HTML5 P2P Transport API

v0.4
Justin Uberti (juberti@google.com)
Sep 23, 2010

Overview
Goals
Non-Goals
Sample Use Cases
Theory of Operation
Sequence of Operations
HTML5 Interface
Pepper Interface
Config Data
References

Overview

The goal of the Transport API is to provide a general-purpose “socket” API to allow web developers to set up peer-to-peer or client-server connections from untrusted web pages, with robust NAT and firewall traversal. This interface is somewhat similar to the WebSocket interface, but is intended to support both connectionless and connection-oriented communications, and does not require the use of HTTP as an underlying protocol.

Goals

- Reliable P2P connectivity between endpoints, including fallback to relayed connections if needed.
- Support for connectionless and connection-oriented transports.
- Support for multiple simultaneous transports.
- Support for TLS/DTLS protection of transports.
- Safe for untrusted content; callers cannot send/receive data to or DoS arbitrary servers.
- High performance; suitable for high-throughput applications such as file transfer
- Robust; detects connectivity loss and auto-reconnects if possible
- Works with any signaling service or protocol.
- Works with any compatible address discovery/allocation services (e.g. STUN); said services can be provided by either the browser manufacturer or web site.
- Interoperates with existing ICE endpoints.

Non-Goals

- Unrestricted socket access, including “raw” UDP support or generic “listen” support

Sample Use Cases

The following are some use cases that we intend to support using this API. It is not an exhaustive list, but is intended to convey some different types of applications that could be built using this API.

- Sending a large file between web clients over a secure, reliable peer-to-peer transport.
- Voice and video chat between web clients, sent peer-to-peer as RTP over UDP, with audio and video sent on separate transports
- Voice and video chat between a web client and a deployed voice/video client, without a gateway.
- Live video streaming from a server to a web client using reliable or unreliable transport.
- Bulk data exchange between multiplayer web games using reliable or unreliable transport.

Theory of Operation

The general concept in the Transport API is to use an interactive connectivity establishment procedure to set up one or more UDP “connections” between endpoints. Interactive connectivity establishment relies on the use of a reliable but low-bandwidth signaling channel to exchange the necessary information to set up a direct, high-bandwidth application connection. We expect to use ICE as described in RFC 5245, but may add extensions (e.g. port prediction) in order to maximize direct connectivity rates. We will also support the use of a relay server, such as TURN or Google’s TURN-like Mediaproxy, to ensure connections can be established when conventional methods fail. This relay server can also provide fallback to TCP in the event that UDP is blocked, as may be the case in an enterprise setting.

For connection-oriented protocols, we will run TCP over the UDP “connection” established with ICE, using Google’s PseudoTCP, an application-layer TCP stack that can run over any datagram protocol, including UDP. This is because it is far easier for UDP to traverse NAT than inbound TCP, and the overhead of TCP/UDP versus TCP/IP is minimal.

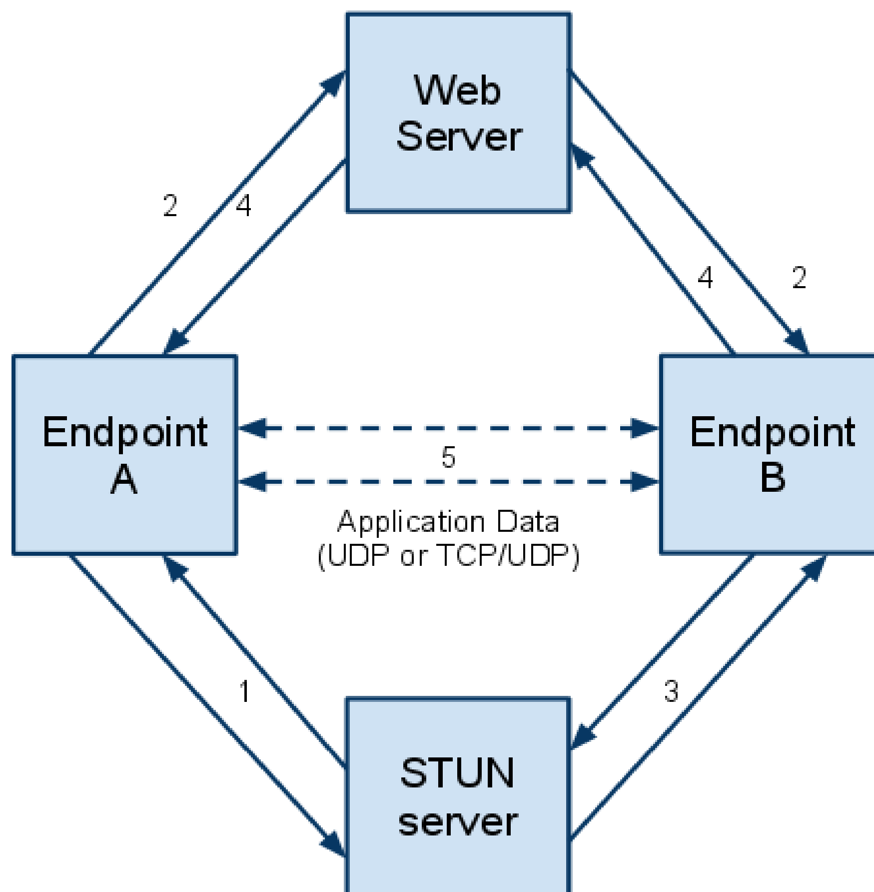
Our choice of ICE as an underlying protocol is based on several benefits that it provides. First, it is a IETF standard, so its workings will be well-understood, and implementations will be readily available. Secondly, it is already often used for media transport on the Internet, making it a good fit for real-time voice and video applications who will be able to talk to existing ICE endpoints without a gateway. Third, it provides reasonable access control for untrusted content. With ICE, no data can be exchanged unless the remote side sends the appropriate responses to ICE’s connectivity checks - meaning that a web page cannot send data to an arbitrary server, preventing unauthorized access to intranet servers and DoS attacks.

