Worcester Polytechnic Institute
Digital WPI

Masters Theses (All Theses, All Years)

Electronic Theses and Dissertations

2009-01-19

Network Performance Evaluation within the Web Browser Sandbox

Artur Adam Janc
Worcester Polytechnic Institute

Network Performance Evaluation within the Web Browser Sandbox

by

Artur Janc

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

January 2009

APPROVED:

____

Professor Craig Wills, Major Thesis Advisor

____

Professor Mark Claypool, Thesis Reader

____

Professor Michael Gennert, Head of Department

Abstract

With the rising popularity of Web-based applications, the Web browser platform is becoming the dominant environment in which users interact with Internet content. We investigate methods of discovering information about network performance characteristics through the use of the Web browser, requiring only minimal user participation (navigating to a Web page). We focus on the analysis of explicit and implicit network operations performed by the browser (JavaScript XMLHTTPRequest and HTML DOM object loading) as well as by the Flash plug-in to evaluate network performance characteristics of a connecting client. We analyze the results of a performance study, focusing on the relative differences and similarities between download, upload and round-trip time results obtained in different browsers. We evaluate the accuracy of browser events indicating incoming data, comparing their timing to information obtained from the network layer. We also discuss alternative applications of the developed techniques, including measuring packet reception variability in a simulated streaming protocol. Our results confirm that browser-based measurements closely correspond to those obtained using standard tools in most scenarios. Our analysis of implicit communication mechanisms suggests that it is possible to make enhancements to existing "speedtest" services by allowing them to reliably determine download throughput and round-trip time to arbitrary Internet hosts. We conclude that browser-based measurement using techniques developed in this work can be an important component of network performance studies.

Acknowledgements

I would like to express my gratitude to my advisor, professor Craig Wills, whose guidance shaped this thesis and who helped me focus on the most relevant aspects of the work despite geographical obstacles.

My thanks are also due to the reader of this thesis, professor Mark Claypool, who provided software tools, statistical know-how and other helpful advice at various points throughout the development of this work.

During the writing of this thesis I have received enormous support from my partner, Kasia Koscielska, who has been understanding and willing to accommodate to my, often inopportune, work schedule. This work wouldn't have been the same without her help. Thank you also to our daughter Maja, whose daily routine introduced order and sanity into my life.

Most of all, however, I would like to thank my parents, Danuta and Alfred. I now understand that many things in life change, but their love and encouragement have always been a constant I could rely on. Throughout the years they have always given me their best; my greatest hope is to be to my children what they've been to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the recent years, software development has been increasingly shifting towards using the Web as a platform, with many of the most popular applications being released as Web 2.0 products available through a Web browser. Some of the most successful applications released recently, such as social networks and social news sites, video and photo sharing software, and various productivity tools are released only as Internet applications. Since the accessibility and performance of such software is closely dependent on the network availability and network conditions between each user and the application server, the ability to assess the quality of each client's connection becomes crucial for ensuring proper operation of any such software.

Our work focuses on evaluating the potential of the limited Web browser environment assessing the performance of a user's Internet connection. We categorize and evaluate communication mechanisms available natively in the browser, as well as those provided by the Flash plug-in. We discuss explicit communication mechanisms, such as the XMLHTTPRequest object in JavaScript and the URLRequest family of functions in Flash, as well as implicit methods utilize including the ability to initiate HTTP connections to retrieve external resources elements such as frames, images, and policy files. We implement network measurement techniques based on those mechanisms in order to infer as much as possible about the network conditions of the connecting client. The project has three main goals:

- to analyze the potential of the Web browser environment to infer network characteristics of the connecting client,
- to develop and validate a set of tests to reliably determine the client's network conditions, and
- to tie in with a larger network evaluation framework by providing an easily accessible environment for basic measurements; details of the more general measurement platform are discussed in related work[CKW07].

By operating in the standard environment of a Web browser, our measuring approach and testing system is available for the vast majority of Internet users without the need to install any software, thus reducing the effort to initiate measurements and obtain results. Designing network measurement tests such that they can be used within a larger network evaluation effort increases the applicability of the results we obtain; it allows such a framework to maximize the coverage of performed measurements by executing tests using all technologies supported by the browser for a large population of users.

One of the main outcomes of this work is the development a Web-based network measurement platform available through the sandbox environment of a Web browser. The platform allows any user to connect and initiate a sequence of measurements, and then display network performance results to the user. The creation of the platform was executed in five stages:

1. Investigation of Web browser mechanisms for retrieving network content, in-

cluding native JavaScript mechanisms, as well as functions provided by widely-installed plugins.

2. Analysis of restrictions placed on the network communication initiated by the browser and plugins (including, but not limited to the effect of same-origin policies), as well as the differences between the policies and implementations of those restrictions in major browsers.

3. Design and implementation of tests measuring basic network parameters, such as the download and upload throughput and round-trip time.

4. Development of a "streaming" HTTP server and the analysis of packet reception time variance (jitter) for data transfers paced at the server-side.

5. Exploratory work in network reconnaissance using available content-retrieval mechanisms, and verification of their applicability for assessing the topology and performance of the user's network environment.

The created network measurement environment is significantly different from available network testing setups in that it functions within the tightly constrained sandbox environment of a Web browser. Therefore, several restrictions apply to tests executed within such an environment:

- lack of direct access to the network layer,

- ability for browser scripts to directly connect only to the server from which the original page was retrieved via the same-origin policy, and

- no persistent storage on the client executing the test.

The first restriction has the most limiting effects for our platform, as it means that Web browsers are in general only capable of generating HTTP (and thus TCP) traffic, without possibility of transmitting UDP or ICMP packets directly (although additional third-party plug-ins, such as Java, might provide such capabilities). Furthermore, because of operating system and Web browser abstractions, scripts executed within the browser environment do not directly interact with the network layer and thus receive no information about events occurring at lower stages of the TCP/IP stack (such as the time of receiving a packet, or potential packet retransmissions).

We analyze the potential of explicit communication techniques to provide information about network layer events, and the accuracy of such information by comparing application layer "progress" events with data from packet traces. We also investigate data transfers similar in nature to streaming UDP traffic through the use of the jitter test, where a custom streaming HTTP server software sends TCP packets at a constant rate and the Web browser client records information about each "progress" event.

The second restriction is the ability to access content only from the originating server controlled by the browser's same-origin policy, built into Web browsers for security and privacy reasons. The same-origin policy specifies that information can only be shared between two HTML frames and iframes if both of them originate from the same domain, port and protocol; thus, explicit communication mechanisms such as XMLHTTPRequest in JavaScript are, by default, prohibited from making request to any host except the originating server.

While same-origin limitations can be relaxed by certain JavaScript and Flash mechanisms, explicit communication mechanisms do not provide a general capability of measuring network performance to an arbitrary Internet host. As implicit communication mechanisms are not subject to the same-origin policy, we compare results obtained through such mechanisms to data obtained by explicit techniques to determine the feasibility of utilizing implicit techniques to measure download throughput and round-trip time to any host connected to the Internet.

The problem with lack of persistent client-side storage is resolved by sending all test results to the testing server immediately after the data gathering phase has been completed. We present the design of our testing platform, which provides capabilities for automatic submission of results from the client's browser, presentation of collected results to the client and researchers, and an extensible mechanism for adding new kinds of network measurement tests to the system.

Based on the novel approach of utilizing the Web browser as a *bona fide* network measurement platform, this work outlines several key contributions to the area of Web-based network performance evaluation:

- We design and implement several scripting techniques to estimate throughput and round-trip time in JavaScript and Flash and determine their accuracy.
- We introduce the concept of implicit measurement techniques to allow reliable measurements of download throughput and round-trip time to arbitrary third-party servers.
- We compare the relative accuracy of all measurement techniques in several popular browsers for broadband and local network clients.
- We develop a "streaming" HTTP server and clients in JavaScript and Flash to measure the variability of packet reception times in a simulated streaming protocol.
- We outline the design of a testing platform to distribute browser-based network measurement tests, as well as automatically gather and present individual and aggregate results.

In the remainder of this thesis, we discuss background material and related work in Chapter 2. In Chapter 3 we present our approach to selecting and implementing explicit and implicit browser communication mechanisms and discuss the capabilities and limitations of each technique. We also present the design of the "jitter" server and client for measuring packet reception variability for a simulated streaming protocol. In Chapter 4 we analyze the relationship between results from our measurement techniques and the corresponding network layer parameters. Additionally, we list the assumptions and limitations affecting each technique. Chapter 5 provides an overview of the design of our measurement system, including the server-side and client-side considerations. Results of our several studies based on the discussed techniques are presented in Chapter 6. We present a validation of browser-based mechanisms by widely-used network measurement tools; we also evaluate browser performance for broadband and LAN clients for each developed measurement method. We also determine the relationship between progress events reported by the application layer with network layer data for explicit communication techniques, and present preliminary jitter test results. We conclude by discussing the implications of our results and presenting possible improvements in Chapters 7.

Chapter 2

Background

Traditionally, most network evaluation studies have, by necessity, focused on performing tests from well-connected university and commercial server locations, such as PlanetLab[PBFM06] or Archipelago[arc] facilities. These measurements, while providing a stable testing environment allowing for various types of network tests, have the drawback of not representing typical end user configurations, comprised mostly of broadband (DSL, cable) connections. To address this issue, the concept of user-centered network measurement has been proposed[CKW07].

An often-taken approach to measuring the performance of home network connections is to encourage users to download and install special-purpose measurement software. While successfully applied in projects such as NETI@Home[SRR06] and DIMES[dim], this approach requires explicit user participation without a clear benefit in return. In addition, such platforms do not usually collect data of any utility to the *user*, providing little incentive for users to participate. To counter this limitation, platforms such as DipZoom[WTR07] were proposed; however, they are still of primary interest to researchers rather than Internet users at large.

There have been studies trying to more closely evaluate the conditions of most home connections by focusing on the ability to infer information from replies to network queries initiated from a remote server[DHGS07]. While this is useful to gain a general understanding of the quality of home Internet connections, such an approach requires long-term probing and analysis, making it unsuitable for an on-demand summary of a particular user's connection.

Another recent approach is to make use of existing traffic from legitimate applications installed by end-users to infer some performance information about their network connections. Measurements have been performed using the BitTorrent[IPKA07] peer-to-peer protocol, as well as by making use of *spurious traffic* generated by worms or network misconfigurations[CGC+05].

Due to the recent popularity of "Web 2.0" applications, users have increasingly used Web browser software to interact with on-line content. Java-based tools have been created to perform measurements within the browser environment using Java applets[Rit07][CM02]. While more convenient than downloading a network-measurement application, Java applets require explicit user authorization to make low-level network calls; they also cannot exchange data with other objects on a Web page (and thus cannot be easily controlled by Web programmers), and are being increasingly replaced by AJAX and Flash-based implementations.

The most common Web-based application for network measurement are commercial "speedtest" services[Ook09][dsl][toa] often visited by users to determine their available bandwidth. Such services universally report only download and upload throughput and the round-trip time from the client to the speedtest's servers and provide no aggregate data about their users' network performance. Little work has been done to verify the accuracy of network performance results obtained from such services.

When using the Web browser to perform network measurements, privacy issues must be considered, as browsers often store sensitive user data. There have been attempts to analyze the privacy implications of existing browser mechanisms for downloading content[JBBM06] including works on the ramifications of the same-origin policy[Moz01] in various Web browsers. Since the increase in popularity of Web applications based on JavaScript, there have also been security analyses of the impact of the XMLHTTPRequest on the integrity and privacy of a user's browser environment[JW07].

Chapter 3

Approach

The primary goal of our work is to determine as much as possible about the network performance characteristics of a client through the use of a Web browser. An important initial assumption is the ability to convince a user to navigate to a website of our choosing, which would then initiate measurements and report results to our database.

The main task of the Web browser environment is to render websites, usually by utilizing a combination of a static markup language (such as HTML) with dynamic scripting capabilities, often in the form of JavaScript, Flash, ActiveX and Java applets.

The ability of a Web browser to initiate network communication is, then, a by-product of other essential browser functionalities. Web browsers do not provide any direct access to the network stack—retrieving content is always done through a high-level interface and is often implicit.

To determine potential methods of evaluating network performance we survey several mechanisms, both those provided natively by browsers as parts of common standards, as well as those provided by popular browser plug-ins. The criteria for using a particular method are:

- ability to initiate network communication and provide information about its results,
- popularity among Internet users, and
- ability to perform basic tests without explicit user permission.

The ability for sending/receiving data to/from the network, often through the form of requesting a resource from a remote location, is crucial for our purposes, as there is no possibility of evaluating network characteristics without the ability to interact with the network.

We consider only technologies with a high adoption rate to ensure that our results are applicable across a wide range of platforms and useful to Internet users at large.

Our final requirement is that our network performance tests can be run without asking the users for explicit authorization. Thus, we elect not to focus on techniques which pop-up digital signature questions, or ask users to install additional browser plug-ins or other software. This decision ensures that our techniques can be used by other network measurement projects or websites without impairing the user experience.

3.1 Selected Technologies

We inspected and evaluated browser mechanisms and plug-ins to select the ones which best match our criteria. We considered several technologies, including JavaScript, ActiveX controls, Flash objects, and Java applets (both signed and unsigned). Those technologies account for the vast majority of dynamic scripting objects available on the World Wide Web.

We chose not to utilize ActiveX due to the fact that the technology is limited to a single browser family and operating system (Microsoft Internet Explorer on Windows), and because ActiveX controls need to obtain user permission to execute under default Internet zone settings.

We also elected not to include Java applets in our study. Signed Java applets have powerful capabilities for inspecting the client's local system and have the ability to access low-level network functions[Rit07]; however, they require explicit user permission to be executed by the client's browser. Fewer capabilities are available to unsigned Java applets, which can execute without interacting with the user. Both signed and unsigned applets are subjects of a study related to this work[Rit07].

Thus, in this study we focus on the capabilities provide by the JavaScript and Flash environments to evaluate the network performance characteristics of a user visiting a website. JavaScript and Flash have become ubiquitous in the recent years, due to a large number of "Web 2.0" applications based on both technologies, and have reached adoption rates of over 95% among Internet users[Sch08][Inc08]. They are also available for almost any browser on any major operating system (Windows, Linux, Max OS X), and are supported by many mobile devices, including PDAs and cellular phones.

3.2 Throughput and Round-trip Time Testing

In both JavaScript and Flash we inspect interfaces that enable the browser to initiate a network connection and transfer data over the network. We investigate possible ways of utilizing those interfaces to receive and send data from and to different hosts on the network, as well as ways of timing this process. The ability to perform those operations would enable us to estimate network-level parameters such as the client's download and upload throughput and round-trip time to Internet hosts. A detailed analysis of such parameters is provided in Chapter 4.

3.2.1 Test Categorization

Based on the specifics of the invocation of network communication we divide available methods into two categories:

- Explicit: the browser environment provides an object with methods to retrieve data from a network URL using HTTP.
- Implicit: the browser can include resources (such as images or scripts) from the network. Dynamically inserting such a resource in the page results in the browser issuing a network request for it and a subsequent retrieval of data.

Internet traffic initiated by a script within the Web browser environment can be classified as belonging to one of the above categories. In all cases, network traffic initiated by JavaScript and Flash scripts is TCP-based and almost always utilizes HTTP. We now analyze the commonly-used explicit and implicit mechanisms in detail; other kinds of network traffic that can be initiated the Web browser, such as DNS queries, are discussed in Chapter 7.

3.2.2 Explicit Communication Mechanisms

Recent versions of JavaScript and Flash provide objects that facilitate asynchronous retrieval of content from the origin server[1] to improve user experience. Such objects are commonly used to request data from the server "in the background", in response to, or in anticipation of, a user's action. This behavior can be used to dynamically

[1] The origin server is the host from which the current page (HTML document) was received.

load a new image in Web-based photo sharing application (such as Flickr[Fli09]) rather than reloading the entire page, or pre-fetching map data to allow smooth scrolling in an on-line mapping service, as done in Google maps[Goo09]. These mechanisms can reduce the perceived loading time of a Web application and have become a significant component of many Web 2.0 applications.

In JavaScript this capability is provided by the XMLHTTPRequest object, also called XHR, mandated by a W3C standard[W3C08] and available in all major browsers. In Flash—or more specifically in ActionScript 3.0, which is the programming language upon which newer versions of Flash are based—the same functionality is provided via multiple interfaces in the `flash.net` package, including the `URLRequest` and `URLLoader` classes[Ado09a]. We refer to this family of functions and objects as "URLRequest", keeping in mind that implementations can use different function calls with the same effect.

The high-level aspects of explicit communication mechanisms in both Flash and JavaScript are similar; in discussing their properties we describe their common features, noting differences where appropriate.

In both cases network communication is achieved after the following phases: instantiating the object, setting appropriate parameters (such as the URL[2] of the resource to fetch, the request type and headers), setting function handlers which receive events about the request status, and sending the request.

The only required parameter that must be set is the URL of the resource to retrieve from the server. The programmer can specify the type of the HTTP request as either GET, POST, or possibly another value. Since in a POST request HTTP parameters are passed separately from the URL and their length is not restricted by the HTTP standard, this provides a way to upload an arbitrary amount of data

[2] The Uniform Resource Locator is a compact string representation for a resource available via the Internet[BLMM94]

to the origin server. It is also possible to set many HTTP header values as desired, except for headers such as *Host* and *Referrer* where such programmer control could introduce a security risk.

In order to receive the data fetched as a result of the request, a callback function must be set. In JavaScript, the request object's `onreadystatechange` handler receives updates at each phase of the request as it is processed; in the event of successful retrieval of data the handler should examine the contents of the response and process the received data. In Flash, the `addEventListener` function of a URLLoader object allows the setting of function handlers for various event types; `Event.COMPLETE` is signaled after the entire server response has been downloaded.

Thus, by recording two timestamps: just before sending the request and when the response has been fully received, it is possible to determine the approximate duration of the network communication, which can in turn provide information about the throughput and round-trip time to the origin server.

3.2.2.1 JavaScript XMLHTTPRequest

Using the XMLHTTPRequest object in JavaScript we developed a technique to fetch data from the origin server, as shown in Figure 3.1. The code can be inserted into any HTML page, either as a `<script>` element, or as an external resource.

The main task of the `getURL()` function is to retrieve the given URL and record the duration of the download process. The XMLHTTPRequest object is created on line 4; depending on the browser and version, it may be implemented using the `XMLHttpRequest` object or as `ActiveXObject("Msxml2.XMLHTTP");` for older versions of Internet Explorer. The `onreadystatechange` handler function defined on line 16 is invoked whenever the state of the request object changes; its purpose is to calculate the duration of the download and the throughput (lines 21-29). The request is sent to the network upon calling the `send()` function on line 43.

Our implementation of this technique closely follows the supplied example, with the addition of error-handling code, several optional parameters, and logging mechanisms.

3.2.2.2 Flash URLRequest

The implementation of an explicit communication technique in ActionScript 3.0 is conceptually similar to the JavaScript example and is given in Figure 3.2 . One difference in publishing Flash content is that the ActionScript code must be compiled into a Flash SWF file, and then inserted into an HTML page through an `<object>` or `<embed>` tag.

In the shown example, the `URDownload()` function on line 14 is automatically executed upon loading the script. The `runDownloadTest()` function creates a `URLLoader` object (line 19), adds a function handler to be executed upon the completion of the download (line 22), and sends the request to the network (line 32). The `completeHandler()` function determines its associated `URLLoader` on line 38, and calculates the duration and throughput of the transfer based on the data from the loader object.

This ActionScript example depicts the core steps necessary to provide throughput measurements in Flash—similar techniques are being employed by various online speed-test services[Ook09].

