 deserializeHL :: TypeDescribable a => ByteString -> IO (Maybe a)
```

This constraint would not be used to define how to serialise a value. That task is still delegated to the runtime system. Instead, the constraint would be used to advertise the boundary type, so that a receiver can reject deserialisation of a serialised value where the boundary types don't match. Descriptors like this could be derived generically, for instance via `Data.Data`, and be made to include additional structural information. This is a more explicit version of the check CloudMicroHaskell already performs for messages: remote messages are sent together with a `TypeRep`, and selective receive compares that representation before deserialising the message payload.

This would sit between the solution of Cloud Haskell and CloudMicroHaskell. There is no need for manual closure conversion or for the programmer to write serialisers, but the communication boundary still becomes explicit in the source program. This solution would not rule out serialisation of runtime-owned resources such as `MVar`s or foreign pointers, but it would turn some type mismatches that are currently detected only at use sites into dynamic checks at deserialisation time. These safer functions would still enable the programmer to accidentally serialise thunks or unintentionally large graphs, however.

Benchmarks. We begin our discussion by observing that the slow path takes significantly longer, increasing the required time from 60 µs to some 550 µs. Of this, roughly 30 µs is added by going through the node loop, and roughly 450 µs comes from serialisation and deserialisation. This is not unexpected, but ideally the cost would be smaller. The serialiser is implemented via two passes to enable sharing of cycles. A one-pass serialiser would be faster, but would not be as capable. We would not be able to serialise recursive graphs, and shared sub-graphs would be replicated as many times as they were referenced, increasing the size of the reconstructed graph.

The serialised process closure occupies roughly 32 KB. The serialised graph is emitted in a textual ASCII format, which is not particularly compact. A new output format could drastically reduce the size of the serialised graph. There would perhaps be a slight gain in performance, but nothing extraordinary.

The results for the ring benchmark indicate that CloudMicroHaskell is significantly slower than Erlang. There are several reasons for this, but a notable one is that Erlang has been optimised for this very task over several decades, whereas CloudMicroHaskell is running on MicroHs, a non-optimising compiler in its infancy. Additionally, even if MicroHs were an optimising compiler, the underlying choice of reducing a mutable combinator graph at runtime will always suffer performance penalties. While not reported in this paper, the authors point out that MicroHs's runtime is generally around two orders of magnitude slower than GHC's.

The heterogeneous-node case study explains why this tradeoff may still be attractive. MicroHs is much slower than mature native-code systems, but its runtime is small and portable. In our prototype, the compiled runtime is roughly 300 KB, small enough to run on the STM32H563ZI microcontrollers used in the case study. The case study therefore supports a portability claim rather than a performance claim: CloudMicroHaskell makes it possible to use the same distributed

Haskell programming model across machines that would not normally be able to run a GHC-based distributed Haskell system.

The distributed prime counter shows significant speedups compared to sequential runtime. We point out that there seems to be a superlinear speedup when going from one to two remote machines, and we are not entirely sure why. The remote instances are separate instances, but it is impossible to say precisely where they are hosted and what else is hosted on the same hardware. While we cannot say for sure, we speculate that the phenomenon may be due to caching, and that using more than one MicroHs instance reduces the frequency with which the garbage collector has to run.

MicroSync. The purpose of MicroSync is not to evaluate file synchronisation performance, but to test whether the CloudMicroHaskell programming model is usable in a larger application. The case study exercises named processes, dynamic process creation, supervisors, monitors, generic servers, and cross-node message passing in one program. It therefore supports an expressiveness claim rather than a performance claim, namely that CloudMicroHaskell is sufficient to structure a nontrivial distributed application while preserving the direct style illustrated in the smaller examples.

6 Related Work

Cloud Haskell. The work described in this paper is strongly related to Cloud Haskell. Cloud Haskell presented an elegant API, as evidenced by the work described in this paper reusing many parts of it. The primary difference is the treatment of `spawn`, where Cloud Haskell required the programmer to manually apply lambda-lifting to their process bodies. Furthermore, the programmer had to apply explicit closure-conversion when spawning such functions, adding cognitive overhead. CloudMicroHaskell removes this source-level closure conversion step. A process body can be written directly at the call to `spawn`, including references to variables in scope, and the runtime serialises the reachable graph when the process crosses a node boundary.

