TCP Socket API Spec

Contents

Overview

Sockets have been a part of the iOS networking infrastructure for over a year, and while their implementation has been satisfactory enough to allow users to add more advanced networking support beyond single-session HTTP requests to their products, there is room for improvement. Currently there are many problems with the iOS implementation of TCP sockets, most notabily with data chunking, the usage of an event-based system, insufficient distinction between 'listening' (accepts incoming connections) and 'connecting' (represents outbound connection) types, and inconsistent representation of sockets (some are objects, some are BSD file descriptors).

With the advent of sockets for Android, this is an ideal opportunity to address these issues and make sockets able to integrate future technologies, such as I/O streaming (or to represent an I/O stream themselves as a ducktype) and zero-configuration networking support for Android.

Where possible, comparisons will be drawn to the existing socket implementation on iOS and the proposed spec.

It should be noted that this spec is in flux based on the pending definition of the IO stream spec. Major changes are not expected, but further changes are possible.

Concepts and Definitions

- Socket: A socket is essentially a data stream that is connected to a host/port pair.
- Transport layer: The portion of a networking protocol which determines the parameters of data transmission (reliability, streaming, multiplexing, etc.) The two most common are UDP and TCP.
- BSD sockets: The standard POSIX implementation of sockets.
- INADDR_ANY: A specialized BSD host identifier for listening sockets which indicates that ALL available local network interfaces should be listened on. Practically, for us, this means sockets would listen on loopback (127.0.0.1), wifi, and data network.
- Listener: A socket which actively listens on a specified host/port for incoming connections.
- Connector: A socket which connects to a specified host/port.

Throughout this document, except where specified, the word "socket" indicates a socket that uses the TCP transport layer.

Basis of spec

The basis of the spec, from an outward facing functionality standpoint, is the BSD socket specification. Functions, where possible, correspond directly to their BSD counterparts.

The reason for this is to keep the interface simple, flexible, minimal, and familiar to developers who may already have experience with developing networking code. Events fired based on the socket lifecycle state (see below) provide the opportunity for more advanced handling than the standard C paradigm and allow a BSD-like interface to fit comfortably into Titanium.

There is precedence for this, as the iOS TCPSocket presents a stripped-down version of the BSD interface with some magic.

Namespacing

It is proposed that we create a new `Ti.Network.Socket` namespace within `Ti.Network`, which will house socket objects which correspond to different transport layers. The rationale for this is as follows:

- Certain socket transport types (in particular, UDP) are incompatible with TCP from an interface standpoint. In particular, UDP sockets use a "datagram" model in which there are no listening/connecting sockets, and they do not behave as I/O streams.
- The name `Ti.Network.TCPSocket` is currently reserved on iOS and cannot be deprecated until a later time (see below). This makes it impractical to use the `Ti.Network` namespace as a container for TCPSocket, UDPSocket, etc.
- This allows us to reserve a specific namespace for implementation of further transport layers, and give users a convenient space to present any custom transport layers they implement.

Transport layer support

As per this spec, only `Ti.Network.Socket.TCP` (TCP support) will be initially available. There are plans to introduce UDP sockets, but that will be covered in a separate spec.

Socket lifecycle

A socket goes through three distinct states through its lifecycle, with one additional state to represent errors. 2a and 2b are mutually exclusive.

As follows:

1. `INITIALIZED` : The socket is ready to have either `connect()` or `listen()` called.
2a. `CONNECTED` : The socket is connected to its specified host/port. Set before the `connected` callback is called.
2b. `LISTENING` : The socket is listening on its specified host/port.
3. `CLOSED` : The socket has been cleaned up via a call to `close()`. A socket in this state may be re-initialized via a new call to `connect()` or `listen()`.
4. `ERROR` : The socket encountered an error. A socket in this state may be re-initialized via a new call to `connect()` or `listen()`. The `error` callback is called if the socket was in either the `LISTENING` or `CONNECTED` state when the error occurred. When a socket enters an ERROR state, the socket is closed.

Currently, iOS TCPSocket only has the concept of a socket being valid. Distinct states provide the user with more information regarding the socket lifecycle, give clearly defined points at which callbacks are triggered, and also allow us to prevent reuse of sockets (undesirable in the case of connections accepted by a listener).

Connecting (outbound) sockets

Connecting sockets are straightforward; they connect to the specified host/port endpoint and act as I/O streams (both read and write) to that endpoint. A user-created socket enters the CONNECTED state by calling `connect()` on it.

All operations other than `listen()` and `accept()` are valid on a connecting socket.

This is equivalent to how a connecting socket functions under iOS TCPSocket.

