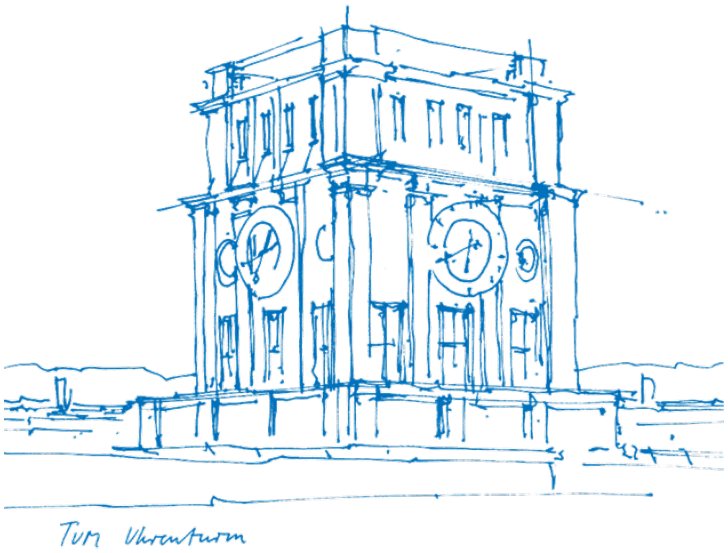
Chair of Connected Mobility
TUM School of Computation, Information and Technology
Technical University of Munich

Enabling WebAssembly Networking in Embedded Devices with Oakestra

Interdisciplinary Project (IDP)

Alessandro Grassi



Enabling WebAssembly Networking in Embedded Devices with Oakestra

Interdisciplinary Project (IDP)

Alessandro Grassi

Chair of Connected Mobility
TUM School of Computation, Information and Technology
Technical University of Munich

Enabling WebAssembly Networking in Embedded Devices with Oakestra

Interdisciplinary Project (IDP)

Alessandro Grassi

Interdisciplinary Project (IDP)

at the TUM School of Computation, Information and Technology of the Technical University of Munich.

Examiner:
Prof. Dr.-Ing. Jörg Ott

Supervisor:
Giovanni Bartolomeo

Submitted:
Munich, 30.03.2025

I hereby declare that this document is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Munich, 30.03.2025 Alessandro Grassi

Abstract

This Interdisciplinary Project (IDP) explores the integration of WebAssembly (Wasm) within embedded Linux devices using the Oakestra orchestration framework. The research evaluates different approaches to enable inter-service networking of WebAssembly workloads on resource-constrained edge devices. After comparing operating system options, Buildroot was selected to create a lightweight embedded Linux distribution for Raspberry Pi hardwares. The project implements two network configuration approaches using Linux namespaces: a simple veth pair for namespace-internal connectivity and a bridged veth pair for cross-host communication. The implementation tries to extend Oakestra's Wasm runtime capabilities with network support.

Contents

1 Introduction

Service orchestration involves the processes needed to package, deploy, manage, and maintain services across distributed computing resources. Traditional frameworks like Kubernetes were built for uniform cloud environments with consistent cluster information and standardized resource handling. However, these frameworks struggle in edge computing settings, which feature diverse hardware, varying computational power, and unstable network conditions.

Oakestra adresses these edge computing challenges with a hierarchical framework that prioritizes flexibility over strict consistency, allowing management of diverse hardware types. Its efficient design cuts resource usage by about ten times and improves application performance by 10% on limited hardware. While Oakestra traditionally worked with standard Linux machines, new developments focus on embedded devices, where connecting services through overlay networks remains difficult.

WebAssembly (WASM) offers a strong solution for service orchestration in embedded environments. Its benefits include portable bytecode, near-native speed without containers, lower memory usage, support for live migration while preserving state, better data locality, and lower latency. This method separates applications from infrastructure, enhancing flexibility and allowing resources to be shared across physical nodes.

This Interdisciplinary Project assesses methods for connecting WebAssembly workloads across services within embedded Linux systems in Oakestra. We will select a lightweight, open-source Linux-based embedded OS suitable for industrial use, review networking approaches, design a system linking WebAssembly workloads to Oakestra's Semantic Overlay Network, and build a prototype that enables services to communicate across cloud/edge systems and embedded devices.

Our work builds on key research areas including lightweight edge orchestration systems, large-scale service mesh designs, container migration methods, and the emerging Computing Continuum model—where computation flows smoothly across diverse environments from data centers to embedded devices. By combining these ideas, we move toward a future where applications contain movable parts that automatically adjust for best performance, energy efficiency, and user experience across all computing levels.

