Design and implementation of a WASM-based process execution service for a distributed-systems middleware

Mikel Solabarrieta Román

Universitat Politècnica de Catalunya

18/10/2023

Supervisor: Jordi Guitart Fernández

Master Degree in Innovation and Research in Informatics
(High Performance Computing)

Outline

1. Introduction and background
2. Development
3. Validation and performance analysis
4. Conclusions
5. Q&A

Introduction

- Design and development of a process executor capable of running and managing programs written in WebAssembly and expose its functionality through object capabilities, as part of a wider distributed systems middleware.
- Validation through a real, distributed application.
- Performance analysis of the developed components.

Distributed systems, IaaS, and PaaS

Infrastructure is rented as a service, often with additional features. Synergizes with distributed systems for its convenience, pricing and ease of horizontal growth; at the cost of vendor lock-in and control relinquishment.

Distributed systems provide:

- Horizontal scalability, availability, concurrency, fault-tolerance, load-balancing, flexibility, data-locality.

At the cost of:

- Complexity, consistency, networking, consensus, failure management, communication bottlenecks[1].

Common examples:

- Content delivery networks, distributed databases, file-sharing networks, VPNs, distributed indexers, web crawlers...

---

[1] Alexey Gotsman et al. "'Cause I'm strong enough: Reasoning about consistency choices in distributed systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 371–384.

Wetware

Definition

Wetware

Modular middleware to build and run peer-to-peer distributed applications through object capabilities.

Provides:

- Peer discovery
- Clustering
- Blob storage
- Inter-process communication
- Message propagation
- **Process execution**
- **Process management**
- ...

Wetware

Usage

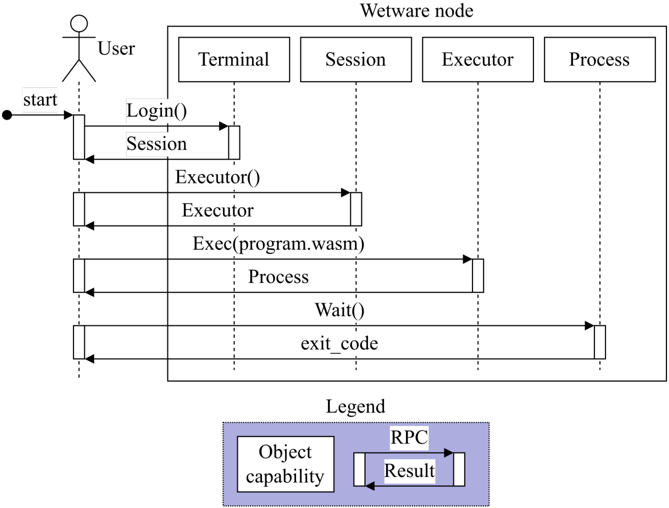


Figure: How Wetware is used to run a process

Cap'n Proto

Object capability and RPC framework

Open source RPC framework and serialization format with user experience similar to OOP.

- Created as successor to Protocol Buffers
- CPU-bound
- Incremental and random reads over serialized data
- Language agnostic
- Time-travel promise pipelining

Security model: objects can only be interacted with through messages sent to their references (object capabilities).

WebAssembly

- ▷ Fast, portable, low-level code[2].
- ▷ Intermediate code format.
- ▷ Processes run in isolation.
- ▷ No real parallelism in a process.
- ▷ WASI: layer over the runtime to enable socket, file system... access.
- ▷ Wazero:
  - Portable
  - Single, statically linked binary
  - Community
  - Collaboration

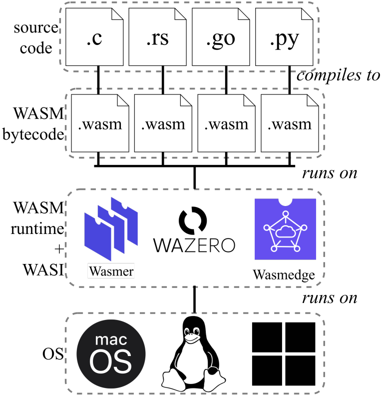


Figure: WASM component hierarchy

---

[2] Andreas Haas et al. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* (2017). ISSN: 0362-1340. DOI: 10.1145/3140587.3062363.

Go

Compiled, garbage-collected language.

Strong focus on concurrency and parallelism.

Concurrency through Goroutines: lightweight threads.

