usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Proceedings of the 3rd USENIX Conference on Web Application Development

Boston, MA, USA June 13, 2012

conference

proceedings

3rd USENIX Conference on Web Application Development (WebApps '12)

Boston, MA, USA

June 13, 2012

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISBN 978-931971-94-2

USENIX Association

Proceedings of the 3rd USENIX Conference on Web Application Development

June 13, 2012
Boston, MA, USA

Conference Organizers

Program Chair
Michael Maximilien, *IBM Research—Watson*

Program Committee
Patrick Chanezon, *VMware, Inc.*
Christopher Grier, *University of California, Berkeley*
Robert Johnson, *Facebook, Inc.*
Emre Kıcıman, *Microsoft Research*
Raffi Krikorian, *Twitter, Inc.*
James Mickens, *Microsoft Research*
Subbu Subramanian, *Facebook, Inc.*
Samuel Talmadge King, *University of Illinois at Urbana-Champaign*

The USENIX Association Staff

External Reviewers

David Huang	Ajith Ranabahu

WebApps '12: 3rd USENIX Conference on Web Application Development
June 13, 2012
Boston, MA, USA

Message from the WebApps ’12 Program Chair

Welcome to WebApps ’12, the third annual USENIX Conference on Web Application Development. Our continuing emphasis and mission is ensuring that attendees are exposed to the most interesting new work from both industry and academia.

The seven papers and four short papers presented (of twenty-one submissions received) were subjected to the rigorous review standards for which USENIX conferences are known. All papers received at least three reviews and some received more; each paper got a fair and thorough discussion at the in-person program committee meeting in San Francisco.

I’d like to thank the authors for taking the time to submit a paper, whether it was accepted or not. Preparing a paper is a lot of work, and we are still exploring ways to engage more industrial authors and get best-of-class work from both academia and industry. I also thank the program committee for their efforts in reviewing, especially some of the industrial participants, whose schedules can be particularly hectic and who have limited budgeted time for conference PC participation.

Finally, as always, USENIX’s professional organization makes the logistical aspects of running a program committee a breeze, for which I especially thank Anne Dickison, Casey Henderson, and Jane-Ellen Long, along with the rest of the USENIX staff.

We hope you enjoy this year’s program and are inspired by our prestigious keynote speakers. We welcome your continued participation in USENIX and hope you consider submitting your own work to our workshop next year.

Michael Maximilien, ***IBM Research, San Jose, CA***

Modeling and Reasoning about DOM Events

Benjamin S. Lerner   Matthew J. Carroll   Dan P. Kimmel
Hannah Quay-de la Vallee   Shriram Krishnamurthi
Brown University

Abstract

Web applications are fundamentally reactive. Code in a web page runs in reaction to events, which are triggered either by external stimuli or by other events. The DOM, which specifies these behaviors, is therefore central to the behavior of web applications. We define the first formal model of event behavior in the DOM, with high fidelity to the DOM specification. Our model is concise and executable, and can therefore be used for testing and verification. We have applied it in several settings: to establish some intended meta-properties of the DOM, as an oracle for testing the behavior of browsers (where it found real errors), to demonstrate unwanted interactions between extensions and validate corrections to them, and to examine the impact of a web sandbox. The model composes easily with models of other web components, as a step toward full formal modeling of the web.

1 Introduction

Modern web applications are fluid collections of script and markup that respond and adapt to user interaction. Because their programming model differs from classic desktop applications, the *analysis* of such programs is still in its infancy. To date, most efforts have focused on individual portions in isolation: huge progress has been made in clarifying the semantics of JavaScript [10, 16, 17], in modeling the tree structure of HTML [9], and in understanding the overall behavior of the browser as a runtime environment [2, 5, 14, 15, 18]. But each of these approaches ignores the crucial element of reactivity: web programming is fundamentally *event-driven*, and employs a powerful mechanism for event propagation. Perhaps counterintuitively, the JavaScript loaded in web applications is largely *inert*, and only executes when triggered by events dispatching through the HTML structure in which it resides. To paraphrase John Wheeler's famous dictum, "HTML tells events how to propagate, and events tell HTML how to evolve."

