Neil Kolban
kolban1@kolban.com
2015-09-24

Espruino Networking Internals

To begin understanding Espruino networking you have to have some background in TCP/IP and especially the "sockets" API. TCP/IP is a protocol specification and sockets is a specific implementation (with API) for programming to TCP/IP. We can't describe the theory of TCP/IP or sockets here but will make some notes as a refresher.

An IP address is a 32 bit value that identifies a host on the network. It is commonly written in dotted decimal notation such as "192.168.1.2" where each decimal represents 8 bits (a byte) of the address.

A port is a 16 bit value that identifies an endpoint of a communication on a host. Only one application at a time can "own" the given port on a given host. The pair of IP address and port specify the endpoint of a TCP connection.

TCP is a connection oriented protocol that, when formed, allows both ends to send and receive simultaneously. There is a second protocol called UDP which is datagram oriented that will not be discussed here as it is not yet implemented in Espruino code.

In order to form a connection between two applications, one side of the conversation has to be listening for incoming connection requests and the other side will initiate a connection request. This means that the coding of TCP is asymmetric. At any given point, one side will be a listener and the other an initiator. To use the proper terms, the application that is listening for an incoming connection is called a "server" and the application that initiates a connection is called a "client".

The sockets API provides both functions and a model of operation. The core concept is the notion of the object known as a "socket". A socket represents one endpoint of a conversation. The implementation and internals of a socket are always considered black box and the socket as seen by the programmer is only treated as a handle. Commonly, it is an integer value which one assumes is internally mapped to all its detailed content.

In classic sockets API, a server application will perform the following steps:

```
int s = socket();      // Create the empty socket.
bind(s, port number); // Bind the socket to a local port number
listen(s);             // Start accepting incoming requests on the bound port
int s2 = accept(s);    // Block waiting for client to connected
```

There is a lot of stuff here but the key point to note is that there is a "server socket" that is used to describe what we are listening upon and there is a "partner socket" that is returned from accept() when a client connects. So a server application will always have a server socket and may have zero or more partner sockets created depending on how may clients are connected.

A client application will have code similar to the following:

```
int s = socket();                  // Create the empty socket
connect(s, address, port number); // Connect to the server
```

When the client connects there will then be a connection between the two applications and each side will have a socket that corresponds to its end of the conversation. Either end can send or receive data.

There is a socket API called send() which sends data and a blocking API called recv() which receives data.

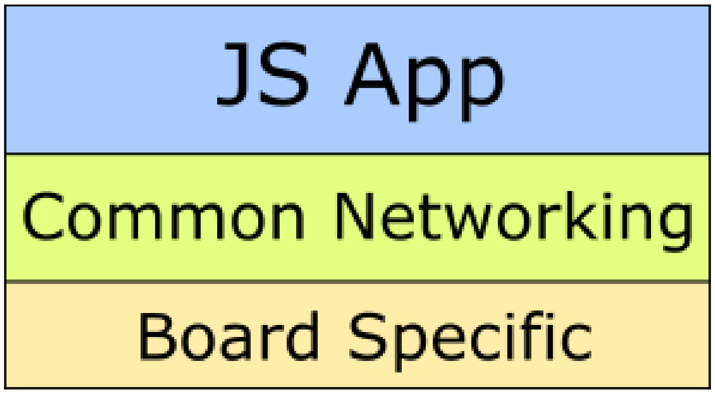
Sockets can be closed using the close() API. Closing one end of a connection causes the closure of the other end of the connection.

The JavaScript language has no inherent standards or implementations for networking. It is left to libraries to choose to implement their own stories. A JavaScript library can choose to model itself on sockets or can choose a different path … perhaps one more appropriate to the nature of JavaScript principles.

Espruino is able to run on a variety of boards and each board has its own story for networking. This means that the semantics and APIs available for networking in the C programming language vary from board to board. Since Espruino runs on a variety of boards, how can we accommodate this?

A brute force approach would be to have completely separate implementations of networking for each board type. However this is a dangerous path. It could quickly lead to dramatically different styles of end user programming in JavaScript based upon the board being used. It would also require duplication of common "algorithms and techniques" for each board that would appear simply not to be needed.

As such, an abstraction layer has been introduced:



A JS App codes to an Espruino architected model of networking and the implementation of that common model invokes board specific implementation functions. For this to work, when we exchange one board specific implementation for another, the interfaces exposed by the board must be the same. What this means is that a board specific networking implementation must expose a "contract" that both sides agree upon.