3.2.2.3 Limitations

Having demonstrated the utility of the described explicit communication mechanisms, there is one important limitation which affects them: both the XML-HTTPRequest and URLRequest objects can only send requests back to the origin

```
// Retrieve the URL and time the duration of the process.
function getURL(url) {
  // Create the XMLHTTPRequest object (cross-browser)
  var request = createXMLHttpRequest();

  // Prepare the request and set the URL. Append current
  // timestamp to the URL to make it unique and avoid
  // caching issues. We could specify "POST" here to be
  // able to send parameters in the request body.
  // The request does not get sent to the server yet.
  request.open("GET", url + ’?’ + (new Date()).getTime(),
               true);

  // This function will get called whenever the status of
  // the request changes. It is called asynchronously.
  request.onreadystatechange = function() {

    // Only log the end time after receiving the complete
    // response and if the HTTP status indicates success.
    if (request.readyState == 4 && request.status == 200) {
      var end_time = (new Date()).getTime();

      // Calculate the duration of the request (in ms)
      var duration = end_time - start_time;

      // And, possibly, the throughput in KB/s, assuming a
      // large amount of transferred data.
      var throughput = (request.responseText.length / 1024)
                       / (duration / 1000);

      // We could now do other processing of the result and
      // send it to the server to log it.
  }

  // We have now set up the request and handler function,
  // so we can finally send the request to the server.

  // Log the time (32-bit timestamp)
  var start_time = (new Date()).getTime()

  // For a POST request we could send a dictionary of
  // parameters rather than null.
  request.send(null);
}
```

Figure 3.1: JavaScript XMLHTTPRequest Technique Implementation.

```
package {
// Import necessary packages

// Implement our test as a generic descendant of Sprite
public class URDownload extends Sprite {

  // An example URL of the resource to retrieve
  private var resourceURL = "/files/1024KB";

  // The start time of the request
  private var startDate:Date = 0;

  // Download a resource from the server on startup
  public function URDownload() { runDownloadTest(); }

  // Set up the request object and send it.
  private function runDownloadTest():void {
    // Create a URLLoader object which for our request
    var loader:URLLoader = new URLLoader();

    // Set the handler to run when download completes
    loader.addEventListener(Event.COMPLETE,
      dompleteHandler);

    // Log start time and add to URL to avoid caching
    startDate = new Date();
    var request:URLRequest = new URLRequest(resourceURL +
                            "?" + startDate.getTime());

    // We could add POST variables to the request here.
    // Send the request to the network.
    loader.load(request);
  }

  // Handle the completed request
  private function completeHandler(event:Event):void {
    // Find our loader object
    var loader:URLLoader = URLLoader(event.target);

    // Log end time and calculate duration (in ms)
    var eT:Date = new Date();
    var d_ms:Number = eT.getTime() - startTime.getTime();

    // Calculate the throughput in KB/s
    var throughput:Number = (loader.data.length / 1024)
      / (d_ms / 1000);

    // We can now send the result to the server to log it.
  }
}}
```

Figure 3.2: Flash URLRequest Technique Implementation.

server, as defined by the same-origin policy[Moz01]. The policy only allows script access to resources located within the same origin, defined as the same:

- hostname,
- port, and
- protocol

as the URL from which the Web page was retrieved. Such a limitation is crucial for maintaining the security and privacy of the user; otherwise, malicious website authors could steal the user's credentials and other private information if the user was concurrently logged into another website[Goo08].

Same-origin policy restrictions can be somewhat relaxed. JavaScript provides a mechanism to allow the scripts on a page to access resources within the parent domain through setting the `document.domain` variable[Mic09], and Flash provides a method for webmasters to allow Flash scripts from arbitrary domains to communicate with their websites via the `crossdomain.xml` file[Ado09b].

However, the XMLHTTPRequest and URLRequest techniques can generally only be used to communicate with the Web server from which the original page was received and potentially with some other hosts whose webmasters collaborate with the author of that page.

While this constraint still allows some useful information to be gathered about the parameters of the client's network, it severely limits the generality of the developed explicit techniques. We now turn to the analysis of implicit communication mechanisms where such restrictions do not exist.

3.2.3 Implicit Communication Mechanisms

Since the beginnings of the World Wide Web, webmasters had the ability to include external resources, such as images, scripts or frames, in their websites. The ability to embed such resources from a third-party server quickly became a core functionality of the Web programming model, and, with the advent of Flash-based video streaming platforms, is still a popular mechanism.

From the perspective of the browser environment this capability is easy to achieve. After the initial HTML document has been loaded and parsed, the browser identifies resources used by the page. If any of them are locally cached, they are retrieved and inserted into the rendered document. If a resource is not locally available, an HTTP GET request for it is issued. The browser can then either parse the response data as it receives it (which is sometimes the case when retrieving images) or wait until the download is complete to parse the resource.

This functionality allows a website author to force the client's browser to issue a HTTP GET request to an arbitrary host to any network port. Combined with the page's ability to receive events when an element has been added to the page through the JavaScript `onload` event, this provides a mechanism to determine the amount of time to request and receive any resource from any HTTP server on the Internet.

A similar implicit method of downloading resources also exists in Flash. It is important to note that these technique can be employed to download resources from the origin server as well as any third-party server and are not subject to the same-origin policy.

3.2.3.1 JavaScript DOM Resource Loading

The Document Object Model (DOM) is a standardized JavaScript mechanism of organizing an HTML document in the browser memory in order to provide website authors with a simple interface of accessing and modifying page elements. An important capability of the DOM is that allows scripts to dynamically insert resources into the current document. Another relevant function of the DOM is the ability to receive events related to user actions (such as `onclick`, `onmouseover`), page/resource load status (`onload`, `onunload`, `onerror`) and others; the programmer also has the ability to define JavaScript handlers for such events.

By combining those functionalities we can measure the time necessary to send a request for a resource and receive a response through the following steps:

1. Log the start time
2. Dynamically insert a resource in the page with a custom onload handler
3. In the onload handler, executed when the resource has finished downloading, log the time and calculate the duration of the process.

An example implementation is provided in Figure 3.3. The `getResource()` function creates a container element to which the requested resource will be appended (line 14). By adding an `<img>` object on line 21, the script is implicitly asking the browser to send a network request for the `url`. The `downloadComplete()` function on line 29 is set as the `onload` handler for the `<img>` object and is called as soon as the network resource has been downloaded. It then calculates the download duration and throughput (lines 32-39).

3.2.3.2 Flash loadPolicyFile Resource Loading

The Flash counterpart of the JavaScript DOM loading technique is the most complex algorithm developed for this thesis. While the implementation details are quite different than the ones presented for DOM resource loading, both algorithms share many high-level similarities and use cases.

```
var startTime;
var endTime;
var resourceSize;

// This function takes a URL to download and the size
// of the resource at the given URL.
function getResource(url, size) {

  // Save the resource size
  resourceSize = size;

  // Get an element to which we'll dynamically a tag
  // referring to the resource we want to download.
  var container = document.getElementById('container');

  // Log the start time
  startTime = (new Date()).getTime();

  // Set the HTML contents of the container element to an
  // image tag with the src parameter set to the URL.
  container.innerHTML = '<img␣src="' + url + '?' + startTime
    + '"' + '␣onload="downloadComplete(this);"␣/>';

  // The browser will fetch the URL, which should be an
  // image. If the resource is not a valid image, we should
  // set a handler for onerror rather than onload.
}

function downloadComplete(el) {

  // Log the end time.
  endTime = (new Date()).getTime();

  // Calculate the download duration
  var duration = endTime - startTime;

  // Get the download throughput in KB/s
  var throughput = (resourceSize / 1024) /
      (duration / 0.001) )).toFixed(1);

  // We can now log the result to the server
}
```

Figure 3.3: JavaScript DOM Resource Loading Technique Implementation.

As mentioned in the discussion of the implications of the same-origin policy, Flash provides webmasters with a way of relaxing same-origin constraints. In earlier versions of Flash the only way to allow scripts from external domains to access resources on a Web server was to place a `crossdomain.xml` policy file in its root directory. The policy file must list all domains from which scripts are authorized to access resources on the given server—thus a webmaster could control which external domains can request information from his or her server.

For example, if a Flash script downloaded from `http://example.com` tried to send a request for a resource at `http://foo.com/resource`, the Flash environment first silently issued a request for `http://foo.com/crossdomain.xml`. If the file existed and included `example.com` in its whitelist of allowed domains, the Flash environment then issued the request to `http://foo.com/resource` and allowed the script to receive the results; otherwise it threw an exception indicating that the script did not have permission to access the resource.

This functionality was often used to allow easy communication between scripts downloaded from different subdomains of a single domain. The webmaster of `www.foo.com` could add the `intranet.foo.com` domain to its `crossdomain.xml` file, allowing Web servers in the intranet subdomain to easily communicate with public server-side script and resources available at `www.foo.com`. This mechanism also allowed the creators of public application programming interfaces (APIs) to allow any other webmaster to use their services, for example to create custom mash-ups of photos available through the Flickr web application[Fli09].

Since the release of Flash 7.0.19, programmers can specify a custom location of the policy file in addition the hardcoded name of `crossdomain.xml` in the server root, through the `loadPolicyFile()` function. The mechanism exists to allow server-side programmers without access to the Web server root to also allow third-party domains to request resources from the server applications.

Thus, a Flash script downloaded from `http://example.com` trying to access `http://foo.com/resource` can first call

```
loadPolicyFile("http://foo.com/custom-policy.xml");
```

and then request the given resource. The Flash environment will then download the `custom-policy.xml` file and if it contains a provision for example.com, the request for the resource will be made; otherwise, an exception will be thrown.

The Flash standard specifies that a URLRequest to a third-party domain will not be executed until all policy files requested through `loadPolicyFile()` have been inspected. This allows a programmer to time the downloading of any HTTP resource by passing it to Flash in place of a valid policy file through the following steps:

1. Log the time.

2. Create a `URLLoader` object and set its error handler to a custom function logging the time.

3. Call `loadPolicyFile(url)`; the function will return immediately and the Flash environment will issue a GET request for the URL;

4. Use `URLLoader.load()` to download the same URL as in step 3; the request will not be made until Flash receives a response to the previous `loadPolicyFile()` call and parses it as a policy file. Once the resource has been parsed, `URLRequest()` will return an error.

5. The time to download the URL disguised as a policy file is the difference between the start time and the time when the error handler for the URLRequest was invoked.

A sample implementation of this novel technique is provided in Figure 3.4. The `downloadResource()` function first creates a `URLLoader` object and sets its `HTTP_STATUS` handler function (lines 15-17). After calling `Security.loadPolicyFile` on line 24, the Flash environment initiates a request for the *url* in the background, and the function immediately returns. Calling the `load()` method of the `URLLoader` on line 31 will wait until Flash receives a response to the `loadPolicyFile()` call (line 24). Only when the resource has been downloaded via `loadPolicyFile()` will the `httpStatusHandlerDownload` (lines 37-45) be executed.

3.2.3.3 Limitations

The implicit communication mechanisms, in both JavaScript and Flash, are more complex than the explicit techniques: their implementation can be non-obvious and is more susceptible to programming errors than well-described applications of the explicit methods.

An inherent property of such a method of requesting and receiving data is that the browser does not provide any information about the current state of the request, and does not signal any intermediate events.

The DOM and loadPolicyFile methods can also only be used for downloading data. While the explicit methods can upload large amounts of data as arguments to the POST request, implicit techniques can only issue GET requests and the amount of data sent is limited by the maximum allowed length of the URL parameter.

However, a significant advantage of those mechanisms over the explicit techniques is that they can be used to retrieve resources from external domains. This makes it possible to determine the download throughput and round-trip time between the client and any Web server on the client's local network or the Internet.

```
// Global variables
private var downloadedFileSize:Number;
private var policyDownloadStartTime:Number;
private var errorTime:Number;

private function downloadResource(url:String,
  size:Number):void {

  // Save the URL and resource size
  downloadingURL = url;
  downloadedFileSize = size;

  // Create a URLLoader and set its event handler. The handler
  // will be invoked if we aren't allowed to retrieve the URL.
  urlLoader = new URLLoader();
  urlLoader.addEventListener(HTTPStatusEvent.HTTP_STATUS,
    httpStatusHandlerDownload);

  // Log the start time of the initial download
  policyDownloadStartTime = (new Date()).getTime();

  // Schedule a request for the URL as a policy file. Returns
  // immediately: the request will be sent in the background.
  Security.loadPolicyFile(url+"?" + policyDownloadStartTime);

  // Try to send the request for the URL as a regular
  // URLRequest. This will only happen once the policy
  // file has been downloaded.
  var request:URLRequest = new URLRequest(url + "?" +
    policyDownloadStartTime);
  urlLoader.load(request);
}

// This function is called when we receive a HTTP_STATUS event
// when trying to send the URLRequest, because we don't have
// permission to access the original URL.
private function
  httpStatusHandlerDownload(event:HTTPStatusEvent):void {

  // Log error time, calculate duration and throughput
  errorTime = (new Date()).getTime();
  var duration = errorTime - policyDownloadStartTime;
  var throughput:Number = (downloadedFileSize / 1024) /
    (duration / 1000));
}}}
```

Figure 3.4: Flash loadPolicyFile Technique Implementation.

3.3 Measuring Streaming Packet Reception Variability

In addition to analyzing the browser's potential to measure throughput and round-trip times, we also evaluated its application for gathering information about scenarios where data is streamed across the network.

While browsers have no mechanism for sending or receiving UDP traffic to emulate a streaming UDP protocol, it is possible for an HTTP server to send data at regular intervals to simulate streaming using TCP. Combined with the ability of explicit communication mechanisms to receive updates about the status of a request as it's being processed, it is, in some cases, possible for the browser to receive information when a new chunk of data has arrived.

Thus, it is possible perform the analysis of the packet reception variability in a simulated streaming protocol by recording times at which data is received by the browser. Such information can be useful for predicting performance of streaming media and VoIP applications which are becoming ported to the Web platform.

3.3.1 Jitter Test Server

For the server-side component of the jitter test we implemented a simple Python Web server based on the `BaseHTTPRequestHandler` class from the `BaseHTTPServer` module.

While the server is capable of sending HTML and script files to a connecting client, its main purpose is to serve resources in a "streaming" manner—by breaking them into fixed-size chunks and sending them at a specified interval.

The server first sets its TCP socket option to NO_DELAY via

```
server.socket.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
```

Then, after a client connects and requests a resource, the server divides the resource into chunks to fit in a single TCP packet and paces their sending, as shown in Figure 3.5.

The `sendResource()` function first creates a list of packets to be sent (lines 11-14) and determines the time to send each packet (lines 17-19). Then, while there are still packets to send the server writes packet data to the socket (line 33) and sleeps until the next packet needs to be sent (lines 40-42).

In our streaming HTTP server, the chunk_size and delay_ms parameters are set dynamically by the client as variables passed in the URL. This way the rate at which data is sent can be quickly modified and tests with various streaming rates can be conveniently executed.

3.3.2 Jitter Test Client

For the jitter test to be useful, the client script which requests a resource from the streaming server must be able to receive events about incoming data as it is being transferred. Therefore, only the explicit download techniques—XMLHTTPRequest and URLRequest—have the potential to provide the necessary information. In both cases it is simple to add intermediate event logging capabilities to our existing implementation.

3.3.2.1 JavaScript

In some browsers, including Mozilla Firefox, the XMLHTTPRequest `onreadystatechange` event gets triggered whenever there is data available in the browser socket. It is, then, easy to record information about times when data was received by modifying the original XHR handler as shown in Figure 3.6. Events are recorded by adding the timestamp, the event's `readyState` and the amount of the data already downloaded

```python
# Send resource (string or byte array) using the server,
# chunk_size bytes every delay_ms milliseconds.
def sendResource(server, resource, chunk_size, delay_ms):
  packets = []
  timestamps = []

  first_byte = 0
  last_byte = chunk_size

  # Break resource into chunks and put them in an array
  for i in range(len(resource) / chunk_size + 1):
    packets.append(resource[first_byte:last_byte])
    first_byte += chunk_size
    last_byte += chunk_size

  # Determine the times at which to send each chunk
  initial_timestamp = int(time.time() * 1000)
  for i in range(len(packets)):
    timestamps.append(initial_timestamp + i * delay_ms)

  # While there are packets to send
  while len(packets) > 0:

    # Inspect the current time
    current_time = int(time.time() * 1000)

    # If there are packets to send which should already
    # have been sent, write them to the socket.
    while packets and timestamps and \
      current_time >= timestamps[0]:

      # Write out data
      server.wfile.write(packets[0])

      # Get rid of first packet and timestamp
      del packets[0]
      del timestamps[0]

     # Sleep until the next packet should be sent
     if timestamps and timestamps[0] and \
       timestamps[0] >= current_time:
       time.sleep((timestamps[0] - current_time) / 1000.0)

     # Log information about packet sending times

  return
```

Figure 3.5: Streaming HTTP Server Paced Sending Implementation in Python.

```
 1  // Record all intermediate events in this array
 2  var eventArray = new Array();
 3
 4  function handleIntermediateEvents(request) {
 5    // Record the timestamp, request state and
 6    // the amount of data received so far
 7    eventArray.push(new Array(new Date(),
 8      request.readyState, request.responseText.length));
 9
10    if (request.readyState == 4 && request.status == 200) {
11      // The resource has been downloaded. Inspect
12      // timestamps and handle data as in the original example.
13    }
14  }
15
16  // handleIntermediateEvents must be set as the
17  // onreadystatechange handler for our XMLHTTPRequest
18  // before its send() method is called.
```

Figure 3.6: JavaScript Jitter Client Implementation.

to the *eventArray* on lines 7-8.

However, some browsers only report onreadystatechange events when the *status* of the request changes, and ignore them in the case when new data became available. In other browsers, such as the Internet Explorer family, it is not possible to inspect the the `request.responseText` variable until the entire resource has been downloaded. Therefore, JavaScript does not provide a general method of determining the amount of partial downloaded data, and its utility for the jitter test is limited to browsers which fully support getting information from intermediate XMLHTTPRequest events.

3.3.2.2 Flash

The behavior of Flash when recording intermediate events during a download is consistent across browsers. Flash allows programmers to set handlers for several different events related to the status of the `URLRequest` object, including `ProgressEvent.PROGRESS` which is triggered when new data is available during a download process. The event handler can then inspect the response and determine how much data was downloaded since the last time the event occurred.

A simple implementation of this behavior, based on the explicit `URLRequest` download method, is shown in Figure 3.7. The main difference is the addition of an event listener for the `PROGRESS` event on line 14. The `progressHandler()` function (lines 21-43) records the timing of events and the amount of downloaded data for each event.

By handling progress events Flash can determine the times when data was received from the server. By comparing this information to the server's log of timestamps when data was sent we can inspect the variability of packet arrival times to see how often and by how much do packets get delayed in the network layer.

3.4 Summary

The work presented in this chapter outlines our main considerations for developing a set of network performance tests for the Web browser environment. We discuss the available scripting technologies and select JavaScript and Flash as the most appropriate scripting environments for our work. We evaluate network communication mechanisms available to programmers within the Web browser and categorize them as explicit or implicit based on the level of control over the network request given to the programmer. We develop four measurement technologies: JavaScript XMLHTTPRequest, Flash URLRequest, JavaScript DOM resource loading and Flash loadPolicyFile resource loading and discuss their potential for gathering network performance information, as well as their limitations. We also present the design of a custom HTTP server for the purpose of determining packet reception variability

```
// Global variables to keep track of download progress
var packetNumber:Number = 0;
var previousDownloadedSize:Number = 0;
var globalEventLog:String;

private function runDownloadJitterTest():void {
  // Create a URLLoader object for sending our request
  var ldr:URLLoader = new URLLoader();

  // Set the handler to execute when the download completes
  ldr.addEventListener(Event.COMPLETE, completeHandler);

  // Handler progress events when data becomes available.
  ldr.addEventListener(ProgressEvent.PROGRESS, updateHandler);

  /* Set up the request and send it as before ... */
  return;
}

// Runs when data becomes available for a URLRequest download.
private function progressHandler(event:ProgressEvent):void {

  // How many new bytes were received?
  var newBytes:Number = event.bytesLoaded - previousSize;

  // If we received new data, log it, otherwise skip.
  if (newBytes > 0) {

    // Keep a text log of progress event times for simplicity
    var event_text:String = packetNumber.toString() + "␣"
      + (new Date()).getTime() + "␣";
    event_text += newBytes.toString() + "␣("
      + event.bytesLoaded.toString() + ")\n";

    // Save event information to global download log
    globalEventLog += event_text;

    // Increase observed packet count and update byte count;
    packetNumber += 1;
    previousDownloadedSize = event.bytesLoaded;
  }
  return;
}
// The downloadCompleteHandler function can inspect the
// globalEventLog and log it to the server.
```

Figure 3.7: Flash Jitter Client Implementation.

in a simulated streaming protocol. The potential of the techniques described in this chapter to determine network performance characteristics is discussed in Chapter 4.