The ability to model web applications more accurately has widespread appeal. Webapps are large codebases in languages with (currently) poor support for modularity: how can we assure ourselves that a program doesn't exhibit unintended behaviors? Many webapps include semitrusted or untrusted content such as ads: how can we ensure that a program is robust in the face of the injected content's activity? And for many web-like applications, foremost among them Firefox or Thunderbird, users avidly install extensions that deliberately and deeply modify the markup and script of the underlying program: what assurance do we have that the composite program will work correctly? Even current tools that do attempt to model both the page structure and the code [3, 4, 6] are hampered by state-space explosion, as without a precise model the potential code paths grow beyond feasibility.

Instead, we propose a simple, executable, testable model of event dispatch in web applications, in the style of λ_{JS} [10, 11, 17]. Our model is engineered to hew closely to the structure of the spec [13], to build confidence in the model's adequacy. For our purposes we abstract JavaScript and model only those APIs dealing with page structure or events; the model is easily extended to include λ_{JS} directly. Likewise we represent the page structure as a simple tree in a heap; again the model can be extended with a richer tree representation [9] for further precision.

Contributions

This paper makes the following concrete contributions:

1. A short, executable, and testable model of event dispatch (Section 4.2). Writing such a model clarifies potential sources of confusion in the spec itself, provides an oracle against which implementations can be tested, and provides a foundation for future program analyses. As a case in point, systematically testing small examples in our model revealed discrepant behavior among the major browsers.

2. Simple proofs (Section 4.1) that the model upholds properties expected of the spec, such as precisely how and when a script's side effects can affect the dispatching of current and subsequent events. Because the model closely resembles the spec, such proofs lend confidence that the spec itself enjoys the same properties; thus far such claims were merely the *intent* of the lengthy, prose spec.

It also presents two initial applications of the model:

1. We examine two Thunderbird extensions to detect a real conflict between them. The model is then used to show that the fix (as implemented by one extension author) currently suffices to correct the bug, that another, simpler fix should be more robust, and this simpler fix in turn reveals a bug in Gecko (Section 4.3).
2. We re-examine the assumptions of ADsafe [1] in light of event dispatch, to determine whether ADsafe widgets may affect the control flow of their host despite the ADsafe sandbox, and suggest directions for more robust widgets (Section 4.4).

2 Web Program Control Flow Unpacked

An intuitive but incomplete model for programming web pages is that of an asynchronous event loop. In this model, events are triggered by user interaction, and event callbacks have access to an object graph representing the tree structure of the HTML, known as the Document Object Model (DOM). The full definition of "the DOM" is, however, spread over many specifications [12, 13, 19, 21, among others], comprising far more than just this tree structure. In reality, the DOM object graph is more interconnected than a mere tree and can be arbitrarily entangled with the JavaScript heap; event callbacks are attached directly to these DOM nodes; and while the event loop itself is not available as a first-class entity through the DOM, nodes may support APIs that implicitly cause further events to be dispatched or that modify the document structure.

In short, it is naïve to think of the execution of a web program as merely an event loop alongside a tree-structured data store. Rather, the structure of the document influences the propagation of events, and the side effects of events can modify the document. Understanding web program behavior therefore requires modeling all the subtleties of event dispatch through the DOM. Like all portions of web-related programming, the event mechanisms were developed over time, resulting in historical quirks and oddities. We explain the main features of event dispatch in this section, and enunciate design goals for our model to support, then develop our model of it in the following section.

2.1 Event Dispatch in N Easy Stages

Static document structure, one event listener We take as a running example a simple document fragment of three nodes: $\langle$**div**$\rangle\langle$**p**$\rangle\langle$**span**/$\rangle\langle$/**p**$\rangle\langle$/**div**$\rangle$. In the simplest case, suppose as the page loads we attach a single event listener to the $\langle$**span**/$\rangle$:

```
spanNode.addEventListener("click",
  function(event) { alert("In click"); });
```

This statement registers the function as a *listener* for mouse "click" events only; any other event types are ignored. When an event is *dispatched* to a particular *target*, the listener on that target for that event type—if there is one—is invoked. Thus a "click" event targeted at the $\langle$**span**/$\rangle$ will yield the alert; a "keypress" event will not, nor will a "click" event targeted at the $\langle$**p**/$\rangle$ node.

Note that scripts can construct new event objects programmatically and dispatch them to target nodes. These events behave identically to browser-generated events, with one caveat addressed later.

Design Goal 1': Every node has a map of installed listeners, keyed by event type. *(To be refined)*

Multiple listeners and the propagation path We now expand the above model in two key ways. First, the suggestively named `addEventListener` API can in fact be used repeatedly, for the same node and the same event type, to add *multiple* listeners for an event. These listeners will be called in the order they were installed whenever their triggering event is dispatched. This flexibility allows for cleaner program structure: clicking on a form button, say, might trigger both the display of new form fields and the validation of existing ones; these disparate pieces of functionality can now be in separate listeners rather than one monolithic one.