Native concurrency tools:

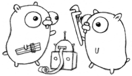
- Channels
- Select statements
- Mutexes, thread-safe structures
- Contexts

Collaborative and preemptive scheduling.

Support for many architectures and operating systems, as well as WebAssembly as a compilation target.

Go

*G*s, *M*s and *P*s



Definitions:

- *P*: Logical cores and network poller
- *M*: OS thread
- *G*: Goroutine

The scheduler's job is to match a *G*, a *M* and a *P*.

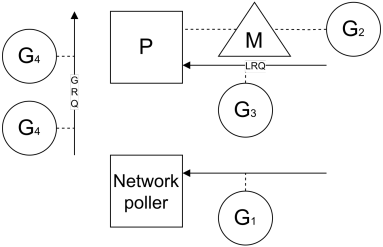


Figure: *G*s, *M*s, and *P*s

Go

Concurrency control

	Blocks			
Interface	**G**	**M**	**P**	**Cost**
mutex	Y	Y	Y	$$$
note	Y	Y	Y/N*	$$
park	Y	N	N	$

Table: Block-levels of native yielding interfaces[3]

* depends on the specific system call

---

[3] Go. *"Scheduling Structures"*. `https://go.dev/src/runtime/HACKING`.

Development

Development is split into:

1. Core executor
2. Host↔Guest communication
3. Process management
4. Integration into Wetware
5. Validation

Core executor

- Based on a Wazero runtime.
- Binds a Process capability to a guest WASM function call.
- Exposed through object capabilities.

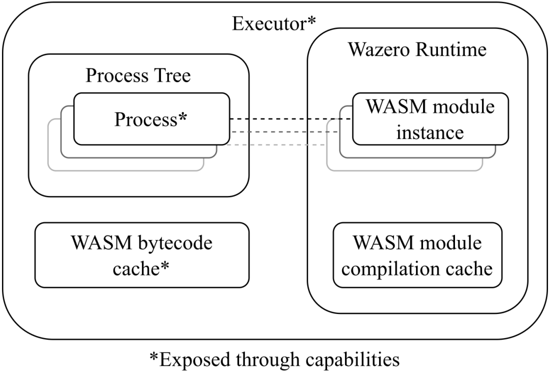
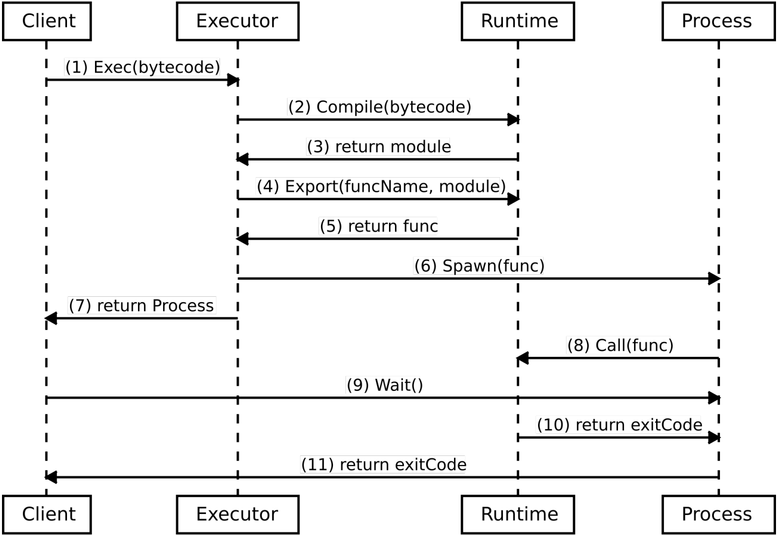


Figure: Executor component overview.

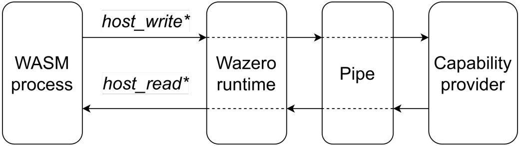
Process execution

Sequence diagram

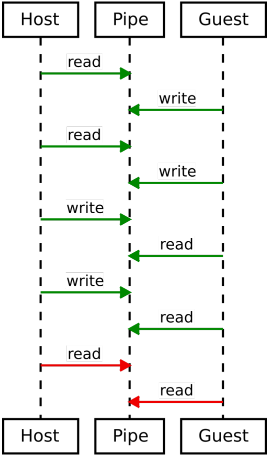


Host-guest asynchronous communication

Host functions



*invoked by the WASM guest, performed by the host.



