# DIG 4104C: WEB DESIGN WORKSHOP

Lecture 07: Messaging: Websockets and cross domain messaging

Summer 2015

D. Novatnak

## Outline

## Objectives

- Expand student knowledge of server and client side methods in messaging
- Expose students to the features of the websockets API in HTML5
- Expose students to the language and syntax necessary to leverage the websockets API in their own applications
- Extend the student’s knowledge of document messaging across domains

## At the end of this lecture, you should be able to….

- Create a connection between pages through the websocket API
- Cross domain message through extending postMessage

## HTML5 WEBSOCKETS

PLUG IT IN, TURN IT ON

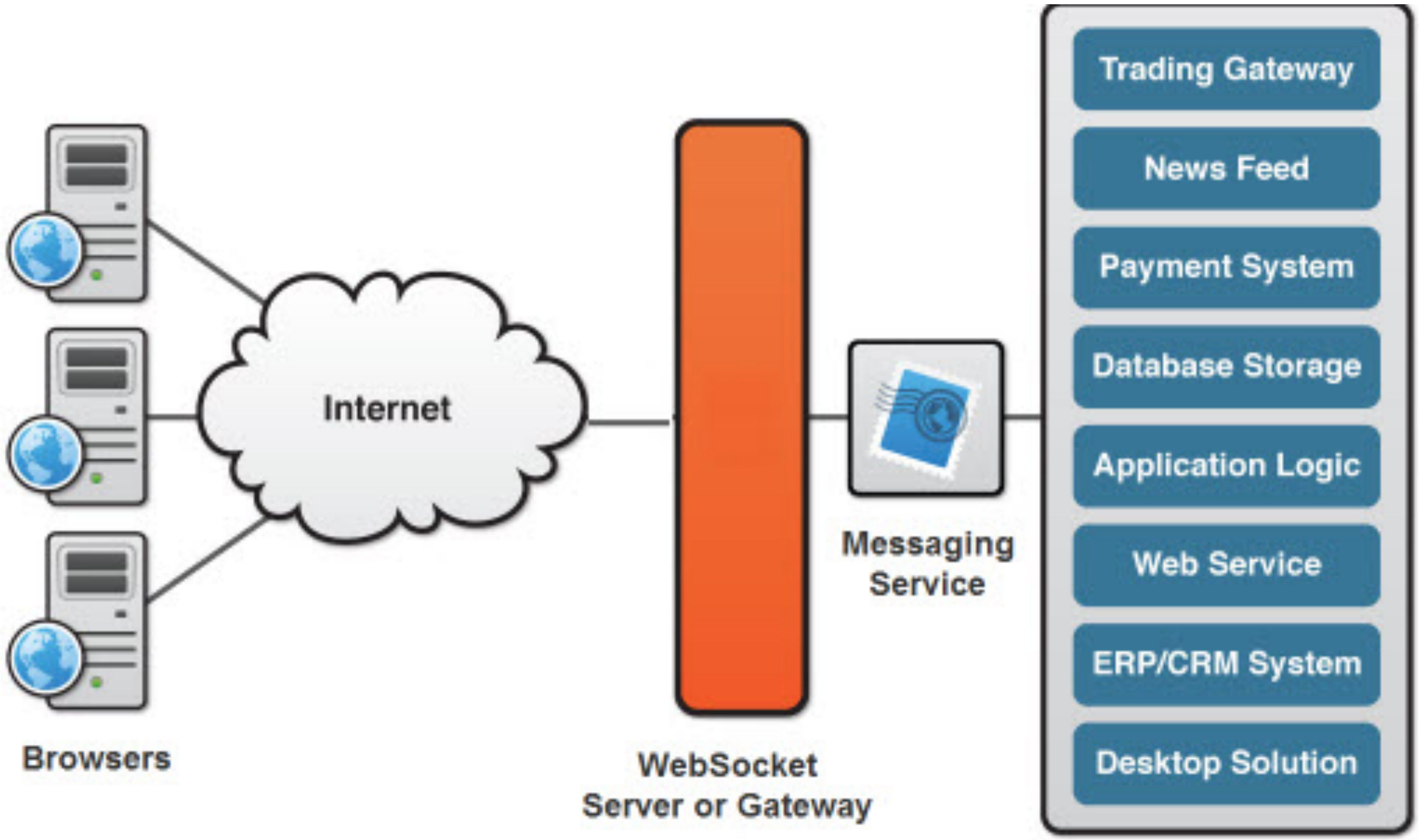
## Introduction

- The HTML5 WebSockets specification defines an API that enables web pages to use the WebSockets protocol for two-way communication with a remote host.
- It introduces the WebSocket interface and defines a full-duplex communication channel that operates through a single socket over the Web.
- HTML5 WebSockets provide an enormous reduction in unnecessary network traffic and latency compared to the unscalable polling and long-polling solutions that were used to simulate a full-duplex connection by maintaining two connections.

