Towards sockets and networking in WebAssembly and WASI

Radu Matei

October 16, 2020

As more compilers add support for emitting Wasm (the Kotlin community officially announced starting the work on a compiler backend based on the WebAssembly garbage collection proposal, and there is an ongoing effort to support an official Wasm target for Swift, besides already being able to execute Rust, Go, AssemblyScript, C#, Zig, and others in the Wasm runtimes), and as Wasmtime implements more and more WebAssembly proposals, one of main scenarios not yet enabled by WASI (the WebAssembly System Interface) is networking applications.

In this article we look at the current state of the networking API in WASI, and add a minimal implementation together with socket clients in AssemblyScript and Rust (with an upfront disclaimer that this should definitely not be used for anything other than experimentation).

Ongoing work to add socket support to WASI

For an overview of how the WASI API is defined, how to add a new API to WASI, an implementation to Wasmtime, and how to use the new API from a module, follow this article I wrote back in March. In a nutshell, the WASI API is declared using `witx`, an experimental file format based on the WebAssembly Text Format, with added support for module types and annotations. The WITX files are used to automatically generate Rust bindings, which are then implemented by Wasmtime. This is an excellent low-level way of defining the WASI API, which means targeting WASI from a new programming language becomes a matter of implementing the layer defined in WITX.

The current WASI snapshot actually contains a few methods for working with sockets: `sock_recv`, `sock_send`, and `sock_shutdown` - they are not quite enough for complete networking support, and they are unimplemented in Wasmtime. However, there is an open pull request which aims to extend the current sockets API in WASI to include the complete Berkeley Sockets API, which so far received quite positive feedback (and with a few changes and clarifications, hopefully it will be merged soon - fingers crossed!) After it gets merged, a Wasmtime

implementation is next, followed by language toolchain support (for example, the Rust standard library's `wasm32-wasi` target). But since official releases that contain socket support will take some time, in this article we explore adding a minimal implementation that allows starting client connections from WebAssembly in WASI runtimes.

The following sections build on multiple ongoing efforts in the community, such as the current PR to add sockets support to WASI, its work-in-progress implementation in Wasmtime, an excellent blog post by Kawamura Yuto adding network support for Decanton, and enabled by the work of all the awesome people in the WASI community and Bytecode Alliance.

Exposing the API for a minimal socket client in WASI and Wasmtime

Adding support for connecting to sockets would allow guest modules running in WASI runtimes to create TCP streams, enabling connections to web servers, databases, or message queues - the guest module attempts to initiate the connection, and if it has enough capabilities, the WASI runtime would start the connection, then send and receive buffered data to and from the module.

It turns out the only missing API in WASI for a socket client is `connect` - so we add a new `sock_connect` function definition to the WASI snapshot. The WITX snippet below adds the `sock_connect` function to WASI (a robust definition and implementation would also add a corresponding `rights::sock_connect` entry to the file descriptor rights to describe the capability of a module to connect to a socket, which is the approach taken by the current proposal):

```
diff --git a/phases/snapshot/witx/wasi_snapshot_preview1.witx
b/phases/snapshot/witx/wasi_snapshot_preview1.witx
index 5604c3e..4c356c6 100644
--- a/phases/snapshot/witx/wasi_snapshot_preview1.witx
+++ b/phases/snapshot/witx/wasi_snapshot_preview1.witx
@@ -491,6 +491,22 @@

+  ;;; Directly connect to a socket.
+  ;;;
+  ;;; This is a temporary workaround that contradicts the
+  ;;; philosophy of WASI, but which is necessary for enabling
+  ;;; an entire suite of networking workloads.
+  ;;;
+  ;;; As the sockets proposal is adopted, this should be
+  ;;; entirely removed and replaced with that proposal.
+  ;;;
+  ;;; See https://github.com/WebAssembly/WASI/pull/312
+  (@interface func (export "sock_connect")
+    (param $ipv4_addr u32)
+    (param $port u16)
```

```
+    (result $error $errno)
+    (result $sock_fd $fd)
+  )
```

As the comment suggests, this method contradicts WASI's principle of least authority - specifically, all modules will be allowed to create connections, which is not ideal, and the main reason why this *should not be used for anything else other than experimentation*. For a much more robust and secure API, see the current sockets proposal described earlier.

