Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications

Ding Li*
University of Southern California
dingli@usc.edu

James Mickens*
Harvard University
mickens@seas.harvard.edu

Suman Nath, and Lenin Ravindranath
Microsoft Research
{suman.nath, lenin}@microsoft.com

Abstract

In a modern web application, a single high-level action like a mouse click triggers a flurry of asynchronous events on the client browser and remote web servers. We introduce Domino, a new tool which automatically captures and analyzes end-to-end, asynchronous causal relationship of events that span clients and servers. Using Domino, we found uncharacteristically long event chains in Bing Maps, discovered data races in the WinJS implementation of promises, and developed a new server-side scheduling algorithm for reducing the tail latency of server responses.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—distributed debugging, testing tools

General Terms Algorithms, Performance, Measurement

Keywords JavaScript, Causality Tracking, Performance debugging, Web application

1. Introduction

A modern web application spans multiple clients and servers. A single, high-level request from a user triggers a flurry of asynchronous events on her client browser as well as on remote servers. Understanding the causal relationships between those events is crucial for diagnosing bugs and optimizing performance.

A rich literature exists on causality analysis for distributed systems, but prior work fails to capture the complex, wide area event causalities that characterize modern web applications. For example, systems like X-Trace [12] and Pinpoint [4] track a client's request ids across multiple low-level events; however, X-Trace focuses on *server-side* event causalities, ignoring the client-side JavaScript events that are often the source of application bugs [15]. AppInsight [18] and Timecard [19] do track fine-grained client-side events of .Net applications, but they lack fine-grained knowledge of *server-side* events. Thus, these systems cannot optimize server-side scheduling using knowledge of the "wide-area" event stream that spans both clients and servers (§3).

In this paper, we introduce Domino, a tool that captures and analyzes end-to-end event causalities in web applications. Domino automatically rewrites client-side JavaScript code, interposing on the event registration interfaces to detect when a new event fires, and how that event causes additional events. To track event dependencies across the wide-area, Domino leverages the fact that the JavaScript runtime is not only used on client browsers, but also on the server-side via Node.js [14]. Thus, Domino can use a single rewriting engine to generate event logs on both clients and servers; by propagating request ids across RPCs, Domino tracks causal relationships that span the wide area. This tracking also works for mobile application frameworks like WinJS [16] and PhoneGap [1] that use client-side JavaScript runtimes.

Using Domino, we analyzed event causalities in Bing Maps, finding an unusually long chain of events that we optimized to reduce network latency. When applied to WinJS apps, Domino discovered potential race conditions that arise from developer misunderstanding about how WinJS implements asynchronous callbacks. Domino's end-to-end causality information is also leveraged by a new server-side scheduling framework for Node applications—compared to shortest-job-first scheduling, our new scheduler reduces client-perceived tail latencies by up to 37%, with at worst 6% increase in average latencies.

2. Design

Domino defines a *causal tree* as a series of asynchronous events that are triggered by an initial, high-level application activity such as a page load or a mouse click. More formally, in a causal tree, (a) each node represents a callback, (b) the execution of a child callback is caused by a parent callback, and (c) the execution of the root callback is caused by an initial, high-level application activity. Domino's causal

*Work done while authors were at Microsoft Research.

SoCC '15, August 27 - 29, 2015, Kohala Coast, HI, USA.

DOI: http://dx.doi.org/10.1145/2806777.2806940

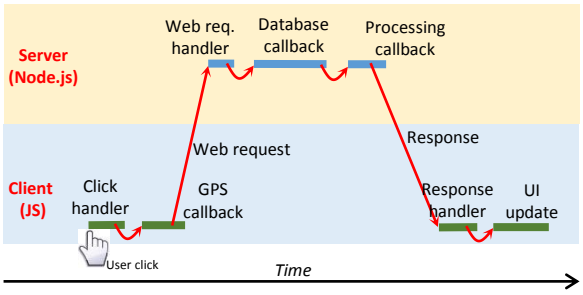


Figure 1. A causal tree from a location-aware web page. Solid horizontal bars represent the execution of asynchronous callbacks, and red arrows represent the causal relationships between callbacks.

```
1  var __setTimeout = setTimeout;
2  setTimeout = function(cb, delayMs){
3    var parentContext = contextStack.top();
4    var wrappedCb = function(){
5      var childContext = getNewContextId();
6      log(childContext, parentContext);
7      contextStack.push(childContext);
8      cb.apply(this, arguments);
9      contextStack.pop();
10   };
11   return __setTimeout(wrappedCb, delayMs);
12 };
```

Figure 2. The Domino JavaScript library interposes on timer registration interfaces like `setTimeout()`.