## Introduction

- HTML5 WebSockets account for network hazards such as proxies and firewalls, making streaming possible over any connection, and with the ability to support upstream and downstream communications over a single connection, HTML5 WebSockets-based applications place less burden on servers, allowing existing machines to support more concurrent connections.

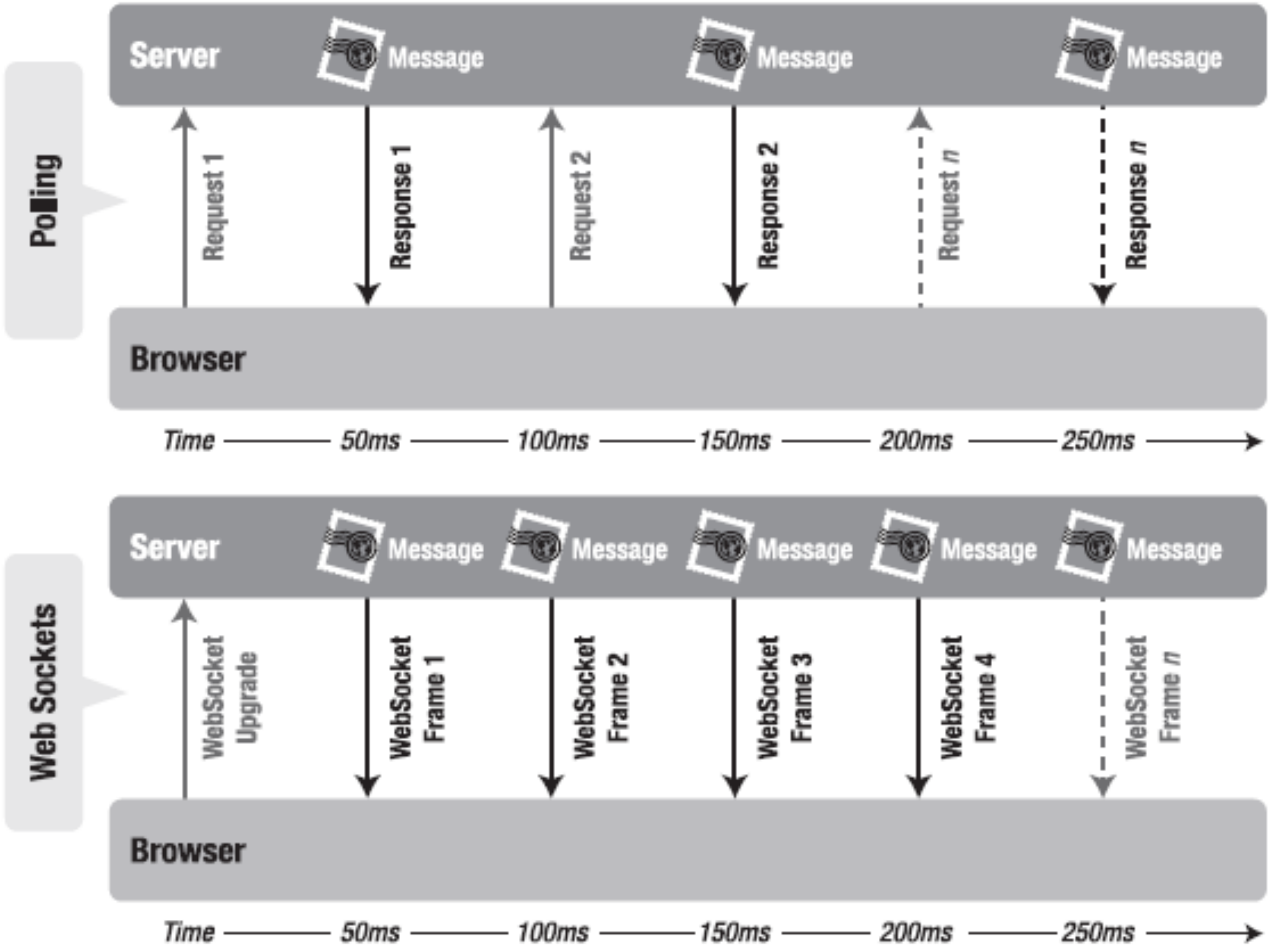
## Basic WebSocket-based Architecture



## Can’t we just use AJAX?

- *AJAX* (*Asynchronous JavaScript And XML*) provides Web clients a means to send mini-requests to the Web server
  - Via the **`XMLHttpRequest`** object
  - Removes need to reload entire page
  - Server has no way to notify the browser unless the client makes a request

# Latency comparison



## In a nutshell

- A full-duplex bidirectional communication channel in javascript
  - over the internet
  - over existing infrastructure
  - very much like a TCP connection
- JavaScript API in all HTML5 complaint browsers

## Features

- One of the more unique features WebSockets provide is its ability to traverse firewalls and proxies, a problem area for many applications.
- Comet-style applications typically employ long-polling as a rudimentary line of defense against firewalls and proxies. The technique is effective, but is not well suited for applications that have sub-500 millisecond latency or high throughput requirements.
- Plugin-based technologies such as Adobe Flash, also provide some level of socket support, but have long been burdened with the very proxy and firewall traversal problems that WebSockets resolve.