Chapter 4

Study

So far, we have presented our approach to obtaining network-related information from the Web browser environment. We now describe and examine the relationship between the indirect values obtained through the application of the developed measurement techniques and actual network layer parameters.

We focus on the types of information that can be inferred through the use of mechanisms available in browsers, and its accuracy. We present the validation of results, comparing them to directly measured values, obtained through other network communication mechanisms and through gathering packet traces.

4.1 Measured Network Parameters

In order for our developed techniques to be of general use, we first establish which types of information can be obtained. The most commonly measured parameters are:

- download throughput,
- upload throughput, and

- round-trip time.

Nearly all on-line speedtest services focus on providing these three values, as measured between the connecting client and the speedtest provider's closest server.

For this thesis we measure those parameters for the connection between the client and our test server. In addition, through the use of the jitter test we also determine packet reception variability times between the client and our server to simulate the behavior of a streaming protocol.

Employing the implicit communication techniques also enables us to test the download throughput and round-trip time to third-party servers.

While these quantities are of primary importance in our work, as they are of highest practical importance, we also investigate other applications of the developed measurement techniques. Such other uses include the ability to gather information about the client's local network setting through the use of querying hosts on the local network and determining open ports. We also consider using the implicit techniques to gather information about the round-trip time to the client's local DNS server; preliminary results are presented in Chapter 7.

4.1.1 Network Parameter Listing

We identify a total of eight parameters of interest that can potentially be obtained through the measurement techniques we developed. All developed techniques, along with the network parameters they are capable of estimating, are presented in Table 4.1; a discussion of the process used to determine each quantity is provided in Section 4.1.2.

Table 4.1: Network Parameter Support by Measurement Technique.

	Download: origin	Upload: origin	Roundtrip: origin	Streaming variability: origin	Download: 3rd-party	Upload: 3rd-party	Roundtrip: 3rd-party	Client net. parameters
JavaScript: XHR	Yes	Yes	Yes	No	No	No	No	No
JavaScript: DOM	Yes	No	Yes	No	Yes	No	Yes	Yes
Flash: URLRequest	Yes	Yes	Yes	No	No	No	No	No
Flash: loadPolicyFile	Yes	No	Yes	No	Yes	No	Yes	Yes
Jitter (JavaScript or Flash)	No	No	Yes	Yes	No	No	No	No

4.1.2 Determining Network Parameters Within the Browser

Throughput and round-trip time parameters listed in Table 4.1 are quantities that are often used to describe network performance. It is important to note that for the purpose of this thesis we make a distinction between the throughput and round-trip time to the origin server and to third-party hosts. This is done because the methods of measurement for the origin server cannot always be used for third-party measurements due to the same-origin policy and related restrictions.

The variability of packet reception in a packet stream, while not often inspected for network performance evaluation, can provide insight into the amount of packet loss and transient network interruptions. The parameters and methods of measuring them are shown below. The methods are abbreviated as: XHR (JavaScript XML-HTTPRequest), URLRequest (Flash URLRequest family), DOM (JavaScript DOM resource loading) and LPF (Flash loadPolicyFile resource loading).

1. Download throughput from origin server.

   A script issues a request for a large resource from the origin server and calculates throughput based on the time to receive a complete response.

Techniques: all.

2. Upload throughput to origin server.

   A script issues a POST request for a small resource from the origin server and attaches a large amount of data as POST variables. Upload throughput is calculated based on the time to receive an HTTP response from the server.

   Techniques: explicit (XHR and URLRequest).

3. Round-trip time to origin server.

   A script issues a request for a small resource (so that its size and the HTTP server headers fit within one IP layer packet), in order to establish a TCP connection with the server and perform any necessary DNS lookups. After the resource is received, the script requests the same resource again—the round-trip time is estimated the time between sending the second request and receiving the resource.

   It is important to ensure that the origin Web server respects the HTTP `Connection: keep-alive` header sent by modern Web browsers to keep the TCP connection open, as discussed in Section 4.1.3. If the server closes the connection after the first request, the second request will need to re-establish the TCP connection and the reported round-trip time will be twice as high as the real value.

   Techniques: all.

4. Reception time variability for streaming packets from origin server.

   A client requests a resource from the streaming HTTP server. The server and client keep track of timestamps when resource data was sent/received. The server determines the potential delay in reception time for each packet.

Techniques: explicit (XHR and URLRequest).

5. Download throughput from third-party servers.

   A script issues a request for a large resource from the remote server and calculates throughput based on the time to receive a complete response.

   Techniques: implicit (DOM and LPF).

6. Upload throughput to third-party servers.

   Determining the upload throughput to remote servers is not possible in general. As an exception, it is possible to measure the upload rate using URLRequest to those servers which use a wildcard whitelist in their crossdomain.xml file to allow clients from any domain to connect.

   Techniques: none (or URLRequest to select servers).

7. Round-trip time to third-party servers.

   A script issues a request for a small resource (so that its size and the HTTP server headers fit within one IP layer packet) from the remote server, in order to perform a DNS lookup and establish a TCP connection. After the resource is received, the script requests the same resource again—the round-trip time is estimated the time between sending the second request and receiving the resource.

   As with the round-trip time to the origin server, it is important to ensure that the target server respects the `Connection: keep-alive` HTTP header. For servers that close the connection after the first request, the reported round-trip time should be estimated as half of the measured value.

   Techniques: implicit (DOM and LPF).

8. Local network configuration discovery.

   A script can request resources to try and establish TCP connections with any host on its local network, even if the network resides behind an Internet firewall. This makes it possible to determine which hosts are available and their open ports. It is also possible to repeatedly query for resources from nonexistent domains to perform timing analysis of the connection to the client's local DNS server. As the main focus of our work is network performance measurement, such tests have not been implemented for this thesis; discovery techniques are presented as a direction for future work in Section 7.1.

   Techniques: implicit (DOM and LPF).

4.1.3 Assumptions

Due to the high-level nature of browser scripts, and the inability to directly interact with the network layer, any tests executed within the browser environment must carefully consider various aspects of the client and server behavior, as well as their interaction. When loading resources from the server, browser scripts might sometimes be affected by delays caused by the server, the network, the browser environment and even other software running on the same hardware as the browser.

While in the discussion of our results presented in Chapter 6 we validate obtained results within the browser by other means, it is important to list factors that can affect the results and our assumptions related to them.

1. Establishing a TCP connection.

   In our download throughput results we do not account for the necessity to establish a connection with the server and send the request for the resource, which takes at least two round-trip times. If the downloaded resource is sufficiently large, we treat the initial delay as negligible compared to the download duration.

2. Server processing delays.

   It is possible that the HTTP server requires a non-negligible amount of time to parse the request and return the appropriate resource. This is especially pronounced in upload throughput tests, where the client sends a large POST request (several megabytes of data), which must be fully parsed before the server can reply.

3. Maximum packet size.

   We assume the maximum size of a network layer packet to be about 1500 bytes. Furthermore, we assume that most data packets from the server are of maximum allowed size. We therefore expect that a HTTP response where the size of the headers and the data does not exceed 1500 bytes will arrive in a single IP packet.

4. HTTP header size.

   In most cases scripts have only access to the data portion of the HTTP response, and not any of the headers. Therefore, we do not account for the extra data in the HTTP header when calculating the throughput. Another potential issue exists in round-trip time measurements. If the server adds headers to the request so that the size of the entire HTTP response exceeds the maximum allowed size of a network packet, the response will be split into multiple packets, which could increase the measured round-trip time. Based on the resources selected for round-trip time measurement and an analysis of server header length, we assume that this does not occur for our measurements.

5. HTTP `Connection: Keep-alive`.

   We expect browsers to set the HTTP `Connection` header to `keep-alive`. We expect the server to respect that setting, so that after establishing a connection to the server through downloading a resource, a subsequent request made within a short time interval does not need to re-establish the TCP connection.

6. Network rate limiting.

   We expect that at no point between the client and server is the data rate intentionally limited, except for ISPs where the client's entire available bandwidth is subject to such treatment (e.g. some cable Internet providers.)

7. Transient network congestion.

   Our approach cannot detect transient network failures or congestion conditions on the client side. Therefore our results do not account for the possibility of other hosts on the user's local network deteriorating the client's performance. It is of note that even if such conditions occur, they will affect the perceived network performance for the user in general, and not only for our measurements.

8. User activity.

   Our approach cannot account for the browser user's activity during the measurement process. Thus, if there are ongoing file downloads, or if the user's browser engages in a computationally intensive task, the perceived network performance will suffer.

9. DNS queries.

   For round-trip time measurements, we expect that the second request for the same resource will not cause a DNS query to be made. While Web browsers generally cache DNS results for performance reasons and to protect against DNS rebinding attacks [JBB$^+$07], we cannot guarantee that no additional queries will be attempted.

4.2 Summary

This chapter presents our work on determining which kinds of network performance information can be obtained from within the Web browser environment using the techniques developed in Chapter 3. We list the network parameters of interest and analyze the potential of each developed measurement method to determine those parameters. In addition to often-measured quantities such as download throughput, upload throughput and round-trip time, we analyze the ability to determine packet reception variability in a streaming protocol and the possibility of performing local network reconnaissance. We present a detailed analysis of how each network parameter can be obtained using browser-based measurement methods and list our assumptions regarding the behavior of the browser environment, as well as the client-server interaction.

Chapter 5

System Design

The main focus of this thesis work is to examine and evaluate techniques browser-based network measurement techniques; due to the specific nature of the browser environment as a measurement platform we were led to create a Web application for performing tests, analyzing and displaying results.

The focus of the server-side architecture and client script organization is to provide a test executing and logging environment that:

- allows for easy addition of new types of tests and modification of test parameters,
- provides common logging mechanisms for various test types,
- provides an automatic result submission mechanisms to log all relevant information to the server without requiring user interaction, and
- allows automatic display of server-parsed results to client and researchers.

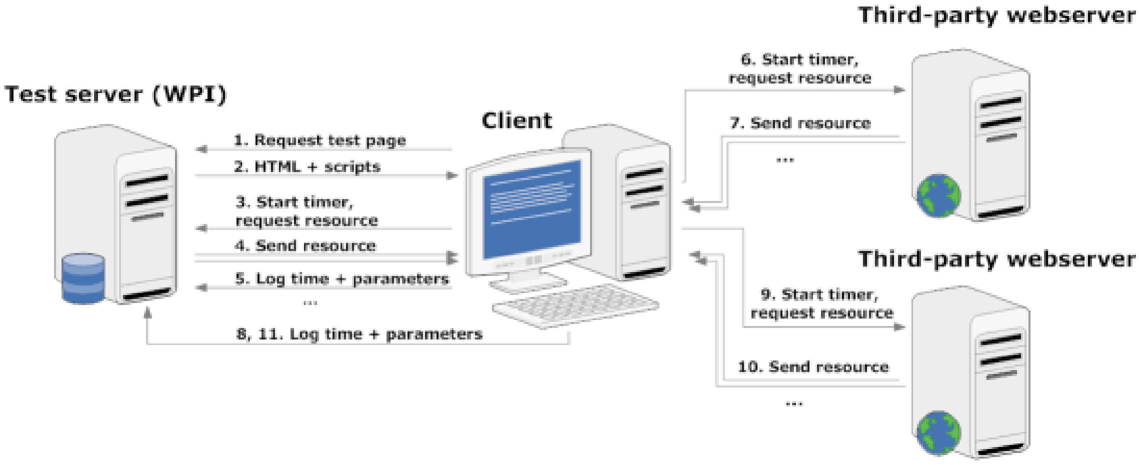


Figure 5.1: Test Platform System Design.

5.1 System Architecture

Our measurement test system is, predictably, implemented as a Web-based application available at `http://cew-research.cs.wpi.edu/sandbox/tests`

A client connecting to the test website is served the script files (in Flash and JavaScript), and a static HTML page for reporting results.

For throughput and round-trip time measurements, tests are set to repeat every 10 minutes, to allow for several tests runs to be executed while the user continued his or her work in other browser tabs. A single test consists of a measurement of the download and upload throughput and the round-trip time using each of the available techniques (XHR, URLRequest, DOM and LPF), to either the server or a third-party host. After each individual measurement the client sends a summary log of the result to the server and performs the next test. As throughput and round-trip results are gathered, the HTML page provides the user with information about each test result. The overall system design is shown in Figure 5.1.

For jitter tests, the client initiates the request for a streaming resource and keeps track of intermediate events related to the transfer. After the download process is complete, the client sends a list of timestamps corresponding to those events to the server. The server combines its own log of packet sending times and the user-submitted data to display statistics and graphs of packet reception variability to the user.

5.2 Client-side Architecture

Due to the specifics of our measurement technique, the organization of the client-side measurement component is especially important.

For throughput and RTT measurements each of the developed techniques is implemented in a separate script file (.js for JavaScript or .swf for Flash). JavaScript files are included through a

```
<script src="...">
```

reference. Flash files are embedded using the `SWFObject` library.

A list of tests to execute and their arguments is specified in the main HTML file. The file schedules the execution of the tests, and refreshes itself after a specified period of time to repeat the measurements.

The core of each test is based on the techniques outlined in Chapter 3. Each test also sets a flag to prevent other tests from running until it finishes, to eliminate the possibility of two measurements being performed at the same time.

5.2.1 JavaScript-Flash Interface

It is important to allow dynamic scheduling of tests and to allow communication between scripts; this is necessary to make ensure that no two tests execute concurrently, and to have common result reporting and logging mechanisms, to minimize the amount of duplicate code. However, this proves to be a non-trivial task, since scripts are implemented in both JavaScript and Flash and there are no simple native ways of passing messages between the two programming environments.

To achieve that purpose we implement a message-passing interface based on the `ExternalInterface` Flash class. This allowes us to call Flash functions from JavaScript and access JavaScript methods from Flash. As the JavaScript environment is a more powerful development platform, due to the number of available debugging tools and the ability to modify scripts by editing a text file (as opposed to re-compiling an ActionScript file and uploading it to the Web server), we develop all common components in JavaScript. Thus, JavaScript is responsible for initiating measurement functions implemented in Flash, receiving their results, and reporting them to the user and the test server.

When the JavaScript `log()` function is called, with results from either a Flash or JavaScript test, the script determines the browser name and version, and other parameters to be reported to the server and submits the data to the server log. It also formats the results in a consistent manner and displays them in the appropriate portion of the page for the user to view.

5.3 Server-side Architecture

Apart from sending the HTML, scripts and resource files to the client, the main purpose of the test server is to collect and store measurement results. The logging component is written in the Python language and utilizes a text file with `name:value` pairs for reported quantities for each test, with one test result per line.

An example line is shown in Figure 5.2.

The following data is gathered by the server:

```
type:download size:2097152 start_time:1228408407373 end_time:1228408407614
platform:IE_7_on_Windows server_time:1228408412131 client_ip:130.215.239.36
url:http%3a//cew-research.cs.wpi.edu/files/2048KB method:flash/urlrequest
```

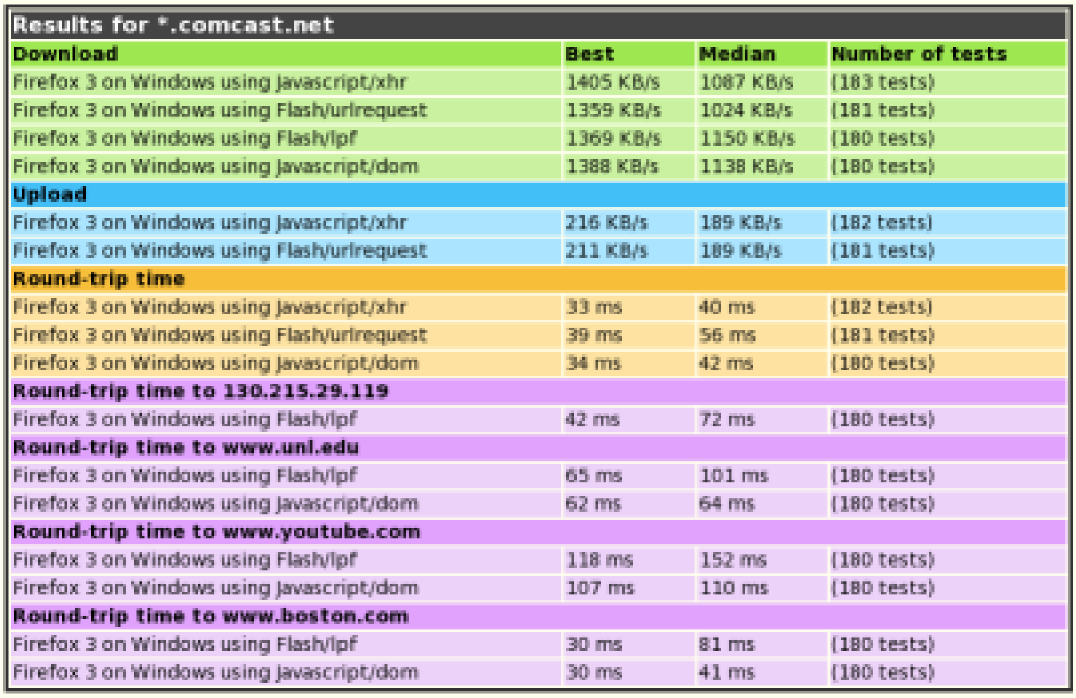
Figure 5.2: Example Server Log Line.

- Test type: download, upload, or ping.
- Method: one of XHR, URLRequest, DOM or LPF.
- URL: address of downloaded resource.
- Size: length of downloaded resource in bytes.
- Platform: The client browser name, version, and operating system.
- Client IP: Address of connecting client.
- Timestamps of test start, end, and submission time.

Due to the flexibility in the logging mechanism, it is easy to add new kinds of tests solely by modifying the *method* parameter and supplying the other data as with any other measurement method.