Second, web programs frequently may respond to events on several elements in the same way. One approach would be to install the same function as a listener on each such element, but this is brittle if the page structure is later changed. Instead, a more robust approach would install the listener once on the nearest common ancestor of all the intended targets. To achieve this, event dispatch will call listeners on *each ancestor of the target node* as well, known as the *propagation path*. Thus adding a listener to the other two nodes in our example:

```
function listener(event) {
  alert("At " + event.currentTarget.nodeName
        + " with actual target "
        + event.target.nodeName);
}
pNode.addEventListener("click", listener);
divNode.addEventListener("click", listener);
```

and then clicking in the $\langle$**span**/$\rangle$ will trigger *three* alerts: "In click", "At p with actual target span", and "At div with actual target span" in that order: the event *bubbles* from the target node through its ancestors to the root of the document.[1]

For symmetry, programs may want to perform some generic response *before* the event reaches the target node, rather than only after. Accordingly, event dispatch in fact defines a so-called *capturing* phase, where listeners are called starting at the root and propagating down to the target node. To install a capture-phase listener, `addEventListener` takes a third, boolean `useCapture` parameter: when true, the listener is for capturing; when missing or false, the listener is for bubbling.

Event dispatch therefore comprises three phases: "capture", from root to the target's parent and running only capture-phase listeners; "target", at the target node and running all listeners; and "bubble", from the target's parent to the root and running only bubble-phase listeners. The `event` parameter to each listener contains three fields indicating the current `eventPhase`, the `currentTarget`, and the intended `target` of the event. For our running example, an event targeted at the $\langle$**span**/$\rangle$ will call listeners

1. On $\langle$**div**/$\rangle$ for phase `capture`, then
2. On $\langle$**p**/$\rangle$ for phase `capture`, then
3. On $\langle$**span**/$\rangle$ for phase `target`, then
4. On $\langle$**p**/$\rangle$ for phase `bubble`, then
5. On $\langle$**div**/$\rangle$ for phase `bubble`.

Design Goal 1": Every node has a map of installed listeners, keyed by event type and phase. *(To be refined)*

Design Goal 2: Dispatch takes as input a node and its ancestor chain, which it will traverse twice.

Aborting event propagation It may be the case that a capture- or target-phase listener completely handles an event, and that the app has no need to propagate the event further. The app could maintain some global flag and have each listener check it and abort accordingly, but this is tedious and error-prone. Instead, the `event` object can be used to stop event propagation in two ways:

- `event.stopPropagation()` tells dispatch to terminate as soon as all listeners on the current node complete, regardless of whether listeners are installed on future nodes of the propagation path. Thus calling this in a target-phase listener on $\langle$**span**/$\rangle$ will abort dispatch between steps 3 and 4 above.
- `event.stopImmediatePropagation()` tells dispatch to terminate as soon as the current listener returns, regardless of whether other listeners are installed on this or future nodes in the propagation path. Thus calling this in a capture-phase listener on $\langle$**p**/$\rangle$ will abort dispatch in the middle of step 2, even if there are more capture-phase listeners on $\langle$**p**/$\rangle$.

Design Goal 3: Dispatch can be aborted early.

Dynamic document structure: no effect! So far our example listeners have had no side effects; in general, however, they often do. This may interact oddly with the informal definitions above: for instance, if a target-phase listener removes the target node from the document, what should the propagation path be? Several options are possible; the currently specified behavior is that the propagation path is *fixed* at the beginning of dispatch, and is unmodified by changes in document structure. Thus in our running example, regardless of whether nodes are deleted, re-parented or otherwise modified, the five steps listed are unaffected.

Design Goal 4: The ancestor chain input to Design Goal 2 is immutable.