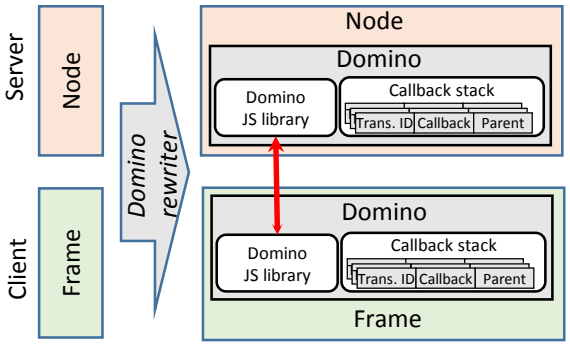


Figure 3. Domino's architecture. The Domino rewriter injects the Domino JavaScript library into the client-side and server-side of an application. The library logs event data and tracks causal relationships between those events. For WinJS applications, the rewriter also instruments code which generates promises (§2.1).

trees are similar to *transactions* in AppInsight [18][1], but Domino's trees can span clients and servers. Figure 1 shows a simple causal tree in a location-aware web application. When a user clicks a button, the client-side of the application gathers GPS data, and then sends that data to a cloud server. In response, the server queries its database for nearby restaurants and returns the results to the client. Finally, the client processes the results and displays them to the user. The end-to-end causal tree consists of the callbacks and causal relationships shown in Figure 1. In the real world, causal trees are often more complex [2, 8, 18].

To capture causal trees, Domino instruments the JavaScript execution contexts that exist on clients and servers. For example, a web browser defines two kinds of contexts: frames and Web Workers [27]. Each execution context is single-threaded and event-driven. Thus, after a context runs its initialization code, the context only executes additional code in response to events like timer expirations and user-driven GUI interactions. To listen for events, a context registers event handlers with the JavaScript runtime; the runtime will fire those handlers when the associated events occur.

As shown in Figure 3, Domino's rewriter injects the Domino JavaScript library into each execution context. The library allows Domino to interpose on the callback registration interfaces within each execution context. Using this interpositioning, Domino can detect the creation of new causal trees, and track causal tree ids across asynchronous events which may span machines.

2.1 Handling Client-side Events

Timers: Interfaces like `setTimeout(cb, delayMs)` and `setInterval(cb, periodMs)` allow an application to provide a callback that the JavaScript runtime will invoke at a preset time in the future. As shown in Figure 2, Domino interposes on timer registration functions using JavaScript's powerful reflection mechanisms. In JavaScript, many of the built-in runtime interfaces are mutable; this means that a JavaScript library which runs before the rest of the application can interpose on runtime interfaces [15], similar to how `LD_PRELOAD` interpositioning works with Linux binaries [6, 7, 28].

Figure 2 shows how Domino uses a *callback stack* to trace causality across multiple events on the same machine. Domino assigns a unique integer id to each top-level event handler h_i. When h_i creates a new event handler h_j, Domino logs the parent-child relationship between i and j (see Line 6 in Figure 2). As discussed in Section 2.3, h_i and h_j may reside on different machines, so Domino's logging statements must identify the host which executed a particular handler.

XMLHttpRequest: Using the `XMLHttpRequest` object, applications can send HTTP requests to remote servers. To send such a request, an application executes code like the following:

```
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "http://foo.com");
    xhr.onreadystatechange = function(){
```

[1] Similar to AppInsight's transactions, Domino's causal trees do not capture *data dependencies*. For example, Domino does not capture the causal dependency between callbacks C_1 and C_2 when C_1 changes a shared state and the change triggers execution of C_2.

```
        //Callback for the arrival of
        //network data.
    };
    xhr.send();
```

The Domino JavaScript library redefines the `XMLHttpRequest` constructor so that the method returns a wrapper object for the newly constructed `XMLHttpRequest` instance. The wrapper allows Domino to interpose on callback registrations for the `onreadystatechange` event; as with `setTimeout()`, Domino uses a stack to associate each `XMLHttpRequest` callback with its parent callback in the causal tree.