5.4 Reporting

The test server provides several ways of inspecting gathered results. Apart from simple text summaries for each test, which lists the computed values of the network parameters of interest, the server is also capable of gathering all test results from a single client and summarizing them in a table. An example result is shown in Figure 5.3



Results for *.comcast.net

Download	Best	Median	Number of tests
Firefox 3 on Windows using Javascript/xhr	1405 KB/s	1087 KB/s	(183 tests)
Firefox 3 on Windows using Flash/urlrequest	1359 KB/s	1024 KB/s	(181 tests)
Firefox 3 on Windows using Flash/lpf	1369 KB/s	1150 KB/s	(180 tests)
Firefox 3 on Windows using Javascript/dom	1388 KB/s	1138 KB/s	(180 tests)
Upload			
Firefox 3 on Windows using Javascript/xhr	216 KB/s	189 KB/s	(182 tests)
Firefox 3 on Windows using Flash/urlrequest	211 KB/s	189 KB/s	(181 tests)
Round-trip time			
Firefox 3 on Windows using Javascript/xhr	33 ms	40 ms	(182 tests)
Firefox 3 on Windows using Flash/urlrequest	39 ms	56 ms	(181 tests)
Firefox 3 on Windows using Javascript/dom	34 ms	42 ms	(180 tests)
Round-trip time to 130.215.29.119			
Firefox 3 on Windows using Flash/lpf	42 ms	72 ms	(180 tests)
Round-trip time to www.unl.edu			
Firefox 3 on Windows using Flash/lpf	65 ms	101 ms	(180 tests)
Firefox 3 on Windows using Javascript/dom	62 ms	64 ms	(180 tests)
Round-trip time to www.youtube.com			
Firefox 3 on Windows using Flash/lpf	118 ms	152 ms	(180 tests)
Firefox 3 on Windows using Javascript/dom	107 ms	110 ms	(180 tests)
Round-trip time to www.boston.com			
Firefox 3 on Windows using Flash/lpf	30 ms	81 ms	(180 tests)
Firefox 3 on Windows using Javascript/dom	30 ms	41 ms	(180 tests)

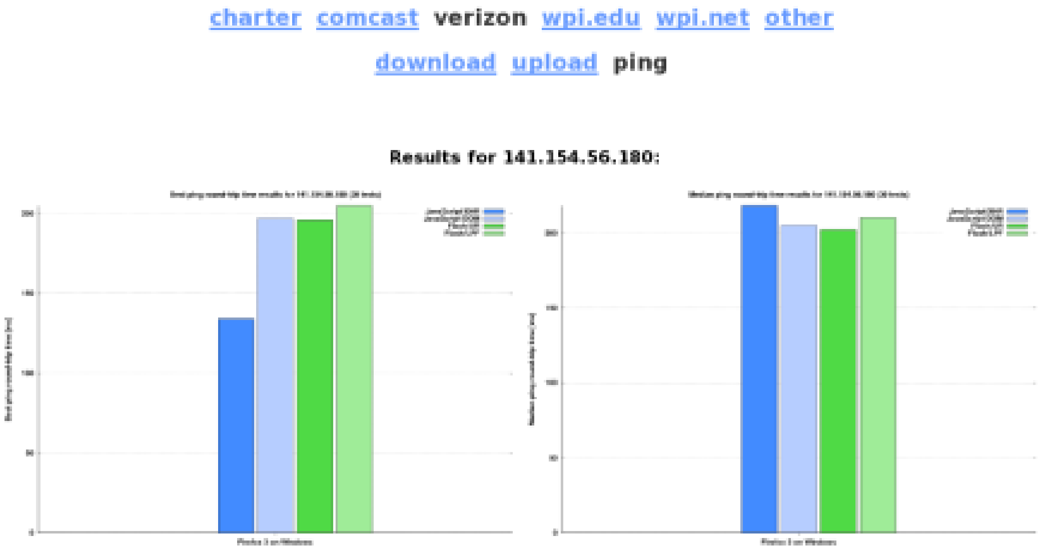
Figure 5.3: Example Host Result Summary.

The server reporting mechanism is also capable of automatic graphing of aggregate results based on the browser software and ISP of each client. An automatically-generated plot is presented in Figure 5.4.

For jitter test results, the server creates a report with a time plot of the download progress and a distribution of packet reception times, as shown in 5.5. The report is shown to the user as a Portable Document Format (PDF) file and stored on the server to allow for quick inspection of all gathered results.

5.5 Summary

This chapter outlines the architecture of our testing server and client platforms. We present the design goals for our system and the benefits they offered for our testing methodology. We outline the implementation of a JavaScript-Flash interface based


charter comcast verizon wpi.edu wpi.net other

download upload ping

Results for 141.154.56.180:

Figure 5.4: Example of Automatic ISP-based Latency Result Summary.

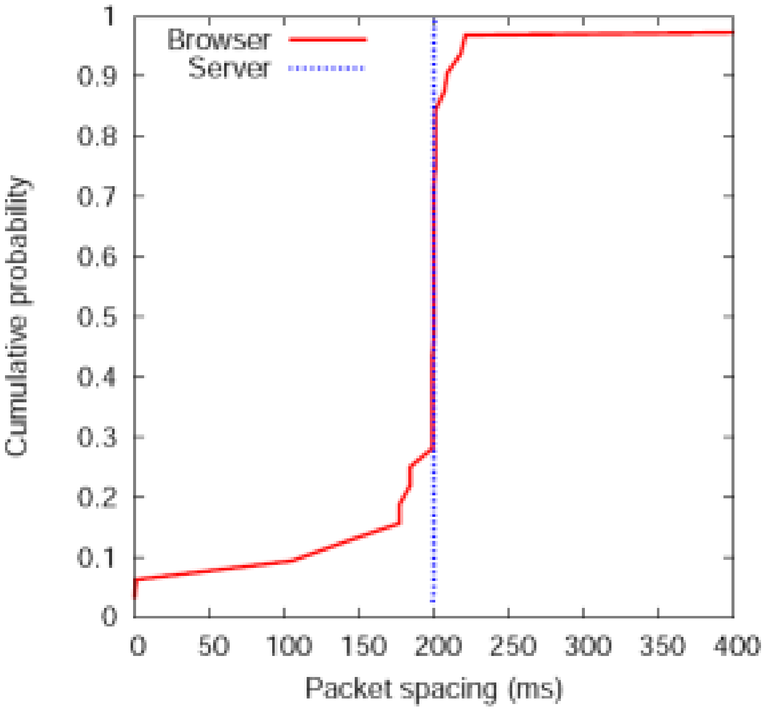


Figure 5.5: Automatically-generated Graph of Jitter Test Results

on the `ExternalInterface` class. We also present the design of our server system and details of the data storage format. We conclude by providing examples of automatically-created visualizations of test data gathered during our experiments. Other examples of dynamically-generated reports are provided in the discussion of our test results in Chapter 6.

Chapter 6

Results

After completing the development of our measurement techniques and finalizing the implementation of the server logging and reporting scripts, we conducted several tests to evaluate the accuracy and feasibility of each technique.

6.1 Controlled Study

To perform initial validation in a controlled environment, we executed a test in several browsers, using all the developed measurement techniques as well as non-browser measurement methods.

The client hosts were located on the US West Coast in a campus residential network, connecting to a server on the WPI academic network. The clients used Windows XP and Linux, and tested several major browsers for throughput and round-trip time. Tested platforms are shown in Table 6.1.

Tests were repeated every 12 minutes for a three-hour period starting at midnight PST. Care was taken to ensure that no two tests are executed at the same moment by spacing out the start time of the measurements in each browser.

Table 6.1: Contolled Study: Tested Browsers.

OS	Browser	Browser Version	Flash Version
Windows XP	Internet Explorer	7.0	9.0
Windows XP	Mozilla Firefox	2.0	9.0
Windows XP	Google Chrome	0.9	9.0
Ubuntu Linux 8.04	Mozilla Firefox	3.0	9.0

6.1.1 Download Throughput

Download throughput results were measured for each of the four developed techniques for four browsers shown in Figure 6.1. The downloaded test file size was 2MB. Validation was done by using the `wget` tool to download the same file from the test webserver.

A cursory examination of the individual results indicates that, while there is variability within the results for each method, most measurement methods seem to give *similar* values for the download throughput.

The mean results for each method and their 95% confidence intervals are shown in Figure 6.2.

Among the download throughput results, one method—XMLHTTPRequest in Google Chrome—reports the obtained values to be noticeably lower than other measurement methods (about 15% difference from the mean of remaining techniques). Combined with the consistency of results obtained using XMLHTTPRequest in Google Chrome, we suspect that there might exist a systematic issue impacting the performance of XHR downloads in this browser.

However, there are no general differences between the four developed techniques in other browsers, indicating that all of them are capable of producing useful download throughput results in most environments.

It is also noteworthy that the implicit technique measurement results do not show

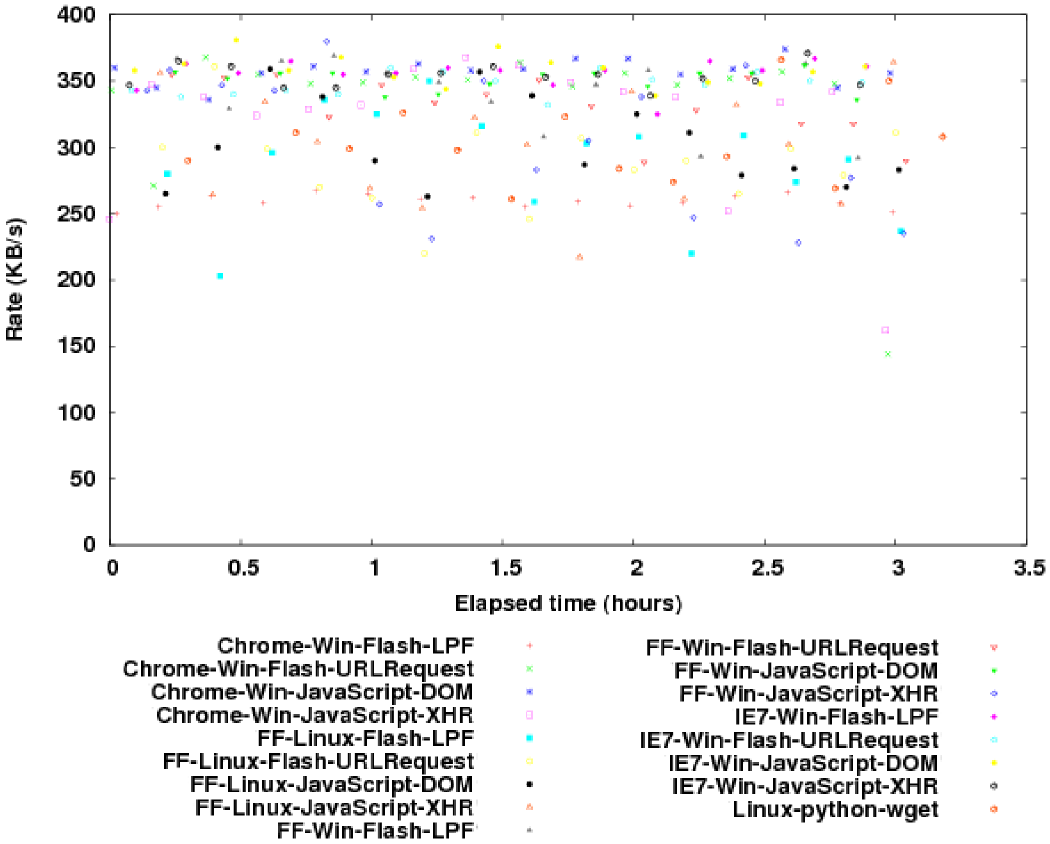


Figure 6.1: Controlled Download Throughput Results.

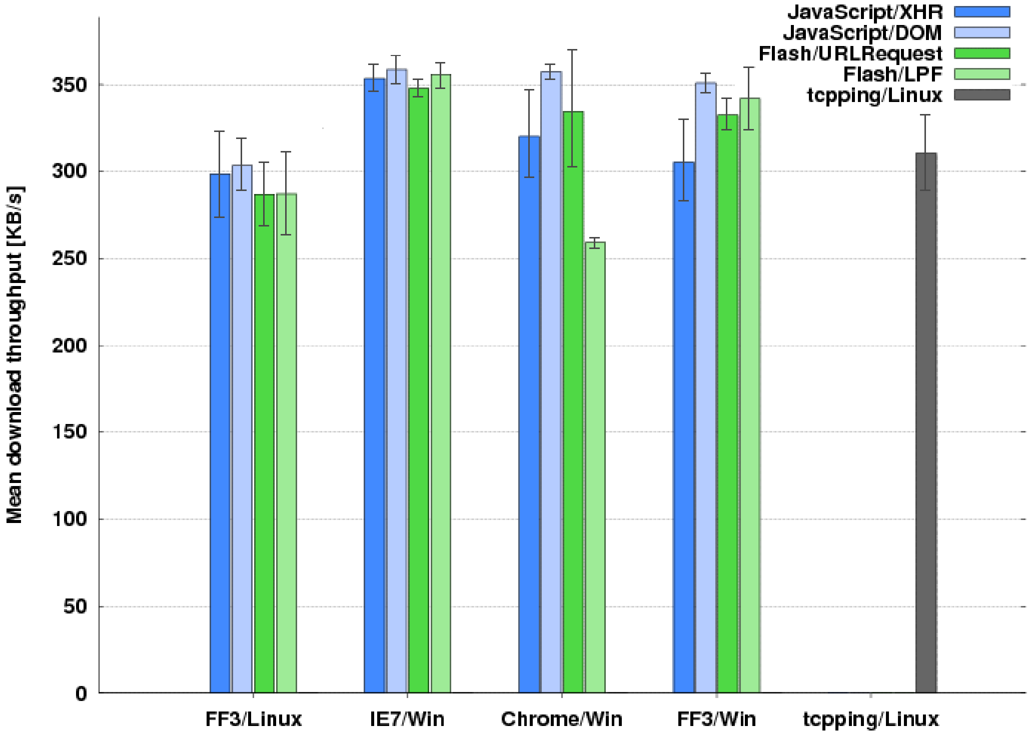


Figure 6.2: Means and Confidence Intervals for Controlled Download.

significant differences from the explicit technique results or the control measurements. These results suggest that it is possible to determine download throughput to third-party hosts as accurately as to the origin server.

6.1.2 Upload Throughput

Upload throughput measurements were done by using the two explicit techniques (XMLHTTPRequest and URLRequest) to send a request for a small (1-byte) file with 2MB of POST data in each of the four tested browsers. Validation was done by using the wget tool to request the same small resource and using the `--post-file` option to attach a 2MB file as a POST variable.

Individual upload results are shown in Figure 6.3. The results, while consistent for each technique show significant differences between measurement techniques.

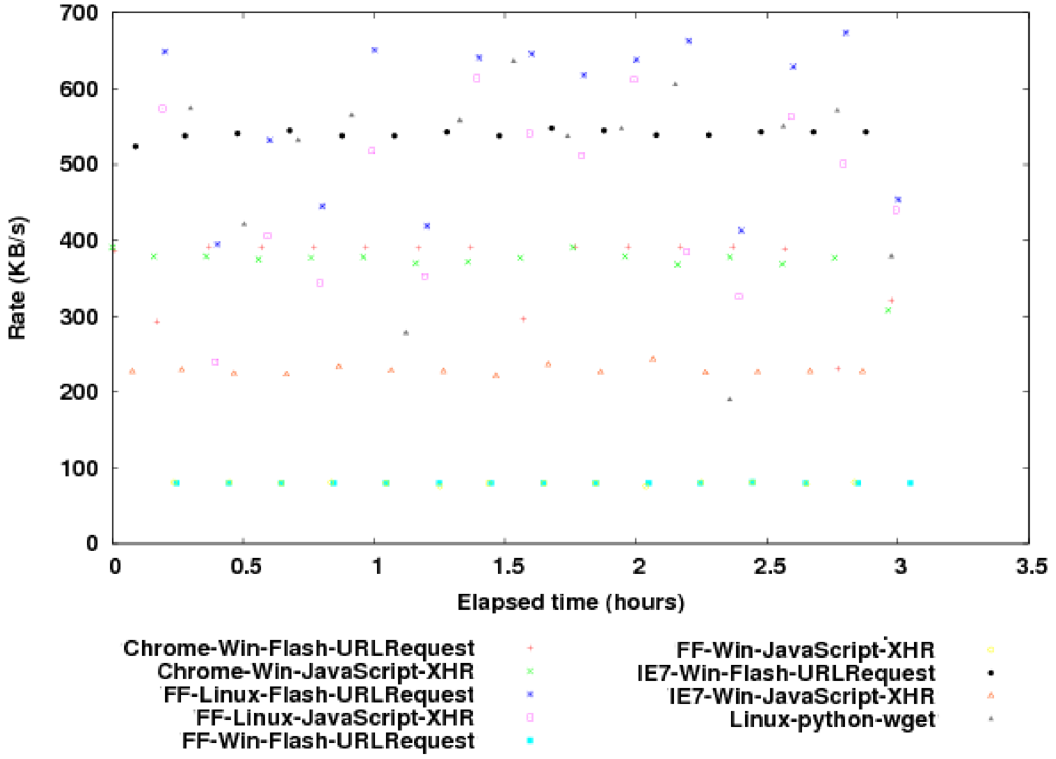


Figure 6.3: Controlled Upload Throughput Results.

It is easily seen that the upload throughput achieved by each method varies significantly depending on the browser used. The results for each method and their confidence intervals are presented in Figure 6.4.

A consequence of the observed behavior is that upload throughput achieved using the developed techniques can vary between different browsers on the same operating system. It is also possible that the XMLHTTPRequest and URLRequest upload techniques will show different throughput in the same browser environment—the URLRequest performance in Internet Explorer 7 is on average over twice that of XMLHTTPRequest.

While investigating this behavior we verified that the shown duration of the upload process is consistent with data obtained from the network layer, ruling out a reporting error in our techniques. We determined that all uploads from the Linux platform use a TCP window size of 128KB; uploads from Windows browsers use a

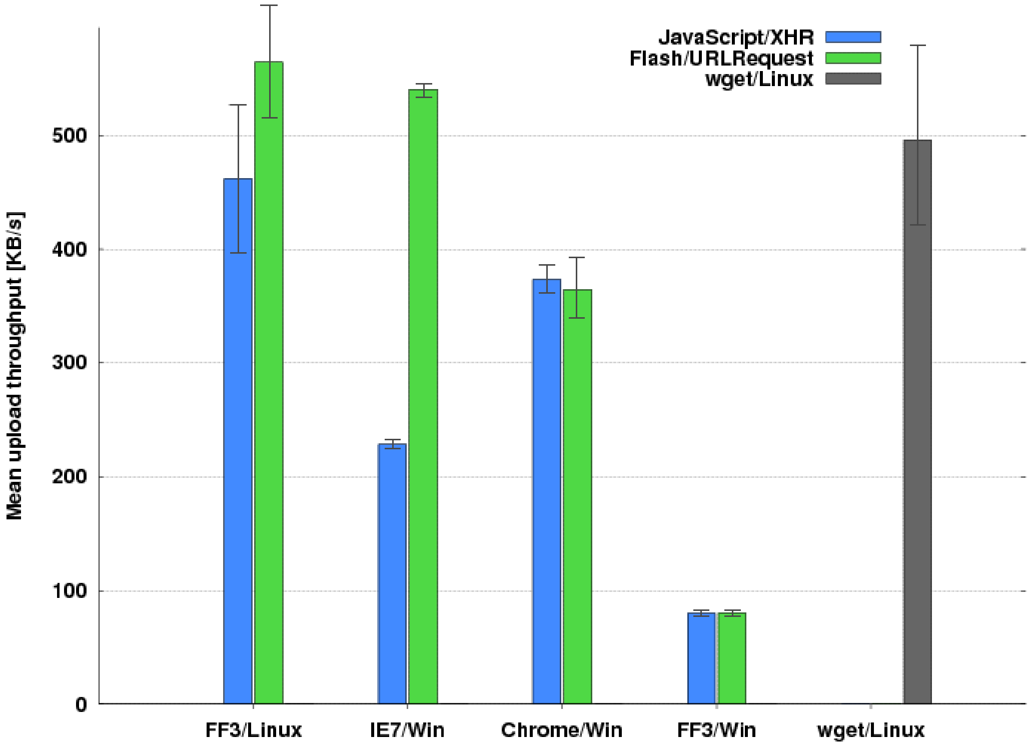


Figure 6.4: Means and Confidence Intervals for Controlled Upload.

window size of 32KB, which could explain why uploads from Linux are characterized by higher throughput, on average, than Windows techniques. However, based on the inspection of TCP packet parameters we were unable to determine the exact cause of the observed discrepancies between Windows browsers.

