AjaxTracker: Active Measurement System for High-Fidelity Characterization of AJAX Applications

Myungjin Lee, Ramana Rao Kompella, Sumeet Singh[†]
Purdue University, [†]*Cisco Systems*

Abstract

Cloud-based Web applications powered by new technologies such as Asynchronous Javascript and XML (Ajax) place a significant burden on network operators and enterprises to effectively manage traffic. Despite increase of their popularity, we have little understanding of characteristics of these cloud applications. Part of the problem is that there exists no systematic way to generate their workloads, observe their network behavior today and keep track of the changing trends of these applications. This paper focuses on addressing these issues by developing a tool, called AJAXTRACKER, that automatically mimics a human interaction with a cloud application and collects associated network traces. These traces can further be post-processed to understand various characteristics of these applications and those characteristics can be fed into a classifier to identify new traffic for a particular application in a passive trace. The tool also can be used by service providers to automatically generate relevant workloads to monitor and test specific applications.

1 Introduction

The promise of cloud computing is fueling the migration of several traditional enterprise desktop applications such as email and office applications (*e.g.*, spreadsheets, presentations, and word processors) to the cloud. The key technology that is powering this transition of the browser into a full-fledged cloud computing platform is Asynchronous Javascript and XML (Ajax) [11]. Ajax allows application developers to provide users with very similar look-and-feel as their desktop counterparts, making the transition to the cloud significantly easier.

The modern cloud applications based on Ajax behave differently from the traditional Web applications that involve users clicking on a particular URL to pull objects from the Web server. Ajax-based cloud applications, however, may involve each mouse movement leading to a transaction between the client and the server. Further, these transactions may potentially involve an exchange of one or many messages *asynchronously* and sometimes, even *autonomously* without user involvement (*e.g.*, auto-save feature in email).

While there are a large number of studies that characterize (*e.g.*, [7]) and model (*e.g.*, [6]) classical Web traffic, we have very limited understanding of the network-level behavior of these Ajax-based applications. A comprehensive study of these applications is critical due to two reasons. First, enterprises are increasingly relying on cloud applications with Ajax as a core technology. As these services can potentially affect the employee productivity, it becomes crucial for operators (both enterprise as well as ISP) to constantly monitor the performance of these applications. Second, network operators need to project how application popularity changes may potentially affect network traffic growth, perform 'what-if' analyses, monitor for new threats and security vulnerabilities that may affect their network.

A standard approach (*e.g.*, [21]) for characterizing these applications is to collect a trace in the middle of the network and observe the network characteristics of these applications in the wild. Due to the reliance on passive network traces, however, this approach has two main limitations. The first limitation is that there is no easy way to isolate the network-traffic produced by individual operations (such as Zoom-in operation in Maps application, or drag-and-drop on Mail application), which may be important to understand which actions are most expensive or how network traffic may change if relative usage of different operations change in future. Second, there is no easy way to understand how network conditions affect the characteristics of these applications. This is since, at the middle of the network, the router only observes aggregate traffic comprising of clients from heterogeneous network environments. For some uses, aggregate view may actually be sufficient, but for certain management tasks such as, say, conducting what-if analyses, this aggregate view is *not* sufficient.

To address these challenges, in this paper, we propose an active measurement system for high-fidelity characterization of modern cloud applications, particularly those that are based on Ajax. Our approach comprises of two key ideas: First, we observe that running an application on an end-host with no other application can allow capturing *all* the packets associated with that application session with zero false positives or false negatives. Second, by controlling the network conditions and what operations we inject in isolation, we can get a deeper understanding of these applications in addition to predicting their impact on the network.

Our system called AJAXTRACKER, works by modeling high-level interaction operations (*e.g.*, drag-and-drop) on a particular Ajax-based cloud application and by injecting these operations through a browser to generate (and subsequently capture) relevant network activity between the client and the server. In addition, it incorporates mechanisms to generate representative client appli-

cation sessions by specifying either an explicit or model-driven sequence of atomic operations. The model that governs the sequence of operations may, for instance, control the distribution of time between two atomic operations. It also utilizes a traffic shaper to control network latencies and bandwidth to study the effects of end-host network conditions on the application performance. We have designed and implemented a prototype of this tool that is available for download[1].

Thus, our paper makes the following contributions: 1) Our first contribution in this paper is the design of AJAXTRACKER that provides a mechanism to automatically interact with Ajax-powered cloud services. We discuss the details of the tool in Section 3. 2) We present a characterization study of popular Ajax-based applications under different bandwidth conditions and different round-trip times. Section 4.2 discusses these results in more detail. 3) Our final contribution is a characterization of network activity generated by popular applications on a per-operation basis. To the best of our knowledge, our study is the first to consider the network activity of individual atomic operations in Ajax applications. We discuss these details in Section 4.3.

While the primary purpose of the tool is to characterize Ajax-based cloud applications, we believe that AJAXTRACKER will prove useful in many other scenarios. For instance, it could provide interference-free access to the ground-truth required to train classifiers in several statistical traffic classification systems [18, 15, 20]. Its fine-grained analysis capabilities will allow network operators to model, predict traffic characteristics and growth, conduct 'what-if' analyses and so on.

2 Background and motivation

Today, many cloud application providers are increasingly focusing on enriching the user interface to make these services resemble desktop look-and-feel as much as possible. Perhaps, the most prominent ones among these are Mail, Documents, and Maps[2] applications, which are now offered by companies such as Google, Microsoft and Yahoo among others.

In the traditional Web, the navigation model of a Web session is quite straightforward: A user first clicks on a URL, then, after the page is rendered, he thinks for some time and requests another object. This process continues until the user is done. On the other hand, the navigation model of modern Ajax web sessions is quite different: A user can click on a URL, drag-and-drop on the screen, zoom in or zoom out (if it is a maps application) using the mouse scroll button among several other such features. In addition, the Javascript engine on the client side can request objects asynchronously and autonomously without the user ever requesting one. For example, when using Ajax-based email services, the browser automatically tries to save data when a user is composing email.

[1] http://www.cs.purdue.edu/synlab/ajaxtracker

[2] While Maps application is not strictly an enterprise cloud application, it exports a rich set of Ajax features making it an interesting Ajax application to characterize.