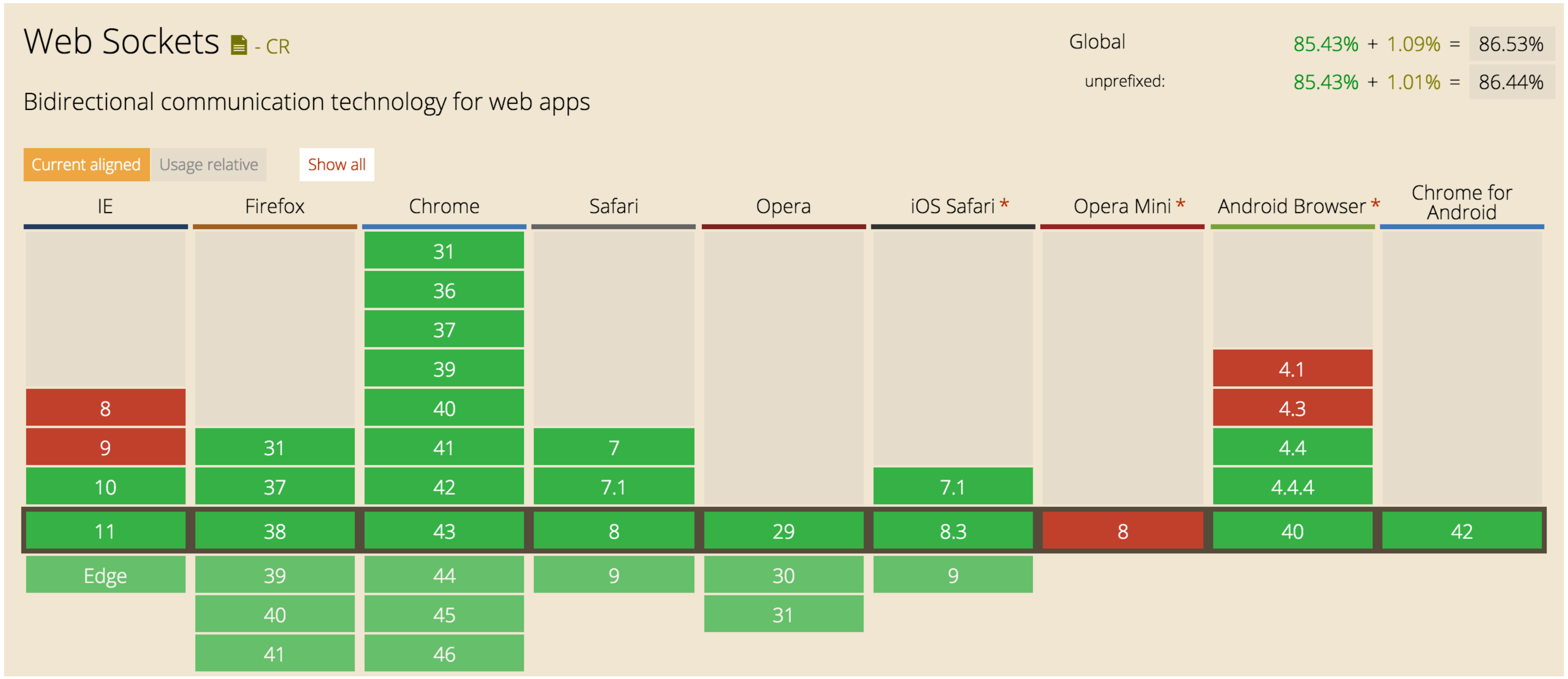
## Proxy Tunneling

- A WebSocket detects the presence of a proxy server and automatically sets up a tunnel to pass through the proxy. The tunnel is established by issuing an HTTP CONNECT statement to the proxy server, which requests for the proxy server to open a TCP/IP connection to a specific host and port.
- Once the tunnel is set up, communication can flow unimpeded through the proxy. Since HTTP/S works in a similar fashion, secure WebSockets over SSL can leverage the same HTTP CONNECT technique.

## But is it HTML5?

- WebSockets—like other pieces of the HTML5 effort such as Local Storage and Geolocation—was originally part of the HTML5 specification, but was moved to a separate standards document to keep the specification focused.

# Browser compatibility



Web Sockets - CR

Bidirectional communication technology for web apps

Global 85.43% + 1.09% = 86.53%

unprefixed: 85.43% + 1.01% = 86.44%

Current aligned | Usage relative | Show all

| IE | Firefox | Chrome | Safari | Opera | iOS Safari * | Opera Mini * | Android Browser * | Chrome for Android |
|---|---|---|---|---|---|---|---|---|
| | | 31 | | | | | | |
| | | 36 | | | | | | |
| | | 37 | | | | | | |
| | | 39 | | | | | 4.1 | |
| 8 | | 40 | | | | | 4.3 | |