`XMLHttpRequests` are typically used for asynchronous network requests, but `XMLHttpRequests` also define a synchronous interface. With the latter execution model, applications do not define `onreadystatechange` callbacks, and `send()` blocks until the network data has arrived. When an `XMLHttpRequest` object is used in synchronous mode, its wrapper does not push entries onto Domino's callback stack.

DOM2 events: JavaScript code uses the Document Object Model (DOM) to collect GUI events [26]. The DOM tree reflects a page's HTML tree into the JavaScript runtime; for each HTML tag, there is a corresponding JavaScript object in the DOM tree. An application attaches event callbacks to DOM objects in order to receive notifications of GUI activity.

The DOM specifies two interfaces by which applications can register callbacks. DOM2 callbacks are registered by invoking the `addEventListener(evtName, cb)` method on the relevant DOM node. For example, to listen for mouse clicks on a <button> HTML tag, an application might execute code like the following:

```
var button = document.getElementById("someId");
button.addEventListener("click", cb);
```

To interpose on DOM2 callback registration, the Domino JavaScript library defines a wrapper for `addEventListener()`.

Understanding how Domino interposes on that method requires an introduction to JavaScript's class system. JavaScript implements classes using prototypes [24] instead of traditional, statically-declared class definitions. In JavaScript, each instance of the class `C` has a special property called `__proto__`; that property points to the "exemplar" or prototype object for instances of `C`. The methods and properties of the prototype object define the default methods and properties for each instance of `C`. To create an inheritance tree, developers set the `__proto__` field of a prototype object reference the prototype object of the parent class.

In a web browser, the DOM elements which export the `addEventListener()` interface are derived from a small number of classes like `Node`. The Domino JavaScript library wraps methods like `Node.prototype.addEventListener()`, allowing Domino to manage the callback stack and record when callbacks fire.

DOM0 events: The DOM defines another mechanism for event registration. In the *DOM0 model*, the callbacks for a DOM element are directly specified in the HTML for that element. Here is an example of such a callback:

```
<button onclick="alert('hi')">Click me</button>
```

In that example, the callback is an anonymous function whose code is `alert('hi')`.

DOM0 registration bypasses the `addEventListener()` interface, and thus bypasses Domino's interpositioning logic for DOM2 event handlers. To detect DOM0 registrations, Domino could rewrite HTML, wrapping DOM0 callback declarations in logging code. However, applications can also declare DOM0 callbacks using JavaScript code. For example:

```
var button = document.getElementById("someId");
button.onclick = function(){alert("hi");};
```

HTML rewriting would not allow Domino to capture these kinds of DOM0 registrations, and browser idiosyncrasies make it difficult for Domino to reliably interpose on DOM0 object properties like `Node.prototype.onclick` [15]. Thus, Domino uses a different approach.

In a JavaScript runtime, the `window` object is the logical root of the DOM tree; the browser passes each new DOM event to `window` before handing the event to the rest of the tree. For each DOM0 event type (e.g., `onclick`, `onkeypress`), Domino's JavaScript library registers a `window`-level callback. Using these handlers, Domino can check whether an event will trigger any DOM0 handlers *before the event actually hits those handlers*. For example, consider this simple HTML:

```
<html>
    <body>
        <button onclick="cb()">
           Click me.
        </button>
        <img src="i.jpg"/>
    </body>
</html>
```

When the user clicks the button, the browser creates a new click event. The browser passes the event to `window`, and then to each DOM node along the path from the <html> tag to the tag that generated the event (in this case, the path will be <html> to <body> to <button>). When the `window` object receives the event, Domino's callback fires. That callback inspects the downstream event path and checks whether any of the tags have DOM0 event handlers. If so, the `window` callback wraps those handlers in logging code before allowing the event to propagate downstream.[2]

The careful reader might wonder why Domino cannot use `window`-level handlers to dynamically wrap DOM2 call-

[2] When an event reaches a DOM node, the browser executes the node's DOM2 handlers before executing any DOM0 handlers. Since Domino's JavaScript library runs before any application-defined code, Domino is guaranteed to register its DOM2 handlers before any application-defined handlers are registered. Thus, Domino cannot miss the registration of `window`-level DOM0 handlers that the application defines.

backs. The reason is that DOM2 callbacks are not enumerable, i.e., given a particular DOM node, Domino cannot retrieve a list of the node's DOM2 handlers. Thus, Domino must interpose on `addEventListener()`, as we described earlier in this section.

postMessage(): A single web page may contain multiple frames or Web Workers [27]. Each frame or Web Worker is a separate JavaScript runtime with a unique `window` object. Different runtimes communicate with each other using `postMessage()`; the recipient defines a callback for the `onmessage` event, and the sender generates a new message by invoking `recipientWindow.postMessage(data)`.