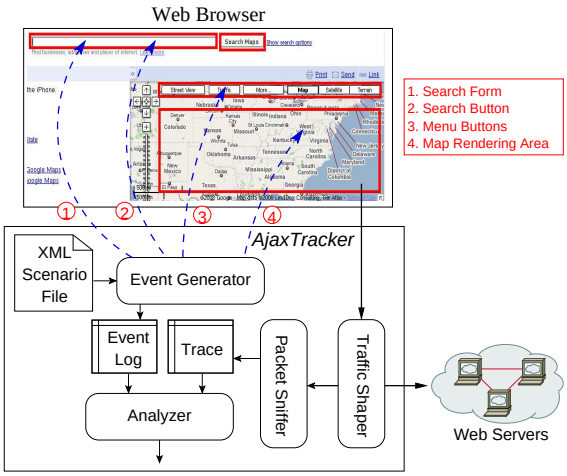


Figure 1: Structure of AJAXTRACKER.

Given their importance in the years to come, as increasingly more applications migrate to the cloud, it is important to characterize these applications and understand their impact on the network. Due to the forementioned shortcomings of passive approaches, we take an *active measurement* approach for characterizing these applications in this paper. The basic idea of our approach is to perform application measurement and characterization at the end-host. By ensuring that there exists only one application session at any given time, we can collect packet traces that are unique for that particular session, even if the session itself consists of connections to several servers or multiple connections to the same server. We, however, need a way to generate user application sessions in an *automated* fashion that can help repeatedly and automatically generate these sessions under different network conditions.

Unfortunately, there exist few tools that can interact with applications in an *automated* manner. Web crawlers lack the sophistication required on the client-side to generate Ajax-behavior. Traditional crawlers have no built-in mechanisms to interact with the interactive controls (like drag-and-drop), which require mouse or keyboard input, and are fairly common in these new applications. In the next section, we describe the design of AJAXTRACKER to overcome these limitations.

3 Design of AJAXTRACKER

The main components of AJAXTRACKER include an event generator, a Web browser, a packet sniffer, and a traffic shaper as shown in Figure 1. The event generator forms the bulk of the tool. It produces a sequence

of mouse and/or keyboard events that simulate a human navigating a cloud-based Web site based on a configured scenario file written in XML. These events are then input to an off-the-shelf Web browser (*e.g.*, Mozilla Firefox) that then executes them in the order it receives individual operations. Note that AJAXTRACKER itself is agnostic to the choice of the Web browser and can work with any browser. Given the goal is to collect representative traces of a client session, AJAXTRACKER employs a packet sniffer (*e.g.*, tcpdump [2]) that captures the packets on the client machine. These packets can then be examined to obtain specific characteristics of the simulated client session.

In addition to the basic components described above, AJAXTRACKER also makes use of a traffic shaper that can be configured to simulate specific bandwidth and delay conditions under which the corresponding cloud application sessions are simulated. This feature enables the tool to obtain many client sessions, each possibly under different network conditions to simulate the real-world settings where each user is exposed to different set of network conditions. Finally, the tool has the ability to perform causality analysis between operations (obtained from the browser's event log) on a cloud Web site and the corresponding network activity captured from the packet sniffer's trace.

AJAXTRACKER works by first configuring the traffic shaper with delay and bandwidth parameters. Next, it runs tcpdump, then launches the Web browser with the corresponding url that is indicative of the cloud application that we wish to simulate. The event generator is then executed until all specified events have been processed. We describe individual components in detail next.

3.1 Scenario file

A scenario file is intended to model a sequence of user operations that need to be executed to emulate a user session. For example, a user session could consist of entering a location, say New York, in the search tool bar in the Maps application and clicking on the submit button. The next action, once the page is rendered, could be to zoom in at a particular location within the map page. After a certain duration, the session could consist of dragging the screen to move to a different location. The scenario file is intended to allow specifying these sequence of operations. In addition to static scenarios, the scenario file will enable the tool to explore random navigation scenarios that for example, execute events in a random order or navigate to random locations within the browser. Random scenarios help the tool users to explore a richer set of navigational patterns that are too tedious to specify using the static mechanisms one-by-one.

The scenario file is composed of mainly three categories—events, objects and actions. There are three broad categories of events: pre-processing events (specified with the tag PRE_EVENTS), main events (MAIN_EVENTS tag), and post-processing events (POST_EVENTS tag). Events in pre- and post-processing category are sequentially executed exactly once—before and after the main events are simulated. Main events can be specified to be executed in either static or random order using the values 'static' and 'random' within the TYPE attribute in MAIN_EVENTS tag.

Model	PDF	Attributes
static	-	PERIOD
uniform	$1/d$	PERIOD
exponential	$\mu e^{-\mu x}$	EXP_MU
weibull	$\frac{b}{a}(\frac{x}{a})b-1_e-(x/a)^b$	W_A, W_B
pareto	$\alpha k^\alpha x^{-(\alpha+1)}$	P_A, P_K

Table 1: Inter-operation time distribution models.

Each event (within the EVENT tag) enumerates a list of objects, with each object described by an identifier, action and a pause period (PAUSE_TYPE attribute) that specifies the amount of time the event generator should wait after executing the specified action on the object. The time specification can be either a constant or could be drawn from a distribution (such as Pareto or exponential). The pause period specification helps model human think time in a sequence of operations. Table 1 shows the five different distributions for pause between operations that are currently supported by our tool. In the EVENT element, if LOG attribute is specified, AJAXTRACKER records event logs into a file that can be later used for correlating with packet logs for causality analysis.

Objects are specified within the broader OBJECTS tag and individually enclosed within the OBJECT tag. Actions are associated with individual objects. Each object defines its position or area and possible actions. Depending on the type of the object, specific tags such as TEXT or READ_FILE are specified that are appropriate for those objects. For example, to fill submission form from a set of predefined texts, the input that needs to be supplied is specified using the READ_FILE object to indicate a file which contains a set of predefined texts. An input text from the file is fetched on a line-by-line basis by the object. The value encapsulated by ACTIONS tag defines supported actions (a list of support actions is depicted in Table 2) and the position where the action needs to be performed. If position values (*e.g.*, X, S_X) in ACTION tag are not defined, the values of POSITION or AREA tags are used.