Thus, we can conclude that the developed techniques are not, by themselves, sufficient to reliably determine the maximum possible upload bandwidth of a connecting client. However, the measurements give us important information about the constraints *within* the JavaScript or Flash environment in each browser—the observed issues affecting upload behavior will impact all Web applications using XMLHTTPRequest or URLRequest in the given browser.

Possible ways of modifying the upload test in order to receive more accurate information about the network upload bandwidth available to the connecting client are discussed in Chapter 7.

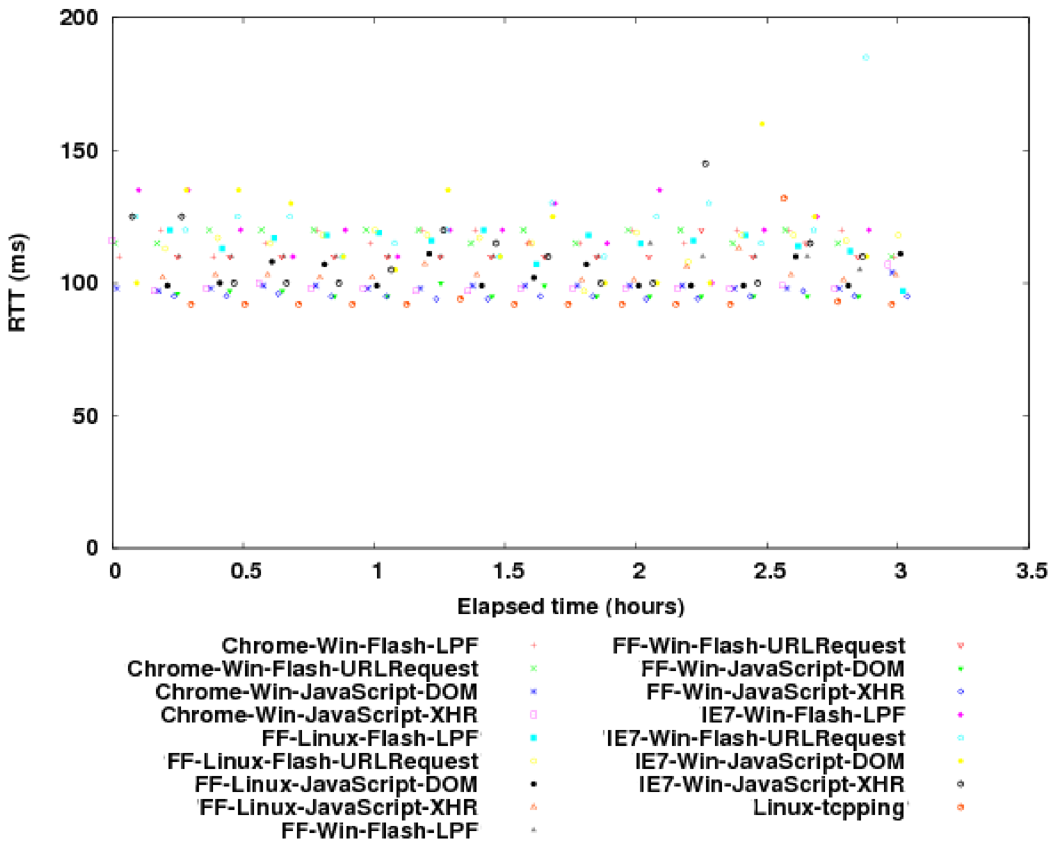


Figure 6.5: Controlled Round-trip Time Results.

6.1.3 Round-trip Time

Round-trip time was measured using the four developed mechanisms in each tested browser by requesting a 1-byte file from the webserver. The control test used the `tcpping` program to determine the time necessary to establish a TCP connection with the server. Individual results are provided in Figure 6.5.

As with download throughput results, the mean values obtained from most measurements techniques fall within a narrow range, The round-trip time measured using `tcpping` shows, predictably, the lowest round-trip time to the server, as it only measures the time to establish a connection and does not request any resources from the webserver. The mean results and confidence intervals are given in Figure

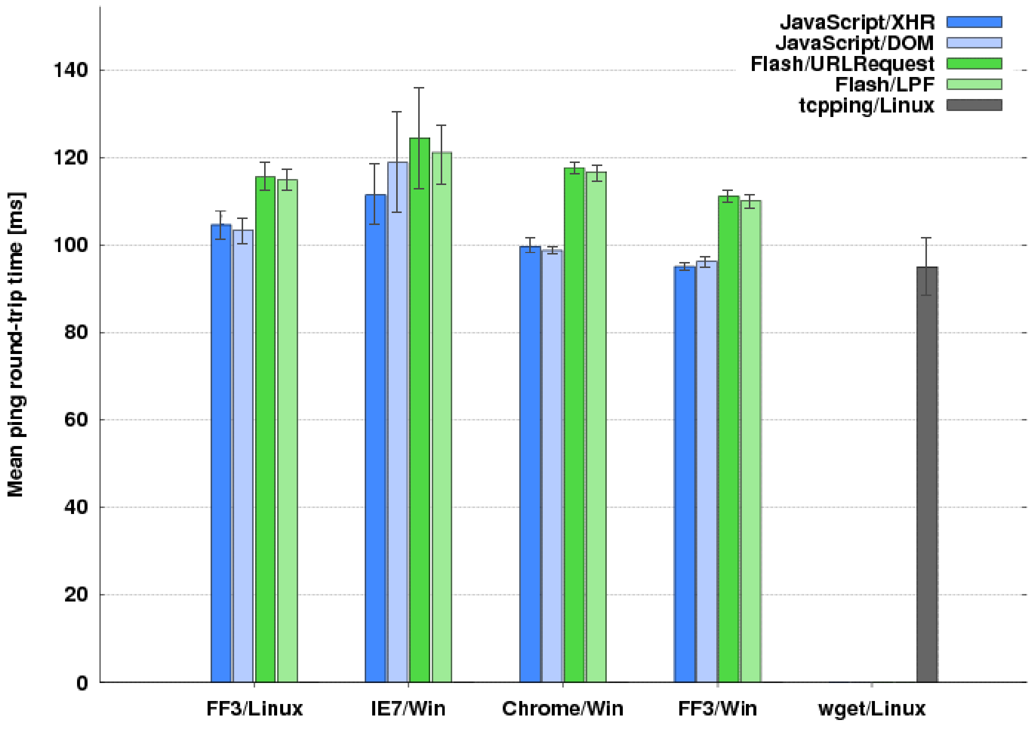


Figure 6.6: Means and Confidence Intervals for Controlled Round-trip Time.

6.6.

An important observation is that Flash techniques report, on average, higher round-trip time than JavaScript techniques. However, those results are still within a 20% margin from the minimum round-trip time reported by `tcpping`. That indicates that all techniques are generally useful for determining round-trip time.

6.2 Global Comparison of Measurement Methods

After establishing the correspondence between throughput and round-trip time metrics obtained using the developed browser techniques and results obtained via standard tools (`wget` and `tcpping`), we conducted a larger-scale study to determine relative differences between the measurement methods themselves.

A total of 70 users participated in the study: 35 from the WPI network (including academic and residential connections), and 35 connecting from another Internet Service Provider (ISP). For both groups of users we compared the throughput and round-trip time results between each available technique.

The tests measured download and upload throughput to the test server, and round-trip time to the server and selected third-party hosts using each of the four implemented techniques. Tests were repeated every 5 minutes until the user decided to close the browser window containing the test page. Incomplete test results (with less than one full execution of the test suite) were not considered. Thus, our results are based on data from about 20 users from the WPI network and 15 non-WPI users of whom 13 connected from a known major broadband ISP (Verizon, Charter, Comcast and Covad). A total of 3610 individual measurements were obtained from WPI-based users (1441 download, 731 upload, and 1438 round-trip time tests). Broadband users performed 3393 tests (1390 download, 660 upload, and 1343 round-trip time measurements).

6.2.1 Broadband Connections (DSL/Cable)

As might be expected, broadband connections were characterized by lower upload and download throughputs and higher round-trip times to the test server. Comparisons between obtained download throughputs between explicit and implicit techniques in JavaScript and Flash are shown in Figure 6.7 and Figure 6.8. In each case the slope of the best fit line indicates which technique reports higher throughput values. For download throughput the slopes are 1.04 +/- 0.10 (95% confidence interval) for JavaScript and 1.05 +/- 0.10 for Flash, indicating that there are no systematic differences between explicit and implicit techniques for broadband download throughput.

The comparison of all download values obtained with both JavaScript and Flash

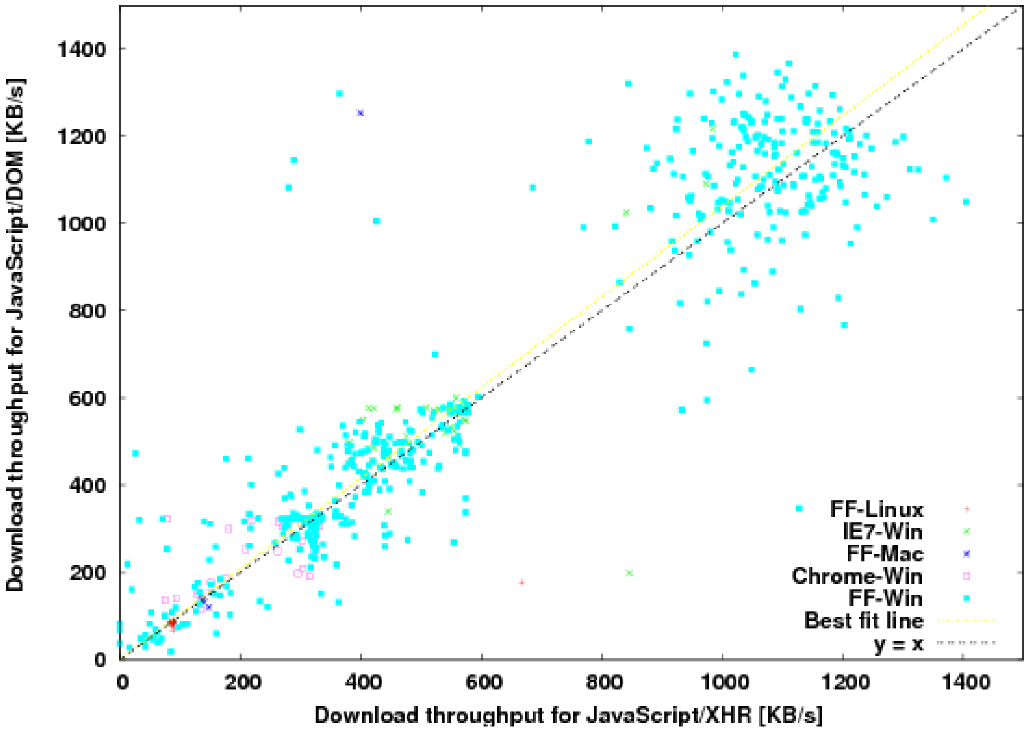


Figure 6.7: Comparison of Download Throughput Measured by XHR and DOM for Broadband Connections

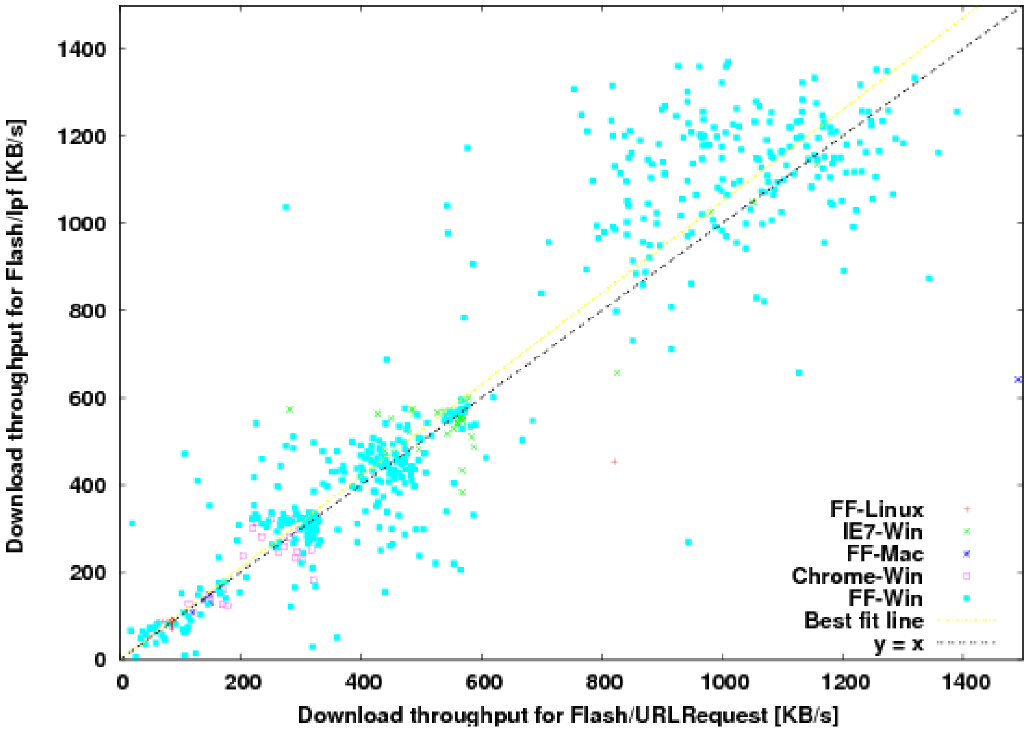


Figure 6.8: Comparison of Download Throughput Measured by URLRequest and LPF for Broadband Connections

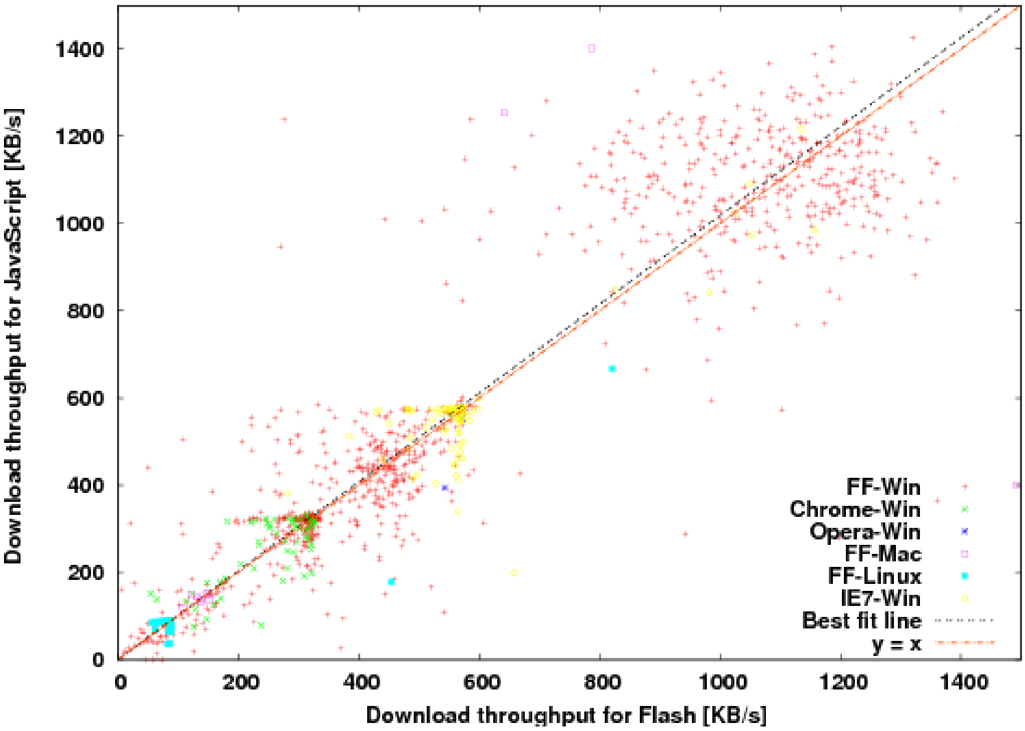


Figure 6.9: Comparison of Download Throughput Measured by JavaScript and Flash Methods for Broadband Connections

is shown in Figure 6.9; the best fit is 1.02 +/- 0.07. The results indicate that for broadband clients there are no significant in the average reported download throughput, both between explicit and implicit techniques and between JavaScript and Flash; consequently, all used techniques provide, on average, similar download throughput estimates.

Upload throughput results for XMLHTTPRequest and URLRequest are provided in Figure 6.10. Our results indicate that for broadband clients, URLRequest upload achieves slightly higher upload rates than XMLHTTPRequest (1.13 +/- 0.08).

For round-trip time, comparisons between explicit and implicit methods in Flash and JavaScript are shown in Figures 6.11 and 6.12. In both cases there is no significant difference explicit and implicit mechanisms (best fits are 0.94 +/- 0.16 in Figure 6.11 and 1.07 +/- 0.11 in Figure 6.12.

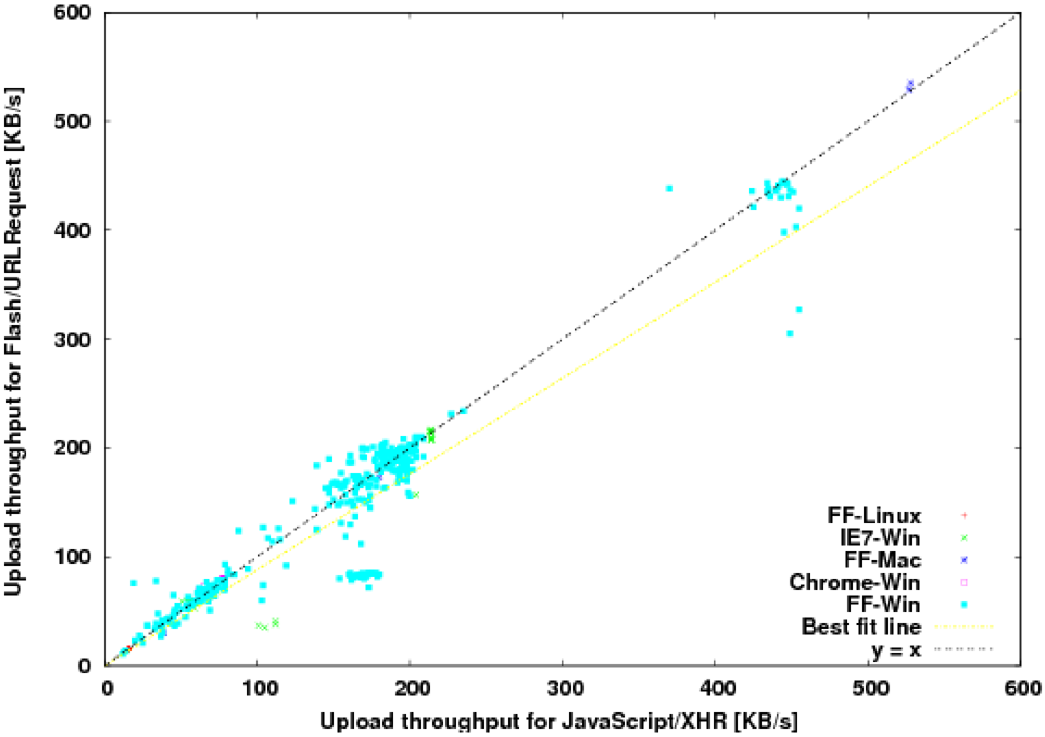


Figure 6.10: Comparison of Upload Throughput Measured by XHR and URLRequest for Broadband Connections