Dynamic listeners: some effect! We can now address the last oversimplification, that event listeners are added once and for all at the start of the program. In fact they can be added and removed dynamically (using the analogous `removeEventListener` API) throughout the program's execution. For example, a common idiom is the "run-once" listener that removes itself the first time it runs:

```
function runOnce(event) {
  node.removeEventListener("click", runOnce);
  ...
}
node.addEventListener("click", runOnce);
```

Such actions have a limited effect on the current dispatch: listeners added to (resp. removed from) a *future* node in the propagation path will (resp. will not) be called by the dispatch algorithm; listeners added to (resp. removed from) the *current* or *past* nodes in the propagation path will be ignored (resp. will still be called). More intuitively, a refinement of the five steps above says that dispatching an event to $\langle$**span**/$\rangle$ will:

1'. Determine the capture-phase listeners on $\langle$**div**/$\rangle$ and run them, then

2'. Determine the capture-phase listeners on $\langle$**p**/$\rangle$ and run them, then

3'. Determine the target-phase (i.e., all) listeners on $\langle$**span**/$\rangle$ and run them, then

[1]Additionally, for legacy reasons it also propagates to the global `window` object; this detail does not substantially change any of our subsequent modeling.

4'. Determine the bubble-phase listeners on ⟨**p**/⟩ and run them, then

5'. Determine the bubble-phase listeners on ⟨**div**/⟩ and run them.

Since the determination of the relevant listeners is lazily computed in each step, dispatch will only notice added or removed listeners that apply to later steps.

Design Goal 5: The listener map is mutable during dispatch, but an immutable copy is made as each node is reached.

Dealing with legacy "handlers" Unfortunately, the mechanism explained so far—multiple listeners, capturing and bubbling, and cancellation—was not the first model proposed. Originally, authors could write ⟨**span** onclick="`alert("In onclick");`"/⟩, and define an event *handler* for the "click" event. There can be at most one handler for a given event on a given node, which takes the form of a bare JavaScript statement.

To incorporate this legacy handler mechanism into the listener model above, handlers are implicitly wrapped in `function(event) { ... }`[2] and their return values are post-processed to accommodate the ad-hoc nature of legacy handler support. Handlers can be altered by modifying the `onclick` content attribute or by modifying the `onclick` property of the node:

```
node.setAttribute("onclick",
         "alert('New handler');");
node.onclick =
  function(event) { alert("New handler"); }
```

and for legacy compatibility, these mutations must not affect the relative execution order of the handler and any other "click" listeners.

Design Goal 1: Every node has a map of installed listeners and handlers, keyed by event type and phase.

Default actions Finally, browsers implement a great deal of functionality in response to events: clicking a link will navigate the page, typing into a text box will modify its contents, selecting one radio button will deselect the others, and so forth. Such *default actions* behave mostly like implicitly-installed listeners, with a few caveats. Default actions are *not* prevented by `stopPropagation` or `stopImmediatePropagation`; instead, listeners must call `preventDefault`. Legacy handlers can return `true` (or sometimes `false`) to achieve the same effect. Also, default actions are *not* run for programmatically constructed events; these events are considered "untrusted" and cannot be used to forge user interaction with the browser.

[2]The expert reader will note that some contortions are needed to supply the right `this` object and scope to the handler.

The default action for many events is in fact to trigger the dispatch of a new event: for example, the default action of a "keydown" event will dispatch a new "keypress" event; likewise, the default action for "mouseup" is to dispatch a "click" event and possibly a "doubleclick" event. Note that these are new dispatches; any and all changes to the document structure made by script-installed listeners will be visible in the propagation path of these new events.

Design Goal 6: Events are equipped with a default action which is the final handler of the dispatch.

2.2 Challenges

Analyzing the full control-flow of an application is difficult enough even in ideal settings when only one developer writes the complete program. Still, a whole-program analysis is possible in principle, since the entirety of the codebase is available for inspection. On the web, however, programmers frequently include code they did not author. We consider two scenarios: the intentional inclusion of third-party code such as ads, and the unforeseeable injection of user-installed extensions.

2.2.1 Invited third-party code

A typical webapp may include ads sourced from various third parties, a Twitter or blog feed, social network sharing operations, and so on. These all take the form of some user-visible UI, and nearly always include additional scripts to make the UI interactive. But such inclusion can have several unpleasant side-effects. The obvious security consequence, in the worst case when the webapp takes no precautions, is that the inserted content runs in the same JavaScript context as the webapp, with the same permissions, and can inadvertently or maliciously break the webapp. Fortunately, several frameworks exist to mitigate such naïve mistakes: tools like ADsafe [1] or Caja [20] attempt to sandbox the inserted content, isolating it within a subtree of the surrounding document and within a restricted JavaScript environment. But these also have weaknesses in the face of DOM events, as we discuss in Section 4.4.