Cloud Haskell explicitly discusses the style of implementation used by CloudMicroHaskell, where the compiler or runtime provides primitives for serialising arbitrary values, and argues against it. The concerns raised there still apply. Some runtime-owned values, such as open file handles, should not be serialisable. A programmer may also accidentally serialise a much larger graph than intended, or serialise an unevaluated computation whose work was expected to happen before transmission.

Runtime-Level Code Mobility. CloudMicroHaskell relies on a runtime representation in which code and data can both be serialised as graph nodes. This approach is less natural for implementations such as GHC, which compile Haskell to native machine code. Machine code is neither portable nor desirable as a communication format. The same idea is more plausible for implementations based on bytecode or another portable intermediate representation. In such systems, the code representation can be included in the serialised data and reconstructed, interpreted, or JIT-compiled by the receiver. This is how the Mu implementation at Standard Chartered worked [1].

Glasgow Distributed Haskell. Glasgow Distributed Haskell [7] (GDH) was an earlier effort to extend Haskell for distributed programming. GDH preceded Cloud Haskell, but the general idea was the same, that a distributed application should be described as a collection of communicating processes. GDH was elegant and well-designed, but practical concerns led to the project slowing down. It required substantial modification to GHC's runtime system of the time (the GUM [11] runtime), which limited its adoption and maintainability.

Elixir. The de facto implementation of actor model concurrency is Erlang, a dynamically typed language. Elixir [10] is a language that is executed with the same underlying runtime system, but whose syntax is more approachable. At the fundamental level Elixir is not much different from Erlang, but it offers a more modern tooling and ecosystem.

Tierless and Choreographic Haskell. Recent work has explored several ways to describe distributed Haskell programs from a single source program. In tierless programming, the behaviours of multiple tiers are written and type-checked together, then compiled into separate executables. The framework of Ekblad and Claessen [3] follows this style for client-server web applications. The client and server are described in one program, and the compiler separates the code so that client code does not appear in the server, and vice versa.

The idea was later adopted by HasTEE [8], a framework for programming *Trusted Execution Environments* in Haskell. They observed that a trusted computing unit running on Intel SGX hardware assumes the role of a server, and that the programming model of Ekblad and Claessen [3] was a good fit because of its separation of client code and server code.

Choreographic programming [6] takes a related whole-program view, but describes communication protocols globally rather than from one endpoint's perspective. A communication action is written once, naming both sender and receiver. When the choreography is interpreted or projected for a particular endpoint, that action becomes a send, a receive, or no local action. HasChor [9] embeds this style of choreographic programming in Haskell. Its presentation is elegant and simple, but the original implementation produced executables that contained the code for every endpoint in the network. Krook and Hammersberg [5] address this limitation by specialising HasChor programs with respect to a known endpoint, producing executables that contain only the code needed by that endpoint.

7 Conclusions & Future Work

We set out to explore what the Cloud Haskell programming model looks like when serialisation is provided by the runtime system rather than exposed as `Closure` and `Serializable` constraints in the source-level API. CloudMicroHaskell shows that this design point is practical: remote processes can be spawned in direct style, messages can contain ordinary values and functions, and familiar Erlang-style abstractions such as generic servers and supervisors can be implemented as ordinary libraries.

The benefit is ergonomic: programs can be written much like local communicating processes, with the runtime serialising the reachable MicroHs graph when computation or data crosses a node boundary. The cost is that this boundary moves to runtime: deserialisation depends on the receiver's expected type, and programmers must avoid moving runtime-owned resources or unevaluated work unintentionally.

Future work should reduce what crosses the wire. One direction is to give nodes a shared vocabulary of known code and send references where possible, falling back to graph serialisation for dynamic code and environments. Another is to move low-level spawning and message delivery into the MicroHs runtime, reducing overhead while centralising dynamic checks for graph types, runtime-owned resources, and version compatibility.

Acknowledgments

The authors would like to thank Koen Claessen, Mary Sheeran, John Hughes, and Simon Peyton-Jones for their helpful discussions and feedback.

A API Reference

This appendix lists the core CloudMicroHaskell API used throughout the paper.