Host-guest asynchronous communication

Pre-opened sockets

Feature officially added to WASI and adopted by Go 1.21[4][5], as well as Wazero[6].

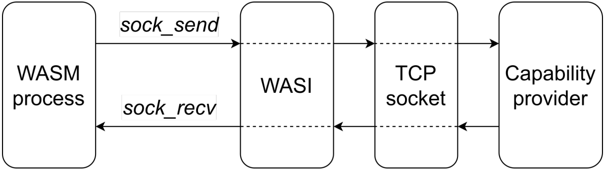


Figure: Asynchronous host-guest communication through pre-opened TCP sockets.

---

[4] Go. *"runtime: implement wasip1 netpoll"*. https://go-review.googlesource.com/c/go/+/493357.

[5] Go. *"net: implement wasip1 FileListener and FileConn"*. https://go-review.googlesource.com/c/go/+/493358.

[6] Wazero. *"WASI: fix nonblocking sockets on *NIX"*. https://github.com/tetratelabs/wazero/pull/1503.

Host-guest asynchronous communication

Host flow diagram

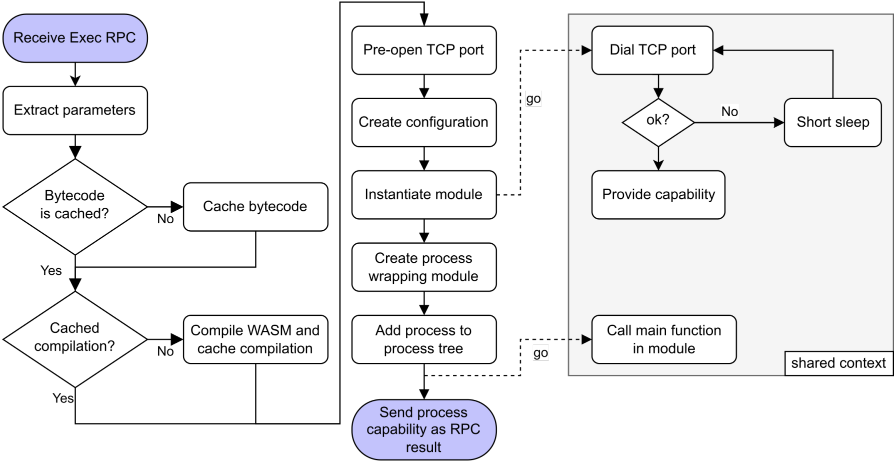
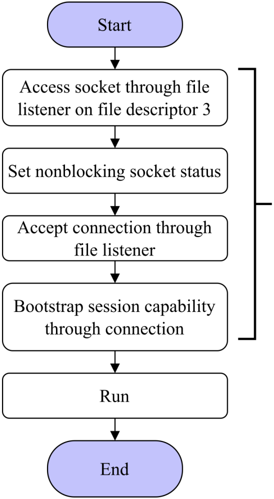


Figure: Flow diagram of execution method.

Host-guest asynchronous communication

Guest flow diagram



Abstracted from the user through

import ww "github.com/wetware/pkg/guest/system"

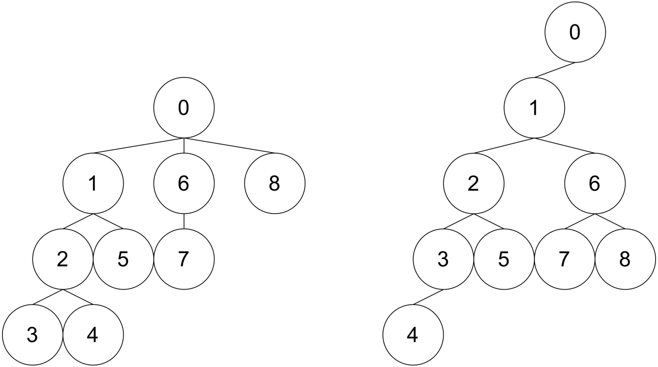
ww.Bootstrap(context.Context)

Figure: Guest bootstrapping flow diagram.

Process management

Process hierarchy

Complemented by a *PID* → *Process* map.



(a) Natural tree (b) Binary representation

Figure: Representations of the process tree

Process management

The Process capability

Processes are interacted with through their object capabilities.
Process capabilities have methods to:

- Wait for completion
- Pause/Resume
- Monitor
- Link/Unlink
- ID
- Kill
- Process listing

Pause and resume processes

Pause and resume send events to Wetware processes, but require user implementation of event management.