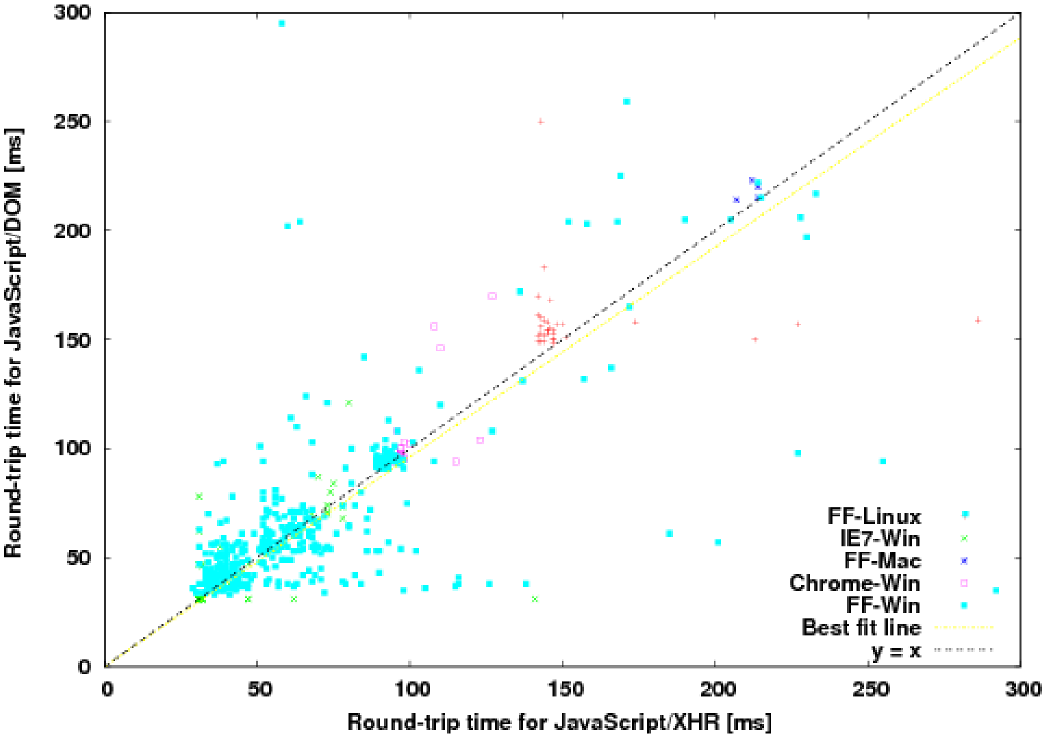


Figure 6.11: Comparison of Round-trip Time Measured by XHR and DOM for Broadband Connections

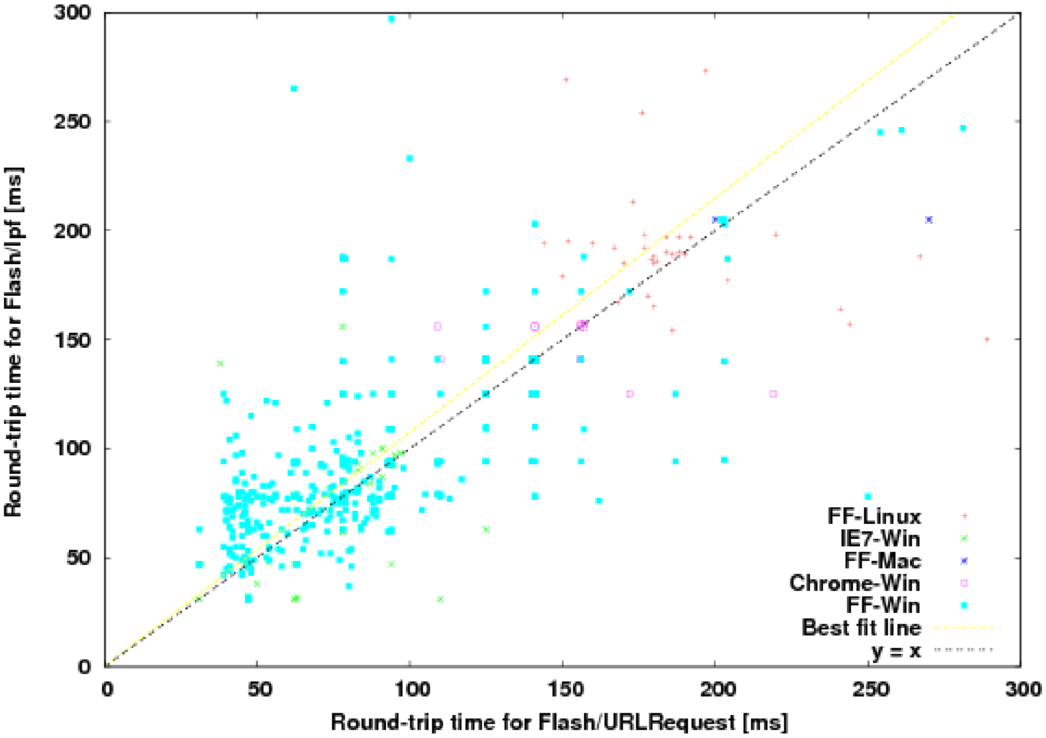


Figure 6.12: Comparison of Round-trip Time Measured by URLRequest and LPF for Broadband Connections

However, there exists a large difference between average round-trip time results as reported by JavaScript and Flash, as depicted in Figure 6.13. Round-trip results reported by JavaScript are on average almost 30% lower than by Flash techniques (0.74 +/- 0.06).

6.2.2 LAN Hosts (WPI Network)

By analyzing the results from WPI-based clients we were able to evaluate the behavior of browser communication mechanisms in a high-bandwidth, low-latency environment. Although such conditions rarely exist for broadband Internet clients, an inspection of the results can provide interesting information about the performance of browser communication schemes.

During the analysis of download throughput results for WPI clients, we observed a significant performance issue in XMLHTTPRequest in Google Chrome for higher

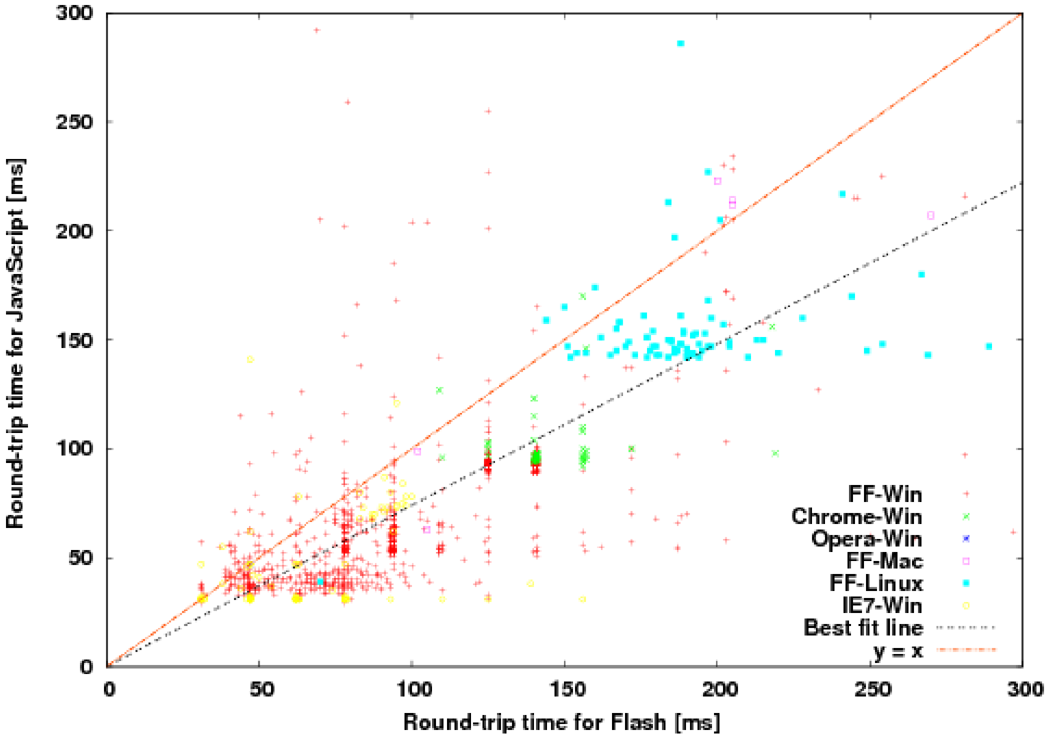


Figure 6.13: Comparison of Round-trip Time Throughput Measured by JavaScript and Flash Methods for Broadband Connections

bandwidths. Thus, for evaluating relative differences between techniques we utilize an additional line of best fit without taking into account results obtained from Chrome.

The performance issue is easily seen from Figure 6.14. While XHR downloads are faster than DOM downloads for higher throughputs in most browsers (1.67 +/- 0.05), results show that the opposite is true in Google Chrome, as XHR downloads are significantly slower than the other three techniques. This confirms our observations from Section 6.1.1 and might indicate a general performance issue with the XMLHTTPRequest implementation in the newly-developed JavaScript V8 engine used by Google Chrome. Both Flash methods achieve similar download throughput values (1.06 +/- 0.09), as shown in Figure 6.15.

For higher connection speeds, without taking into account results from Google Chrome, JavaScript and Flash give close estimates of download throughput, with

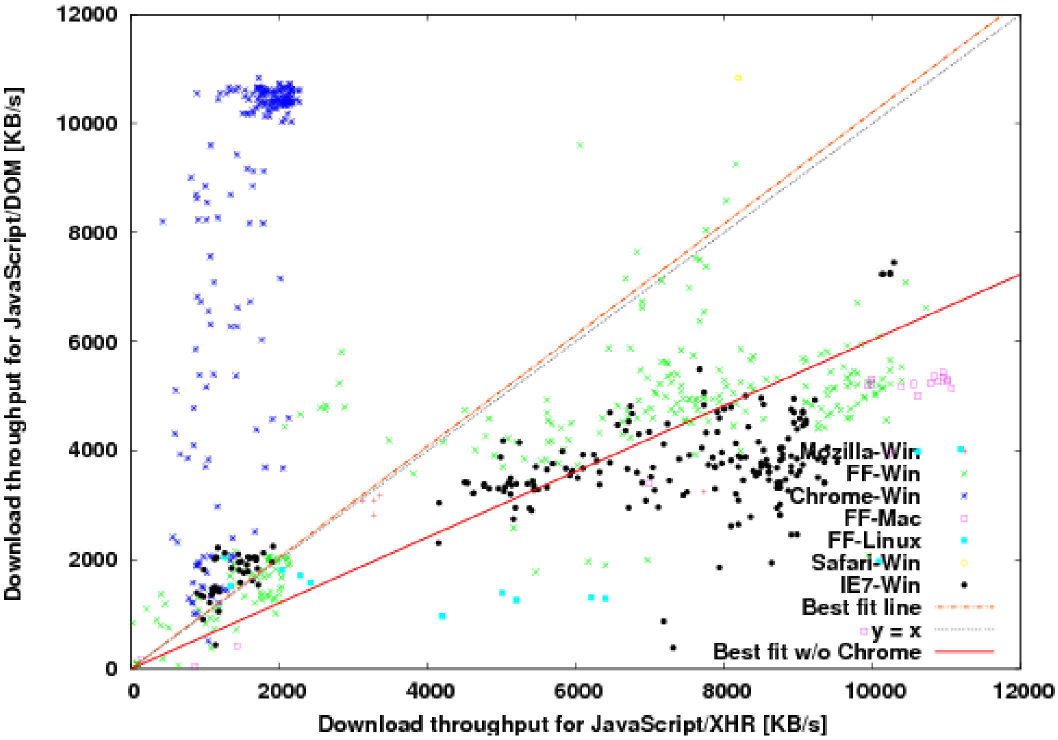


Figure 6.14: Comparison of Download Throughput Measured by XHR and DOM for Local Network Connections

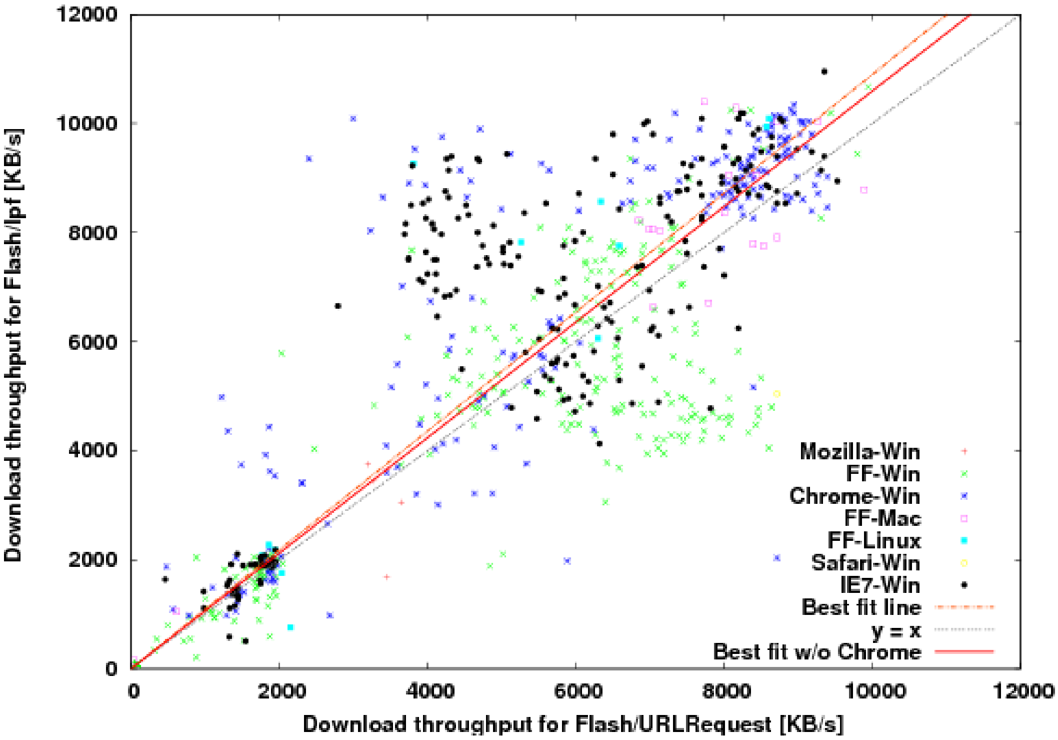


Figure 6.15: Comparison of Download Throughput Measured by URLRequest and LPF for Local Network Connections

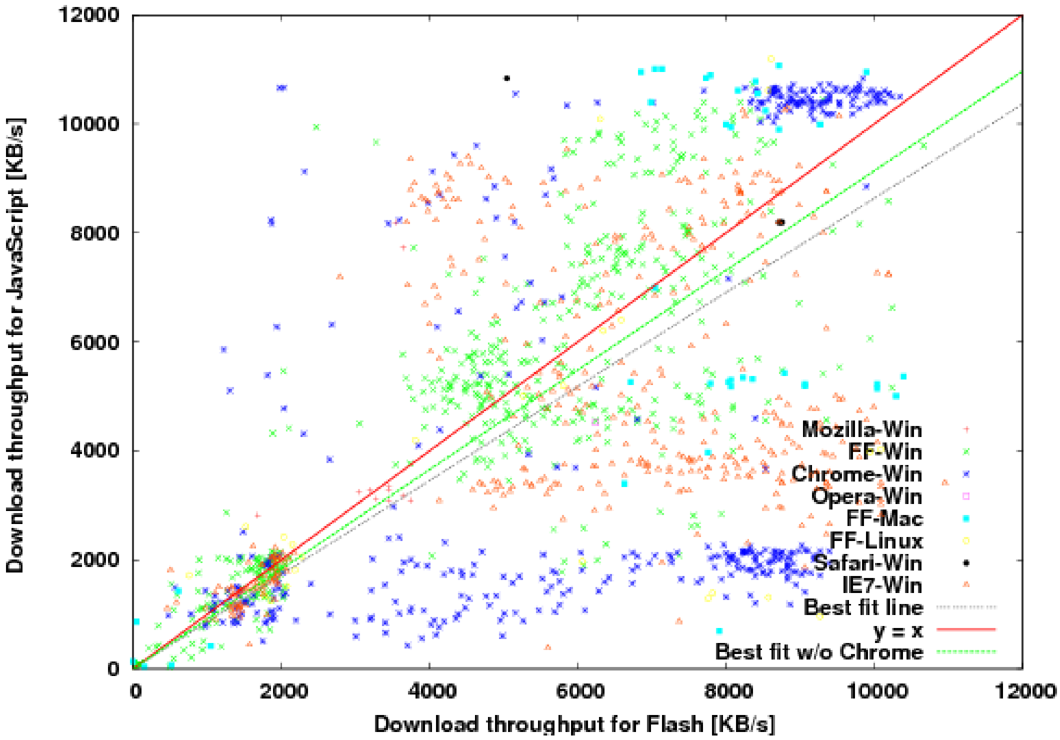


Figure 6.16: Comparison of Download Throughput Measured by JavaScript and Flash Methods for Local Network Connections

JavaScript performing slightly faster (1.10 +/- 0.06), as indicated in Figure 6.16.

For upload throughput, JavaScript and Flash performed equally well in all browsers (0.96 +/- 0.08), except for Google Chrome where URLRequest outperformed XHR by about 40%. Upload results are presented in Figure 6.17.

Measurements in the local environment indicate that explicit techniques report lower round-trip time than implicit methods (0.81 +/- 0.20 for XHR vs. DOM, and 0.90 +/- 0.10 for URLRequest vs. LPF). Thus, explicit techniques seem better suited for low-latency measurements. We speculate that the delay observed in results from implicit methods might be caused by the use of a different mechanism for scheduling the original GET request for the resource; it is also possible that more time is required for the browser to parse the server's response, which is not necessary in explicit methods.

Additionally, a large difference is observed in the aggregate results for JavaScript

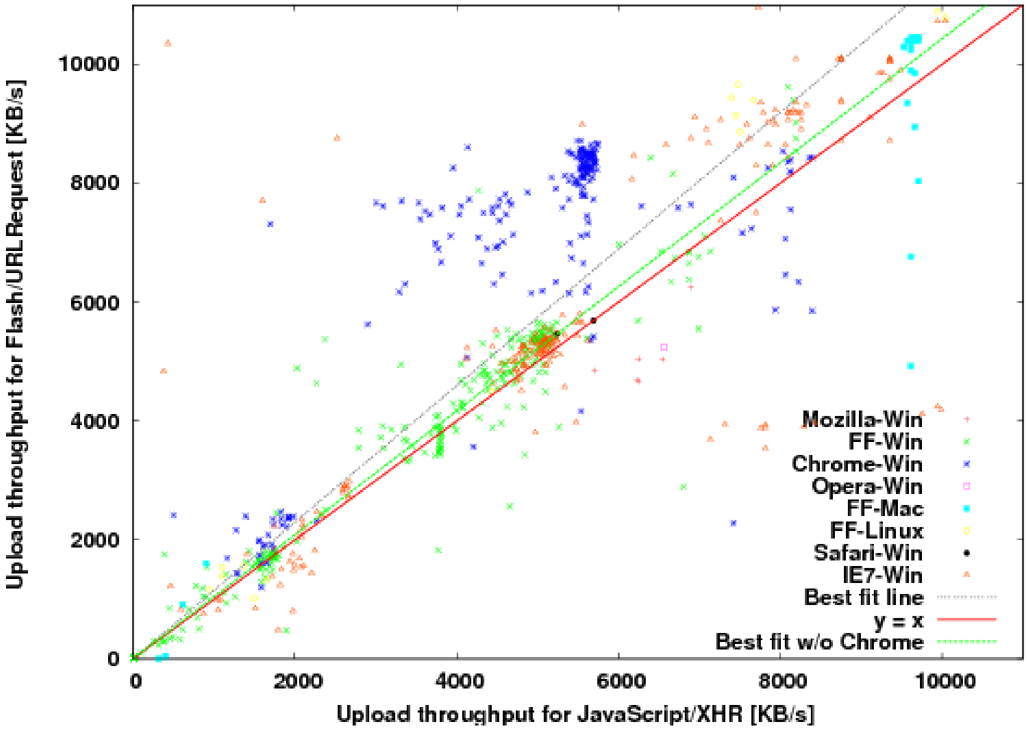


Figure 6.17: Comparison of Upload Throughput Measured by XHR and URLRequest for Local Network Connections

and Flash, as shown in Figure 6.18 (0.31 +/- 0.05). However, the majority of results for Flash fell under 60ms (and under 20ms for JavaScript), suggesting the existence of constant overhead when measuring round-trip time in Flash, which disproportionately impacts results for low-latency environments.