1.1 Timeline

- **Nov 2024**: Hardware handout, initial meetings with project partner in University of Padua and Definition of target OS
- **Dec 2024**: Experiments deploying WASM Workloads on the Embedded OS. Literature review and real-world experiments of the networking approaches.
- **Jan 2025**: Final network architecture and initial implementation effort for the final prototype
- **Feb 2025**: Continuing implementation, finalization, and IDP related exam
- **Mar 2025**: Final Report and IDP Presentation

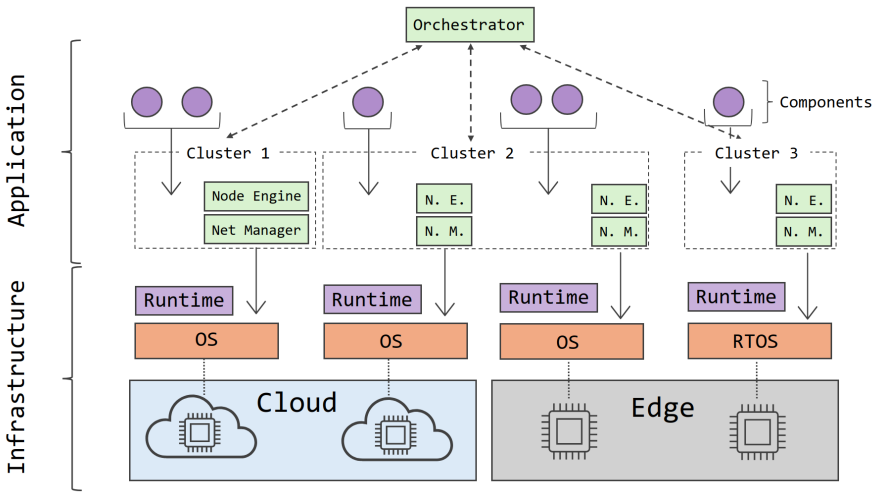


Figure 1.1 Oakestra Architecture

2 Environment

In this chapter we explore our process for selecting an appropriate operating system for embedded devices. Beginning with Zephyr OS, we ultimately pivoted to Linux-based solutions due to WebAssembly runtime support requirements and development constraints. After comparing Yocto and Buildroot, we selected Buildroot for its simplicity and efficiency. We detail our Buildroot configuration for Raspberry Pi, including the integration of Wasmtime for WebAssembly support and the Oakestra Worker Node Package for edge computing orchestration.

2.1 Zephyr OS

Our initial operating system candidate was Zephyr OS, a small-footprint RTOS designed for resource-constrained devices. Zephyr offers several advantages that initially made it attractive for this project. However, we encountered mainly three integration challenges:

1. **WebAssembly Runtime Support**: Zephyr currently lacks native support for Wasmtime, our chosen WebAssembly runtime.
2. **Development Environment Complexity**: The specialized toolchain required for Zephyr development would introduce additional complexity compared to more Linux-like alternatives.
3. **Limited Time**

2.2 Linux-Based Alternatives

After deciding to pursue a Linux-based approach, we evaluated two primary build systems for creating custom embedded Linux distributions:

1. **Yocto Project**: A comprehensive framework for creating custom Linux distributions
2. **Buildroot**: A simpler, more straightforward embedded Linux build system

2.3 Yocto

The Yocto Project provides templates, tools, and methods for creating custom Linux-based systems for embedded applications. Built for highly customized Linux distributions with precise control over components and optimizations. Despite these advantages, we ultimately discarded Yocto for this project due to its steep learning curve, resource-intensive build process requiring substantial hardware resources, and long development cycles that impeded rapid iteration.

2.4 Buildroot

Buildroot offers a simple approach to embedded Linux system development that addresses many of Yocto's limitations. Unlike Yocto's multi-layered architecture, Buildroot utilizes a simpler, make-based build system that significantly reduces complexity while maintaining powerful customization capabilities. This simplicity means substantially faster build times and a shorter learning curve for developers.

2.5 Buildroot on Raspberry

1. **Buildroot Configuration**
   - Buildroot-2024.02.9 configured for Raspberry Pi 4 (64-bit)
   - Custom Raspberry Pi Linux kernel
2. **Wasmtime Integration**
   - Integrated Wasmtime 28.0.0 runtime
   - Compiled specifically for aarch64 architecture (Raspberry Pi 4 64-bit)
   - Cargo target hard-coded to aarch64-unknown-linux-gnu
   - Wasmtime installed to /usr/bin/wasmtime
3. **Network and Utilities**
   - Wireless connectivity via iwd, pre-configured for network eth0
   - Installed network tools: iptables, iproute2, dropbear (SSH), openresolv
   - Default WiFi PSK configured for network raspberrypi4-64
4. **Additional Components**
   - System monitoring via collectd
   - SSH/SFTP capabilities through Dropbear
   - Standard utilities and basic editors (nano, etc.)

2.6 Buildroot configuration

2.6.1 Buildroot Basic Commands

This section covers the essential commands for working with Buildroot to create and manage your embedded Linux system for Raspberry Pi.