The `sock_connect` method takes an IP address and port as arguments, and returns the socket's file descriptor back to the client - so its Rust implementation follows the signature defined in WITX, directly starts a TCP connection to the desired IP address and port, and retains a handle for the socket's file descriptor:

```
fn sock_connect(
    &self,
    ipv4_addr: u32,
    port: u16,
) -> Result<types::Fd> {
    use std::net::{Ipv4Addr, SocketAddrV4, TcpStream};
    let addr = SocketAddrV4::new(Ipv4Addr::from(ipv4_addr), port);
    println!("wasi_snapshot_preview1::sock_connect to addr {:#?}", addr);
    let stream = TcpStream::connect(addr)?;
    let handle: Box<dyn crate::handle::Handle> =
        Box::new(SocketHandle(std::cell::RefCell::new(stream)));
    let entry = Entry::new(EntryHandle::from(handle));
    self.insert_entry(entry)
}
```

Next, `sock_recv` and `sock_send` are implemented in a similar manner - get the socket's file descriptor, data buffers, and flags as arguments, and either read from the buffer, or write into it:

```
fn sock_recv(
    &self,
  fd: types::Fd,
  ri_data: &types::IovecArray<'_>,
    _ri_flags: types::Riflags,
) -> Result<(types::Size, types::Roflags)> {
  use std::convert::TryFrom;
  use std::io::IoSliceMut;
  let mut bufs = Vec::with_capacity(ri_data.len() as usize);
  for iov in ri_data.iter() {
      let iov = iov?;
      let iov: types::Iovec = iov.read()?;
      let buf = iov.buf.as_array(iov.buf_len).as_slice()?;
      bufs.push(buf);
```

```
    }
    let mut iovs: Vec<_> = bufs.iter_mut()
                    .map(|s| IoSliceMut::new(&mut *s)).collect();
    let total_size = self
        .get_entry(fd)?
        .as_handle(&HandleRights::empty())?
        .read_vectored(&mut iovs)?;
    println!(
        "wasi_snapshot_preview1::sock_recv: {} bytes written",
        total_size
    );

    Ok((total_size as u32, types::Roflags::try_from(0)?))
}
```

Here, the `rights::fd_read` and `rights::fd_write` rights (describing the capabilities of a module to read from and write into sockets) should also be checked. Besides this, additional rights (describing the address pool a module is allowed to connect to, read from, and write to) would also have to be added - this would require more substantial changes to Wasmtime, which for the purpose of this article will not be explored.

> You can find a fork of Wasmtime with these changes on GitHub.

Building from this branch, we should now have a version of Wasmtime that exposes starting connections, as well as sending and receiving on sockets.

Writing a WASI socket client in AssemblyScript

AssemblyScript is a strict variant of TypeScript which natively compiles to WebAssembly. While it retains significant portions of the TypeScript syntax, it is a fundamentally different language, with a different compiler and standard library.

AssemblyScript's standard library contains the *bindings* for WASI - and we need to add the definition for `sock_connect` (which is the only API missing from the current WASI snapshot):

```
diff --git a/std/assembly/bindings/wasi_snapshot_preview1.ts
b/std/assembly/bindings/wasi_snapshot_preview1.ts
index 2470e06c..b206a20a 100644
--- a/std/assembly/bindings/wasi_snapshot_preview1.ts
+++ b/std/assembly/bindings/wasi_snapshot_preview1.ts
@@ -531,6 +531,34 @@

+/** See https://github.com/WebAssembly/WASI/pull/312*/
+// @ts-ignore: decorator
+@unsafe
```

```
+export declare function sock_connect(sock: fd, ipv4: u32, port: u16): errno;
```

You can find a fork of AssemblyScript with these changes on GitHub.

While the bindings for WASI are defined in the standard library, the actual implementation is currently a separate project which provides a layer for WASI system calls - `as-wasi`. The project already implements functionality such as reading and writing from files, so adapting reading and writing from sockets is straightforward, since both are modeled using file descriptors.