Listening (accepting) sockets

Listening sockets accept incoming connections. A user-created socket enters the `LISTENING` state by calling `listen()` on it. The user is responsible for manually accepting incoming connections with the `accept()` command, which flags the socket to accept the next incoming connection (however, unlike BSD, it does not block). The rationale behind giving users control over inbound connection acceptance is to allow fine-grained control over system resources; in particular, due to the limitations of mobile device network device speeds or the OS, the user may want to only allow a limited number of concurrent connections at one time.

A listening socket **only** may have the `accept()` and `close()` operations called on it.

This is distinctly different from how iOS TCPSocket handles listening right now. In particular, iOS listening sockets both auto accept all incoming connections and act as a "hub" for all connections, rather than creating distinct socket objects for them; this introduces peculiarities such as associating file descriptors, rather than objects, with read/write data, and allowing the listener to "broadcast" information to all of its connected sockets. No such functionality will exist in the new socket implementation.

This current implementation is inefficient, cumbersome, presents a fragmented interface, and does not conform to what developers with network experience would consider a standard socket interface. These are all excellent reasons for changing existing behavior.

Accepted sockets

Sockets which are connecting to the local host as an endpoint are created when a listening socket accepts a new connection. These sockets arrive in the CONNECTED state and are functionally equivalent to outbound sockets.

Currently, iOS TCPSocket represents inbound sockets as file descriptors. This is unacceptable.

Changes to existing `Ti.Network.TCPSocket` on iOS

For the present, `Ti.Network.TCPSocket` will remain as-is on iOS, and continue to be the only way to interface with `Ti.Network.BonjourClient` and `Ti.Network.BonjourBrowser` as these are iOS-only features. For release 1.7.0 `Ti.Network.TCPSocket` will **not** become deprecated (due to it being the exclusive way to interface with other iOS-only features). There will not be a `Ti.Network.TCPSocket` namespace alias backported to Android as these sockets behave in a fundamentally different manner.

`Ti.Network.TCPSocket` will not be removed until there is Bonjour/Zero-configuration networking support for Android. At this point it will become deprecated for removal in the following release.

Due to the fundamental differences in interface and operation, there is no plan to provide a transitional bridge from `Ti.Network.TCPSocket` to `Ti.Network.Socket` at any point on iOS.

Security issues

Presenting sockets to the world introduces a host of security issues, including but not limited to malicious data injection via a connection over the CDN (celluar data network) via a host listening on the IP address assigned to the radio (or, equivalently, via `INADDR_ANY`). In order to try and reduce the potency of these attacks, or the ability to conduct them, we should consider:

- Revoking access to any connection that comes in directly to the CDN. **NOTE**: This is not necessarily feasible; socket implementations on both iOS and Android contain information about the originating host, but not necessarily the interface the connection came in over.
- Disallowing `INADDR_ANY`
- Limiting the possibility for standard attacks (i.e. buffer overflow)
- Providing specialized training for end developers specifically for advanced network programming

It is worth noting that on iOS, an application listening over the CDN **is** considered grounds for rejection.

`INADDR_ANY`

As above, we do not support BSD's `INADDR_ANY`. References to it are included in the spec for historical reasons and to clarify why it is not supported. In particular, we have no way to filter connections (inbound or outbound) based on the CDN, and `INADDR_ANY` is global with no scoping, meaning we can't have it translate to "all interfaces except for CDN." For these reasons, support for this feature is removed.

I/O Layer

I/O is intended to be handled entirely through Streams and Buffers. For this reason no I/O operations are specified in this document.

The I/O Layer is intended to only be available on CONNECTED sockets. I/O does not make sense on LISTENING sockets, and sockets in the ERROR state do not have an "active" buffer.

The current iOS TCPSocket does not present a unified I/O layer with any other interface, and also confuses availability of I/O on LISTENING and CONNECTED sockets, and handles it in a clumsy way.

Proposed API

Ti.Network

- Namespace
  - `Ti.Network.Socket` : Namespace for all socket types and related constants.

Ti.Network.Socket

- Properties
  - `INITIALIZED` : Constant representing the "initialized" state a socket
  - `CONNECTED` : Constant representing the "connected" state for a socket
  - `LISTENING` : Constant representing the "listening" state for a socket
  - `CLOSED` : Constant representing the "closed" state for a socket
  - `ERROR` : Constant representing the "error" state for a socket
- Functions
  - `Ti.Network.Socket.TCP` createTCP(Object args)}} : Creates a new TCP socket.
  - `Ti.Network.Socket.UDP` createUDP(Object args)}} : Creates a new UDP socket. RESERVED; not intended to be

implemented immediately.