2.6.2 Getting Started with Buildroot

1. **Downloading Buildroot**: Begin by obtaining the Buildroot source

```
$ wget https://buildroot.org/downloads/buildroot-2024.02.9.tar.gz
$ tar -xf buildroot-2024.02.9.tar.gz
$ cd buildroot-2024.02.9
```

2. **Initial Configuration**: Load the default Raspberry Pi 4 (64-bit) configuration

```
$ make raspberrypi4_64_defconfig
```

Configuration Commands

1. **Main Configuration Interface**: Launch the menuconfig tool to customize your build

```
$ make menuconfig
```

2. **Linux Kernel Configuration**: Modify the Linux kernel settings

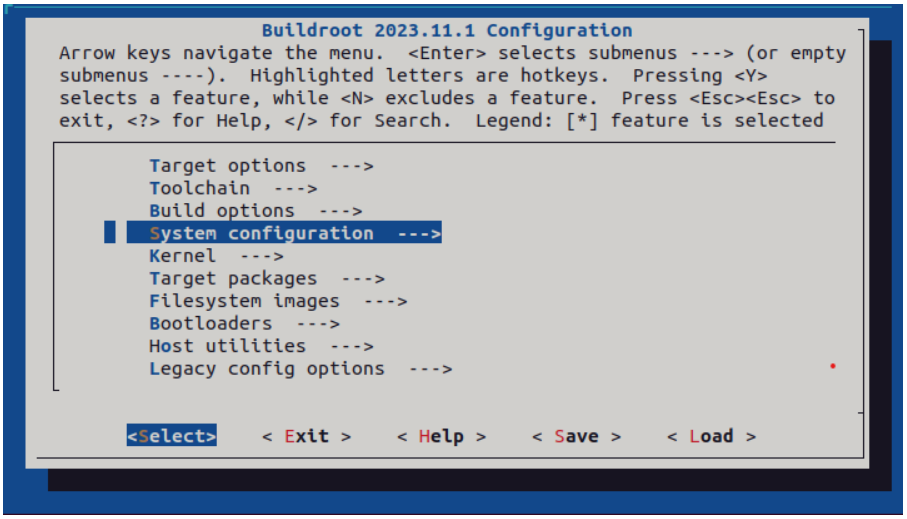


Figure 2.1 Buildroot Menuconfig

```
$ make linux-menuconfig
```

3. **Save Configuration**: Save your configuration as a defconfig file

```
$ make savedefconfig
$ cp defconfig board/raspberrypi4-64/my_raspberrypi4_64_defconfig
```

Build Commands

1. **Parallel Build**: Accelerate the build using multiple cores

```
$ make -j$(nproc)
```

2. **Building Specific Packages**: Rebuild a single package

```
$ make wasmtime-rebuild
$ make oakestra-worker-rebuild
```

Clean Commands

1. **Clean a Specific Package**: Remove build artifacts for a single package

```
$ make wasmtime-clean
```

2. **Clean All Packages**: Remove all package build artifacts while preserving configuration

```
$ make clean
```

2.6.3 System Configuration

- **Device Management**: Changed from devtmpfs to mdev - Switched to a lighter-weight device manager, which is typically used in more resource-constrained environments

- **Custom Overlay**: Added `board/raspberrypi4-64/rootfs_overlay` - Including custom files/-configurations specific to Raspberry Pi 4 64-bit
  - I created this pre-configured WiFi network credential for the iwd wireless daemon. `board/raspberrypi4-`

2.6.4 Connectivity

- **WiFi Support**: Added BR2_PACKAGE_BRCMFMAC_SDIO_FIRMWARE_RPI_WIFI=y - Firmware for the Raspberry Pi's built-in WiFi
- **Wireless Management**: Added BR2_PACKAGE_IWD=y - A modern wireless daemon, alternative to wpa_supplicant
- **Network Tools**:
  - Added iproute2 - Modern networking utilities for routing, traffic control, etc.
  - Added iptables - Firewall configuration tool
  - Added openresolv - DNS resolver configuration utility

2.6.5 Remote Access

- **SSH Server**: Added BR2_PACKAGE_DROPBEAR=y - Lightweight SSH server/client
  - Configured with small footprint option (DROPBEAR_SMALL=y)
  - Client functionality enabled

2.6.6 System Libraries & Support

- **Compression**: Added zlib - Essential compression library required by many applications
- **Networking**: Added libcurl with proxy and cookies support - Library for transferring data with URLs
- **XML Support**: Added expat - XML parsing library
- **IPC**: Added dbus - Inter-process communication system
- **Development**:
  - Added binutils - GNU binary utilities
  - Added elfutils - Utilities and libraries for handling ELF files
  - Added ell (Embedded Linux Library) - Core library for embedded devices

2.6.7 Monitoring & Diagnostics

- **System Monitoring**: Added collectd with specific plugins:
  - COLLECTD_LOGFILE=y - Log to files
  - COLLECTD_SYSLOG=y - Log to system log
  - COLLECTD_CURL=y - HTTP data collection

2.6.8 Runtime Environments

- **WebAssembly**: Added wasmtime - Runtime for WebAssembly applications
- **BPF Tools**: Added bpftool - Tools for working with Berkeley Packet Filter programs

2.6.9 Build Tools (Host)

- **Password Generation**: Added host-mkpasswd - Tool for generating encrypted passwords
- **Rust Support**: Added host-rustc (binary version) - Rust compiler, likely needed for WebAssembly and some packages

2.7 Buildroot Custom Packages Structure Overview

Both the Wasmtime and Oakestra Worker packages follow Buildroot's standard organizational pattern to ensure consistency and maintainability. Each package includes several key components: a `Config.in` file defining dependencies and features accessible through Buildroot's menuconfig system, accompanied by respective `.mk` implementation files (`wasmtime.mk` and `oakestra-worker.mk`) that handle build logic and installation procedures. Verification is ensured through cryptographic hash files, while service integration is provided through both systemd unit definitions and traditional SysV init scripts. Each package creates a dedicated configuration directory hierarchy (`/etc/wasmtime/` and `/etc/oakestra/` respectively) with sensible defaults that can be customized through either package-specific files or board-specific rootfs overlays. This modular approach allows for streamlined deployment across different architectures while maintaining the flexibility needed for various embedded applications.