Domino's rewriter injects the Domino JavaScript library into each frame and Web Worker in an application. The library wraps `postMessage()` in code which silently tags each outbound message with a unique integer id. The library also defines its own `onmessage` handler; this handler, which runs before any application-defined handler, logs the message id and the sending frame of a message before passing it to subsequent `onmessage` handlers in the recipient frame.

WinJS: WinJS [16] exposes the Windows Runtime via JavaScript bindings. For example, using WinJS, JavaScript code can access the local file system, and leverage native interfaces for touch-based GUI interaction. WinJS allows developers to write standalone (i.e., non-browser hosted) applications which use HTML, CSS, and JavaScript.

Many of the WinJS APIs are asynchronous and *promise-based*. A promise `p` is an object that represents a potentially uncompleted asynchronous operation. Applications invoke `p.then(cb)` to register a callback which WinJS will fire when the operation completes. For example, WinJS exports a promise-based version of the standard `XMLHttpRequest` interface:

```
var p = WinJS.xhr({url:"http://foo.com"});
p.then(function(result){
    alert(result.responseXML);
});
```

To track causality across promise callbacks, Domino interposes on callback registration via `then()`, wrapping callbacks in code which maintains the callback stack. Logically speaking, this is similar to how Domino interposes on `setTimeout()`. However, `then()` is implemented by the WinJS runtime, not the JavaScript engine, and WinJS prevents JavaScript code from replacing `then()` with wrapper code. So, Domino's rewriter is forced to manually instrument each WinJS call that returns a promise:

```
var p = wrapPromise(WinJS.xhr({url:"http://
    foo.com"}));
```

The `wrapPromise()` allows Domino to define a proxied promise object which maintains the callback stack.

2.2 Handling Server-side Events

Node.js provides a JavaScript environment to the server-side of a web application. Since the server-facing portion of the application lacks a GUI, Node does not expose DOM0 or DOM2 events. However, Node does support timers and Web Worker-like execution contexts called processes. Domino tracks causality across these components using the strategies that are described in Section 2.1.

Node defines a variety of IO mechanisms that are not supported by web browsers. For example, Node allows JavaScript to access databases, open network sockets, and read and write the local file system. To generate such IOs, developers must first use the `require()` statement to include the relevant IO interfaces:

```
var fs = require("fs");
fs.readFile("/etc/hosts", "utf8", cb);
     //Invoke cb() when file data
     //is ready.
```

Domino's server-side JavaScript library interposes on the `require()` statement, wrapping the returned objects in code which maintains the callback stack.

2.3 Tracking Wide-Area Trees

Sections 2.1 and 2.2 describe how Domino manages callback stacks on the client-side and the server-side. To track wide-area causal trees, Domino must link the two stacks. Domino does this by propagating event handler ids across `XMLHttpRequests`. On the client-side, Domino's wrapper for `XMLHttpRequest.send()` silently injects a new HTTP header which contains the id for the handler which invoked `send()`. On the server-side, Domino wraps the Node methods that accept incoming HTTP requests; the wrapped methods extract the handler id in the HTTP header, and push that id onto the server's callback stack.

3. Case Studies

In this section, we describe three case studies which demonstrate how Domino can help developers to find bugs and improve performance.

Bing Maps: Domino's causal trees can span both clients and servers, but in some situations, Domino may lack insight into server-side events. For example, the server-side code might not run atop a JavaScript engine, or the developer may be unable to modify the server-side code to pass through Domino's rewriting engine. In these scenarios, Domino can still uncover interesting behavior in the client-side portion of the spplication. To demonstrate this, we used the Fiddler web proxy [23] to inject Domino.js into the client-side of Bing Maps. Domino discovered several causal trees that were surprisingly deep. Figure 4(bottom) depicts such a tree. The tree corresponds to Bing Maps asynchronously (but sequentially!) downloading eight JavaScript libraries; as shown in the figure, one event handler is fired after each download completes.