```
data Node a
data NodeId
data NodeConfig -- = (ip, port)
data ProcessM a; instance MonadIO ProcessM
data Pid
runNode :: NodeConfig -> Node () -> IO ()
```

A node says *where* something is: a location in a network. When a node connects to an existing network of nodes, they form a mesh, whereby every node is connected to every other node.

```
connect  :: NodeConfig -> ProcessM ()
```

Each node can host zero or more processes, which are independent computation units. Processes (almost) don't share any memory with other processes, with the exception(s) described in section 3. There are some basic operations for processes to identify themselves, their node, the nodes that are connected to the network, and to terminate.

```
self      :: ProcessM Pid
node      :: ProcessM NodeId
nodes     :: ProcessM [NodeId]
terminate :: ProcessM a
```

A process that wishes to retrieve a list of every node in the network, aside from its own host node, may execute

```
do mynode <- node
   allnodes <- nodes
   return $ filter (/= mynode) allnodes
```

Processes are started either by running an initial process on a node, or by spawning a new process from an existing one. A derived `call` operation spawns a process and waits for its result.

```
runProcessM :: ProcessM () -> Node ()
spawn :: NodeId -> ProcessM () -> ProcessM Pid
call  :: NodeId -> ProcessM a -> ProcessM a
```

As the processes are independent and don't share any memory, they cannot perform any meaningful task without interacting with each other. Interactions are done via message passing, where processes can `send` and `expect` messages to and from one another. `send` is an asynchronous call that returns immediately, whereas `expect` is a synchronous call that doesn't return until it has received a message of the correct type. It is appropriate to think of it as if every process had a mailbox where messages of any type can go in, and where `expect` goes through the letters in the order they arrived, looking for one of the right type.

```
send   :: Identifiable process => process -> a -> ProcessM ()
expect :: ProcessM a
```

For selective receive, CloudMicroHaskell follows the matcher interface introduced by Cloud Haskell [4]. A matcher specifies one acceptable message type, optionally guarded by a predicate, and `receiveWait` blocks until any matcher succeeds. There is additionally a `matchUnknown` handler that will match a message of any type, removing it from the mailbox.

```haskell
data MatchM q a
match          :: (a -> ProcessM q) -> MatchM q ()
matchIf        :: (a -> Bool) -> (a -> ProcessM a) -> MatchM q ()
matchUnknown   :: ProcessM q -> MatchM q ()
receiveWait    :: [MatchM q ()] -> ProcessM q
receiveTimeout :: Int -> [MatchM q ()] -> ProcessM (Maybe q)
```

A matcher is one branch of a selective receive. It recognises a message and runs a handler producing a value of type `q`. `receiveWait` blocks until one of the supplied matchers succeeds, while `receiveTimeout` bounds the wait and returns `Nothing` if no message matches in time.

For example, a process can wait for either a request to divide two numbers, or for a command to shut down

```haskell
data Div      = Div Pid Int Int
data Shutdown = Shutdown

server :: ProcessM ()
server = receiveWait
  [ matchIf (\(Div _ _ y) -> y /= 0) $ \(Div sender x y) ->
      send sender (x `div` y)
  , match $ \Shutdown -> terminate
  ]
```

The two matchers receive messages of different types, but both branches are part of the same blocking receive. The first branch accepts only non-zero division requests and replies to the sender. The second branch accepts a shutdown command and terminates the process. Messages that do not match either branch remain in the mailbox.

So far, the API describes how live processes communicate. For fault-tolerant programs, processes must also be able to observe one another's lifetime, where a missing reply may mean that another process has terminated, crashed, or disappeared with its node. CloudMicroHaskell exposes this through monitors.

```haskell
data ProcessDied = ProcessDied Pid ExitReason
data MonitorAction = TrapExit | Succumb

monitor :: MonitorAction -> Pid -> ProcessM ()
```

When process `p` monitors process `q`, `p` is notified if `q` terminates. With `TrapExit`, the notification is delivered as a `ProcessDied` message in `p`'s mailbox. With `Succumb`, `p` terminates as well.

Aside from observing process termination, processes can also cause process termination, using the `exit` function. This makes it possible to designate application-specific roles to processes, such as workers and supervisors. Supervisor processes might send an exit signal to a process that has entered an unhealthy state.

```haskell
data ExitReason
  = ExitNormal     -- ^ Process exited normally
  | ExitShutdown   -- ^ Process exited upon request of another
                   --   process, e.g. a supervisor
  | ExitKill       -- ^ Process was forcibly killed, equivalent to kill -9.
  | ExitOther String -- ^ Other custom reasons why a process was terminated

exit :: Pid -> ExitReason -> ProcessM ()
```

An example that illustrates how `monitor` and `exit` can be used to control behaviour is shown below

```
do child <- spawn n childBody

   _ <- spawn n $ do
     monitor TrapExit child
     ProcessDied _ _ <- expect
     liftIO (putStrLn "the child died!")