Calls to *Process.Pause* and *Process.Resume* will cause an event on the guest *OnPause* and *OnResume* channels.

```
func main() {
    ...
    urls = make(chan string)
    for {
        select {
        case <-eventHandler.OnPause():
            <-eventHandler.OnResume()
        case <-crawl(ctx, urls, <-urls):
        }
    }
}
```

Listing: Event loop management in a Wetware process

Process linking and unlinking

Link

Bidirectional relation between two processes $A \leftrightarrow B$ in which if either A or B end, the other will end as well.[a]

---

[a] Erlang. *"link"*. `https://www.erlang.org/doc/man/erlang#link-1`.

Four methods: *Link*, *LinkLocal*, *Unlink*, *UnlinkLocal*.

Idempotence: there can only be one or zero links between two processes at any given time. Unlink operations have no effect between processes with no link.

Propagation: If $A \leftrightarrow B$ and $B \leftrightarrow C$, terminating A will indirectly cause C's termination. Every initial call $A \rightarrow B$ to *Link* or *Unlink* will cause a roundtrip call $B \rightarrow A$.

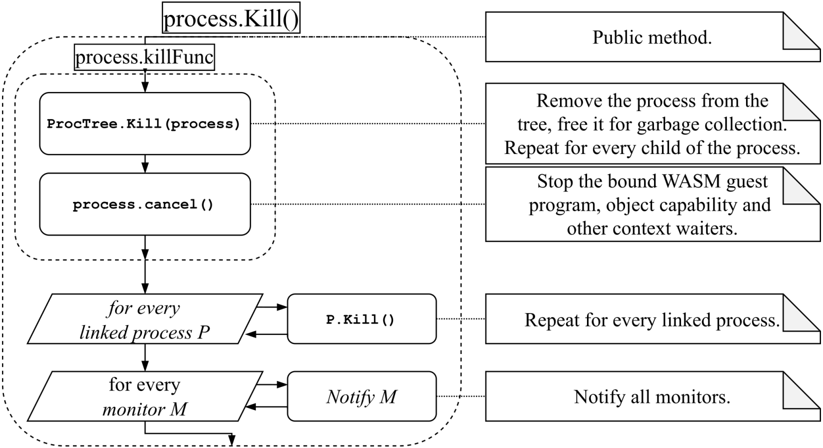
Process monitoring

If a process A monitors a process B, A will be notified when B ends.

Calls to monitor are blocked, and only released when the monitored process ends or connection is lost.

Same as *Wait*, but returns a reason instead of an exit code.

Process ending



Public method.

Remove the process from the tree, free it for garbage collection. Repeat for every child of the process.

Stop the bound WASM guest program, object capability and other context waiters.

Repeat for every linked process.

Notify all monitors.

Figure: End of a process

Process listing

Unitary *Executor.Ps()* method, which provides the executor, PID, PPID, CID, creation date and arguments of every process running in the executor.

Aggregated by CLI tool *ww ps* to show every process in the cluster.

```
mikel@laptop$ ww ps
Executor   PID  PPID  Creation       CID     Args
38908...   2    1     Wed Oct 11...  z26...  [iPH...]
38908...   3    2     Wed Oct 11...  z26...  [iPH...]
92149...   2    1     Wed Oct 11...  z26...  [j92...]
92149...   3    1     Wed Oct 11...  z26...  [j92...]
7158b...   2    1     Wed Oct 11...  z26...  [mLw...]
7158b...   3    1     Wed Oct 11...  z26...  [mLw...]
mikel@laptop$
```

Listing: Output of “ww ps”

Integration into Wetware

Each Wetware node has an executor, exposed through a capability and initialized along with the node.

Introduction of the *ww cluster run* command.

Guest bootstrapping and QoL features in the *guest/system* package.

Considerations:

- The Wazero runtime is configured with `WithCloseOnContextDone(true)`, for propagating stops to processes.
- Any *select* statements that might prevent a correct shutdown must contemplate the case of the context ending.
- Every spawned goroutine is tied to a context bound to the “original” context.

Validation prerequisites

Developing a web crawler application as validation showed us the need for:

- A Capability Storage, accessible through the executor capability.
- Implementation of any additional functionalities outside WASM, in this case HTTP requests.
- Utilization of the Capability Storage to provide the functionalities to the application.

Basic validation

Through a real application

The initial version of the web crawler has centralized coordination and distributed work.

