SONY
make.believe

W3C SysApps WG

TCP and UDP Socket API

W3C Santa Clara TPAC meeting - 2014

Claes Nilsson
Technology Research / Sony Mobile
claes1.nilsson@sonymobile.com

Summary of TCP and UDP Socket API status

SONY
make.believe

- W3C SysApps TCP and UDP Socket API provides interfaces to UDP sockets, TCP Client sockets and TCP Server sockets.
- Latest efforts mainly spent on rewriting this API to be based on the general Streams API.
- Specification ready for publication as a new public working draft.
- The major remaining action is specifying support for secure sockets

Note on trust and permissions for this API

SONY
make.believe

- There is ongoing work on trust and permissions in W3C. For example see Workshop on trust and permissions for Web applications 3–4 September 2014, Paris, France. The assumption is that this API must only be exposed to trusted content according to a security model. The details of that security model as such is out of scope for this specification as this model should apply to any security and privacy sensitive API.
  - One example is a model based on existing web security mechanisms such as tls/ssl, signed manifests, csp, etc. Sony Mobile is exploring such a model. See Trusted Hosted Web Apps in FFOS

SONY
make.believe

Summary of Streams API status

- Main specification is WHAT WG Streams API
  - Public API stabile
  - Reference implementation exist:
    - **JavaScript polyfill**
    - **Testsuite**
- The W3C Streams API specification defines a few extensions to the WHAT WG Streams API to meet requirements specific to the browser environment
- Some ideas on making Streams a part of ECMAScript, TC39

SONY
make.believe

Support for secure sockets

- Create a secure socket, issue-10-Support for secure sockets
  - Do we need to define certificate pinning?
  - Do we need to define cipher suites?
  - Do we need client authentication?
  - Select server certificate for TCPServerSocket?
- Add support for upgrading existing connection to a secure connection, issue-35-startTLS
  - Could we make it simple by using default certificates and algorithms?

SONY
make.believe

Supporting slides on adapation to Streams follow

Motivations for basing the API on Streams

SONY
make.believe

- Reusing a general standardized solution for handling the complexity of sending, receiving, buffering, backpressure and other issues related to streaming and asynchronous APIs.

- Reusing a solution for piping a source stream to a destination stream.

SONY
make.believe

What is a Streams API?

- Provides an interface for creating, composing, and consuming streams of data.
- Is designed to map efficiently to low-level I/O primitives,
- Deals with similar issues as we do with the TCP and UDP Socket API, e.g.:
  - ”don’t lose data”
  - ”don’t overflow send buffers”
  - ”keep it simple for developers”
- Is designed to be used in conjunction with other APIs.

SONY
make.believe

Stream Producers

APIs which can produce a Stream object are identified as *Producers*.

Examples:

- XMLHTTPRequest
- FileReader
- Media Capture
- MediaStream Recording API
- Web Cryptography API
- Text Encoder
- Text Decoder
- WebSockets
- RTCPeerConnection
- TCP and UDP Sockets

SONY
make.believe

Stream Consumers

APIs which read and act on a Stream object are identified as *consumers*.

Examples:

- XMLHttpRequest
- Web Audio
- Media Source Extensions
- Web Cryptography API
- Text Encoder
- Text Decoder
- WebSockets
- RTCPeerConnection
- FileWriter
- TCP and UDP Sockets

Reading push-based data sources (such as TCP) - requirements

SONY
make.believe

- Handling new data pushed from the source
- Mechanism for pausing and resuming the flow of data.
- A way to signal that the source has no more data
- A way to signal when there is an error in getting data
- Buffering logic in the stream primitive itself to assure that we don’t lose data.

Writing data - requirements

SONY
make.believe

- The Stream object must handle the complexity of buffering sequential writes, e.g. the case when the send buffer becomes full due to slow network. For example:
  - A method to write data
  - A way to signal that the buffer is getting full (reached the “high water mark”)
  - A way to signal that the buffer is drained and can receive more data
- Must be possible to signal that the underlying sink should be closed.
- Must be possible to detect “abort” signal

SONY
make.believe

Piping streams - requirements

- A common way of consuming streams is to pipe them to each other. This is one essence of streaming APIs: getting data from a readable stream to a writable one, while buffering as little data as possible in memory.