2.8 Wasmtime Package

2.8.1 Introduction and Architecture

The Wasmtime Package is a Buildroot integration solution that enables deployment of the Wasmtime WebAssembly runtime on embedded Linux systems. Built for the latest stable version, the package provides a streamlined method for installing the Wasmtime runtime engine, which serves as a lightweight, standalone runtime for executing WebAssembly modules on resource-constrained devices. The package is designed to support both `ARM64` (`aarch64`) and `x86_64` (`AMD64`) platforms, ensuring broad compatibility across modern embedded computing hardware while maintaining minimal resource requirements appropriate for edge computing environments.

2.8.2 Configuration Management

Configuration management is handled through a layered approach that prioritizes customization while providing sensible defaults. The package creates a structured configuration directory hierarchy in the target filesystem at `/etc/wasmtime/` with appropriate subdirectories for modules and runtime settings. Default configuration files are provided with essential parameters pre-configured, but the system also supports custom configuration files placed in the package's files directory, allowing for site-specific deployments with minimal modification to the package itself. Alternatively, board-specific rootfs-overlays can be defined to overwrite the default configuration, providing flexibility for different deployment scenarios.

2.8.3 Service Integration

The package implements dual service management strategies to accommodate different init systems. For systemd-based targets, it provides comprehensive unit files that include proper service dependencies, ensuring that any prerequisite system services start before Wasmtime, and that the service properly restarts on failure with appropriate backoff intervals. For traditional SysV init systems, the package delivers init scripts with carefully planned execution order and includes thorough status checking and dependency verification to prevent cascading failures. Both service implementations include proper logging configurations to direct output to the standard system journals as well as component-specific log files, facilitating debugging and operational monitoring.

2.8.4 Build Process Flow

The build process follows a sequence of operations designed to maximize reliability and reproducibility. Upon selection in the Buildroot configuration, the package first determines the target architecture and selects the appropriate binary paths. The build system verifies the integrity of downloaded binary files using SHA-256 hashes to ensure authenticity. During the installation phase, binaries and configuration files are properly installed to their target locations, services are configured if enabled, and all necessary directories are created with appropriate permissions. The process includes comprehensive error checking to fail with descriptive messages if required components are missing or corrupted.

2.8.5 Security Considerations

Security is a core design principle evidenced throughout the package. The systemd service files implement best practice security measures including process isolation, capability restriction, and resource limiting to mitigate potential vulnerabilities. Binary integrity is enforced through cryptographic hash verification of all downloaded components. Configuration files are installed with restrictive permissions to prevent unauthorized modification. The package leverages Wasmtime's inherent sandboxing capabilities, allowing WebAssembly modules to execute in a secure, memory-safe environment with controlled access to system resources. The service management scripts perform runtime verification to ensure dependencies are properly met before components are started, preventing security vulnerabilities that might arise from improperly sequenced service initialization.

2.9 Oakestra Worker Node Package

2.9.1 Introduction and Architecture

The Oakestra Worker Node package is a Buildroot integration solution that enables deployment of Oakestra edge computing orchestration components on embedded Linux systems. Built for version 0.4.400, the package provides a streamlined method for installing both the NodeEngine and NetManager components, which together form the worker node portion of the Oakestra distributed computing platform. The package is architecturally designed to support both ARM64 (aarch64) and x86_64 (AMD64) platforms, ensuring broad compatibility across modern embedded computing hardware.

2.9.2 Configuration Management

Configuration management is handled through a layered approach that prioritizes customization while providing sensible defaults. The package creates a structured configuration directory hierarchy in the target filesystem at `/etc/oakestra/` with subdirectories for both the NodeEngine and NetManager components. Default configuration files are provided with essential parameters pre-configured, but the system also supports custom configuration files placed in the package's files directory, allowing for site-specific deployments with minimal modification to the package itself. Otherwise, it's possible to define board-specific rootfs-overlays in order to overwrite the default configuration.

2.9.3 Service Integration

The package implements dual service management strategies to accommodate different init systems. For systemd-based targets, it provides comprehensive unit files that include proper service dependencies, ensuring that NetManager starts before NodeEngine, and that both services properly restart on failure. For traditional SysV init systems, the package delivers init scripts with carefully planned execution order (S98 for NetManager, S99 for NodeEngine) and includes thorough status checking and dependency verification to prevent cascading failures. Both service implementations include proper logging configurations to direct output to the standard system journals as well as component-specific log files.

2.9.4 Build Process Flow

The build process follows a sequence of operations designed to maximize reliability and reproducibility. Upon selection in the Buildroot configuration, the package first determines the target architecture and selects the appropriate binary paths. During the build phase, it either processes the downloaded binary files or creates placeholder scripts if the binaries are unavailable. The build system also generates or copies configuration files to a staging directory. In the installation phase, binaries and configuration files are properly installed to their target locations, services are configured, and all necessary directories are created with appropriate permissions.