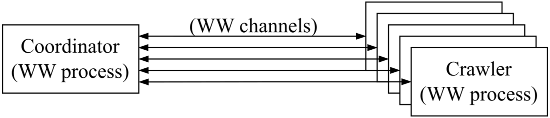


Figure: Distributed web crawler with centralized coordination

It allowed us to validate:

- Creation and deployment of a distributed application in a Wetware node and cluster.
- Creation of sub-processes from a Wetware process.
- Inter-process communication through Wetware channels.
- Overcoming of WASM limitations through external object capabilities.

Raft

A comprehensible consensus algorithm[7]

- ▷ Standalone Raft-over-Cap'n Proto library based on Etcd's Raft implementation.
- ▷ Cap'n Proto as a transport.
- ▷ Nodes communicate through each other's capabilities.
- ▷ Some method implementation up to the user: required for Wetware compatibility.
- ▷ Missing snapshot features.

---

[7] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. ISBN: 9781931971102.

Feature-rich validation

Design

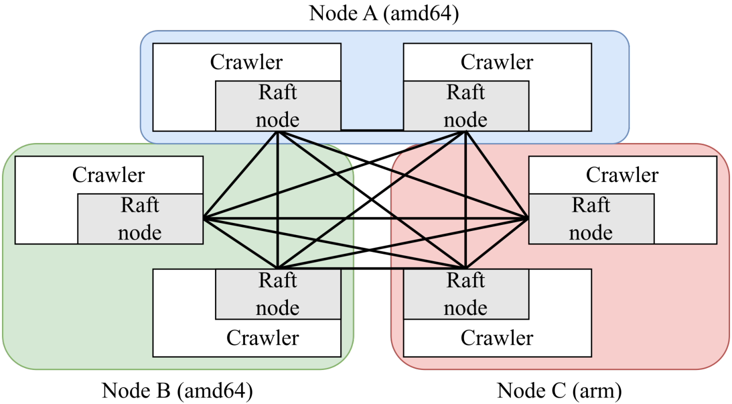


Figure: Fully distributed web crawler

Feature-rich validation

By creating a complex application

It demonstrated:

- Creation and deployment of a fully distributed application across nodes in a Wetware cluster.
- Nodes in a Wetware cluster can run in different architectures.
- Consensus algorithm utilization in Wetware applications.
- Fault tolerance through *claim* sets.
- Management of local and global queues.
- Transparent IPC without the need of channels.
- Stability of the executor in a 30 minute application run.
- Successful integration of the executor as part of the Wetware middleware.