Example. A small snippet of a sample scenario file for Google Maps is shown in Figure 2. Note that this example is not meant to exhaustively capture all the capabilities of the tool. The scenario forces AJAXTRACKER

Action	Meaning
left_click	click with left mouse button
right_click	click with right mouse button
select	pushing Ctrl+A
delete	pushing Backspace
copy	pushing Ctrl+C
cut	pushing Ctrl+X
paste	pushing Ctrl+V
drag	move object holding mouse left button
wheel_up	scroll up an object
wheel_down	scroll down an object

Table 2: Supported actions.

```
 1: <SCENARIO>
 2:  <NAME> Google Maps </NAME>
 3:
 4:  <PRE_EVENTS>
 5:  </PRE_EVENTS>
 6:  <MAIN_EVENTS TYPE="static" EXEC_COUNT="2">
 7:   <EVT_REF IDREF="navigate_map" />
 8:  </MAIN_EVENTS>
 9:  <POST_EVENTS>
10:   <EVT_REF IDREF="close_window" />
11:  </POST_EVENTS>
12:
13:  <EVENT ID="navigate_map">
14:   <OBJ_REF IDREF="search_form" ACTION="paste"
15:    PAUSE_TYPE="static" PERIOD="1" />
16:   <OBJ_REF IDREF="search_button" ACTION="click"
17:    PAUSE_TYPE="static" PERIOD="1" />
18:   <OBJ_REF IDREF="map_area" ACTION="drag"
19:    LOG="drag map"
20:    PAUSE_TYPE="pareto"
21:    PARETO_K="1" PARETO_A="1.5" />
22:  </EVENT>
23:
24:  <OBJECTS>
25:   <OBJECT ID="search_form">
26:    <POSITION X="359" Y="225" />
27:    <ACTIONS>
28:     <ACTION ID="paste">paste</ACTION>
29:     <ACTION ID="click">left_click</ACTION>
30:    </ACTIONS>
31:    <READ_FILE>/ajax/env/list.site</READ_FILE>
32:   </OBJECT>
33:
34:   <OBJECT ID="map_area">
35:    <AREA LEFT="500" TOP="333"
36:          RIGHT="1241" BOTTOM="941" />
37:    <ACTIONS>
38:     <ACTION ID="drag" S_X="600" S_Y="400"
39:      E_X="900" E_Y="900" COUNT="1">drag</ACTION>
40:    </ACTIONS>
41:   </OBJECT>
42:  </OBJECTS>
43: </SCENARIO>
```

Figure 2: Example scenario file for Google maps.

to work as follows: Events are executed in the order of 'navigate_map' and 'close_window'. If there exist events in PRE_EVENTS, these events are executed first. Then, the tool statically executes 'navigate_map' event twice (as specified in the EXEC_COUNT attribute). If there are more than one event listed, the tool will sequentially execute each event twice. For each instance of 'navigate_map' event, operations specified between lines 13 to 22 of the scenario file are executed. We only allow the tool to execute the operations serially because we defined an event as a series of operations to accomplish a task. Thus, in this event, a query string retrieved from list.site file in line 31 is put into 'search_form' object, and the tool takes inter-operation time of 1 second. Then, 'search_button' object is clicked and the tool lets another 1 second elapsed. After that, the tool generates 'drag mouse' window event which is followed by inter-operation time probabilistically selected by Pareto distribution. In addition, the tool records a log in the file "drag map" specified as part of the OBJ_REF description. For the tool to identify the coordinate of object or actions to be taken, the tool searches objects which are defined from lines 24 to 42 by using object ID whenever it executes an object.

Note that the scenario file can describe events at both semantic level or in the form of coordinates; our system allows both types of input. The choice of one over the other depends on the particular event that needs to be described. Drag-and-drop actions on maps applications, for instance, are better represented using coordinates, while actions that involve submitting forms (*e.g.*, save file) are better represented at the semantic level.

A scenario is presented in a hierarchical fashion. One can first list events to generate and flexibly compose events with one or more objects and actions against the objects. Multiple actions can be defined within an object which can be reused in several events. While the specification allows users to build various scenarios covering many user interactions, it is hard to cover all user actions due to complexity of user actions and coordinate-based specification of an object. Note that although the specification appears complicated, we have found in our experience that coding the scenario file does not take too long. In our experience with 14 scenario files, the longest was 454 lines that took us less than an hour to specify. Once the scenario file is specified, the tool itself performs completely automatically and can work repeatedly and continuously. Thus, the cost of specifying the scenario file is amortized over the duration over which the scenario is monitored. As part of our future work, we are working on automating the generation of the scenario file by recording and replaying user/client activities passively using a browser plugin.

3.2 Event generator

Given a scenario file, the event generator first parses it and builds data structures for elements listed in the scenario file. Then, the event generator registers a callback function called run_scenario() for a timer. The callback function plays a core role in generating mouse and keyboard events. Every time the function is called, it checks if the whole events in the scenario file were executed. If there is any event left, the function generates mouse or keyboard events accordingly. As discussed before, the

event generator supports two basic navigation modes—static and random. In the static navigation mode, AJAXTRACKER generates the sequence of events exactly in the order specified in the scenario file. In the random mode, it provides different levels of randomness. First, the event generator can randomly select the order of events in main event class by assigning TYPE attribute as 'random' which implies uniform distribution. Second, in an event, it can adjust inter-operation time with four different probabilistic distributions if PAUSE_TYPE is defined as one of values (except 'static') listed in Table 1. Other distributions can optionally be added. Third, action can be executed randomly. For instance, if TYPE attribute in ACTION element is set 'random', the tool ignores position values (*e.g.*, X, S_X), and executes the action by uniformly selecting position or direction (in case of drag) within values of AREA element and the number of clicking objects within the value of COUNT attribute.

In case of action name called 'paste', the tool can randomly select one from text list which it manages and pastes it in the input form of a Web site. Moreover, by simply changing the number of main events and reorganizing the execution procedure of an event, we can let the event generator work completely differently. Thus, through this way of providing randomness, the event generator strives to generate random but guided navigation scenarios to simulate a larger set of client sessions.

We implemented the event generator as a command-line program with 3500+ lines developed using C++, GTK+, the X library and Xerces-C++ parser [4].

3.3 Traffic shaper

Often, it is important to study the characteristics of these applications under different network conditions. Given that the tool works on an isolated end-host, the range of network conditions it can support is quite dependent on the capacity of the bottleneck link at the end host. For example, if the tool is being used by an application service provider (ASP), typically, the ASP is going to use it in a local area network (close to the Web server) where the network conditions are not as constrained as clients connected via DSL or Cable Modem or Dial-up or some such ways to access the Web service. The traffic shaper, in such cases, provides a way to study the application performance by artificially constraining the bandwidth as well as increasing the round-trip times of the network.

The traffic shaper mainly implements bandwidth throttling and delay increases, and does not factor in packet drop rates. Packet losses are not directly considered in our tool at the moment since available bottleneck bandwidth, to some extent, forces packets to be lost as soon as the bottleneck capacity is reached. We can, however, augment the tool with arbitrary loss fairly easily. We used an open-source software router called Click [16] in our tool for implementing the traffic shaping functionality. We ran Click as a kernel module in our system. Note that any software that provides the required traffic shaping functionality would work equally well.