2.9.5 Security Considerations

Security is a core design principle evidenced throughout the package. The systemd service files implement best practice security measures including process isolation, capability restriction, and resource limiting. Binary integrity is enforced through cryptographic hash verification of all downloaded components. Configuration files are installed with restrictive permissions to prevent unauthorized modification. The service management scripts perform runtime verification to ensure dependencies are properly met before components are started, preventing security vulnerabilities that might arise from improperly sequenced service initialization.

3 Network Implementation and Configuration

The implementation focuses on adding network management support for WebAssembly (Wasm) runtimes in the Oakestra orchestration platform. This integration represents a significant advancement in enabling Wasm workloads to participate fully in distributed edge computing environments. The changes span two main repositories:

- The `oakestra` repository adds network setup/teardown functionality to the WebAssembly runtime in the node engine
- The `oakestra-net` repository implements the actual network management for Wasm modules

Associated Pull Requests:

- Oakestra Node Engine: `https://github.com/oakestra/oakestra/pull/397`
- Oakestra Network Manager: `https://github.com/oakestra/oakestra-net/pull/205`

3.1 Network Architecture

This section outlines the approaches for configuring network connectivity for WASI HTTP applications, with detailed implementation steps provided in Appendix B. We have implemented two primary approaches for networking WASI applications, each with different characteristics and use cases:

3.1.1 Approach 1: Simple veth Pair

This approach creates a network namespace with a single veth pair connecting it to the host. The WASI app can be reached inside the namespace via curl localhost:80 or using the assigned IP address. This is useful for quick testing of isolated network functionality.

3.1.2 Approach 2: Bridged veth Pair

This approach runs the WASI app in a namespace with a veth interface attached to a host bridge. The WASI app is assigned an IP on a dedicated subnet and can be reached from the host using that IP. This configuration is more suitable for production-like environments where the application needs to be accessible on the broader network.

3.1.3 hello-wasi-http

For testing and demonstrating network connectivity with WebAssembly, we utilize the `hello-wasi-http` example application. This serves as a lightweight HTTP server implemented as a WASM component using the WASI HTTP "proxy" world.

- Repository: `https://github.com/sunfishcode/hello-wasi-http`

Both approaches are implemented using Linux network namespaces, providing isolation while maintaining connectivity. The detailed configuration steps for each approach are provided in Appendix B.

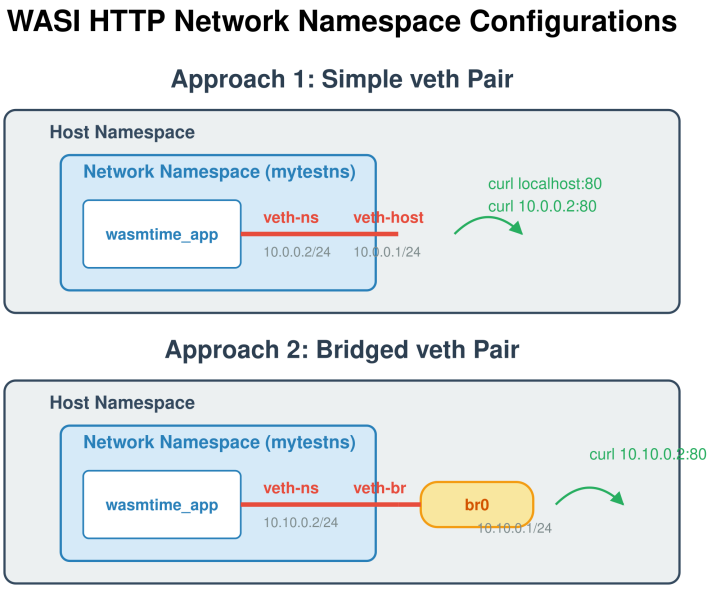


Figure 3.1 Network Configuration Diagram

3.1.4 Implementation Details

Oakestra-Net Changes (Network Manager)

- Added "wasm" as a new runtime type alongside containers and unikernels
- Created new files to handle Wasm networking:
  - `WasmNetDeployment.go`: Implements network setup for Wasm (similar to containers)
  - `WasmManager.go`: Provides HTTP endpoints for Wasm network operations
- The Wasm networking implementation:
  - Creates a network namespace for each Wasm module
  - Sets up virtual Ethernet (veth) pairs
  - Configures IPv4 and IPv6 addresses
  - Sets up routing and firewall rules
  - Manages port mappings

Oakestra Changes (Node Engine)

- Added new network management functions for Wasm in `NetManagerRequests.go`:
  - `CreateNetworkForWasm`: Sets up networking for a Wasm module
  - `DeleteNetworkForWasm`: Cleans up networking when a Wasm module terminates
- Enhanced `WasmManagement.go` to integrate with network overlay:
  - Added network setup before running Wasm modules
  - Added network cleanup in all error paths and after module completion
  - Added proper error handling for network operations

3.1.5 Technical Discussion of the Implementation

The WebAssembly networking solution leverages Linux network namespaces to provide network isolation without requiring the complete containerization of the process. This approach allows WebAssembly modules to have their own network stack while maintaining WebAssembly's native execution model.