Let's add a new `Socket` class to `as-wasi` with a single member, the socket's file descriptor:

```
diff --git a/assembly/as-wasi.ts b/assembly/as-wasi.ts
index b8c844e..a66bdb6 100644
--- a/assembly/as-wasi.ts
+++ b/assembly/as-wasi.ts

@@ -771,6 +777,94 @@

+@global
+export class Socket {
+  fd: Descriptor;
+ }
```

In order to connect to a socket, we call WASI's `sock_connect` with the IP address and port desired, and store the file descriptor returned by Wasmtime:

```
connect(ipv4: u32, port: u16): void {
  let fd_buf = memory.data(8);
  let res = sock_connect(ipv4, port, fd_buf);
  if (res !== errno.SUCCESS) {
    Console.write("as_wasi::socket::connect: error: " + res.toString());
    abort()
  }

  this.fd = new Descriptor(load<u32>(fd_buf));
}
```

An implementation that writes an AssemblyScript string to the socket UTF-8 encodes it (and its length) to a scatter-gather memory block and calls `sock_send`. Receiving is analogous, this time *reading* from a scatter-gather memory block and decoding into AssemblyScript strings:

```
write(data: string): void {
  let s_utf8_buf = String.UTF8.encode(data);
  let s_utf8_len: usize = s_utf8_buf.byteLength;
  let iov = memory.data(16);
  store<u32>(iov, changetype<usize>(s_utf8_buf));
  store<u32>(iov, s_utf8_len, sizeof<usize>());
```

```
  let written_ptr = memory.data(8);
  sock_send(this.fd.rawfd, iov, 1, 0, written_ptr);
}
```

You can find a fork of `as-wasi` with these changes on GitHub.

Now we can write a WebAssembly guest module in AssemblyScript that uses the high level socket API we just implemented. The only thing to note here is that since `sock_connect` takes in the integer value of the IP address, we have to manually convert it from `127.0.0.1` (you can read about the conversion here). This is because our WASI API is lacking DNS resolution to translate from a URL to an IP address (which is left as exercise for the reader):

```
import { Console, Socket } from "as-wasi";

export function _start(): void {
  let s = new Socket();
  // 127.0.0.1:3333
  s.connect(2130706433, 3333);

  s.write("writing to an echo server...");
  let res = s.receive();
  Console.write(res);
}
```

Assuming there is an echo server running on `localhost:3333` (there is a simple C socket server you can find in this `as-wasi` fork that can be used), we should be able to send and receive data:

```
$ asc test/sockets.ts -t test/sockets.wat --use abort=wasi_abort
$ wasmtime test/sockets.wat

wasi_snapshot_preview1::sock_connect to addr 127.0.0.1:3333
wasi_snapshot_preview1::sock_send: 29 bytes written
wasi_snapshot_preview1::sock_recv: 29 bytes written
wasi_snapshot_preview1::sock_recv: 0 bytes written
writing to an echo server...
```

At this point, the socket can be used for sending and receiving data using any protocol that works on top of sockets (with the mention that while our implementation only sends and receives strings, it can manipulate arbitrary data as well) - for example, if we start a static HTTP file server on the same port:

```
$ echo "this is a file that will be served by an
        HTTP server to a WebAssembly module" > file.txt
$ ls
.rw-r--r-- 77 radu file.txt
$ http-server --port 3333
```

```
Starting up http-server, serving ./
Available on:
  http://127.0.0.1:3333
Hit CTRL-C to stop the server
```

And modify our AssemblyScript source code to make a `GET` request for `file.txt`:

```
import { Console, Socket } from "as-wasi";

export function _start(): void {
  let s = new Socket();
  // 127.0.0.1:3333
  s.connect(2130706433, 3333);
  s.write("GET /file.txt HTTP/1.1\r\n\r\n");
  let res = s.receive();
  Console.write(res);
}
```

We can see the response from the HTTP server:

```
$ asc test/sockets.ts -t test/sockets.wat --use abort=wasi_abort
$ wasmtime test/sockets.wat

wasi_snapshot_preview1::sock_connect to addr 127.0.0.1:3333
wasi_snapshot_preview1::sock_send: 27 bytes written
wasi_snapshot_preview1::sock_recv: 365 bytes written
wasi_snapshot_preview1::sock_recv: 0 bytes written

HTTP/1.1 200 OK
server: ecstatic-3.3.2
cache-control: max-age=3600
last-modified: Oct 2020 ... GMT
content-length: 77
content-type: text/plain; charset=UTF-8
Connection: keep-alive

this is a file that will be served by an HTTP server to a WebAssembly module
```

Going from manually creating HTTP requests and printing HTTP responses to an actual HTTP client is a matter of following the HTTP specification (and is also left as an exercise for the reader).