Espruino has architected that contract to be the following C language interfaces:

- `int createSocket(struct JsNetwork *net, uint32_t host, unsigned short port)`
- `int send(struct JsNetwork *net, int sckt, const void *buf, size_t len)`
- `int recv(struct JsNetwork *net, int sckt, void *buf, size_t len)`
- `int accept(struct JsNetwork *net, int sckt)`
- `void closeSocket(struct JsNetwork *net, int sckt)`

- `void idle(struct JsNetwork *net)`
- `bool checkError(struct JsNetwork *net)`
- `void getHostByName(struct JsNetwork *net, char *hostName, uint32_t *outIp)`

It is important to note that these functions exposed by the board specific interface do **not** have to have these specific names but they must accept the identical arguments and honor the correct semantics.

Before we go into the details of the semantics, it is important to understand the "socket". In Espruino, a socket is a "handle" to a connection. It is implemented as an integer but no-one should ascribe any meaning to its value other than as the key to the internal data. This allows the generic Espruino common networking to isolate itself from any board specific data. Saying this again, think of it that any particular board may need to maintain a bunch of data items related to any one connection. Since these are distinct by board, we use this "generic" socket integer to refer to this collection of board specific private data.

For example ... when the common networking portion says "Create me a socket", what will be returned by the board specific code will be an integer "handle" (the socket handle). Later on, if the common networking wishes to refer to that previously created connection, it will pass in the returned socket handle. The board specific components are responsible for mapping to and from socket handles to the board specific data.

Networking is, by its nature, a non-instantaneous set of activities. When we wish to connect to a partner over TCP/IP, we may find that it only takes a few tens of milliseconds but in computing terms, this is an eternity. This introduces a distinction between blocking and non-blocking calls. A call which is blocking is asked to perform a function and not return until it has completed. This "blocks" the caller until the work is complete. A "non-blocking" call is one where the caller asks for work to be performed and control is returned immediately (or as soon as possible) with the result being available some time later. This is also known as asynchronous processing. Some boards implement networking in a blocking fashion while others implement networking in a non-blocking/asynchronous fashion. We will find that this will complicate our story.

In the following, when we see the symbol:

That means that there is a discussion on asynchronous processing.

Now we will start to take apart the semantics of each of these interfaces and what they mean.

createSocket

Signature:

```
int createSocket(struct JsNetwork *net, uint32_t host, unsigned short port)
```

- `net` – Unknown
- `host` – The remote host to connect with or 0 to be a local server

- `port` – The port number to listen upon or connect to

This call is responsible for creating a socket that is associated with a particular host and port. Since we have already seen that there are two fundamental uses for a socket (as a client caller or as a server listener), we get to specify that through the host IP address. If its value is 0 then that is the indication that we are to be a server. If its value is not zero, then we are being a client.

Note: There is some thought that an IP address of `0xFFFFFFFF` may also be special.

When the socket is to be a server, the port number supplied is the port that the server will listen upon. When the socket is to be a client, the port number is the port of the remote host application.

The return value is the identity of the socket. This is the handle that will be passed back to further board specific calls. The board implementation will map socket handles to internal data on these further calls.

Question: Can this return an error indication ... for example ... -1?

Let us split up our discussion on more detailed semantics into two parts … one for servers and one for clients.

For a client, we might assume that this causes an underlying connection request to be made to the partner but the semantics don't dictate that. It is assumed that a common implementation would be to perform something similar to (in real sockets) a "socket" call followed immediately by a "connect" call. Bot of these are blocking.

For asynchronous boards however, the story becomes murkier. Again we might assume that a connection request is made but the connection will **not** be immediately usable after returning from `createSocket()`. We should assume that the board implementation will flag the connection has been requested but not yet formed. Think of the board implementation as being in the state *WAITING_FOR_CONNECTION*.

For a server, we should assume that client connection requests will be allowed return from `createSocket()`. This tells us that the board implementation had better do what ever it needs to start listening.

send

Signature:

```
int send(struct JsNetwork *net, int sckt, const void *buf, size_t len)
```

- `net` – Unknown
- `sckt` – The socket handle identifying the connection to be used
- `buf` – A pointer to the data to be sent
- `len` – The size of the data to be sent

This call is responsible for sending data through the connection identified by the socket. The data to be transmitted is pointed to by the `buf` pointer and the length of the data to send is supplied in `len`.