6.2.3 Round-trip Time to Third-party Hosts

As part of the general study we measured the round-trip time to three third party servers: `boston.com`, `unl.edu` and `youtube.com`. We selected these servers because of their geographic location (US East Coast, Midwest and West Coast), to provide different latency conditions for test clients, most of which connected from the East Coast. For this experiment we only gathered results using implicit techniques as the same-origin policy would prohibit making explicit requests to third-party hosts.

Round-trip results to `boston.com` from WPI hosts fall into the range of 10-50ms for DOM measurements and 25-80ms for LPF in over 95% of executed tests and are

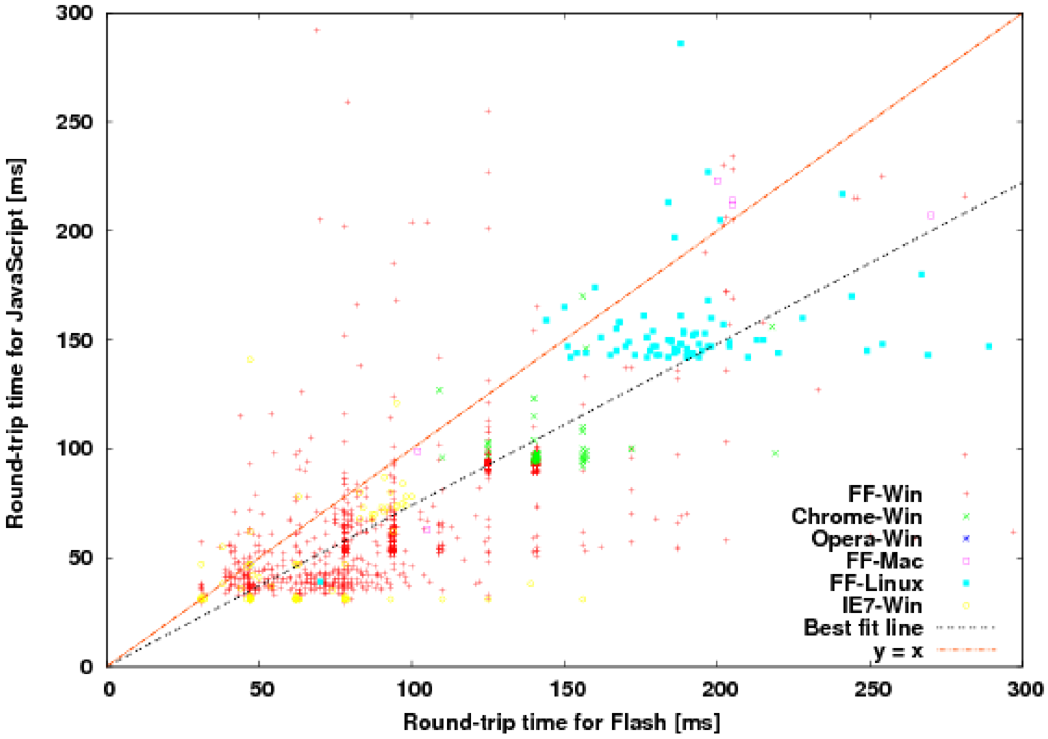


Figure 6.18: Comparison of Round-trip Time Throughput Measured by JavaScript and Flash Methods for Local Network Connections

shown in Figure 6.19.

Results for broadband users are shown in Figure 6.20 and are characterized by higher round-trip times than those from the WPI network, falling in the range of 25-100ms for both techniques.

In both cases, the DOM technique reports smaller round-trip times (0.56 +/- 0.08 for WPI hosts and 0.71 +/- 0.10 for broadband hosts). Since higher average round-trip times reduce the difference between results from both techniques, we suspect that a large part of the discrepancy is caused by a constant overhead inherent in our Flash measurement techniques.

Results for `unl.edu` follow a similar pattern as data for `boston.com`, albeit with a slightly higher minimum and average round-trip time, as shown in Figure 6.21 and Figure 6.22. Also in this case measurements using DOM object loading provide a smaller round-trip time estimate (0.70 +/- 0.05 for WPI hosts and 0.76 +/- 0.08

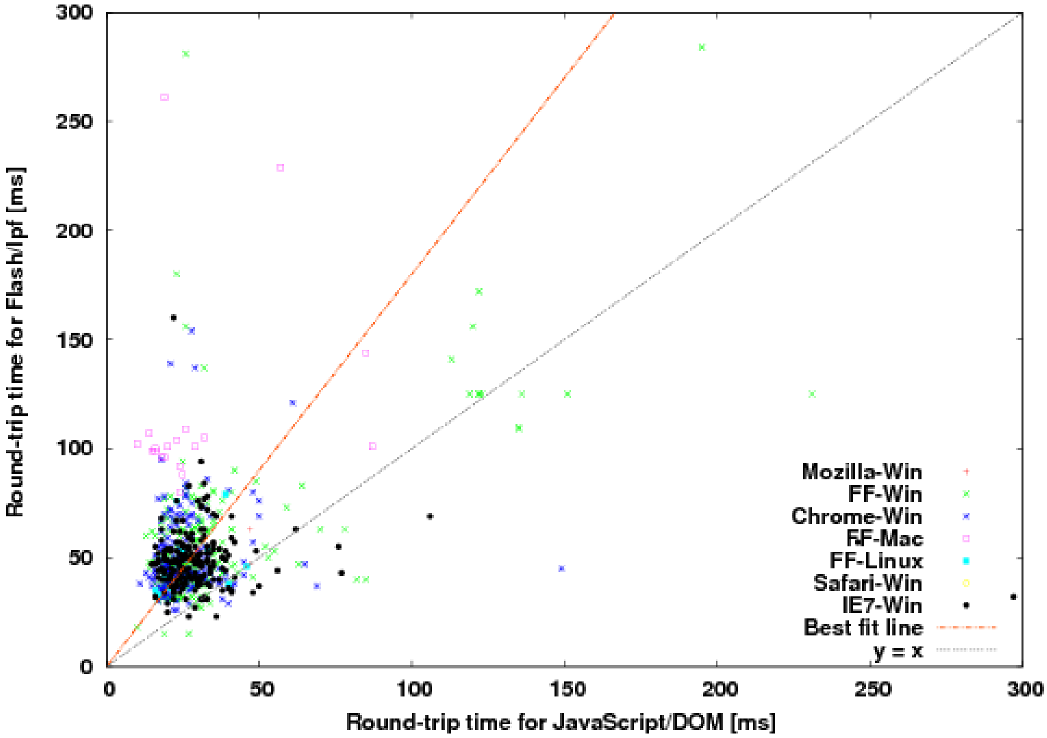


Figure 6.19: Round-trip Time to boston.com for WPI Hosts

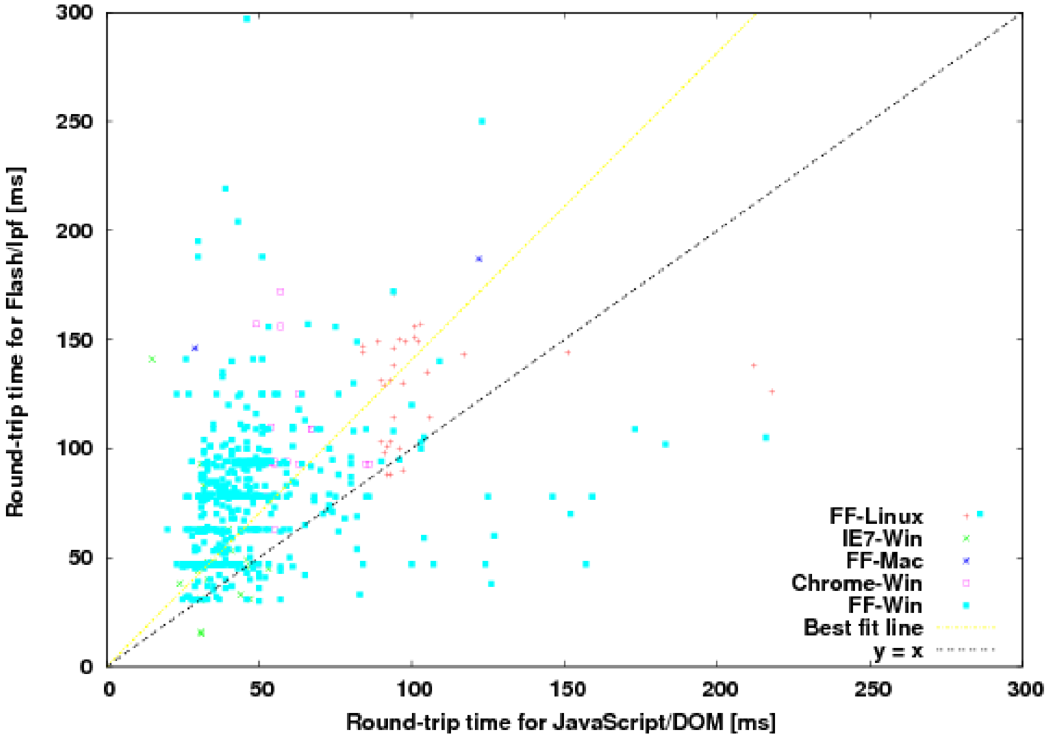


Figure 6.20: Round-trip Time to boston.com for Broadband Hosts

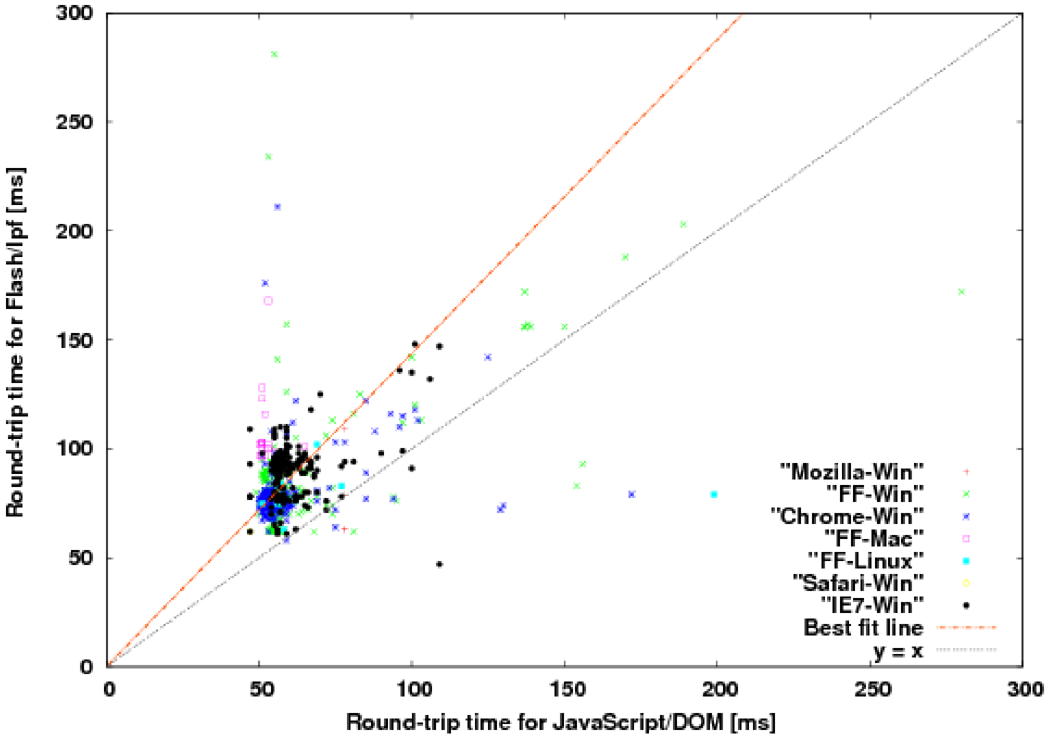


Figure 6.21: Round-trip Time to unl.edu for WPI Hosts

for broadband hosts).

Round-trip data to `youtube.com` shows interesting behavior, evidenced in Figure 6.24 and Figure 6.23. For broadband clients the result distribution is similar to the two other round-trip servers, with a majority of results under 200ms for both techniques. However, for hosts connecting from the WPI network, results are evenly distributed in the range 100-1000ms for both techniques.

The probable reason for this behavior is service-based packet filtering at the ISP level affecting the `youtube.com` server. The existence of a bandwidth-throttling policy for that domain was confirmed by the WPI Network Operations office.

The best fit lines are 0.83 +/- 0.08 for broadband clients and 0.95 +/- 0.09 for WPI hosts, supporting our observation about constant overhead affecting round-trip results in Flash.

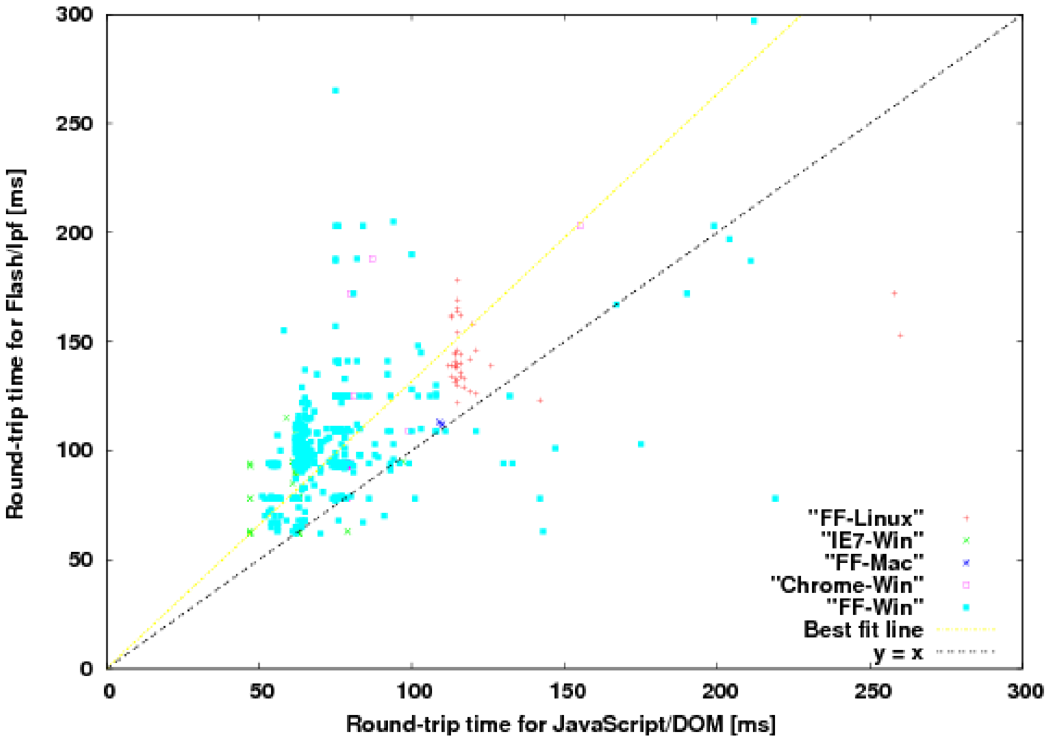


Figure 6.22: Round-trip Time to unl.edu for Broadband Hosts

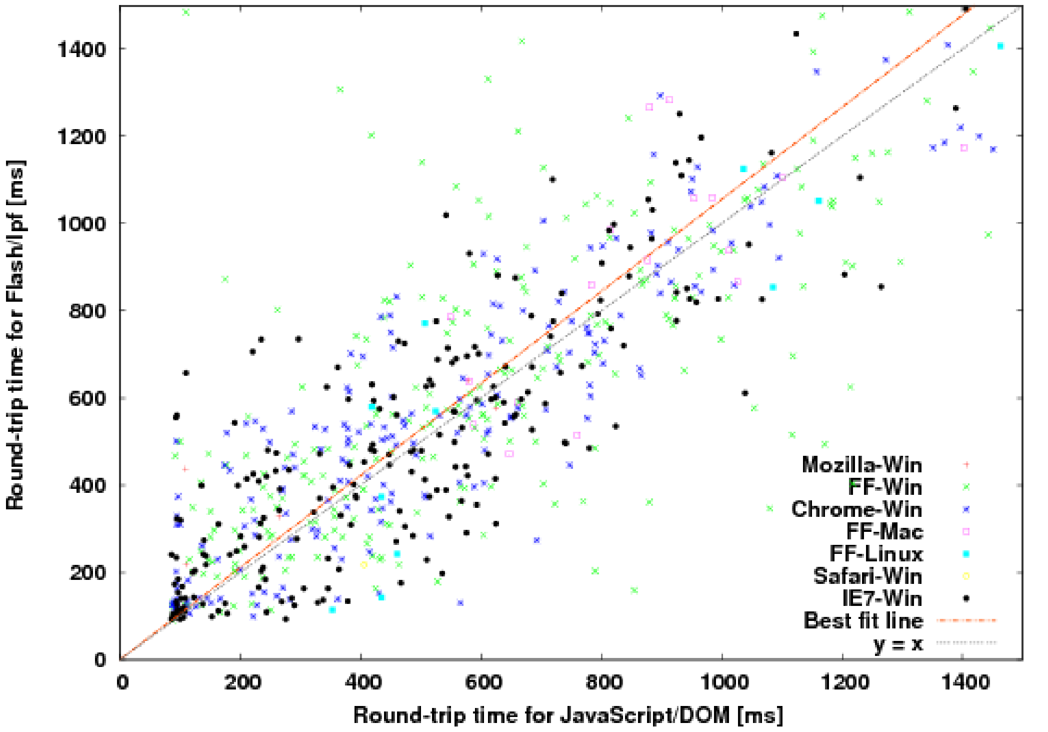


Figure 6.23: Round-trip Time to youtube.com for WPI Hosts

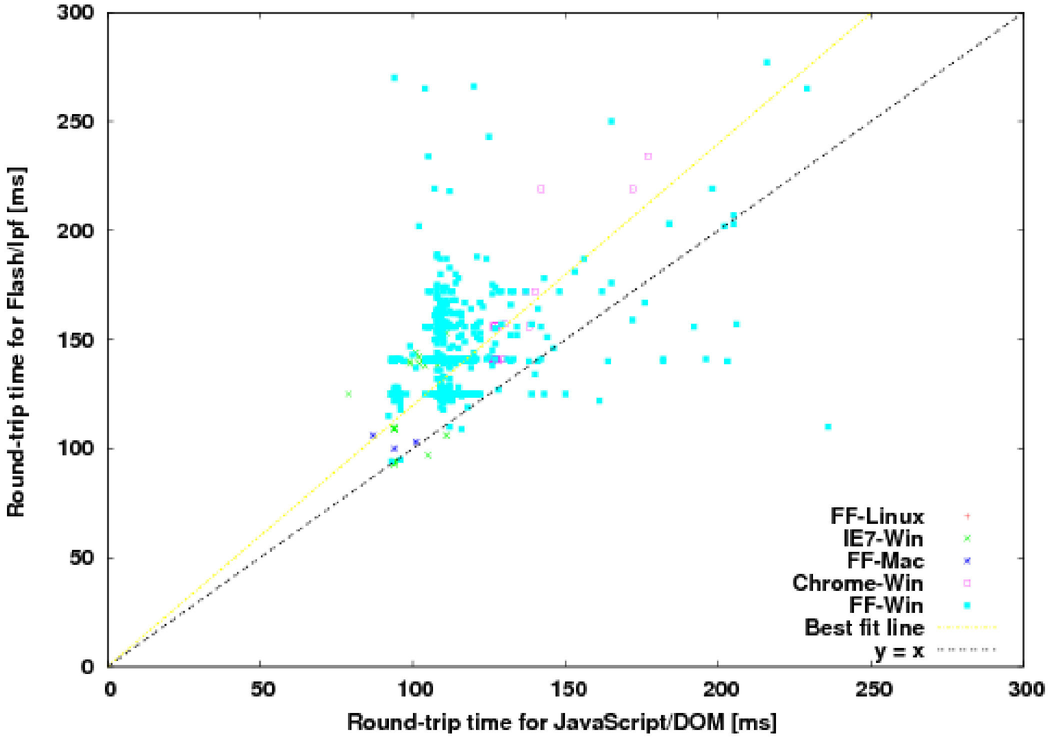


Figure 6.24: Round-trip Time to youtube.com for Broadband Hosts

6.3 Jitter

The jitter test allows the browser environment to simulate the behavior of a streaming protocol, and can be valuable in determining service quality. It also has the potential to detect transient network service degradation and interruption.