- Example: Create a read stream from a file, transforming it, and pipe it to a write TCP socket stream.

How to use the Streams API for TCP and UDP Sockets? 1(5)

SONY
make.believe

Before:

```
[Constructor (DOMString remoteAddress, unsigned short remotePort,
      optional TCPOptions options)]
interface TCPSocket : EventTarget {
  readonly attribute DOMString      remoteAddress;
  readonly attribute unsigned short remotePort;
  readonly attribute DOMString      localAddress;
  readonly attribute unsigned short localPort;
  readonly attribute boolean        addressReuse;
  readonly attribute boolean        noDelay;
  readonly attribute unsigned long  bufferedAmount;
  readonly attribute ReadyState     readyState;
           attribute EventHandler   ondrain;
           attribute EventHandler   onopen;
           attribute EventHandler   onclose;
           attribute EventHandler   onerror;
           attribute EventHandler   ondata;
  void      close ();
  void      halfclose ();
  void      suspend ();
  void      resume ();
  boolean send ((DOMString or Blob or ArrayBuffer or ArrayBufferView) data);
};
```

How to use the Streams API for TCP and UDP Sockets? 2(5)

SONY
make.believe

Now:

```
[Constructor (DOMString remoteAddress, unsigned short remotePort,
      optional TCPOptions options)]
interface TCPSocket : {
   readonly attribute DOMString         remoteAddress;
   readonly attribute unsigned short    remotePort;
   readonly attribute DOMString         localAddress;
   readonly attribute unsigned short    localPort;
   readonly attribute boolean           addressReuse;
   readonly attribute boolean           noDelay;
   readonly attribute ReadyState        readyState;
   readonly attribute Promise           opened;
   readonly attribute Promise           closed;
   readonly attribute ReadableStream    readable;    // ReadableStream is defined by Streams API
   readonly attribute WriteableStream   writeable;   // WritableStream is defined by Streams API
   void close ();
   void halfClose ();
};
```

How to use the Streams API for TCP and UDP Sockets? 3(5)

SONY
make.believe

- Each Streams API based API must provide an *adaptation layer* to the Streams API.

- The *adaptation layer* to Streams API is created through implementation of a number of functions that are given as input arguments to the constructors of the Readable/WritableStreams objects and called by the Streams API implementation.

- These functions then calls the internal methods of the Streams API to do stuff.

How to use the Streams API for TCP and UDP Sockets? 4(5)

SONY
make.believe

For example, the ReadableStream's constructor is passed the following functions that must be implemented by the TCP and UDP Socket API:

- start(): Called immediately by Streams implementation. Used to adapt to the underlying TCP implementation.
- pull(): Used to start the flow of TCP data after a "buffer getting full" condition.
- cancel(): Called when the readable stream is canceled. Used here to close the TCP connection.

How to use the Streams API for TCP and UDP Sockets? 5(5)

SONY
make.believe

- For example, the ReadableStream's constructors start() function does the following:
  - Performs TCP connection setup handshake.
  - Pushes received TCP data into the internal buffer by calling the Streams API's internal enqueue() function.
  - When enqueue() return value says "high watermark reached" then stops receiving TCP data through the TCP flow control mechanism.

Application code example

SONY
make.believe

```
// Echo client

var mySocket = new TCPSocket("127.0.0.1", 6789);

mySocket.writeable.write("Hello World").then(
   () => {
      console.log("Data has been sent to server");
      mySocket.readable.wait().then(
         () => {
            console.log("Data received from server:" + mySocket.readable.read());
            mySocket.close();
         },
         e => console.error("Receiving error: ", e);
      );
   },
   e => console.error("Sending error: ", e);
);
```