Performance analysis

Highly parallelizable workload - Busy

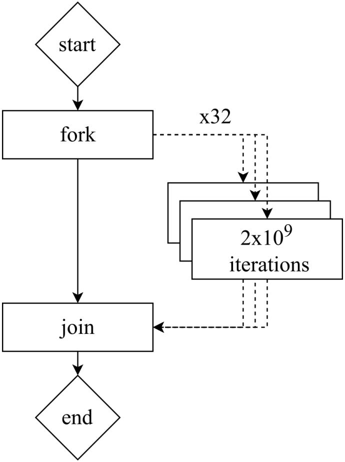


Figure: Multi-process busy program flow diagram

Performance characterization

Speedup

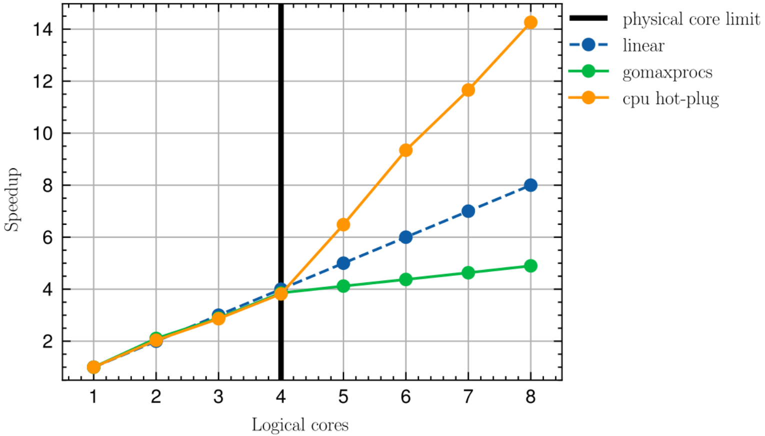


Figure: Busy workload strong scalability

Performance characterization

Total runtime

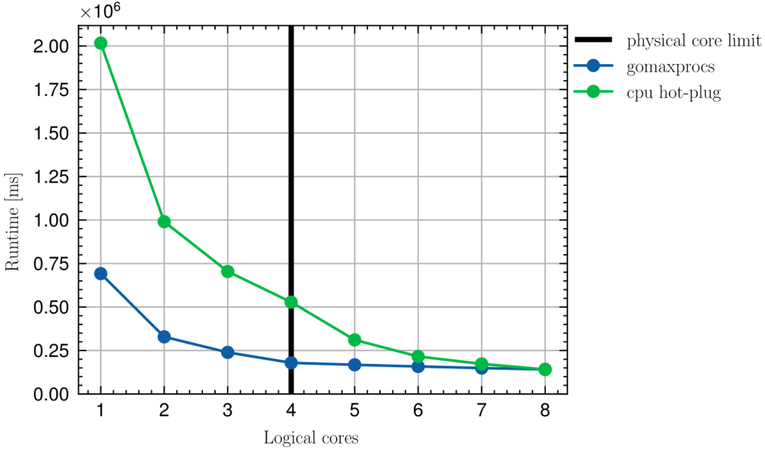


Figure: Total runtime of the busy workload

Benchmark against native programming languages

Corrected results

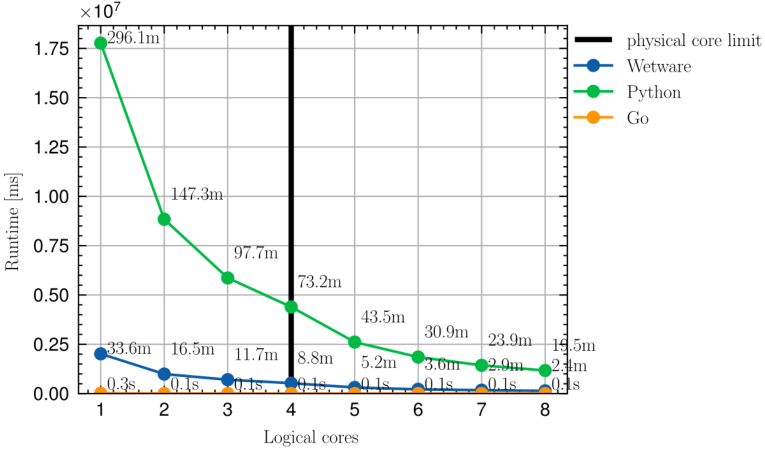


Figure: Runtime comparison for equivalent programs

Concurrency characterization

Utilization of concurrency control mechanisms

Flat	**Flat%**	**Sum%**	**Cum**	**Cum%**	**Name**	**Cost**
11645	82.55%	82.55%	11645	82.55%	runtime.selectgo	$
1656	11.74%	94.29%	11645	11.74%	runtime.chanrecv1	$
515	3.65%	97.29%	515	3.65%	sync.(*Mutex).Lock	$$$
279	1.98%	99.92%	279	1.98%	runtime.chanrecv2	$
0	0.00%	99.92%	131	0.93%	net.(*Buffers).WriteTo	?
0	0.00%	99.92%	402	2.85%	io.ReadFull	?
0	0.00%	99.92%	402	2.85%	io.ReadAtLast	?
0	0.00%	99.92%	5287	38.19%	golang.org/x/sync/errgroup.(*Group).Go.func1	$

Table: Top concurrency contentions on Wetware server running a webcrawling application, excluding system calls

Conclusions

- We designed and developed WebAssembly and Cap'n Proto-based process execution and management tools for the Wetware distributed systems middleware.
- Studied and used the background to make design and development decisions.
- Provided isolated WebAssembly modules access outside their sandbox only though object capabilities.
- Created and deployed a real distributed application.
- Concluded that Wetware is a viable tool for building and deploying distributed P2P applications with comparable performance to higher-level languages.

Future work

- Re-approach asynchronous communication between WASM host and guest through host functions.
- Consider adoption of Cap'n Proto third-party hand-off.
- Further performance improvements.
- Quality of life and user experience improvements.

Contributions to open source projects

- github.com/wetware/pkg
- github.com/mikelsr/pkg
- github.com/mikelsr/ww-webcrawler
- github.com/mikelsr/raft-capnp
- github.com/mikelsr/ww-raft-example

End of the presentation

Qustions and answers

Thank you for attending!
Time for Q&A.