Time	Event	#click	S_X	S_Y	E_X	E_Y
1224694605.059651	click search button	1	620	220	620	220
1224694975.651213	zoom out	2	940	832	940	832
1224695045.303020	drag map	1	1128	537	1021	470
1224695062.083703	zoom in	10	824	554	824	554
1224695175.203356	drag map	1	858	693	867	411

Figure 3: Example event log snapshot generated.

3.4 Packet capture

To characterize the network-level performance of an application session, it is important to capture the packet-level traces that correspond to the session. One can potentially instrument the browser to obtain higher-level characteristics, such as URLs accessed and so on. However, our goal is to characterize network activity; thus, we employ off-the-shelf packet sniffer such as tcpdump to capture packet traces during the entire session. Since AJAXTRACKER aims to characterize mainly cloud Web applications, it filters out non-TCP non-Web traffic (i.e., packets that do have port 80 in either the source or destination port fields). AJAXTRACKER considers all captured 5-tuple flows of <*src, dst, src_port, dst_port, protocol*> to form the entire network-activity corresponding to a given session. We do not need to perform TCP flow-reassembly as we are mainly interested in the packet-level dynamics of these sessions.

Advertisement data, however, which are parts of a Web site but are not Ajax-based cloud application related, can also be included in the trace file. If we do not wish to analyze the non-Ajax content, depending on the application, we apply a simple keyword filter which is similar to ones used by Schneider *et al.* [21] to isolate Ajax-based flows alone. We find related flows whose HTTP request message contains keywords of interest and retrieve bidirectional flow data.

3.5 Causality analysis

Our tool generates traces as well as logs about which operations were performed with their timing information as shown in Figure 3. While information in the first two columns is mainly used for causality correlation, other five columns provide auxiliary information. The third column denotes the number of mouse clicks. The fourth and fifth columns denote the screen coordinates where an event begins to occur and the last two columns represent the screen coordinates where it ends. We do not at the moment use this auxiliary information however, and focus mainly on the first two columns. Based on these two pieces of information, i.e., timestamp and event name,

we can reconstruct the causality relationship between operations in the Web browser and the corresponding network activity by correlating the network activity using the timestamp.

Using this causality, we can isolate the network effects of individual operations, such as finding what Web servers are contacted for a given operation, the number of connections that are open or the number of requests generated by each operation. Such causality analysis helps when anomalies are found in different applications as one can isolate the effects of individual operations that are responsible for the anomalies. In addition, the causality analysis helps predict how the application traffic is going to look like, when we change the popularity distributions of different operations in a given sequence. For example, if users use the zoom features much more than drag-and-drops, we can study the underlying characteristics of such situations.

Note that while this timing-based causality works well for simple applications we considered in this paper such as Gmail and maps, it may not work easily for all events and applications. For instance, if we investigate 'autosave' event of Google Docs, we need to know when the event is triggered while a user composes, which may not be simple to know unless the browser is instrumented appropriately. Modifying the browser, however, introduces an additional degree of complexity that we tried to avoid in our system design.

3.6 Limitations

As with perhaps any other tool, AJAXTRACKER also has some limitations. First, since it works depending on the layout of user interface, significant changes to the user interface by the application provider may cause the tool to not operate as intended. Though this limitation may seem serious, observations made by the tool over a period of weeks shows considerable consistency in the results of the static navigation mode, barring a few user interface changes that were easy to modify in the scenario file. Second, in our current setup, we specify the mouse clicks in the form of coordinates, which assumes that we have access to the screen resolution. If the setup needs to run a separate platform, the scenario files need to be readjusted. One way to address this issue is to specify them relative to the screen size; we did not implement this feature yet and is part of our future work. Third, because the operation of our tool depends on a specified scenario file, the generated workloads cannot cover all possible user space. Instead, we try to configure scenario files with functions which are most likely to be used by users in each explored application. Currently, while we program these scenarios ourselves, we are also investigating representative client session models and deploying them into our tool. Note that these models are orthogonal to our tool design itself. Third, given the nature of the traffic shaper, we cannot emulate all types of network conditions; we can either reduce the bandwidth or increase the RTT in comparison with the actual network conditions at the location where the tool is deployed.

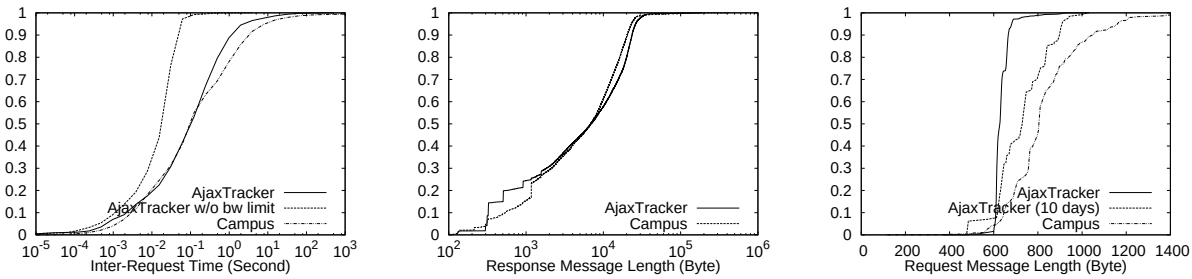
4 Evaluation

In this section, we present our measurement results obtained using the tool on real Ajax applications. We categorize our results into three main parts. First, we demonstrate that our tool produces representative traces by comparing our results with a passive campus trace. Second, we perform macroscopic characterization of full application sessions generated using our tool. We also show how Ajax application traffic characteristics change with different network conditions. Third, we show the characterization of individual operations such as 'click' and 'drag-drop' in two canonical Ajax applications—Google Maps and Mail—with the help of the causality analysis component of our tool.

4.1 Comparison with a real trace

The representativeness of our tool is completely dependent on the expressiveness of our tool and the scenario files specified. In order to demonstrate that the scenarios we have specified in our tool are representative, we show comparisons with a real passive trace. For the purposes of this experiment, we have obtained a real trace of Google Maps traffic from a campus switch of Purdue university. There are approximately 300 machines connected to the switch and users are mainly campus students. The Google Maps trace we collected represents 24 hours worth of client activity over which we observed about 182 unique clients totaling about 13,200 connections. While our campus trace is not representative of all settings, our trace is representative of network infrastructure environment that corresponds to typical enterprise networks, and hence, the use of Google Maps in this environment is arguably similar to that of any other organization's use of Google Maps.