Ti.Network.Socket.TCP

While not currently used in this proposal, the "options" property name would be reserved for setting socket options in the future.

- Properties
  - `host` : The host to connect to. **Cannot** be modified when not in the `INITIALIZED` state. Supports both IPv4 and IPv6.
  - `port` : The port to connect to. **Cannot** be modified when not in the `INITIALIZED` state.
  - `listenQueueSize` : Max number of pending incoming connections to be allowed when `listen()` is called. Any incoming connections received while the max number of pending connections has been reached will be rejected.
  - `timeout` : The timeout for `connect()` and all I/O `write()` operations. **Cannot** be modified when not in the `INITIALIZED` state.
  - `options` : Options for the socket (such as reuse, multicast, etc.) RESERVED; not to be implemented immediately.
  - `connected` : The callback to be fired after the socket enters the "connected" state. Only invoked following a successful `connect()`call.
    - Argument parameters:
      - `socket` : The socket which was connected
  - `error` : The callback to be fired after the socket enters the `ERROR`state.
    - Argument parameters:
      - `socket` : The socket that experienced the error
      - `error` : A stringified description of the error
      - `errorCode` : The error code of the error (potentially system-dependent)
  - `accepted`: The callback to be fired when a listener accepts a connection.
    - Argument parameters:
      - `socket` : The socket which received the connection
      - `inbound` : A `Ti.Network.Socket` object which represents the inbound connection; this should be considered a "connected" socket and is created in the CONNECTED state.
  - `state`[spe:read-only] : The current state of the socket.

- Functions
  - `void connect()` : Attempts to connect the socket to its host/port. Throws exception if the socket is in a CONNECTED or LISTENING state. Throws exception if a valid `host` and `port` has not been set on the proxy. Nonblocking; connection attempts are asynchronous.
  - `void listen()` : Attempts to start listening on the socket's host/port. listen() call will attempt to listen on the specified host and/or port property for the socket if they are set. This function blocks execution and throws an exception on error (and sets the socket state to `ERROR`) but does not fire the `error` callback in this event. Throws exception if the socket is in a LISTENING or CONNECTED state.
  - `void accept(Object params)` : Tells a `LISTENING` socket to accept a connection request at the top of a listener's request queue when one becomes available. Takes an argument, a box object which assigns callbacks to the created socket. Note that the `connected` callback is **not** called (the socket does not "transition to" the `CONNECTED` state - it's created in the `CONNECTED` state) on the newly created socket. The accepted callback **is** called when a new connection is accepted as a result of calling accept(). If the socket is already flagged to accept the next connection, the existing accept options will be update to use the newly specified options object. Throws an exception if the socket is not in a LISTENING state.
  - `void close()` : Closes a socket. Throws exception if the socket is not in a `CONNECTED` or `LISTENING` state. Blocking.

Pseudo Code Examples

Create a socket to connect

```
/*
 * Assumes the existence of a `Ti.Blob Ti.createBlob(string text)` method
 */

var connectingSocket = Ti.Network.createTCP({
 host: 'www.externalhost.com',
 port: 4747,
 connected:function(e) {
  e.socket.write(Ti.createBuffer({data: "Well, hello there!"}));
 },
 error:function(e) {
  Ti.UI.createAlertDialog({
   title: "Socket error: "+e.errorCode,
   message: e.error
  }).show();
  Ti.API.info("CONNECTION has been closed: " + e.socket.host+":" + e.socket.port);
 }
});
connectingSocket.connect();
```

Create a socket to listen

```
// NOTE: Under iOS, Ti.Platform.address always resolves to wifi; Android
// behavior may differ

var hostWhitelist = ['192.168.0.1', '192.168.0.2'];
var connections = [];

var acceptedParams = {
 read: function(e) {
  // Do something with data
 },
 error: function(e) {
  // Do something with error
 }
};

var listeningSocket = Ti.Network.createTCP({
 host: Ti.Platform.address,
 port: 4747,
 listenQueueSize: 10,
 error: function(e) {
  Ti.UI.createAlertDialog({
   title: "Listener error: " + e.errorCode,
   message: e.error
  }).show();
  Ti.API.info("CONNECTION has been closed: " + e.socket.host + ":" + e.socket.port);
 },
 accepted:function(e) {
  var socket = e.inbound;

  // NOTE: We only have the host information after accept()ing the
  // connection
```

```
  var whitelisted = hostWhitelist.indexOf(socket.host);
  if (whitelisted == -1) {
   Ti.API.warn("Attempted connection from socket not on whitelist: " + socket.host);
   socket.close();
  }
  else {
   Ti.API.info("Accepted connection from: " + socket.host);
   connections.push(socket);
  }

  // Check for a maximum number of connections
  if (connections.length < 10) {
   listeningSocket.accept(acceptedParams);
  }
 }
});

try {
 listeningSocket.listen();
 Ti.API.info("Now listening on: " + listeningSocket.host + ":" +
listeningSocket.port);

 // NOTE: We do not block JS execution on 'accept', unlike BSD.
 // It is an asynch call which fires the 'accepted' callback when a
 // new inbound connection is available to be pulled off the queue.
 listeningSocket.accept(acceptedParams);
}
catch (e) {
 Ti.API.info("Error occured while configuring listener: "+e);
}
```

```
// Maybe do something with connections somewhere...
```