Such deep trees often leave network bandwidth underutilized, lengthening page load time [25]. We contacted the

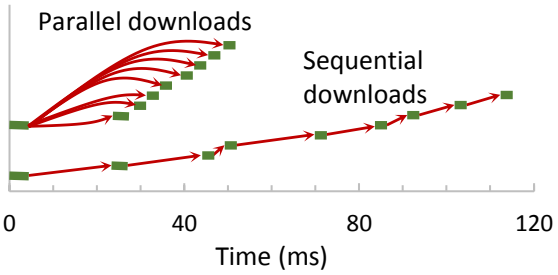


Figure 4. A long, sequential AJAX chain in Bing Maps (bottom), and an optimized, parallel version (top).

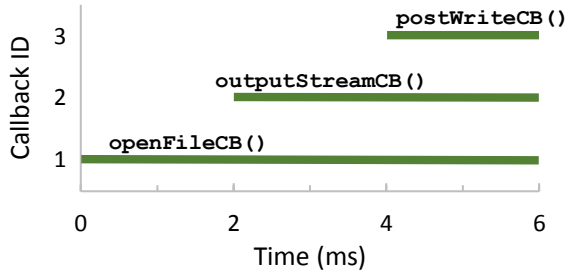


Figure 5. Overlapping events in the same causal tree indicate a potential race condition.

Bing Maps team, who informed us that the sequential downloads were optimistic prefetches that were not on the critical path for the loading of Bing Map's primary, foreground content. Nevertheless, such long, serial chains may lead to incomplete prefetching if a user visits and then quickly navigates away from Bing Maps. Thus, we implemented a new prefetching scheme which aggressively issues parallel downloads if Domino detects that there are no outstanding causal trees that involve network requests (the condition ensures that prefetching does not interfere with other network requests). We omit the implementation and evaluation details due to space constraints, and merely note that the new scheme is 59%–70% faster in a variety of network conditions.

WinJS: According to the JavaScript language specification [9], each JavaScript execution context (§2) is single-threaded, i.e., at any given time, at most one call chain can be active. This frees developers from worrying about race conditions on JavaScript state. However, as new asynchronous constructs are added to the JavaScript language, call chain atomicity may be violated due to implementation errors or developer confusion about how new constructs interact with old ones. Domino can automatically detect these problems by scanning causal trees and finding event handlers whose executions overlap in time.

For example, JavaScript promises [5] are a new mechanism for deferred computation—an application launches an asynchronous method, passing an `onFulfilled` callback which is invoked when the method completes. There are two main specifications for promises: Promises/A [5] and Promises/A+ [17]. Promises/A+ mandates that callbacks *must* be executed asynchronously. However, Promises/A is ambiguous about whether `onFulfilled` can be fired immediately and synchronously, i.e., as soon as the application launches the ostensibly asynchronous method. On a Promises/A system, if (1) `onFulfilled` is invoked synchronously, (2) it modifies state that is accessed by the rest of the call chain, and (3) the developer interpreted Promises/A as disallowing synchronous behavior, then race conditions can arise.

Many JavaScript runtimes implement Promises/A+, but WinJS implements Promises/A, meaning that WinJS may synchronously invoke promise callbacks. As a result, Domino found potential race conditions in several WinJS applications. For example, the Deluxe Mahjong app [11] contains code that is similar to the following:

```
function openFileCb(f){
    f.openOutputStream().then(outputStreamCb);
}
function outputStreamCb(s){
    s.write(data).then(postWriteCb);
}
function postWriteCb(){
    alert("Finished");
}
OpenFileAsync(fName).then(openFileCb);
```

Figure 5 depicts timing information of callbacks of a causal tree that Domino generated during a run of the application. The callback executions overlapped because WinJS executed the callbacks *synchronously*; for example, WinJS invoked `outputStreamCB` at 2 ms, *before* `openFileCB` finished at 6 ms. Such behavior is compliant with the Promises/A specification, but it can lead to subtle bugs if developers assume the presence of traditional JavaScript concurrency semantics.