Figure 4 shows the comparison results in terms of inter-request time (IRT), response and request message length (QML). IRT is an important metric because it can show or measure how proactive the application is. If IRTs are much smaller than RTT (if we measure RTT), it implies that the application is more proactive and relies on parallel connections for fast retrieval of traffic. For IRT, we calculated intervals between request messages sent to the same server through multiple flows, instead of calculating intervals between every request messages regardless of the destination. We believe that this is a reasonable approach to calculate IRT because ignoring the destination may lead to a much higher request frequency for Ajax application's traffic, but it is misleading



(a) CDF of inter-request time (b) CDF of response message length (c) CDF of request message length

Figure 4: Comparison between AJAXTRACKER's trace and Campus trace of Google Maps.

as some of the traffic will be destined to multiple destinations (typically within the same cluster).

First, we plot the comparison between AJAXTRACKER (the curve that says AJAXTRACKER without bandwidth limit) and campus trace in terms of their IRT distributions in Figure 4(a). When we compare the two, we can observe clearly that they are not similar. This is because, IRT distributions are easily affected by the available network bandwidth. Since the clients in the trace may potentially have a different throughput constraints from the machine we use AJAXTRACKER from, we need some calibration to match the trace. We first analyzed the average instantaneous throughput of Maps with a bin of size 1 second for every host in the trace. We excluded the case where there is no traffic in considering instantaneous throughput. The average instantaneous throughput was about 580Kbps. Specifically, in cumulative distribution, 82% of instantaneous throughput were less than 1Mbps, 16% were between 1-5Mbps, and 2% were between 5-20Mbps.

Based on the above observation, we ran our tool with different network bandwidth constraints to adjust available network bandwidth artificially. To simulate the distribution of instantaneous throughput, we differentiated the proportion of each trace generated by AJAXTRACKER under different network bandwidth conditions based on the distribution we have observed in the campus trace. Specifically, traces generated by our tool has fixed data rate configured by Click: 500Kbps, 1Mbps, 5Mbps, and 10 Mbps. On the other hand, campus trace has continuous distribution from around 500Kbps to 20Mbps. We envision that there are only a few different quantized access bandwidths for different clients within a network. By empirically finding these numbers, one can run the tool and mix different traces with different weights. Thus, we empirically gave 10% weight to a trace by 500Kbps constraint, 72% weight to a trace by 1Mbps constraint, 16% weight to a trace by 5 Mbps, and 2% weight to a trace by 10Mbps constraint, and conducted weighted IRT simulation.

We found that IRT distribution generated by AJAXTRACKER is quite close to the IRT distribution of the campus trace as shown in Figure 4(a). There are still a few minor discrepancies; at around 0.6-2 seconds, there is 10% discrepancy between two curves. IRTs larger than one second are typically because of human think time, as has been described by Schneider *et al.* in [21]. Thus, we believe this area of discrepancy that represents human think time exists because of the discrepancies between our scenario models that model the inter-operation duration and real user's behavior. If needed, therefore, we can carefully tune the scenarios to easily match the campus trace. Such tuning may or may not be necessary depending on the particular use of the tool; the more important aspect is that the tool allows such calibration.

The distribution of response messages (shown in Figure 4(b)) are quite similar between AJAXTRACKER and the campus trace for the most part. The big difference ranging from about 300 to 1,000 bytes is related to whether basic components (*i.e.*, icons, thumbnail images, etc.) that constitute Maps application are already cached or not. Because we ran AJAXTRACKER ensuring that the browser has no cached data, the fraction of that area in AJAXTRACKER's distribution is larger than that of campus.

While we have not conducted extensive experiments to study the impact of cached data in this paper, we note that storage size generally for cached data is limited and stale data is typically removed from cache storage. Thus, in general, the fact that Web browser caches data does not mean that it cached data for a particular application of interest. In addition, Ajax applications often consist of two portions: Application source (e.g., javascript code) that is static and needs to be downloaded only once when a user accesses the application for the first time and, application data that is typically more dynamic and diverse than application source. Thus, caching typically impacts only the application source download and not so much the data portion of the session.

Finally, in Figure 4(c), we can observe that QML dis-

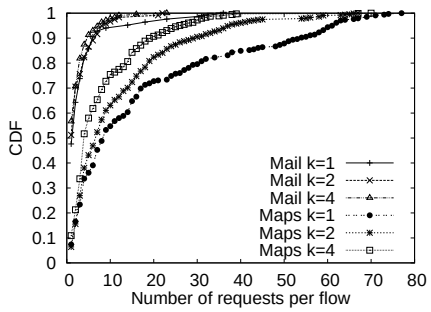


Figure 5: CDF of number of requests per flow. k is the scale parameter. The shape parameter α is fixed at 1.5.

tribution is significantly different from what is observed using AJAXTRACKER and the campus trace. This discrepancy is because QML can vary quite a bit because of the variation of cookie length. We collected 10 days of traces to find if there are changes in the length of request messages, and their QMLs showed diversity due to different cookie length. When we joined together those traces, we found that the distribution comes close to the curve of campus in Figure 4(c). There are also browser differences in campus trace can cause different size of requests for the same objects. When we adjust for these differences, the trends of the distribution by AJAXTRACKER and campus will be similar. The bigger takeaway from these experiments is that the results obtained using our tool can be made to be easily consistent with those observed in real life. These experiments also suggest an interesting side-application of our tool in reverse engineering the traffic shaping policy at the monitoring network. For the rest of the experiments, we do not put any restriction on the bandwidth, since there is nothing particularly fundamental about any given aggregate bandwidth and can vary depending on the situation.

4.2 Characterization of full sessions

In this section, we characterize overall application sessions. We first describe some flow-level characteristics of Ajax applications. Next, we discuss our results by simulating different network conditions at the end-host.

Number of requests per flow. Since persistent connections are supported by modern Web browsers, the number of requests per flow depends on the inter-operation duration. If there is no network activity, the flows are usually terminated. To understand this dependency, we varied the inter-operation duration as a Pareto distribution, which has been used before to model human think time for traditional Web traffic [6]. Given that we do not yet have representative parameters for Ajax applications, we chose to experiment with those reported in [6] for regular Web traffic and some new ones. For both Google Maps and Mail, we chose the Pareto scale parameter k to be 1, 2 and 4 and fixed the shape parameter α as 1.5 (as suggested in [6]).