Current KS->Platform->Sockets example, using new sockets

```
var win = Titanium.UI.currentWindow;

var connectedSockets = [];

var acceptedCallbacks = {
 read: function(e) {
  messageLabel.text = "Read from: " + e.socket.host;
  readLabel.text = e.data.text;
 },
 error: function(e) {
  Ti.UI.createAlertDialog({
   title: "Socket error: " + e.socket.host,
   message: e.error
  }).show();
  var index = connectedSockets.indexOf(e.socket);
  if (index != -1) {
   connectedSockets.splice(index,1); // Removes socket
  }
 }
};

var socket = Titanium.Network.createTCPSocket({
 hostName: Ti.Platform.address,
 port: 40404,
 type: Ti.Network.TCP,
 accepted: function(e) {
  var sock = e.connector;
  connectedSockets.push(sock);
  socket.accept(acceptedCallbacks);
 },
 closed: function(e) {
  messageLabel.text = "Closed listener";
 },
 error: function(e) {
  Ti.UI.createAlertDialog({
   title: "Listener error: "+e.errorCode,
   message: e.error
  }).show();
 }
});

var messageLabel = Titanium.UI.createLabel({
 text: 'Socket messages',
 font: {fontSize: 14},
 color: '#777',
 top: 220,
 left: 10
});
win.add(messageLabel);

var readLabel = Titanium.UI.createLabel({
```

```
 text: 'Read data',
 font: {fontSize: 14},
 color: '#777',
 top: 250,
 left: 10,
 width: 400
});
win.add(readLabel);

var connectButton = Titanium.UI.createButton({
 title: 'Listen on 40404',
 width: 200,
 height: 40,
 top: 10
});
win.add(connectButton);
connectButton.addEventListener('click', function() {
 try {
  socket.listen();
  messageLabel.text = "Listening on" + e.socket.host + ":" + e.socket.port;
  e.socket.accept(acceptedCallbacks);
 } catch (e) {
  messageLabel.text = 'Exception: ' + e;
 }
});

var closeButton = Titanium.UI.createButton({
 title: 'Close',
 width: 200,
 height: 40,
 top: 60
});
win.add(closeButton);
closeButton.addEventListener('click', function() {
 try {
  socket.close();
 } catch (e) {
  messageLabel.text = 'Exception: ' + e;
 }
});

var stateButton = Titanium.UI.createButton({
 title: 'Socket state',
 width: 200,
 height: 40,
 top: 110
});
win.add(validButton);
validButton.addEventListener('click', function() {
 var stateString = "UNKNOWN";
 switch (socket.state) {
  case Ti.Network.SOCKET_INITIALIZED:
   stateString = "INITIALIZED";
   break;
  case Ti.Network.SOCKET_CONNECTED:
   stateString = "CONNECTED";
   break;
  case Ti.Network.SOCKET_LISTENING:
   stateString = "LISTENING";
```

```
   break;
  case Ti.Network.SOCKET_CLOSED:
   stateString = "CLOSED";
   break;
  case Ti.Network.SOCKET_ERROR:
   stateString = "ERROR";
   break;
 }
 messageLabel.text = "State: " + stateString;
});

var writeButton = Titanium.UI.createButton({
 title: "Write 'Paradise Lost'",
 width: 200,
 height: 40,
 top: 160
});
win.add(writeButton);
writeButton.addEventListener('click', function() {
 var plBlob = Titanium.Filesystem.getFile(Titanium.Filesystem.resourcesDirectory,
'paradise_lost.txt').read();

 for (var sock in connectedSockets) {
  sock.write(plBlob);
 }
 messageLabel.text = "I'm a writer!";
});

// Cleanup
win.addEventListener('close', function(e) {
 try {
  socket.close();
 }
 catch (e) {
  // Don't care about exceptions; just means the socket was already closed
 }
 for (var sock in connectedSockets) {
  try {
   sock.close();
  }
  catch (e) {
   // See above
  }
```

```
  }
});
```