The return code for this API is very important. Specifically, there are three distinct variants.

- -1 – An error occurred
- 0 – No data was actually transmitted
- >0 – Data was transmitted but may be less than ALL the data. The size of the data actually sent is the value.

From a semantic perspective, it at first glance appears quite straight forward. However, there are some subtleties that can be found through deeper examination.

We must presume that the caller of this board specific function (the Espruino common networking layer) is prepared to handle sending of no data or less data that was desired. How that is accommodated is not part of the board specific contracts though.

But what of an asynchronous board? For an asynchronous sender what would that mean? The best guess is that when called, we accept as much data as we can and then schedule it for board transmission. We return an indication of how much data we have consumed. However, because we haven't REALLY finished transmitting the data, we place the socket in a state of *SENDING_DATA* that is cleared when the data has actually been transmitted.

recv

Signature:

```
int recv(struct JsNetwork *net, int sckt, void *buf, size_t len)
```

- `net` – Unknown
- `sckt` – The socket handle identifying the connection to be used
- `buf` – A pointer to a buffer that can be populated with received data
- `len` – The maximum data that can be written into the buffer

This call is responsible for returning any data that may have been received over the connection.

The return code for this API is very important. Specifically, there are three distinct variants.

- -1 – An error occurred
- 0 – No data was received
- >0 – Data was received and the amount is the value. The data can be found in the buffer.

accept

Signature:

```
int accept(struct JsNetwork *net, int sckt)
```

- `net` – Unknown
- `sckt` – The socket handle identifying the connection to be used

This call is responsible for returning a **new** socket handle for a new client connection. This needs some explanation. If the socket identified by `sckt` is a server socket, then that implies that clients can connect to it to form a new conversation. When this accept call is made, it is made against the server socket and if there is a partner connection ready, a new socket handle will be returned for that new connection.

The return values are:

- -1 – No new client connection is present.
- >=0 – The socket handle for the new conversation.

The semantics of this seem pretty clear but there may be some edge cases. What, for example, should happen when a client connects to our server socket and then a second client connects before we have "accepted" the first? It feels like the board must implement a queuing mechanism for client connections that have not been forwarded onwards.

closeSocket

Signature:

```
void closeSocket(struct JsNetwork *net, int sckt)
```

- `net` – Unknown
- `sckt` – The socket handle identifying the connection to be closed

This call is responsible for closing an open socket. This is a socket that was returned by either a call to `createSocket` or a socket returned by `accept`. The purpose of this API is to close the conversation and release any board specific resources that may have been allocated for its internal networking operations. Following this call, no subsequent API calls against this socket should occur.

For asynchronous boards, this again poses a problem. What does it mean to receive a request to close a socket (a connection) when there is still asynchronous work in flight. One guess is that a request to close state be maintained by the socket and that it close when it can.

idle

Signature:

```
void idle(struct JsNetwork *net)
```

- `net` – Unknown

It is possible that the board specific networking may wish to perform idle work. This is an opportunity for it to perform those tasks. Since there is never an assurance that there will be idle time, no required functions for board operations should be included in here.

Question: What is the design intent for this function? How might it be used?

checkError

Signature:

```
bool checkError(struct JsNetwork *net)
```

- `net` – Unknown

The purpose of this call is to determine if an error has occurred at the network level. Since no socket has been provided, this should be considered an indication of an unspecified global network issue.

Question: What is the design intent for this function? How might it be used?

getHostByName

Signature:

```
void getHostByName(struct JsNetwork *net, char *hostName, uint32_t *outIp)
```

- `net` – Unknown
- `hostName` – A NULL terminated string that identifies the host name to be resolved
- `outIp` – The address of a 4 byte storage area that will hold the IP address corresponding to the hostname.

At the TCP/IP networking level, all addressing occur via 4 byte IP addresses. However, these aren't memorable by people so we have the concept of DNS which maps symbolic names to IP addresses (eg. www.google.com). When called, this function takes in a DNS hostname and returns its corresponding IP address.

Question: What is the value of an IP address if the hostname can't be found?

For asynchronous processing, again we have a puzzle. What might it mean to return an IP address when we can't block?