| 9 | 31 | 41 | 7 | | | | 4.4 | |
| 10 | 37 | 42 | 7.1 | | 7.1 | | 4.4.4 | |
| 11 | 38 | 43 | 8 | 29 | 8.3 | 8 | 40 | 42 |
| Edge | 39 | 44 | 9 | 30 | 9 | | | |
| | 40 | 45 | | 31 | | | | |
| | 41 | 46 | | | | | | |

## The WebSocket Protocol

- The WebSocket protocol was designed to work well with the existing Web infrastructure.
- As part of this design principle, the protocol specification defines that the WebSocket connection starts its life as an HTTP connection, guaranteeing full backwards compatibility with the pre-WebSocket world.
- The protocol switch from HTTP to WebSocket is referred to as a the WebSocket handshake.

## Typical handshake

- The browser sends a request to the server, indicating that it wants to switch protocols from HTTP to WebSocket.
- The client expresses its desire through the Upgrade header:

```
GET ws://echo.websocket.org/?encoding=text HTTP/1.1
Origin: http://websocket.org
Cookie: __utma=99as
Connection: Upgrade
Host: echo.websocket.org Sec-WebSocket-Key: uRovscZjNol/
umbTt5uKmw==
Upgrade: websocket
Sec-WebSocket-Version: 13
```

## Typical handshake

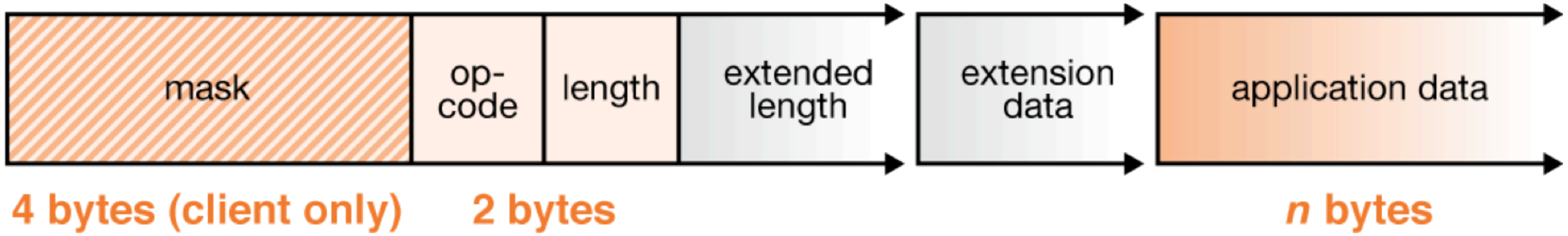
- If the server understands the WebSocket protocol, it agrees to the protocol switch through the Upgrade header.