The implementation is thorough in its handling of both IPv4 and IPv6 addressing, ensuring forward compatibility with modern networking requirements. Each WebAssembly module receives dynamically allocated addresses from the same pools used for other runtimes, maintaining consistency across the platform.

Particularly noteworthy is the careful attention to proper resource cleanup. The code includes multiple error handling paths that ensure network resources are reclaimed if anything goes wrong during setup or execution. This attention to detail prevents resource leakage that could otherwise accumulate and degrade system performance over time.

3.1.6 Comparing Runtime Networking Approaches

While the WebAssembly networking shares many characteristics with container networking in Oakestra, there are important distinctions stemming from WebAssembly's different execution model. Unlike containers, WebAssembly modules don't have associated process IDs that can be used for namespace binding. Instead, the implementation creates standalone network namespaces identified by service and instance names.

Another key difference is in the lifecycle management. For containers, network setup typically happens after container creation. In contrast, WebAssembly network setup occurs before module download, requiring more careful error handling throughout the subsequent operations.

These differences reflect the fundamental architectural differences between containerized and WebAssembly environments while maintaining consistent networking capabilities from the user's perspective.

4 Conclusion

This project designed and developed approaches for WebAssembly networking within embedded Linux devices using the Oakestra orchestration framework, though complete implementation testing remains blocked by ecosystem limitations. Our architecture leverages Linux namespaces to provide network isolation for WebAssembly modules without requiring full containerization, thus preserving Wasm's lightweight execution model while theoretically enabling proper network functionality.

The primary contributions of this work include:

- A lightweight embedded Linux distribution using Buildroot, optimized for WebAssembly workloads on resource-constrained devices
- Two effective network configuration approaches using veth pairs that enable both internal and cross-host communication
- Integration of WebAssembly networking capabilities within the Oakestra orchestration platform
- A comprehensive analysis of the current state of WASI networking and its implementation challenges

4.1 WebAssembly (WASM)

WebAssembly (WASM) is a binary instruction format designed as a portable compilation target for programming languages. It enables deployment on the web for client and server applications while ensuring high performance and security through sandboxed execution.

4.1.1 Modules

WebAssembly Modules are the original units of WASM code. They function as single, monolithic compilation units with limited interface capabilities that only support basic numeric types. These modules cannot easily share complex data structures between host and module, which limits interoperability.

4.1.2 Components

WebAssembly Components represent a newer architectural model built on top of modules. Using the WebAssembly Interface Type (WIT) language, components support richer type systems including strings, records, and variants. This enables better composition of WASM artifacts from different sources and languages, allowing for more sophisticated application architectures.

4.1.3 WASI Preview 1

The WebAssembly System Interface (WASI) standardizes how WASM accesses system resources. WASI Preview 1 is the initial version based on the module system, providing basic filesystem and clock access but intentionally excluding networking capabilities due to security concerns. Preview 1 is currently widely supported across WebAssembly runtimes.

4.1.4 WASI Preview 2

WASI Preview 2 represents a significant redesign built on the Component Model. It takes a modular approach with separate "worlds" for different functionality domains, including expanded capabilities for networking, HTTP, and cryptography. Preview 2 offers better interoperability between different language ecosystems but is still evolving with varying support across runtimes.

4.2 Our Implementation Status

Our current runtime implementation is based on WASI Preview 1, which lacks native networking capabilities. This creates significant limitations for our embedded system applications that require network functionality. For testing and advancing our implementation, we need the networking features available in WASI Preview 2, particularly its explicit network APIs through component-based "worlds" (wasi:sockets, wasi:http, and wasi:tls).

The technical assessment confirms that proper network functionality in WebAssembly fundamentally requires the Component Model to expose network interfaces in a standardized way. This dependency creates a technical gap in our implementation.

4.3 Implementation Blocker

Our progress is currently blocked by limitations in wasmtime-go, which does not support WASM Components or WASI Preview 2. This support is specifically blocked by pull request #9812 in the Bytecode Alliance repository. This limitation affects not only wasmtime-go but extends to most C API-based Wasmtime embeddings, indicating a broader ecosystem challenge rather than an isolated issue.

Without resolution of this PR, we cannot test the network functionality required for our applications using our current approach.

4.4 Alternative Approaches

Given these constraints, we have identified several alternative approaches:

4.4.1 TinyGo

TinyGo appears to be the most promising path forward for working with WASI Preview 2 components in Go. It offers compatibility with WASI Preview 2 components, with examples available in the wasmCloud/go repository. Using TinyGo, we could access network support in Preview 2, including HTTP connections, raw socket connections, and TLS.

4.4.2 Gravity

Arcjet's Gravity project provides another alternative, aiming for component support in pure Go. Based on Wazero (a zero dependency WebAssembly runtime for Go), this could potentially replace wasmtime-go for our component support needs.

4.4.3 wasi-http-go

The wasi-http-go project offers a more direct approach. While there are some implementation complexities with Preview 2 integration, it's functional and shows promise for HTTP-based applications in our context.