Regardless of values of k, we can observe clear difference between Maps and Mail in Figure 5. While Mail has at most 31 requests per flow at $k = 1$, Maps generates 67 requests per flow at $k = 4$. In the head of distribution, while about 50% of all Mail's flows have only one request, only 10% of Maps' flows have one request. Interestingly, the top 93% exhibit similar trends for Mail, after which the $k = 1$ curve exhibits higher number of requests per flow. This phenomenon is expected since, smaller values of k imply better re-use of connections, which in turn leads to larger number of requests per flow.

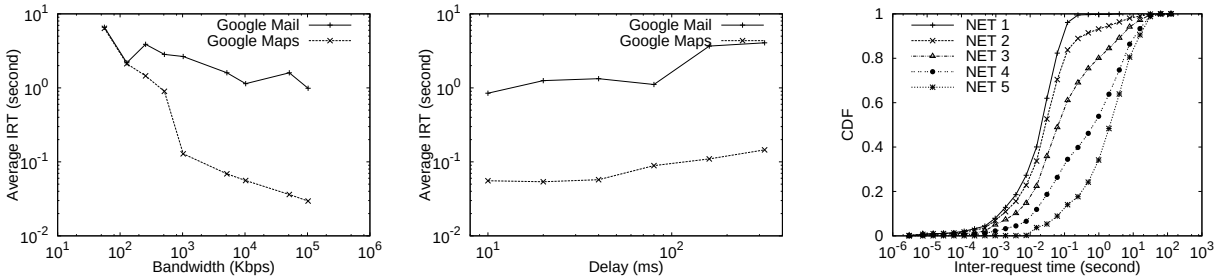
In Maps, on the other hand, the number of requests per flow exhibits big difference depending on values of k. The reason for the difference could be because of the lack of a mechanism in Maps similar to asynchronous updates in Mail. While Maps prefetches map tile images, the connections are closed faster than Mail's connections.

Our analysis on number of requests per flow uses flow-level granularity while Schneider *et al.* report session-level results in [21]. Despite this difference, our results roughly match their observations, in that Maps generates more requests than Mail.

Effects of different network conditions. To understand how these applications behave under different network conditions, we let AJAXTRACKER run on emulated network environments using its traffic shaping functionality. We conducted two experiments on Maps and Mail: The first is a bandwidth test where the traffic shaper throttles link capacity. The second is a delay test where it injects additional delays in the link to artificially increase the RTT. In our configuration, the set of bandwidths we have chosen include {56Kbps, 128Kbps, 256Kbps, 512Kbps, 1Mbps, 5Mbps, 10Mbps, 50Mbps, 100Mbps}. For delay, we chose values from 10ms to 320ms in multiples of 2 (*i.e.*, 10ms, 20ms, 40ms, etc.).

While our framework allows adding delay to both outbound as well as inbound packets, we added the simulation delay only to the inbound packets. This is because, it is the RTT that typically dictates the connection characteristics and hence it suffices to adding it in either of the directions. The inter-operation times were statically configured in the scenario files.

Figure 6 shows how IRT is affected according to the change of bandwidth and delay, respectively. We used the causality between operation and network activity from log information and traces in order to remove large IRTs which come from the the interval between the time when last request message of previous operation was seen and the time when the first request message of current operation is seen because these large IRTs affected by inter-operation time (which is decided by a user) restrain our understanding about applications' behavior.



(a) Bandwidth with no added delay (b) Delay with 50 Mbps (c) Simulated IRT

Figure 6: Average IRT variation for different network parameters.

From Figure 6(a), we can observe that, as network bandwidth increases, the average IRT of Maps decreases fast, but the extent to which Mail's IRT distribution decreases is small. The graph shows that Maps fetches contents more aggressively than Mail does. On the contrary, Figure 6(b) shows that IRTs of both applications are relatively less affected by the increase of delay. The figure indicates that IRT feature is more sensitive to bandwidth rather than delay. Because Ajax applications can actively fetch data before the data are actually required by users, network delay may have little sensitivity. On the other hand, network bandwidth directly impacts the performance of Ajax applications since it takes more time to pre-fetch the content. We believe that our tool helps answer whether a given Web site is more sensitive to either bandwidth or latency by allowing different settings for the Web site.

	10 Mbps	1 Mbps	56 Kbps
NET1	100%	0%	0%
NET2	70%	20%	10%
NET3	20%	50%	30%
NET4	10%	20%	70%
NET5	0%	0%	100%

Table 3: Configuration for weighted IRT simulation.

Since these results indicate a direct dependency of IRT on the network conditions, we consider how IRT distributions change when different clients with different network conditions are mixed together, as a router in the middle of the network would probably observe. Thus, we conduct a simple simulation by mixing together different proportion of clients with different network conditions (particularly 10Mbps, 1Mbps and 56Kbps clients) in Figure 6(a). The parameters for the simulation are summarized in Table 3. Figure 6(c) shows these different IRT distributions for these different combinations.

We believe our mechanism provides interesting projections into how the IRT distribution varies according to the traffic mixes. For example, as we move from NET1 which consists of users with extremely high bandwidth (100% users have 10 Mbps) to the NET5 (100% users have 56Kbps), we see the progressive shift in the curves to the right, indicating a stark increase in the IRT distributions.

We also investigate how the number of requests per flow is affected by changes of bandwidth and network delay constraints. (Due to space constraints, we omit showing the graphs.) We observed that as more bandwidth becomes available, both Maps and Mail services increase the number of requests per flow. As we increase the delay, we found that Mail application showed a slight dip in the number of requests (4 reqs/flow at 10ms down to about 2 reqs/flow at 160ms) while Maps shows variations of 9-15 reqs/flow at 10-80ms but decreases 7 reqs/flow from 160ms because the number of flows itself increases (about 200 flows to 80ms and about 340-400 flows from 160ms).

4.3 Characterizing individual operations

We begin our discussion with explaining methodology for characterizing individual operations. In Table 4, we show the candidate operations within Google Maps and mail applications we selected for our analysis. These operations are by no means exhaustive; we hand-selected only basic operations that seem to represent the main objective of the application. We can easily instrument the tool for characterizing other operations as well. For this analysis, we collected about 250 MB of data representing about 10,000 flows across the two candidate applications for the operations included within Table 4. Each operation has been repeated several times for statistical significance, with exact counts for each operation outlined in Table 4. The counts are different for different operations because we wrote different scenario files, each of which represent different sequences of operations and extracted out the individual operations from the sessions.