Server-side scheduling: In our last case study, we highlight the benefit of Domino's wide-area causal trees. For web applications, reducing the average response latency is obviously important. However, reducing *tail latency* is an increasingly critical goal as datacenters strive to meet SLAs and provide interactive services [30, 31]. Prior work has shown that, for a heterogeneous workload, servers can use shortest-job-first (SJF) scheduling to dramatically reduce average response latencies over FIFO [10]. Unfortunately, SJF can result in high tail latencies since it ignores how much time a request has already spent before arriving at the server.

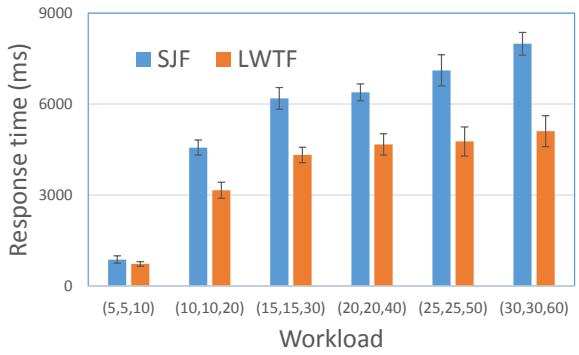
To address this, we created a new server-side scheduling policy, which we call longest-wait-time-first (LWTF). It prioritizes jobs based on the sum of two values. The first is the estimated *processing time* (PT) of the job (note that SJF selects job with the smallest PT). The second is the total *waiting time* (WT) of the job at the client, in the network, and in the server queue. When the server needs to choose a job to run, it picks the job with the largest value of

(PT+WT). In our implementation, PT (for both LWTF and SJF) is estimated by offline profiling, while WT is extracted from Domino's causal trees.[3]

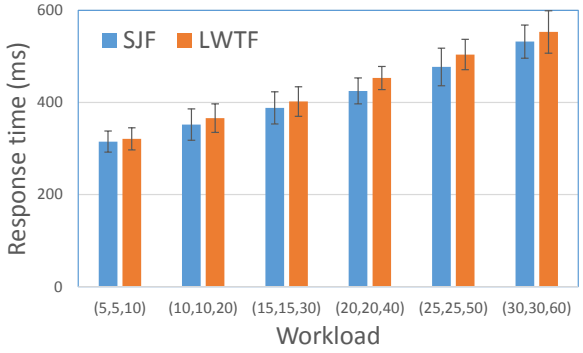
To evaluate the scheduling algorithms, we built a simple web application whose architecture was inspired by that of Happy Pancake,[4] a popular dating website in Sweden. A client-side webpage allows users to register, discover potential dates, and message them. On the server-side, a stateless Node server acts as a front-end, directing user requests to one of three storage back-ends: a MongoDB database which stores user profiles, a RESTful blob store which contains user messages, and an in-memory cache which serves recently accessed messages.

In our scheduling experiments, we examined three types of client requests: (1) user registration, where the front-end stores profile information in MongoDB; (2) user search, where the front-end queries MongoDB to find people matching certain criteria; and (3) messaging, where the front-end accesses the in-memory cache and the blob storage to read or write messages. Message sizes were distributed according to empirical observations of instant messaging traffic [29], with an average message size of 52 bytes. Profile sizes were distributed according to [13], with an average size of 10KB.

As shown in Figure 6, LWTF dramatically reduces tail latencies compared to SJF, while only slightly increasing average latencies. The reductions in tail latency are unlocked by knowledge of Domino's end-to-end causal trees.



(a) LWTF has 17%–37% lower tail (99^{th} percentile) latencies than SJF.



(b) LWTF's average latencies are at worst 6% higher than those of SJF.

Figure 6. SJF scheduling vs LWTF scheduling using various workloads. Each workload is labeled (XX,YY,ZZ), where XX, YY, and ZZ represent the number of simultaneous requests to MongoDB, HTTP blob, and in-memory cache. Each result represents the average of 5 trials; error bars depict standard deviations.

4. Related Work

AppInsight [18] is a profiling tool for mobile applications. Like Domino, AppInsight uses rewriting to interpose on event handling interfaces; AppInsight uses binary rewriting of .NET applications, whereas Domino uses source code rewriting of JavaScript applications. Unlike Domino, AppInsight lacks fine-grained knowledge of server-side events. Timecard [19], an extension of AppInsight, does track events across the client/server boundary. However, Timecard assumes that the server-side portion of a causal tree has a single link. In contrast, Domino allows the server-side causal graph to be a full tree. As a result, Domino can handle more complicated scheduling decisions than Timecard.

