Networking

TCP/UDP sockets &
Domain name lookup

https://github.com/badeend/WASI-Networking

Agenda

- Quick overview of proposal
  - Goals
  - Challenges
  - API’s
- Next steps

FYI

This presentation is just a quick overview. More details available in repo. Feel free to take a look!

Code snippets will be Rust-*ish*.

I'm going to assume basic knowledge of the BSD sockets API.

Please interrupt!

Goals

Goal: "MVP"

Start with the bare minimum amount of APIs that are required to create a basic functioning TCP/UDP application.

Goal: Portable

Windows, Linux, FreeBSD, Mac OS, etc…

Though this is mostly a given, given the initial set of APIs.

Goal: POSIX-compatibility

A POSIX compatible layer should be able to be built on top of the proposal in wasi-libc. Heads up: the proposal itself is *not* POSIX compatible.

Challenges

Challenge: Genericity

BSD sockets interface is highly generic. The same functions have different semantics depending on which type of socket they're called on. Man-pages are riddled with conditional documentation.

Challenge: Genericity

BSD sockets interface is highly generic. The same functions have different semantics depending on which type of socket they're called on. Man-pages are riddled with conditional documentation.

Solution?

- Provide one function returning the kind of socket, that can be called on all sockets.
  - wasi-libc can then use this return value to dispatch calls to the correct wasi module. (examples in draft text.)
- Push all other socket functions into protocol-specific modules.
  - This enables every socket function signature to be tailored to the needs of a specific protocol. Removing a lot of cruft, optional parameters and edge-cases.
  - Each protocol can be a standalone WASI module, and progress the standardization process independently.
- Con: Not all functions differ per protocol (bind, local_address, connect, ...), yet those are currently duplicated as well.
- Con: many switch-statements in wasi-libc.

Challenge: Security

Challenge: Security — Capability handles

Maybe kicking in an open door, but:

Capability handles! 🪄

```
fn create_udp_socket(cap: &UdpCapableNetwork, address_family: IpAddressFamily);
fn create_tcp_socket(cap: &TcpCapableNetwork, address_family: IpAddressFamily);
```

Challenge: Security — Deny by default

Having possession of a capability handle should not mean having keys to the entire kingdom/internet!

Even with capability handles, WASI implementations should deny all network access by default. Access should be granted at the most granular level possible. The proposal does not dictate how this must happen, but one can image: ->

Challenge: Security — Deny by default (cont.)

```
--allow-resolve=example.com            # Allow the lookup of a specific domain name
--allow-resolve=*.example.com          # Allow the lookup of all subdomains
--allow-resolve=*                      # Allow any lookup

--allow-outbound=tcp://127.0.0.1:80    # Allow TCP connections to 127.0.0.1 on port 80
--allow-outbound=tcp://127.0.0.1:*     # Allow TCP connections to 127.0.0.1 on any port
--allow-outbound=tcp://*:80            # Allow TCP connections to any server on port 80

--allow-outbound=tcp://example.com:80  # Allow connections to any IP address resolved from example.com
                                       # on port 80. This also implies `--allow-resolve=example.com`

--allow-inbound=tcp://localhost:80     # Allow listening only on loopback interfaces on port 80
```

In these examples the capability handle is basically a firewall.

APIs: Sockets

```
mod socket {
  trait Socket : Handle {
    fn protocol(&self);
  }
}
```

```
mod socket_udp {
  trait UdpCapableNetwork {
    // `socket(AF_INET or AF_INET6, SOCK_DGRAM, 0)`
    fn create_udp_socket(&self, address_family);
  }

  trait UdpSocket : Socket {
    fn bind(&self, local_address);
    fn local_address(&self);
    fn receive(&self, iovs);
    fn peek(&self, iovs);
    fn send(&self, iovs, options);
    fn connect(&self, remote_address);
    fn remote_address(&self);
  }
}
```

```
mod socket_tcp {
  trait TcpCapableNetwork {
    // `socket(AF_INET or AF_INET6, SOCK_STREAM, 0)`
    fn create_tcp_socket(&self, address_family);
  }

  trait TcpSocket : Socket
                  + InputByteStream
                  + OutputByteStream {
    fn bind(&self, local_address);
    fn local_address(&self);
    fn connect(self, remote_address);
    fn listen(self, backlog_size_hint);
    fn accept(&self);
    fn peek(&self, iovs);
    fn remote_address(&self);
    fn shutdown(&self, shutdown_type);
  }
}
```

APIs: Address lookup

Significantly simpler than `getaddrinfo`:

- Input: only domain names. Not serialized IP addresses.
- Output: only IP addresses.

```
mod ip_resolve_address {
  trait IpAddressResolver {
    fn resolve_addresses(&self, name, options) -> Result<Vec<IpAddress>, errno>;
  }
}
```

This function is a major departure from `getaddrinfo`. The dissimilar name is chosen to reflect this.

APIs: Address lookup – Why not getaddrinfo?

`getaddrinfo` is very generic and multipurpose by design. This proposed WASI module is *not*. This module focuses strictly on translating internet domain names to ip addresses. That eliminates many of the other "hats" getaddrinfo has (and potential security holes), like:

- Mapping service names to port numbers ("https" -> 443)
- Mapping service names/ports to socket types ("https" -> SOCK_STREAM)
- Network interface name translation (%eth0 -> 1)
- IP address deserialization (“127.0.0.1” -> Ipv4Address(127, 0, 0, 1))
- IP address string canonicalization (“0:0:0:0:0:0:0:1” -> “::1”)
- Constants lookup for INADDR_ANY, INADDR_LOOPBACK, IN6ADDR_ANY_INIT and IN6ADDR_LOOPBACK_INIT

Many of these functionalities can be shimmed in the libc implementation. But some require future WASI additions.

APIs: Address lookup

TBD: Add the inverse of `resolve_addresses`? Eg `resolve_name` (based on `getnameinfo`). Although they would be each other's mirror image, their use cases are very different. `resolve_addresses` is used by almost every application that is connected to the internet, while its inverse `resolve_name` is very niche.

Next steps

Next steps: Split

My suggestion is to split the current draft up into 4 separate modules/proposals:

- socket (that one function)
- socket_tcp
- socket_udp
- ip_resolve_address

Next steps: ?

-