2.2.2 Uninvited third-party code

Virtually every major browser now permits the installation of extensions. These are specifically intended for users to modify individual webapps or the browser itself. For example, there are app- or site-specific extensions that, say, supplant existing webmail auto-completion methods, or replace ads with contacts, or customize the UI of a particular newspaper or social networking site. While these extensions are sometimes written by the creators of the original application or site, in other cases they

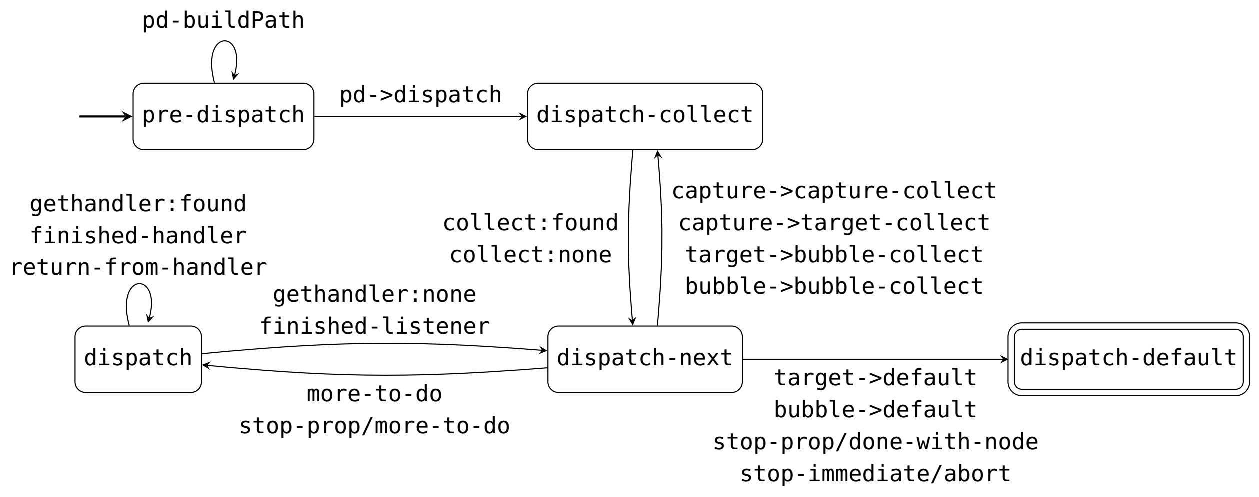


Figure 1: The core reduction steps in our model, implementing the event dispatch state machine.

are written by third parties. Other browser extensions personalize the browser's whole look-and-feel. All these extensions can be highly invasive to apps, and there is no way for app authors to anticipate these modifications. Instead, they must code defensively in *all* event listeners, for which they need a model of what to defend against.

3 Modeling DOM Events

Having informally laid out how event dispatching works, we are ready to model it precisely. We will first describe the model itself, then explain how we account for its relationship to the actual DOM specification. Design goals 1, 3, and 6 are used to construct the model; the other three express properties about that model that we prove in Section 4.1. Section 5 presents extensions to the model.

3.1 Model Highlights

Because the DOM is essentially a large automaton that determines what operations will execute next, we model it using an operational semantics. In particular, because of the ability to abort dispatch in subtle ways (see Design Goal 3), we find it most effective to use the evaluation context style of Felleisen and Hieb [8], which was initially designed to model control operators (such as exceptions and continuations) in regular programming languages and is thus well suited for that purpose.

Our full model, which can be found at `http://www.cs.brown.edu/research/plt/dl/domsemantics/`, is 1200 lines of commented code. It is implemented using the PLT Redex modeling language [7], which provides programming environment support for models in the Felleisen-Hieb style. Here we present the highlights that will help the reader navigate that document.

3.1.1 Stages of a Dispatch

The Events spec defines the procedure for synchronously dispatching a single event in careful detail, and the prose is full of challenging nuances. Conceptually, however, the spec defines a single event dispatch as an automaton with five states. The states and their transitions, as named in our model, are shown in Fig. 1; we discuss the key transitions below. Our model identifies eight transitions, with eighteen triggering conditions: a reasonable size, given the many interacting features of event dispatch, and certainly more concise than the original spec.

1. Determining the propagation path. Event dispatch begins by determining the propagation path for the event: the ancestors of the target node at the time dispatch is initiated. Our model builds this path in the `pre-dispatch` state. The spec states that "once determined, the propagation path must not be changed," regardless of any page mutations caused by listeners that are triggered during dispatch *(Design Goal 1).* This is trivially maintained by our model: every transition between the `dispatch-next`, `dispatch-collect` and `dispatch` states (described below) preserve the path without modification.