Prior to Timecard, a variety of additional systems provided some notion of wide-area event tracing [3, 4, 12, 20–22]. The key difference between those systems and Domino is that Domino tracks client-side events, not just network messages and server-side component flows, because in modern, interactive web applications, client-side events are just as important as server-side activity. Moreover, Domino tracks only JavaScript events, while systems like XTrace [12] and Pinpoint [4] can track low level network events as well. Prior tracing frameworks like Dapper [22], Pinpoint [4], XTrace [12], and Pip [20] may also require developers to expend non-trivial effort to port applications to the tracing framework. In contrast, Domino's rewriting is automatic. Magpie [3] uses detailed knowledge of application semantics provided by the developer to extract causality from system event logs. In contrast, Domino automatically captures causality. WAP5 [21] uses black-box techniques to infer the causal relationship between network messages in a wide-area application. In contrast, Domino can track more fine grained causality.

5. Conclusion

Domino is a tool for analyzing the asynchronous causal trees in modern web applications. Domino automatically rewrites JavaScript code, interposing on the event registration interfaces and tracking causal relationships that may span clients and servers. Domino has found performance and correctness bugs in several real systems. We also used Domino to create a new server-side scheduling algorithm for reducing tail latencies.

[3] Clocks at client and server are synchronized using techniques from Timecard [19].

[4] http://www.happypancake.com/

References

[1] Adobe Systems. PhoneGap. http://phonegap.com/.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DTCP). *ACM SIGCOMM computer communication review*, 41(4):63–74, 2011.

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[4] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of DSN*, 2002.

[5] CommonJS. Promises/A Promise Specification. http://wiki.commonjs.org/wiki/Promises/A.

[6] J. Conrod. Tutorial: Function interposition in linux. http://jayconrod.com/posts/23/tutorial-function-interposition-in-linux, June 2009.

[7] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *SOSP*, pages 388–405. ACM, 2013.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[9] ECMA International. ECMAScript 2015 Language Specification, 6th Edition, June 2015. http://www.ecma-international.org/ecma-262/6.0/index.html.

[10] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of WWW*, pages 276–286. ACM, 2004.

[11] EnsenaSoft. Mahjong Deluxe Free Application. http://apps.microsoft.com/windows/en-us/app/mahjong-deluxe-free/abf22535-69ca-4511-9946-e3a69016cdf3.

[12] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of NSDI*, 2007.

[13] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 295–312. ACM, 2011.

[14] Joyent. Node.js. https://nodejs.org/.

[15] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*, 2010.

[16] Microsoft. WinJS: The Windows Library for JavaScript. https://dev.windows.com/en-us/develop/winjs.

[17] Promises/A+ organization. Promises/A+ Promise Specification. https://promisesaplus.com/.

[18] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of OSDI*, 2012.

[19] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling User-perceived Delays in Server-based Mobile Applications. In *Proceedings of SOSP*, 2013.

[20] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, volume 6, pages 115–128, 2006.

[21] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th international conference on World Wide Web*, pages 347–356. ACM, 2006.

[22] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. *Google research*, 2010.

[23] Telerik. Fiddler Web Proxy. http://www.telerik.com/fiddler.

[24] w3school. http://www.w3schools.com/js/js_object_prototypes.asp.

[25] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of NSDI*, pages 473–485, 2013.

[26] World Wide Web Consortium (W3C). Document Object Model (DOM), . http://www.w3.org/DOM/.

[27] World Wide Web Consortium (W3C). Web Workers, . http://www.w3.org/TR/workers/.

[28] Y. Wu, S. Sathyanarayan, R. H. Yap, and Z. Liang. Codejail: Application-transparent Isolation of Libraries with Tight Program Interactions. In *Proceedings of ESORICS*, pages 859–876. Springer, 2012.

[29] Z. Xiao, L. Guo, and J. Tracey. Understanding instant messaging traffic characteristics. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 51–51. IEEE, 2007.

[30] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of NSDI*, pages 329–341, 2013.

[31] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing The Flow Completion Time Tail in Datacenter Networks. *ACM SIGCOMM Computer Communication Review*, 42(4):139–150, 2012.