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Fri, 10 Feb 2015 17:38:18 GMT
Connection: Upgrade
Server: Kaazing Gateway
Upgrade: WebSocket
Access-Control-Allow-Origin: http://websocket.org Access-Control-
Allow-Credentials: true
Sec-WebSocket-Accept: rLHCkw/SKsO9GAH/ZSFhBATDKrU=
Access-Control-Allow-Headers: content-type
```

- At this point the HTTP connection breaks down and is replaced by the WebSocket connection over the same underlying TCP/IP connection. The WebSocket connection uses the same ports as HTTP (80) and HTTPS (443), by default.

# The WebSocket Protocol

- Once established, WebSocket data frames can be sent back and forth between the client and the server in full-duplex mode.
- Both text and binary frames can be sent in either direction at the same time.
- The data is minimally framed with just two bytes. In the case of text frames, each frame starts with a 0x00 byte, ends with a 0xFF byte, and contains UTF-8 data in between.
- WebSocket text frames use a terminator, while binary frames use a length prefix.



## Using the HTML5 WebSocket API

With the introduction of one succinct interface (see the following listing), developers can replace techniques such as long-polling and "forever frames," and as a result further reduce latency.

```
[Constructor(in DOMString url, optional in DOMString protocol)]
interface WebSocket {
readonly attribute DOMString URL; // ready state const unsigned
short CONNECTING = 0; const unsigned short OPEN = 1;
const unsigned short CLOSED = 2;
readonly attribute unsigned short readyState; readonly attribute
unsigned long bufferedAmount;  // networking attribute Function
onopen;
attribute Function onmessage;
attribute Function onclose;
boolean send(in DOMString data);
void close();
};
WebSocket implements EventTarget;
```

## Using the HTML5 WebSocket API

- Utilizing the WebSocket interface couldn't be simpler. To connect to an end-point, just create a new WebSocket instance, providing the new object with a URL that represents the end-point to which you wish to connect, as shown in the following example.
- ws:// and wss:// prefix are proposed to indicate a WebSocket and a secure WebSocket connection, respectively.

```
var myWebSocket = new WebSocket("ws://www.websockets.org");
```

## WebSockets readyState

- The readyState property is a number:
  - **`0`** (**`socket.CONNECTING`**): client is establishing the connection to the server
  - **`1`** (**`socket.OPEN`**): client is connected; use **`send()`** and an **`onmessage`** event handler to communicate
  - **`2`** (**`socket.CLOSING`**): the connection is in the process of being closed
  - **`3`** (**`socket.CLOSED`**): connection is closed

## Using the HTML5 WebSocket API

- A WebSocket connection is established by upgrading from the HTTP protocol to the WebSockets protocol during the initial handshake between the client and the server.
- The connection itself is exposed via the "onmessage" and "send" functions defined by the WebSocket interface.
- Before connecting to an end-point and sending a message, you can associate a series of event listeners to handle each phase of the connection life-cycle.

```
myWebSocket.onopen = function(evt) { alert("Connection
open ..."); };
myWebSocket.onmessage = function(evt) { alert( "Received Message:
" + evt.data); };
myWebSocket.onclose = function(evt) { alert("Connection
closed."); };
```

## Using the HTML5 WebSocket API

- To send a message to the server, simply call "send" and provide the content you wish to deliver.
- After sending the message, call "close" to terminate the connection.
- As you can see, it really couldn't be much easier.

```
myWebSocket.send("Hello WebSockets!");
myWebSocket.close();
```

## WebSockets server side

- On the server side, we need a server that supports WebSockets
  - **`mod_pywebsocket`**: a Python-based module for Apache
  - **`Netty`**: a Java network framework that includes WebSocket support
  - **`node.js`**: a server-side JavaScript framework that supports WebSocket server implementation

## Websockets Demo & Tutorial

- Since we need a server that supports sockets an online demo is required. Let’s take a look at a quick demo.