2. Determining the next listener. The flow of an event dispatch may be truncated in one of three ways: after the completion of the current listener, after the completion of any remaining listeners on the current node and phase, or the default action may be canceled. Further, some events may skip the bubble phase entirely. When any given listener completes execution, the dispatch algorithm must check whether any of these truncations have been signaled, and abort dispatch accordingly *(Design Goal 3).* If none have, then dispatch proceeds to the next listener

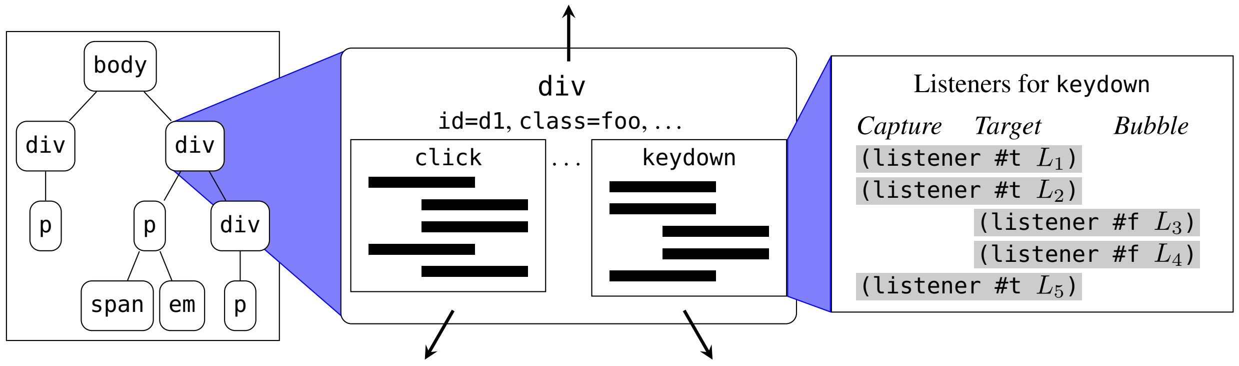


Figure 2: Schematic representation of the model's store. The store contains nodes connected in a tree structure (left). Each node stores its name, attributes, pointers to its parent and children, and suites of event listeners, grouped by event type (middle). Each suite contains three lists, one for each phase of dispatch; listeners for either $\langle capture\rangle$ or $\langle bubble\rangle$ phases also apply—in order—to the $\langle target\rangle$ phase (right). Each listener holds a pointer (L_i) to its actual code.

for the current node and phase or, if no such listener exists, begins collecting listeners for the next node (and possibly phase) on the propagation path. Precisely identifying these conditions is the crux of our model, which reifies them as ten transitions out of the `dispatch-next` state.

3. Determining listeners for the current node and phase. Perhaps one of the subtlest requirements of the spec determines which listeners must be called when dispatch reaches a node on a propagation path—and not all browsers currently get this right (see Section 4.2). As noted in Section 2.1, the list of listeners for a given node and phase is fixed only when dispatch reaches that node; this step is accomplished by the `dispatch-collect` model state *(Design Goal 5)*. Unfortunately here the spec conflates specification and representation: it implicitly assumes a flat list of the installed event listeners, and must include qualifiers to predicate which listeners should run. Our model avoids making assumptions about representation, using a structure that exposes the semantic intent (see Section 3.1.2 below), and merely copies the relevant list of installed listeners for the current phase into the dispatch context, thereby insuring that any changes to the installed listeners on the current node and phase *will not take effect* until a subsequent dispatch. (Still, any modifications to listeners installed on nodes for phases later in the current dispatch *will* be visible.) Accordingly, the model's transitions here are far clearer than the spec they implement.

4. Executing a listener. Either transition from `dispatch-next` to the `dispatch` state determines that a given listener should be executed. The model then records both the current listener and the remaining target nodes, and begins executing the listener body. While in this state, listeners may invoke additional, reentrant ("synchronous") event dispatches, may cancel the current event dispatch, or generally may modify the DOM however they choose. Once a `dispatch` context completes its listener body, it transitions back to `dispatch-next` to determine the next listener to call.