6.3.1 Correspondence Between Browser Events and Network Events

To establish how closely the “data available” software events correspond to network layer events, we compared event timing from the application layer and a packet trace obtained using the `wireshark` network sniffer.

We focused on evaluating the behavior of Flash `Event.PROGRESS` events, as JavaScript XHR events are not available in all major browsers, as mentioned in Chapter 4. In this test we downloaded a 1MB resource from the origin server using

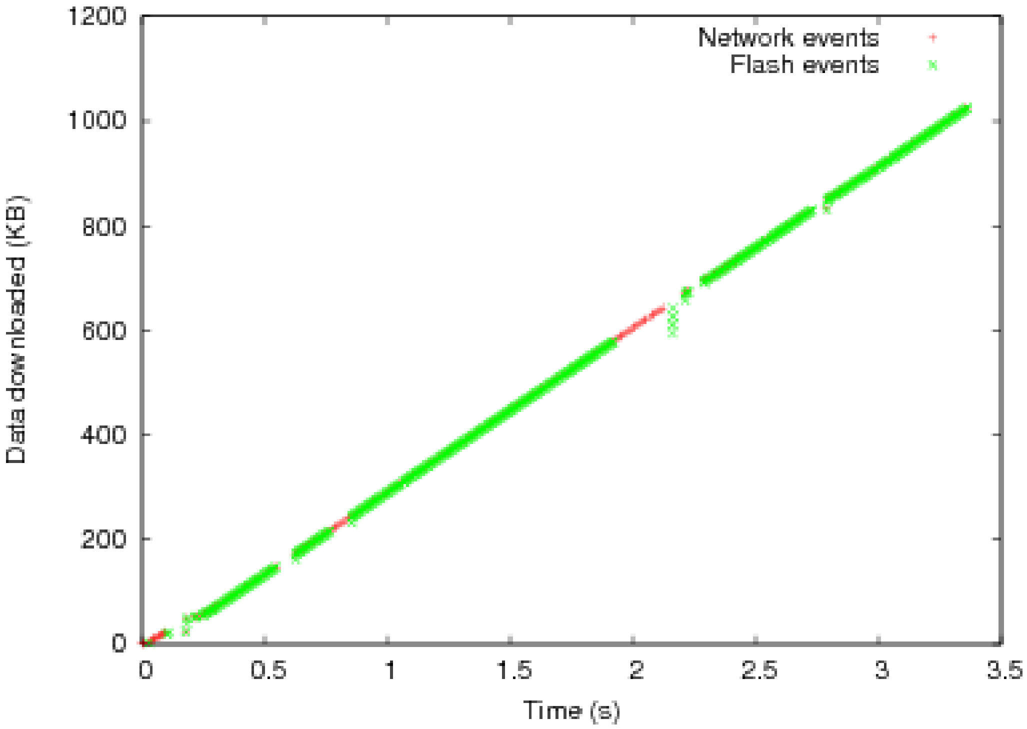


Figure 6.25: Timing of Network Layer Events and Flash Progress Events for a 1MB download.

Table 6.2: Differences in Timing of Network Layer and Flash Progress Events

0-5ms	6-10ms	Over 10ms
96.1%	2.1%	1.8%

the URLRequest technique in Firefox 3 under Ubuntu Linux using the methodology outlined in Section 3.3.2.

A time-based plot of the relationship between network events and Flash events is shown in Figure 6.25; a summary of the mean differences between the arrival of a packet and the corresponding software event is presented in Table 6.2.

We conclude that application-layer progress events in Flash correspond closely to the times when network data became available, making it possible to reliably determine the variability of packet reception times in our "streaming" jitter test.

6.3.2 Jitter Results

Jitter results were obtained by streaming data from the test server at WPI to a client connecting from a campus residential network on the US West Coast. A 10000-byte file was divided into 40 chunks of 250 bytes, spaced at 100ms. The transfer was performed using the Flash URLRequest technique analyzed in Section 6.3.1.

We tested the jitter client behavior under two conditions:

1. Without interference.

2. With a competing concurrent file download from the origin server.

The corresponding distributions of packet reception times are shown in Figure 6.26 and Figure 6.27. In the no-interference scenario, there is almost no variation in the reception times for all packets, indicating *good* network conditions. The results with interference show that about 30% of packets were delayed, with 15% delayed by over 50ms.

The jitter measurement technique shows considerable potential for evaluating network conditions outside of the often-measured throughput and round-trip time metrics; the technique also provides an opportunity to detect transient network interruptions and give insight into the performance of streaming data protocols within the Web browser environment.

6.4 Discussion

In Section 6.1 we determined that all developed measurements report download throughput and round-trip time within 20% of the control value.

We also observed that the developed upload techniques vary significantly between browsers on the same client host, indicating that they are not sufficient to determine

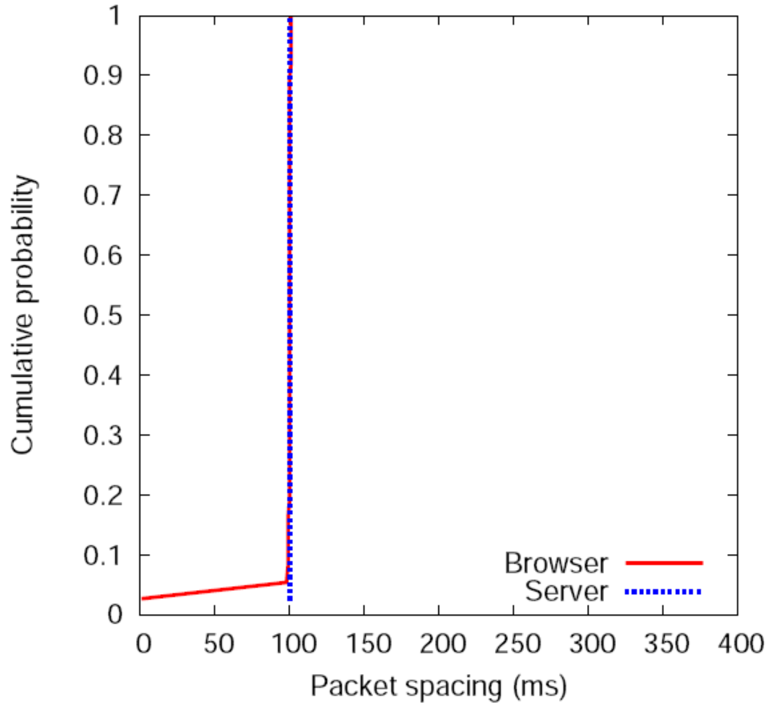


Figure 6.26: Packet Streaming Without Interference.

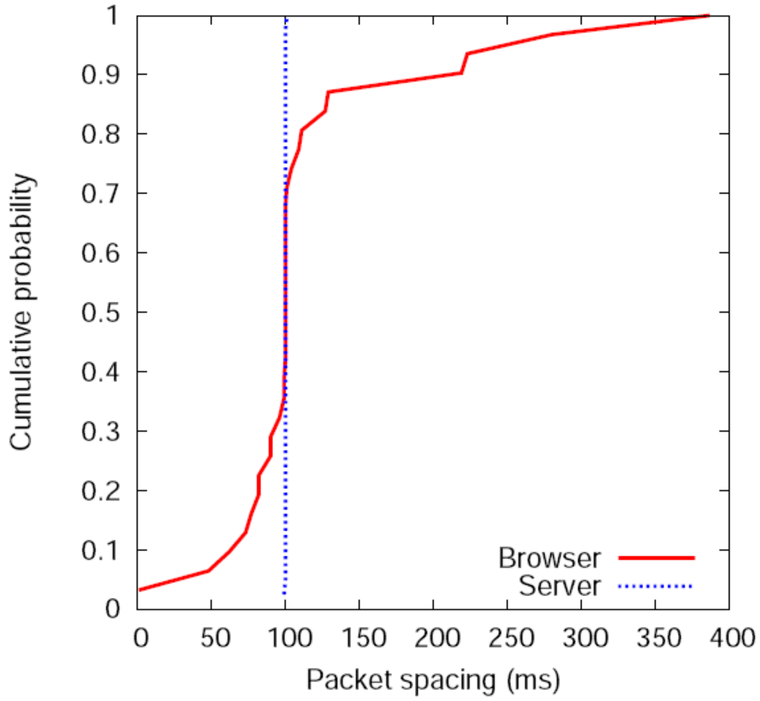


Figure 6.27: Packet Streaming With Interference.

a connecting client's network upload bandwidth in general. We present possible corrections to the upload measurement techniques in Chapter 7.

Through analysis shown in Section 6.2 we discovered that for broadband users, all developed techniques perform equally well in the case of upload and download using the same browser. We also determined that average reported round-trip time in Flash is higher than for JavaScript.

For download and round-trip time measurements we determined that implicit techniques perform on par with explicit techniques. This suggests that the JavaScript DOM and Flash LPF techniques can be used to reliably estimate client download throughput and round-trip times to arbitrary third-party hosts.

In Section 6.3 we showed that there is a close correspondence between the time when data reached the network layer and application layer progress events in Flash, with over 95% of software events occurring within 5ms of the respective network event.

We also demonstrated the utility of the jitter test to measure packet reception variability and give additional information about the characterstics of the network connection between the client and the server.

Chapter 7

Conclusions

In this chapter we present the summary of our work and suggest further improvements to the measurement techniques developed for this thesis. We discuss ongoing work as well as other potential improvements that can be made in work based on the techniques and methodology contributed in this thesis.

7.1 Future Work

During the implementation of the measurement techniques presented in this work we made several corrections and added assumptions affecting our work, as outlined in Chapter 3 and Chapter 4. In this section we present additional improvements which can be made to our approach to improve measurement accuracy as well as obtain other kinds of useful network performance data with little or no modification to the existing techniques and platform.

1. Modifications to upload throughput test.

   In Chapter 6 we determined that in some browsers the upload throughput achieved by browser-based methods is substantially lower than what can be obtained using standard techniques. In our analysis we noticed the effect of the TCP window size parameter on the reported throughput, but were unable to determine the exact cause for the slowness of the upload. A potential improvement to the measurement methodology is to initiate several concurrent uploads and report the aggregate throughput. Thus, even if each stream is artificially limited due to browser limitations, the aggregate throughput will more closely approximate the maximum available value.

2. Large-scale jitter study.

   We propose to conduct a large-scale study of packet reception variance based on the jitter test, conducted similarly to the throughput and round-trip time tests in Section 6.2. The sending parameters on the server can also be modified to simulate the behavior of real streaming protocols, such as those used in VoIP applications to provide insight into the performance of such applications for each client.

   We also propose a controlled jitter measurement experiment where artificial network service interruptions are introduced, and an analysis of their effect on the variability of packet reception. Such an experiment could help determine the exact relationship between packet reception delays and packet loss and the jitter results measured by the client.

3. Local network discovery.

   The implicit communication techniques introduced in Section 3.2.3 can be used to issue HTTP GET requests to arbitrary hosts and network ports. Thus, it is possible to perform timing analysis to determine if a particular host is available and determine its open ports by using the information about the time of the error event associated with the request—a request to a nonexistent host will time out after several seconds, whereas active hosts will immediately close any attempted connection to a closed port. This information can also be used to help determine local IP addresses of hosts behind a router performing Network Address Translation (NAT).

4. DNS performance evaluation.

   Before a request is made using one of the implicit communication methods, the browser must perform a DNS lookup for the target domain if information about that domain is not available in the browser DNS cache. Thus, for a request to a nonexistent domain, the timing of the error event reveals information about the time necessary to receive a response from the DNS server. As DNS latency has been shown to affect perceived browser network performance, evaluating DNS performance can provide additional information about the client's network parameters. More work is needed to determine the feasibility of this technique.

5. Creation and distribution of a browser-based network measurement package.

   For the research presented in this thesis we developed a set of client-side scripts to measure various network parameters and report their results to the server. Work is ongoing to distribute a modified version of the test scripts to allow for their use in other projects.

7.2 Summary

This thesis presents our work on evaluating the network performance by utilizing network communication techniques available within the Web browser environment. We presented the rationale for selecting the browser scripting environments to maximize the utility of our results; we categorized available communication methods as explicit and implicit and discussed our implementations of two explicit and two implicit techniques using JavaScript and Flash. We determined which types of information can be obtained using each technique and described measurement procedures for estimating relevant network-layer parameters. We also provided an overview of our client-side and server-side architecture. Finally, we presented results of a controlled measurement of throughput and round-trip time, as well as a larger-scale study using the developed measurement techniques to determine the network performance of broadband and local clients.

Our work made several key contributions to the area of Web-based network measurement:

- We designed and implemented four techniques to estimate throughput and round-trip time in JavaScript and Flash. We compared their results with values reported by control measurements obtained with standard tools.
- We introduced the concept of implicit measurement techniques to allow measurements of download throughput and round-trip time to arbitrary third-party servers, and validated their results.
- We compared the relative accuracy of all measurement techniques in several popular browsers for broadband and local network clients.
- We developed a "streaming" HTTP server and clients in JavaScript and Flash to measure the variability of packet reception times in a simulated streaming protocol.
- We outlined the design of a testing platform to distribute browser-based network measurement tests and automatically gather as well as present individual

and aggregate results. The platform allows for easy addition of new kinds of measurements.

Our results indicate that all developed measurement techniques can provide useful information about the download throughput and round-trip time between the Web server and the connecting client. In our control experiment, all techniques of estimating download throughput provide results within 20% of the control value in several popular browsers; additionally, all round-trip time techniques reported mean results within 25ms of the control value.

We have also established that upload throughput cannot be, in general, reliably estimated using the developed techniques. While several browsers report upload throughput consistent with the control measurements, others report lower upload rates and do not utilize the entire available upload bandwidth.

The results of our larger-scale study confirm that there are no significant differences between our measurement techniques for the download and upload throughput for broadband clients. Round-trip time results indicate that Flash-based techniques report values higher by about 20-30ms than their JavaScript counterparts.

We demonstrate that implicit measurement methods report download throughput and round-trip times as reliably as explicit techniques in both JavaScript and Flash. Because of the lack of same-origin policy restrictions for implicit methods, we conclude that it is possible to reliably determine download throughput and round-trip time to an arbitrary Internet host using these techniques.

The timing analysis of network layer events and "download progress" events in Flash shows that there is a high correspondence between the arrival of a packet and the related software event, with over 95% of events occurring within 5ms of the reception of the packet. Thus, we determine that the jitter test has the potential to accurately report on network-layer packet delays in a streaming download.

We conclude that browser-based performance measurements using the presented techniques can be a valuable tool for assessing network parameters of a connecting client.

Bibliography

[Ado09a] Adobe. ActionScript 3.0 Language and Components Reference: URLLoader. *http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/net/URLLoader.htm* 1 2009.

[Ado09b] Adobe. External data not accessible outside a Flash movie's domain. *http://kb.adobe.com/selfservice/viewContent.do?externalId=tn_14213*, page 1, 01 2009.

[arc] Archipelago measurement infrastructure. `http://www.caida.org/projects/ark/`.

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (url), 1994.

[CGC+05] Martin Casado, Tal Garfinkel, Weidong Cui, Vern Paxson, and Stefan Savage. Opportunistic measurement: Extracting insight from spurious traffic. In *Proceedings of the Fourth Workshop on Hot Topics in Networks*, College Park, MD USA, November 2005.

[CKW07] Mark Claypool, Robert Kinicki, and Craig Wills. User-centered network measurement. Technical Report WPI-CS-TR-07-08, Computer Science Department, Worcester Polytechnic Institute, August 2007.

[CM02] Liang Cheng and Ivan Marsic. Java-based tools for accurate bandwidth measurement of digital subscriber line networks. *Integr. Comput.-Aided Eng.*, 9(4):333–344, 2002.

[DHGS07] Marcel Dischinger, Andreas Haeberlen, Krishna P. Gummadi, and Stefan Saroiu. Characterizing Residential Broadband Networks. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, New York, NY, USA, 2007. ACM Press.

[dim] The DIMES project. `http://www.netdimes.org/`.

[dsl] Broadband reports.com speed test. `http://www.dslreports.com/stest`.

[Fli09] Flickr. Flickr Web Application. *http://www.flickr.com*, 1 2009.

[Goo08] Google. Browser Security Handbook. *http://code.google.com/p/browsersec/wiki/Part1*, page 1, 12 2008.

[Goo09] Google. Google Maps. *http://maps.google.com*, 1 2009.

[Inc08] Adobe Systems Incorporated. Flash Player Version Penetration. *http://www.adobe.com/products/player_census/flashplayer/version_penetration.html*, 9 2008.

[IPKA07] Tomas Isdal, Micahel Piatek, Arvind Krishnamurthy, and Thomas Anderson. Leveraging BitTorrent for end host measurements. In *Proceedings of the Passive and Active Measurement Workshop (PAM)*, Louvain-la-neuve, Belgium, April 2007.

[JBB+07] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from dns rebinding attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 421–431, New York, NY, USA, 2007. ACM.

[JBBM06] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting browser state from web privacy attacks. In *WWW*, pages 737–744, 2006.

[JW07] Martin Johns and Justus Winter. Protecting the Intranet Against JavaScript Malware and Related Attacks. In *Lecture Notes in Computer Science*, volume 4579, pages 40–59. Springer Berlin / Heidelberg, 2007.

[Mic09] Microsoft. About Cross-Frame Scripting and Security. *http://msdn.microsoft.com/en-us/library/ms533028(VS.85).aspx*, page 1, 01 2009.

[Moz01] Mozilla.org. Mozilla: The Same Origin Policy. *http://www.mozilla.org/projects/security/components/same-origin.html*, 8 2001.

[Ook09] Ookla. Speedtest.net - The Global Broadband Speed Test. *http://speedtest.net*, 1 2009.

[PBFM06] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building PlanetLab. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA USA, November 2006.

[Rit07] Alan Ritacco. Java home based scanning and reporting tool, December 2007. Computer Science Department, Worcester Polytechnic Institute. In progress.

[Sch08] W3 Schools. W3 Schools: Browser Statistics. *http://www.w3schools.com/browsers/browsers_stats.asp*, 1 2008.

[SRR06] Charles Robert Simpson, Jr., Dheeraj Reddy, and George F. Riley. Empirical models of TCP and UDP end-user network traffic from NETI@home data analysis. In *20th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2006)*, pages 166–174, May 2006.

[toa] Internet performance and speed test. `http://performance.toast.net/`.

[W3C08] W3C. The XMLHttpRequest Object. *http://www.w3.org/TR/XMLHttpRequest/*, 4 2008.

[WTR07] Zhihua Wen, Sipat Triukose, and Michael Rabinovich. Facilitating Focused Internet Measurements. In *SIGMETRICS: Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, June 2007. ACM Press.