4.5 Future

Outside of these Go-specific alternatives, we've demonstrated how Linux namespaces can provide network isolation for WebAssembly modules without requiring full containerization. This hybrid approach preserves WASM's execution model advantages while enabling proper network functionality.

Moving forward, we must carefully evaluate the tradeoffs between waiting for official wasmtime-go support and adopting one of these community-driven alternatives based on our specific requirements and timeline constraints.

A Build on Raspberry

The following instructions for building a Raspberry Pi image using Buildroot are adapted and extended from the Buildroot documentation. For complete details and the most up-to-date information, please refer to the official Buildroot Raspberry Pi guide.

A.1 Default Configuration

There are several Raspberry Pi defconfig files in Buildroot, one for each major variant, which you should base your work on:

For models A, B, A+ or B+:

```
$ make raspberrypi_defconfig
```

For model Zero (model A+ in smaller form factor):

```
$ make raspberrypi0_defconfig
```

or for model Zero W (model Zero with wireless LAN and Bluetooth):

```
$ make raspberrypi0w_defconfig
```

For model Zero 2 W (model B3 in smaller form factor):

```
$ make raspberrypizero2w_defconfig
```

For model 2 B:

```
$ make raspberrypi2_defconfig
```

For model 3 B and B+:

```
$ make raspberrypi3_defconfig
```

or for model 3 B and B+ (64 bit):

```
$ make raspberrypi3_64_defconfig
```

For model 4 B:

```
$ make raspberrypi4_defconfig
```

or for model 4 B (64 bit):

```
$ make raspberrypi4_64_defconfig
```

For model CM4 (on IO Board):

```
$ make raspberrypicm4io_defconfig
```

or for CM4 (on IO Board - 64 bit):

```
$ make raspberrypicm4io_64_defconfig
```

A.2 Build the rootfs

Note: you will need to have access to the network, since Buildroot will download the packages' sources.

You may now build your rootfs with:

```
$ make
```

A.3 Result of the build

After building, you should obtain this tree:

```
output/images/
+-- bcm2708-rpi-b.dtb          [1]
+-- bcm2708-rpi-b-plus.dtb     [1]
+-- bcm2708-rpi-cm.dtb         [1]
+-- bcm2708-rpi-zero.dtb       [1]
+-- bcm2708-rpi-zero-w.dtb     [1]
+-- bcm2710-rpi-zero-2-w.dtb   [1]
+-- bcm2709-rpi-2-b.dtb        [1]
+-- bcm2710-rpi-3-b.dtb        [1]
+-- bcm2710-rpi-3-b-plus.dtb   [1]
+-- bcm2710-rpi-cm3.dtb        [1]
+-- bcm2711-rpi-4-b.dtb        [1]
+-- bcm2711-rpi-cm4.dtb        [1]
+-- bcm2837-rpi-3-b.dtb        [1]
+-- boot.vfat
+-- rootfs.ext4
+-- rpi-firmware/
|   +-- bootcode.bin
|   +-- cmdline.txt
|   +-- config.txt
|   +-- fixup.dat              [1]
|   +-- fixup4.dat             [1]
|   +-- start.elf              [1]
|   +-- start4.elf             [1]
|   `-- overlays/              [2]
+-- sdcard.img
+-- Image                      [1]
`-- zImage                     [1]
```

1

Not all of them will be present, depending on the RaspberryPi model you are using.

2

Only for the Raspberry Pi 3/4 Models (overlay miniuart-bt is needed to enable the RPi3 serial console otherwise occupied by the bluetooth chip). Alternative would be to disable the serial console in cmdline.txt and /etc/inittab.

A.4 How to write the SD card

Once the build process is finished you will have an image called "sdcard.img" in the output/images/ directory. Copy the bootable "sdcard.img" onto an SD card with "dd":

```
$ sudo dd if=output/images/sdcard.img of=/dev/sdX
```

Insert the SDcard into your Raspberry Pi, and power it up. Your new system should come up now and start two consoles: one on the serial port on the P1 header, one on the HDMI output where you can login using a USB keyboard.

A.5 Useful resources

- Creating a custom Linux distribution for the Raspberry Pi (with Buildroot and OpenEmbedded/Yocto)

B Network Configuration

This appendix provides detailed configuration steps for setting up network connectivity for WASI HTTP applications using Linux network namespaces.

B.1 Prerequisites

Before configuring network for WASI HTTP applications, ensure you have the following prerequisites:

- A Linux distribution with support for network namespaces.
- iproute2 (for ip commands)
- Wasmtime 18.0+ installed.
- cargo-component installed (v0.13.2+)
- The hello-wasi-http repository cloned and built.

Building the WASM component:

```
cd hello-wasi-http
cargo component build
# The component is built at:
# target/wasm32-wasip1/debug/hello_wasi_http.wasm
```

B.2 Approach 1: Simple veth Pair (Connectivity Inside Namespace)

This approach creates a network namespace and connects it to the host using a single veth pair. This is useful if you want to test whether networking "just works" inside a namespace (e.g. using curl localhost:80).