5. Default actions. When `dispatch-next` reaches the end of the propagation path, or when the bubble phase would begin but the current event does not bubble, the algorithm must execute the *default actions*, if any, for the given event and event target. We model this with a `dispatch-default` state *(Design Goal 6)* and a meta-function (not shown) to compute the relevant default actions. This meta-function is the only portion of the dispatch algorithm that inspects the detailed form of the event and target; everything else is agnostic. A model of the full HTML spec would supply this meta-function, specializing the event dispatch mechanism to the events applicable to HTML documents.

3.1.2 Representing Event Listener Lists

The precise storage for event listeners encodes several requirements culled from disparate portions of the spec, and embodies Design Goal 1. We give the precise type in Fig. 3b, and explain its design in four stages.

First, the spec mandates that event listeners installed for the same node, event type and dispatch phase must be called in the order they were installed. Accordingly, every node contains a map (LS) from event type and phase to a vector of listeners.

Second, the spec elsewhere states listeners may be installed for either $\langle capture\rangle$ and $\langle target\rangle$ phases, or $\langle target\rangle$ and $\langle bubble\rangle$ phases. At first glance, it seems that we might simply maintain separate lists for $\langle capture \rangle$- and $\langle bubble \rangle$-phase listeners, but that runs afoul of the ordering requirement when dispatch reaches the $\langle target \rangle$ phase. Instead, we must also maintain a list of $\langle target \rangle$-phase listeners, and when adding a new listener, we must update two lists in our map: this is accomplished by the `addListener` meta-function.

Third, the spec requires that the triple of arguments $(\langle eventType \rangle, \langle usesCapture \rangle, \langle listener \rangle)$ must be unique within each node: while a given function may be installed both as a capture-phase listener and as a bubble-phase listener on the same node, subsequent installations will have no effect. In combination with the previous requirement, one implicit consequence is that a function may be called *twice* during the `target` phase; though true, this is not immediately obvious from the spec wording, but is evident from our rules.

Finally, the spec defines how event listeners may be removed from a node: again, from both capture and target phases, or from both target and bubble phases. Thanks to the uniqueness requirement and our modeling of `addEventListener`, we know that a given listener may be present twice in the target-phase list, so we must record *which* target-phase listeners were also installed on the capture phase, and which were not, or else we might remove the wrong listener and violate the ordering requirement. Consider the following program fragment:

```
node.addEventListener("click", true, f1);
node.addEventListener("click", true, f2);
node.addEventListener("click", false, f1);
node.removeEventListener("click", true, f1);
```

While it is intuitively clear that the call to `removeEventListener` must remove `f1` in the `capture` phase, it must also remove the *corresponding* `f1` in the target phase, i.e., the first one.

Remark: In prior work [15], in which the first author implemented the event dispatch algorithm, he read the documentation for `addEventListener` too quickly; it is excerpted in Fig. 3a. Note the emphasized text: in fact, the specification is inconsistent in defining on which phases listeners may be installed! By contrast, the meta-function in Fig. 3b uses the `useCapture` flag exactly once, and hence avoids and resolves this error.

The store as described here is redundant: the $\langle target \rangle$ list by itself contains sufficient information to produce the spec-defined behavior. However, this redundancy is intentional: it simplifies the determination of relevant listeners (the `dispatch-collect` state earlier), emphasizes the "doubled" effect of `addEventListener`, and indirectly encourages implementers to treat the model as a specification rather than an implementation guide.

3.2 Modeling Challenge: Adequacy

Whenever researchers build a model of a system, they must demonstrate why the model adequately represents the system being examined, or else the model is of no relevance. This is an inherently informal process, as the system here is the prose of the specification; if the spec were amenable to formal methods in the first place, there would be no need for a formal model!

To simplify the case for our model's adequacy, we have annotated each paragraph of the spec with a link to the relevant definitions and reduction rules of our model. A reasonably knowledgeable reader could flip back and forth between the spec and the model, and convince herself that the model faithfully represents the intent of the spec. An excerpt of this is shown in Fig. 3, where we show the spec's definition for `addEventListener` and the corresponding Redex metafunction that installs the listener into our model.[3]

Of course, the DOM also lives through many implementations. We can therefore test our model to determine whether it conforms to the behavior of actual implementations. We have begun doing so, and discuss the results in Section 4.2. Ultimately, we have to choose between modeling a specific browser or the spec; we have chosen the latter, but the decisions we have made are localized and can thus be altered to reflect one particular implementation, if desired.

4 Applications

We now demonstrate the utility of our model by discussing its application in various settings.