Rust TCP streams on top of WASI sockets

Kawamura Yuto's blog post does a fantastic job of introducing the required steps for patching the Rust standard library to use the new WASI bindings and implement `TcpStream` on top of the WASI socket API. In short, we have to take a similar approach to the one in AssemblyScript: update the bindings to

include the `sock_connect` API, and add use `sock_connect`, `sock_send`, and `sock_receive` in order to send and receive data.

These steps result in a forked Rust standard library (together with a stage 2 compiler build) that can be used to compile Rust programs - for example, attempting to execute the same request for `file.txt`:

```
use std::io::{Read, Write};
use std::net::TcpStream;
use std::str::from_utf8;

#[no_mangle]
pub unsafe extern "C" fn _start() {
  println!("wasm::_start: attempting to make a TCP stream to localhost:3333");
  match TcpStream::connect("localhost:3333") {
      Ok(mut stream) => {
          println!("wasm::_start: successfully connected to server in port 3333");

          let msg = b"GET /file.txt HTTP/1.1\r\n\r\n";
          stream.write(msg).unwrap();

          let mut data = [0 as u8; 365];
          match stream.read_exact(&mut data) {
              Ok(_) => {
                  let text = from_utf8(&data).unwrap();
                  println!("wasm::_start: received {}", text);
              }
              Err(e) => {
                  println!("wasm::_start: failed to receive data: {}", e);
              }
          }
      }
      Err(e) => {
          println!("wasm::_start: failed to connect: {}", e);
      }
  }
  println!("wasm::_start: exit");
}
```

The important thing to note here is that this is using (a fork of) the Rust standard library directly:

```
$ rustup run wasi32-sockets rustc --target wasm32-wasi --crate-type=cdylib
-C linker=build/x86_64-apple-darwin/lld/bin/lld src/lib.rs
$ wasmtime lib.wasm

wasm::_start: attempting to make a TCP stream to localhost:3333
rust_stdlib_sys_wasi_net::TcpStream::connect
```

```
wasi_snapshot_preview1::sock_connect to addr 127.0.0.1:3333
wasm::_start: successfully connected to server in port 3333
wasi_snapshot_preview1::sock_send: 26 bytes written
wasm::_start sent request, awaiting reply.
wasi_snapshot_preview1::sock_recv: 365 bytes written
wasm::_start: received

HTTP/1.1 200 OK
server: ecstatic-3.3.2
cache-control: max-age=3600
last-modified: Oct 2020 ... GMT
content-length: 77
content-type: text/plain; charset=UTF-8
Connection: keep-alive

this is a file that will be served by an HTTP server to a WebAssembly module

wasm::_start: exit
```

You can find a fork of Rust with these changes on GitHub.

What about socket listeners and servers ?

So far, this article exclusively considered socket clients - partly because they are easier to implement. Implementing a server requires additional API functions exposed - specifically, `sock_bind`, `sock_listen`, and `sock_accept`, followed by reading and writing. And while implementing them is done similarly to what was described here so far, there is an underlying issue: there is no multi-threading support in WebAssembly - which means that a server running in WebAssembly is either going to be single-threaded, or its implementation would have to be significantly more complex (see Node's event loop). At the same time, there is a WebAssembly threads proposal that would define operations for handling atomic memory access across threads, but it is only at stage 2 of the Wasm standardization process.

While actually creating a server within a Wasm module might not be feasible in the near future (see limitations above), it doesn't mean that WebAssembly modules cannot be used to handle HTTP requests, for example - deferring the server creation and threading to the underlying host, and passing request data to the module (which can now create client connections on its own) could be a reasonable approach (this has the benefit of not compiling the entire HTTP stack for simple request handlers), and could enable really exciting projects in the future.

Ongoing WebAssembly proposals (such as SIMD, threads, garbage collection, or interface types), together with WASI's capability-oriented API and the vision for nanoprocesses, have the potential to make server-side Wasm a true contender

in the cloud native ecosystem, and WASI's networking proposal is one of the main enablers of this.