```
https://www.websocket.org/echo.html
```

- Tutorial

```
http://www.webcodegeeks.com/html5/html5-websocket-example/
```

## More information on the API from w3

- The specification from the W3 has more information on implementation.

http://dev.w3.org/html5/websockets/

## Summary

- No more long polling, comet, ajax, etc.
- WebSockets use half the amount of connections as traditional methods
- Up to 1000:1 improvement on HTTP header traffic
- Up to 3:1 improvement on latency
- Based on specification
- Ability to support real-time callbacks
- Firewall and proxy supported
- Simple API

## HTML5 CROSS DOMAIN MESSAGING

MESSAGE ACROSS THE VOID

## Introduction

- We discussed previously that you can message on the same domain with postMessage() in HTML5
- The window.postMessage method also safely enables cross-origin communication.
- Normally, scripts on different pages are allowed to access each other if and only if the pages that executed them are at locations with the same protocol (usually both https), port number (443 being the default for https), and host (document.domain being set by both pages to the same value).
- window.postMessage provides a controlled mechanism to circumvent this restriction in a way which is secure when properly used.

## Security concerns

- If you do not expect to receive messages from other sites, do not add any event listeners for message events. This is a completely foolproof way to avoid security problems.
- If you do expect to receive messages from other sites, always verify the sender's identity using the origin and possibly source properties.
- Any window (including, for example, http://evil.example.com) can send a message to any other window, and you have no guarantees that an unknown sender will not send malicious messages.
- Having verified identity, however, you still should always verify the syntax of the received message.
- Otherwise, a security hole in the site you trusted to send only trusted messages could then open a cross-site scripting hole in your site.
- Always specify an exact target origin, not *, when you use postMessage to send data to other windows. A malicious site can change the location of the window without your knowledge, and therefore it can intercept the data sent using postMessage.

## Example

- In window A's scripts, with A being on <http://example.com:8080>:

```
var popup = window.open(...popup details...);

// When the popup has fully loaded, if not blocked by a popup blocker:

// This does nothing, assuming the window hasn't changed its location.
popup.postMessage("The user is 'bob' and the password is 'secret'",
                  "https://secure.example.net");

// This will successfully queue a message to be sent to the popup, assuming
// the window hasn't changed its location.
popup.postMessage("hello there!", "http://example.org");

function receiveMessage(event)
{
  // Do we trust the sender of this message?  (might be
  // different from what we originally opened, for example).
  if (event.origin !== "http://example.org")
    return;

  // event.source is popup
  // event.data is "hi there yourself!  the secret response is: rheeeeet!"
}
window.addEventListener("message", receiveMessage, false);
```

## Example

- In the popup's scripts, running on <http://example.org>:

```
// Called sometime after postMessage is called
function receiveMessage(event)
{
  // Do we trust the sender of this message?
  if (event.origin !== "http://example.com:8080")
    return;

  // event.source is window.opener
  // event.data is "hello there!"

  // Assuming you've verified the origin of the received message (which
  // you must do in any case), a convenient idiom for replying to a
  // message is to call postMessage on event.source and provide
  // event.origin as the targetOrigin.
  event.source.postMessage("hi there yourself!  the secret response " +
                           "is: rheeeeet!",
                           event.origin);
}

window.addEventListener("message", receiveMessage, false);
```

## Caveats

- Any window may access this method on any other window, at any time, regardless of the location of the document in the window, to send it a message.
- Consequently, any event listener used to receive messages must first check the identity of the sender of the message, using the origin and possibly source properties.
- This cannot be overstated: Failure to check the origin and possibly source properties enables cross-site scripting attacks.
- As with any asynchronously-dispatched script (timeouts, user-generated events), it is not possible for the caller of postMessage to detect when an event handler listening for events sent by postMessage throws an exception.

## Caveats

- The value of the origin property of the dispatched event is not affected by the current value of document.domain in the calling window.
- The value of the origin property when the sending window contains a javascript: or data: URL is the origin of the script that loaded the URL.

## DISCUSSION

How would you use websockets or cross domain messaging?

## Questions?

## References

- Crowther, R., & Lennon, J. (n.d.). HTML5 in action.
- Websockets API (n.d.). Retrieved May 25, 2015, from http://dev.w3.org/html5/websockets/
- https://developer.mozilla.org/
- https://www.websocket.org/