B.2.1 Steps

1. **Create a New Network Namespace**

   ```
   sudo ip netns add mytestns
   ```

2. **Create a veth Pair**

   - One end remains in the host (named veth-host).
   - The other end will be moved to the namespace (named veth-ns).

   ```
   sudo ip link add veth-host type veth peer name veth-ns
   ```

3. **Move the Namespace End into the Network Namespace**

   ```
   sudo ip link set veth-ns netns mytestns
   ```

4. **Assign IP Addresses and Bring Up Interfaces**

- **Host side:**

```
sudo ip addr add 10.0.0.1/24 dev veth-host
sudo ip link set veth-host up
```

- **Inside the namespace:**

```
sudo ip netns exec mytestns ip addr add 10.0.0.2/24 dev veth-ns
sudo ip netns exec mytestns ip link set veth-ns up
sudo ip netns exec mytestns ip link set lo up
```

5. **(Optional) Enable Outbound Connectivity (if your WASI app makes external calls)**
   If your WASI app (like hello-wasi-http) makes outbound HTTP requests, set up a default route and NAT:

```
sudo ip netns exec mytestns ip route add default via 10.0.0.1
sudo sysctl -w net.ipv4.ip_forward=1
sudo iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -o eth0 -j MASQUERADE
```

6. **Run the WASI HTTP App in the Namespace**
   Since the hello-wasi-http component uses the WASI HTTP "proxy" world, use wasmtime serve (with the default listening address 0.0.0.0:8080 or change with –addr if needed):

```
sudo ip netns exec mytestns wasmtime serve \
  --addr 0.0.0.0:80 \
  target/wasm32-wasip1/debug/hello_wasi_http.wasm
```

   Tip: If you previously used older flags like –tcplisten, note that they are not supported with wasmtime serve.

7. **Test the Setup**

   - Inside the namespace:

```
sudo ip netns exec mytestns curl http://localhost:80
```

   - Or from the host (using the assigned IP):

```
curl http://10.0.0.2:80
```

B.3 Approach 2: Bridged veth Pair (Connectivity via a Host Bridge)

In this approach the WASI app's network interface is attached to a Linux bridge. This allows the namespace to use a different subnet (e.g. 10.10.0.x) and be accessible via that IP.

B.3.1 Steps

1. **Clean Up (if needed)**

```
sudo ip netns del mytestns 2>/dev/null
sudo ip link del veth-br 2>/dev/null
sudo ip link del br0 2>/dev/null
```

2. **Create a New Namespace**

```
sudo ip netns add mytestns
```

3. **Create a veth Pair**
   - Host side: `veth-br`
   - Namespace side: `veth-ns`

   ```
   sudo ip link add veth-br type veth peer name veth-ns
   ```

4. **Create a Bridge in the Host**

   ```
   sudo ip link add br0 type bridge
   sudo ip link set br0 up
   ```

5. **Move the Namespace End to the Network Namespace**

   ```
   sudo ip link set veth-ns netns mytestns
   ```

6. **Attach the Host-End to the Bridge and Bring It Up**

   ```
   sudo ip link set veth-br master br0
   sudo ip link set veth-br up
   ```

7. **Assign IP Addresses**
   - **On the bridge in the host:**

     ```
     sudo ip addr add 10.10.0.1/24 dev br0
     ```

   - **Inside the namespace (on veth-ns):**

     ```
     sudo ip netns exec mytestns ip addr add 10.10.0.2/24 dev veth-ns
     sudo ip netns exec mytestns ip link set veth-ns up
     sudo ip netns exec mytestns ip link set lo up
     ```

8. **Test Connectivity**
   - From the host:

     ```
     ping -c 3 10.10.0.2
     ```

   - From the namespace:

     ```
     sudo ip netns exec mytestns ping -c 3 10.10.0.1
     ```

9. **Run the WASI HTTP App in the Namespace**
   For example, to bind the WASI HTTP server on port 80:

   ```
   sudo ip netns exec mytestns wasmtime serve \
     --addr 0.0.0.0:80 \
     target/wasm32-wasip1/debug/hello_wasi_http.wasm
   ```

10. **Test the WASI HTTP Service**
    From the host, use:

    ```
    curl http://10.10.0.2:80
    ```

    You should see the response from the WASI HTTP proxy, for example:

    ```
    Hello, wasi:http/proxy world!
    ```

B.4 Troubleshooting Tips

- **RTNETLINK "File Exists" Errors:**
  These errors typically indicate an interface is already assigned or a naming conflict exists. Use ip link show and sudo ip netns exec mytestns ip link show to inspect which interfaces are present. If needed, delete the interfaces and namespace with:

  ```
  sudo ip netns del mytestns
  sudo ip link del veth-br 2>/dev/null
  sudo ip link del br0 2>/dev/null
  ```

  Then restart the steps.

- **No Route to Host Errors:**
  If ping or curl returns "No route to host," verify that only one IP is assigned per interface and the routes are set correctly. In Approach 2, avoid assigning conflicting IPs (for example, do not assign both 10.0.0.x and 10.10.0.x to the same interface).

- **Wasmtime Flags:**
  Use wasmtime serve with –addr to specify the listening address. The older –tcplisten flag is not valid for components running in the WASI HTTP "proxy" world.