   linked <- spawn n $ do
     monitor Succumb child
     expect

   monitor TrapExit linked
   exit child ExitKill
   ProcessDied _ _ <- expect
```

The first monitor survives the child's death and receives a `ProcessDied` message. The second monitor uses `Succumb`, so it dies together with the child. By monitoring the second process with `TrapExit`, the parent observes this cascading termination as an ordinary mailbox message.

The process registry maps node-local names to process identifiers. In the example above, the two monitoring processes had to have a way of knowing the process identifier of the child process. Sometimes it is not ergonomic to distribute process identifiers to everyone who has to know them. Furthermore, a process may be restarted by a supervisor process, in which case the pre-distributed process identifier will no longer be valid. The process registry lets a process register itself under a name on the node it is running on.

```
register   :: Pid -> String -> ProcessM ()
unregister :: String -> ProcessM ()
whois      :: String -> ProcessM (Maybe Pid)
```

Since `String` is an instance of `Identifiable`, `send` may target a registered name.

```
send :: Identifiable i => i -> a -> ProcessM ()
```

Types that are `Identifiable` are process identifiers and strings. If a message is sent to a named process, the process identifier is fetched from the process registry before the message is sent. If the name is not registered, the `send` fails with an exception, much like it would in Erlang.

We emphasise that the name registry is node-local, meaning that messages can only be sent to named processes on the same node. The process identifier of a remote named process can be retrieved without much fuss, however. We can simply issue a synchronous call, reading the remote registry

```
remoteProcess <- call n' (whois "child") -- n' is the NodeId of some remote node
case remoteProcess of
  Just pid -> send pid "hello"
  Nothing -> return ()
```

References

[1] Lennart Augustsson. 2011. Keynote: Pragmatic Haskell. In *Proceedings of the 8th ACM SIGPLAN Workshop on Commercial Users of Functional Programming (CUFP)*. ACM, Tokyo, Japan. Discussions on the design and deployment of the Mu language at Standard Chartered Bank.

[2] Lennart Augustsson. 2024. MicroHs: A Small Compiler for Haskell. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, Niki Vazou and J. Garrett Morris (Eds.). ACM, 120–124. doi:10.1145/3677999.3678280

[3] Anton Ekblad and Koen Claessen. 2014. A seamless, client-centric programming model for type safe web applications. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 79–89. doi:10.1145/2633357.2633367

[4] Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. 2011. Towards Haskell in the cloud. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, Koen Claessen (Ed.). ACM, 118–129. doi:10.1145/2034675.2034690

[5] Robert Krook and Samuel Hammersberg. 2024. Welcome to the Parti(tioning) (Functional Pearl): Using Rewrite Rules and Specialisation to Partition Haskell Programs. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, Niki Vazou and J. Garrett Morris (Eds.). ACM, 27–40. doi:10.1145/3677999.3678276

[6] Fabrizio Montesi. 2014. Choreographic programming. (2014).

[7] Robert F. Pointon, Philip W. Trinder, and Hans-Wolfgang Loidl. 2000. The Design and Implementation of Glasgow Distributed Haskell. In *Implementation of Functional Languages, 12th International Workshop, IFL 2000, Aachen, Germany, September 4-7, 2000, Selected Papers (Lecture Notes in Computer Science)*, Markus Mohnen and Pieter W. M. Koopman (Eds.). Springer, 53–70. doi:10.1007/3-540-45361-X_4

[8] Abhiroop Sarkar, Robert Krook, Alejandro Russo, and Koen Claessen. 2023. HasTEE: Programming Trusted Execution Environments with Haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium, Haskell 2023, Seattle, WA, USA, September 8-9, 2023*, Trevor L. McDonell and Niki Vazou (Eds.). ACM, 72–88. doi:10.1145/3609026.3609731

[9] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 541–565. doi:10.1145/3607849

[10] The Elixir Team. 2026. The Elixir Programming Language. Accessed: 2026-05-21. https://elixir-lang.org/

[11] Philip W Trinder, Kevin Hammond, James S Mattson Jr, Andrew S Partridge, and Simon L Peyton Jones. 1996. GUM: a portable parallel implementation of Haskell. *ACM SIGPLAN Notices* 31, 5 (1996), 79–88.

[12] D. A. Turner. 1979. A new implementation technique for applicative languages. *Software: Practice and Experience* 9, 1 (1979), 31–49. doi:10.1002/spe.4380090105

Received 2026-05-22; accepted 2026-06-30