As mentioned above, configuration information needs to be exchanged between the endpoints to establish a direct connection. The mechanism via which this information is transmitted is independent of this specification. Typically, it will be sent over some HTTP-based signaling channel, but it could also be sent over an existing Transport, if desired.

Security is provided via TLS/DTLS and self-signed certificates, which are verified over the aforementioned out-of-band signaling channel (presumably secure) to prevent man-in-the-middle attacks.

Sequence of Operations

The diagram below illustrates the basic steps involved in setting up a Transport connection. The signaling exchange here is done via a web server, but the details of this are left up to the web application.



1. Endpoint A creates a Transport object (or objects) and chooses its type (connection-oriented or connectionless). It calls connect() to start the connection process. This results in queries to the STUN server (and potentially other servers) to gather all possible addresses to be used in connecting.
2. Endpoint A receives callbacks with the addresses that are discovered. Endpoint A passes these addresses to Endpoint B using an out-of-band mechanism, such as HTTP

POST to a web server.
3. Endpoint B creates a Transport object of the same type (based on the out-of-band signaling). It calls connect() to start the connection process, and also passes the data received from Endpoint A to the appropriate Transport.
4. Endpoint B receives callbacks with the addresses that it discovers, and passes these addresses to Endpoint A using an out-of-band mechanism.
5. The Transports determine a viable path for application data by processing the received addresses and performing connectivity checks against them. When connectivity is established, the onconnect callback is issued on both sides.
6. Data is exchanged via the send and onmessage APIs.
7. If connectivity is interrupted, the onunwritable callback will be fired, and the endpoints will again exchange addresses to determine a new data path.
8. When the Transport is no longer needed, either endpoint can call close(), which will trigger the ondisconnect callback on the remote side.

HTML5 Interface

The following is the intended interface for Javascript clients, expressed in WebIDL format. Where practical, the interface has been kept similar to the WebSocket interface. Key differences from the WebSocket interface include:

- Support for address gathering and transmission. This is required for ICE.
- Events to indicate nonfatal connectivity loss and restoration. This is to allow for graceful handling of IP mobility and similar network events.
- An event to indicate that the Transport is currently flow controlled. This is to prevent excessive buffering of data in high-throughput applications such as file transfer.

```
[Constructor]
interface Transport {
  // Creates the transport with the specified component name
  // (e.g. “0”) and protocol (currently either “udp” or “tcp”).
  Transport(in DOMString component, in DOMString protocol);
  readonly attribute DOMString name;
  readonly attribute DOMString protocol;

  // Sets the servers (e.g. STUN) to use for establishing
  // connectivity.
  // TBD: this might make more sense on a global object, so that it
  // doesn’t have to be set on each Transport.
  void setServerConfig(in DOMString config);

  // For connection-oriented transports, sets whether the transport
  // will act as a client (initiating connection) or server
  // (receiving connection). By default the Transport will be in
  // client mode.
  void setMode(in boolean server);
```

```
// Sets whether this connection should be protected using TLS/DTLS.
// For this to be useful, the config data must be sent over a
// secure channel.
// TBD: format of the config data, and whether external
// certificates will be allowed.
void setSecure(in boolean secure);

// Starts acquiring addresses and attempting to connect.
void connect();

// Adds a remote address provided by the peer.
void addAddress(in DOMString address);

// Sends the specified data over the transport.
boolean send(in DOMString data);

// Disconnects the transport.
void close();

// Gets the current state of the Transport.
const unsigned short CONNECTING = 0;
const unsigned short OPEN = 1;
const unsigned short CLOSING = 2;
const unsigned short CLOSED = 3;
const unsigned short UNWRITABLE = 4;
readonly attribute unsigned short readyState;

// Gets the amount of queued data.
// For connection-oriented Transports only.
readonly attribute unsigned long bufferedAmount;

// Gets the current local address in use.
readonly attribute DOMString localAddress;
// Gets the current remote address in use.
readonly attribute DOMString remoteAddress;

// Called when the connection completes.
attribute Function onconnect;
// Called when writability is established, e.g. when the connection
// first completes, or when connectivity is restored after a loss.
attribute Function onwritable;
// Called when an open connection loses connectivity.
attribute Function onunwritable;
// Called if the connection is closed by the remote peer, or
// connectivity is lost and cannot be re-established.
attribute Function ondisconnect;
```

```
  // Called when a new local address is available.
  attribute Function onaddressready;
  // Called when a message has been delivered.
  attribute Function onmessage;
  // For connection-oriented transports, called when flow control
  // is lifted, i.e. buffered data has been successfully flushed.
  attribute Function onreadytosend;
```