We configured the scenario files for each application with a 60 second interval between two adjacent operations (120 seconds for 'idle state' of Mail) to eliminate any interference between them. We then matched the

App.	Operation	Meaning	Count
Google Maps	drag map	dragging a mouse on the map area	140
	zoom in	zooming in map area by moving wheel button on a mouse	69
	zoom out	zooming out map area by moving wheel button on a mouse	62
	click search button	clicking the search button for finding a location	102
Google Mail	delete mail	clicking mail deletion button after selecting a mail	52
	attach file	inputting a file path, and then waiting to be uploaded	37
	idle state	letting Google Mail idle without operations for 120 seconds	56
	click inbox	clicking inbox and listing mails	125
	read mail	reading a mail by clicking mail	70
	send mail	clicking send button to send a mail	96

Table 4: Candidate set of operations selected within Google Maps and Mail applications for characterization. These are some of the most commonly used operations within these applications.

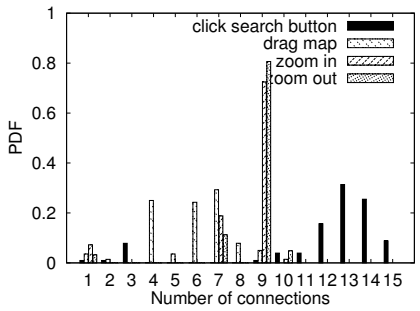
mouse event timing recorded in the logs with the trace files to extract out the relevant network activity. To determine the causal relationship, we collected all flows generated within 5 seconds of the mouse events and analyze their network-level behavior. In the means of correlating operation and traffic, the time value is selected empirically. While in different applications we may have to use the time value different from one used in this paper or require different approaches, we find that the approach works fine for the given two applications in our setting. Thus, only flows that are a direct result of the event are considered for the analysis; some asynchronous flows reflective of the polling mechanisms, such as those employed in Mail, are considered separately through an 'idle state'. To ensure minimal interference with active user activity, the 'idle state' is not mixed with any other user action.

We had to make a few other minor adjustments. The first involves the 'file attach' operation in Mail. File attachments in Mail are handled by specifying a file directory path information in the form, which then attaches and uploads the file automatically after a fixed 10 seconds because there is no explicit upload button; thus, we configured the tool to collect all network activity within 11 seconds (which ensures that any flows that occur as a result of this activity are started within 11 seconds). The other adjustment we required was for the 'idle state' of Mail, which unlike Maps, continuously exchanges information with the server even in the absence of any particular mouse event. To accommodate this, we chose 60 seconds as the window interval to include flows generated during 'idle state' to obtain a finite snapshot of the polling mechanisms in use. Note that once we identify the flows belonging to a specific operation, we analyzed every packets until the flows are closed. The time thresholds are only to identify which flows to keep track of.

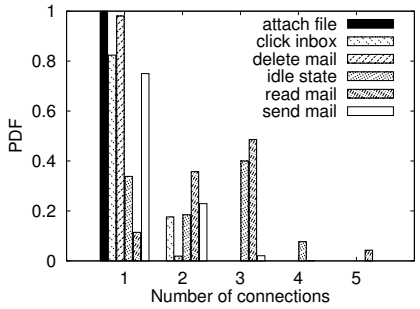
To obtain flow-concurrency, we counted the number of concurrent flows per distinct host (*i.e.*, on a per-IP address basis) and the number of distinct hosts contacted within a window of α seconds of each other. In our analysis, we set α to 0.1 seconds, which is an adjustable parameter in our analyzer. This ensures that flows that are generated automatically as a result of a particular action are obtained and not necessarily those that are related to human actions.

We considered two types of network-level metrics: flow- and message-oriented. Along flow-based metrics, we mainly considered concurrency in terms of number of simultaneous flows generated. For message-oriented metrics, we selected number of request messages per flow. We have analyzed a few more, such as flow inter-arrival time, inter-request time, bytes transferred, request and response message length, and so on, but in the interest of space, we do not show them in this paper. We can potentially also consider packet-level metrics, such as packet size distribution and their timing, but we chose to model mainly Ajax application's characteristics; packet size distributions are dependent on how the TCP layer segments individual packets, and thus is less intrinsic to the Ajax application itself.

Connection Distribution. We analyze connection distribution along three dimensions—total number of connections, number of connections per server, and the number of servers contacted per operation. However, due to space limitation, we only provide a graph about total number of connections in this paper, but briefly explain the other two results. From Figures 7(a) and 7(b), we observe that Maps opens the most number of total connections (up to 15) as well as the most number of connections per server (up to 8). This phenomenon is because Maps requires the fastest responsiveness as every user activity leads to a large number of requests for fairly large map image tiles. While the Mail application also requires fast responsiveness, the responses are usu-

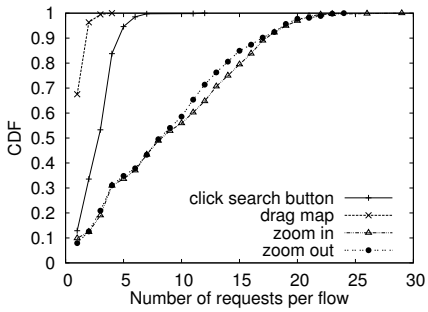


(a) Google Maps

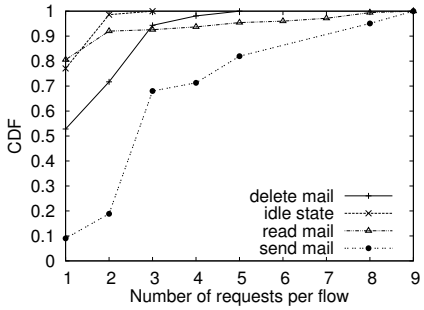


(b) Google Mail

Figure 7: Total connection distributions.



(a) Google Maps



(b) Google Mail

Figure 8: Number of requests per flow.

ally small, and thus fewer connections are sufficient to fetch the required information.

Among operations within Maps, 'click search button' starts the most number of connections with up to 8 connections per server and up to 5 different servers. This is because clicking on the search button typically leads to the whole refresh of a page thus involving most amount of response data. Other operations on Maps involve more connections (around 4–9) as well, but typically to a much lesser number of servers (around 1 or 2). We believe this might be because operations such as 'drag and drop' typically fetch a few tiles that are adjacent to the current location on the displayed map, thus resulting in smaller amount of data compared to the total refresh when the search button is clicked.

For Mail, we observe that most connections are found in the case of 'read mail' operation. To identify the reason, we inspected the HTTP headers and observed that along with the mail content, several advertisements were fetched from different servers causing more number of connections. The 'idle state' operation came next in terms of the number of connections and servers involved. This is because we use a window interval of 60 seconds, which in turn results in a lot of client requests generated to synchronize with the server for any further updates.