Pepper Interface

The following is the intended interface for Pepper clients, expressed in “C” format. Instead of exposing event callbacks, it uses “blocking” methods with completion callbacks.

```
struct PPB_Transport {
  // Creates a new transport object with the specified name
  // using the specified protocol.
  PP_Resource (*CreateTransport)(PP_Module module,
                                 const char* component,
                                 const char* proto);

  // Returns whether the transport is currently writable
  // (i.e. can send data to the remote peer)
  bool (*IsWritable)(PP_Resource transport);
  // TODO: other getters/setters
  // connect state
  // connect type, protocol
  // RTT

  // Establishes a connection to the remote peer.
  // Returns PP_ERROR_WOULD_BLOCK and notifies on |cb|
  // when connectivity is established (or timeout occurs).
  int32_t (*Connect)(PP_Resource transport,
                     PP_CompletionCallback cb);

  // Obtains another ICE candidate address to be provided
  // to the remote peer. Returns PP_ERROR_WOULD_BLOCK
  // if there are no more addresses to be sent.
  int32_t (*GetNextAddress)(PP_Resource transport,
                            PP_Var* address,
                            PP_CompletionCallback cb);
  // Provides an ICE candidate address that was received
  // from the remote peer.
  int32_t (*ReceiveRemoteAddress)(PP_Resource transport,
                                  const PP_Var* address);

  // Like recv(), receives data. Returns PP_ERROR_WOULD_BLOCK
```

```
  // if there is currently no data to receive.
  int32_t (*Recv)(PP_Resource transport,
                  void* data,
                  size_t len,
                  PPCompletionCallback cb);
  // Like send(), sends data. Returns PP_ERROR_WOULD_BLOCK
  // if the socket is currently flow-controlled.
  int32_t (*Send)(PP_Resource transport,
                  const void* data,
                  size_t len,
                  PPCompletionCallback cb);

  // Disconnects from the remote peer.
  int32_t (*Close)(PP_Resource transport);
};
```

Config Data

The following is the format of the opaque data used in the setServerConfig and addAddress/ onaddressready APIs. It is intended that this data be opaque to applications.

```
Server Config: {
 "stunServer": "stun.l.google.com:19302",
 "relayServer": "relay.google.com:80",
 "relayToken": "ABCDEFGH"
};

Address: {
 "component": "0",
 "foundation" : "",
 "ip" : "1.2.3.4",
 "port" : 49152,
 "protocol" : "udp",
 "type" : "host",
 "preference" : 65535,
 "username" : "ABCD1234",
 "password" : "WXYZ7890",
};
```

Security Considerations

As mentioned in the Goals section, protection of data and safe usage by untrusted web pages are requirements for this API.

Protection of data will be accomplished by use of TLS/DTLS, as appropriate for the transport type. The client that calls setMode(false) will be the TLS Client; the client that

calls setMode(true) will be the TLS Server. Each will use a self-signed certificate to identify themselves to the other side during the TLS handshake; protection against MITM attacks will be performed by exchanging hashes of the certificates via the out-of-band, secure, signaling channel.

Protection against untrusted web pages falls into two categories: prevention of unauthorized access, typically on an intranet, and prevention of DoS.

Unauthorized access will be prevented by the fact that we will not allow applications to send or receive data on Transports until they complete an ICE connectivity check. Therefore, an attempt to contact an arbitrary server will fail, as it will either ignore the STUN traffic sent to it, or reply with an incorrect response.

DoS will be prevented by limiting the number of Transports that can attempt to contact a given host at the same time. The exact limits have not yet been determined; for voice/video applications, the limit here probably should be at least 6, for compatibility with existing applications. We will also need to impose similar restrictions on the addresses supplied for the server config.

For enterprise firewall configuration, we will want to be able to globally specify which source ports will be used by this API, so that the minimum set of ports need to be opened up.

References

- Introduction to ICE: http://www.isoc.org/tools/blogs/ietfjournal/?p=117
- ICE: http://www.rfc-editor.org/authors/rfc5245.txt, http://xmpp.org/extensions/xep-0176.html
- STUN: http://tools.ietf.org/html/rfc5389
- PseudoTcp: http://code.google.com/apis/talk/libjingle/file_share.html
- WebSockets: http://dev.w3.org/html5/websockets/

Document History

- 0.4: Cleanup, addressed some comments.
- 0.3: Rewrote Pepper API based on input from Darin Fisher.
- 0.2: Fleshed out supporting text, added diagrams
- 0.1: Initial HTML5 API