One concern is that the characteristics of Gmail sessions may be dependent on the inbox content. Our approach cannot effectively model the application if there is a lot of variation based on different inboxes, unless appropriate state is created in the form of different logins and different numbers of emails at the server side. In our future work, we will study how sensitive the characteristics of these applications are to the size of the inbox contents or the amount of content at the server side.

Number of request messages per flow. The middle graphs of Figures 8(a) and 8(b) show the CDF of the number of request messages per flow. In the case of Maps, most operations tend to have multiple requests in a flow. Specifically, zooming in and out results in changes of the map scale, and, as a result, the client needs to fetch whole new tile images repeatedly. On the other hand, in the case of 'drag map' operation, the probability of repeated fetching is lower. The 'click search button' operation also requires to fetch several tile images, but it achieves its goal through multiple connections to servers (shown later in Figure 7(a)). Unlike Maps, most operations in Mail have less than 9 requests in a flow.

5 Related Work

Given the recent emergence of Cloud-based applications into the mainstream Web, there has been limited research work in characterization of these applications. A recent paper by Schneider *et al.* [21] made one of the first attempts to characterize these application protocols by examining the contents of HTTP headers over ISP traces. In this work, they study the distributions of several fea-

tures such as bytes transferred, inter-request times, etc., associated with popular cloud-based application traffic. Our work is significantly different from (and in many respects complimentary to) theirs, as we focus on generating user interactions with a Web service in a controlled fashion to generate and obtain our traces.

Web traffic has been extensively studied in the past. While some studies focus on the user-level behavior in terms of the number of request and response messages and application-specific properties such as reference counts and page complexity (*e.g.*, [5, 9]), network-level characteristics were examined in others such as [12, 6, 7]. Several traffic generators based on statistical models have also contributed to understanding the impact of Web traffic (see [3] for a list of these) on the network. These models and tools, however, do not factor the exact cloud application traffic characteristics.

There also exist a lot of tools developed for the classical Web. Web automation tools such as Chickenfoot [8], CARENA [19], and SWAT [1] enable one to navigate a Web site automatically and repeatedly. However, these tools are mainly tailored for static navigation of Websites using automatic filling of forms and lack the functionality to interact with cloud applications.

Traditionally, characterizing and modeling network-level behavior of Web applications is largely trace-driven. For example, prior measurement efforts [21, 7, 10, 14] used Gigabytes to Terabytes of traces collected from core routers. On the other hand, we use end-host based active profiling to study the network-level behavior of individual operations within application.

The idea of end-host based active profiling is not new by itself. There have been several contexts where the idea has been applied. For example, Cunha *et al.* characterized Web traffic by collecting measurements from a modified Web browser [13]. In [17], Krishnamurthy *et al.* use Firefox add-on features to demonstrate how to collect statistics about page downloads. Our tool, shares some similarity, but is more tuned towards cloud applications and is designed to be browser agnostic as opposed to the other approaches.

6 Conclusion

As the popularity of cloud Web services increases, it becomes critical to study and understand their network-level behavior. A lot of tools designed for classical Web characterization are difficult to adapt to cloud applications due to the rich interface used by them. While trace-based approaches are standard for characterization, they do not allow characterizing individual operations within an application or provide any understanding on how these applications behave under different network conditions. We described the design and implementation details of AJAXTRACKER that is designed to address these limitations. It successfully captures realistic network-level flow measurements by imitating the set of mouse/keyboard events specified in the scenario file. The traffic shaper functionality allows it to simulate arbitrary network conditions.As part of ongoing work, based on our tool's capability, we hope to classify and detect cloud applications in the middle of the network. This is particularly useful given that cloud applications cannot easily be detected using port number approaches alone.

Acknowledgments

The authors are indebted to the anonymous reviewers and Marvin Theimer, our shepherd, for comments on previous versions of this manuscript. We also thank the IT staff at Purdue, Addam Schroll, William Harshbarger, Greg Hedrick for their immense help in obtaining the network traces. This work was supported in part by NSF Award CNS 0831647 and a grant from Cisco Systems.

References

[1] Simple Web Automation Tool. http://swat.sourceforge.net/.
[2] tcpdump. http://www.tcpdump.org/.
[3] Traffic Generators for Internet Traffic. http://www.icir.org/models/trafficgenerators.html.
[4] Xerces C++ Parser. http://xerces.apache.org/xerces-c/.
[5] M. Arlitt and C. Williamson. Internet Web Servers: Workload Characterizations and Implications. *IEEE/ACM Transactions on Networking*, 5(5), Oct. 1997.
[6] P. Barford and M. E. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of Performance '98/SIGMETRICS '98*, 1998.
[7] N. Basher, A. Mahanti, A. Mahanti, C. Williamson, and M. Arlitt. A comparative analysis of web and peer-to-peer traffic. In *WWW*, 2008.
[8] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *Proc. UIST '05. ACM Press*, pages 163–172, 2005.
[9] F. Campos, K. Jeffay, and F. Smith. Tracking the evolution of web traffic: 1995 - 2003. In *MASCOTS*, 2003.
[10] J. Cao, W. S. Clevel, Y. Gao, K. Jeffay, F. D. Smith, and M. Weigle. Stochastic models for generating synthetic http source traffic. In *INFOCOM*, pages 1546–1557, Mar. 2004.
[11] D. Crane, E. Pascarello, and D. James. *Ajax in Action*. Manning, 2006.
[12] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, Dec. 1997.
[13] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW Client-based Traces. Technical Report BUCS-TR-1995-010, Boston University, 1995.
[14] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube Traffic Characterization: A View From the Edge. In *ACM IMC*, 2007.
[15] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. In *ACM SIGCOMM*, 2005.
[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.
[17] B. Krishnamurthy and C. E. Wills. Generating a privacy footprint on the internet. In *ACM IMC*, 2006.
[18] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. In *ACM IMC*, 2006.
[19] I. J. Nino, B. de la Ossa, J. A. Gil, J. Sahuquillo, and A. Pont. Carena: a tool to capture and replay web navigation sessions. In *E2EMON '05: Proceedings of the End-to-End Monitoring Techniques and Services on 2005 Workshop*, pages 127–141, 2005.
[20] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-Service Mapping for QoS: A Statistical Signature-based Approach to IP Traffic Classification. In *ACM IMC*, 2004.
[21] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The New Web: Characterizing AJAX Traffic. In *International Conference on Passive and Active Network Measurement*, 2008.