4.1 Provable Properties of the Model

We concern ourselves here only with *well-formed* states of the model: a finite statement/heap pair (S, H) is well-formed if

1. There are no dangling pointers from S into H.
2. The heap is well-typed: Every heap location mentioned in S is used consistently either as a node or as a listener.
3. There are no dangling pointers within H: the parents and children of every node n must be present in H.
4. The nodes in the heap are tree-structured: No node is its own ancestor, descendant or the child of two distinct nodes.
5. For every listener (`listener` b L) or handler (`handler` L), L points to a statement S' and (S', H) is well-formed.

[3]The spec requirement that `addEventListener` be idempotent is in fact defined elsewhere; that text in turn corresponds to the (elided) `addListenerHelper` metafunction.

addEventListener

Registers an event listener, depending on the `useCapture` parameter, *on the capture phase of the DOM event flow or its target and bubbling phases.*

Parameters:

type of type `DOMString`: Specifies the `Event.type` associated with the event for which the user is registering.

listener of type `EventListener`: ...

useCapture of type `boolean`: If true, `useCapture` indicates that the user wishes to add the event listener *for the capture and target phases only*, i.e., this event listener will not be triggered during the bubbling phase. If false, the event listener must *only be triggered during the target and bubbling phases.*

(a) Excerpt from the specification of `addEventListener`; emphasis added to highlight self-inconsistencies.

$$\begin{array}{rrcl}
P \in & \textsc{Phase} & ::= & \langle \mathit{capture} \rangle \mid \langle \mathit{target} \rangle \mid \langle \mathit{bubble} \rangle \\
L \in & \textsc{Listener} & ::= & \texttt{listener}\ S \\
S \in & \textsc{Stmt} & ::= & \texttt{skip} \mid \texttt{return}\ \mathit{bool} \mid S; S \\
 & & \mid & \texttt{stop-prop} \mid \texttt{stop-immediate} \\
 & & \mid & \texttt{prevent-default} \\
 & & \mid & \texttt{addEventListener}\ N\ T\ \mathit{bool}\ L \\
 & & \mid & \texttt{remEventListener}\ N\ T\ \mathit{bool}\ L \\
 & & \mid & \texttt{debug-print}\mathit{string} \\
T \in & \textsc{EvtType} & ::= & \texttt{"click"} \mid \texttt{"keydown"} \mid \cdots \\
LS \in & \textsc{LMap} & ::= & (T \times P) \rightharpoonup (\overrightarrow{\mathit{bool} \times L}) \\
N \in & \textsc{Node} & ::= & (\texttt{node}\ \mathit{name}\ LS\ \ldots)
\end{array}$$

```
(define-metafunction DOM
  [(addListener
      LS string_type bool_useCapture L)
   (addListenerHelper
     (addListenerHelper
       LS string_type target L)
     string_type
     ,(if (term bool_useCapture)
        (term capture)
        (term bubble))
     L)])
```

(b) Excerpt from our Redex model of `addEventListener`. Note that the impact of the `useCapture` is defined exactly once, leaving no room for self-inconsistency.

Figure 3: Defining and modeling `addEventListener`

We can now prove that our model upholds the invariants stated in our Design Goals:

Theorem 1. *Once computed, the event propagation path is fixed for each dispatch.* (Design Goal 4)

Proof sketch. By inspection of the reduction rules: every rule between `dispatch-next`, `dispatch-collect` and `dispatch` leaves the path unchanged. Rules leading to `dispatch-default` vacuously leave the path unchanged, since dispatch has passed the end of the path. The remaining rules in `pre-dispatch` compute the path itself. □

Theorem 2. *During dispatch, once the event listener list for a given node and phase is computed, it is unaffected by calls to `addEventListener` or `removeEventListener` in any invoked listeners.* (Design Goal 5)

Proof sketch. This is the express purpose of `dispatch-collect`: only it examines the store to collect the currently installed listeners, and copies that list into the `dispatch-next` context. All further reductions from `dispatch-next` use the copy, and are unaware of any changes in the store. □

Theorem 3. *Event dispatch is deterministic.*

Proof sketch. By inspection of the reduction rules: the left hand sides of the rules never overlap. □

Additionally, we can prove several other key properties:

Lemma 1. *Preservation of well-formedness: given a well-formed (S, H) such that $(S, H) \to (S', H')$, (S', H') is well-formed.*

Lemma 2. *Progress: a well-formed term is either $(\texttt{skip}, H)$ or it can take a step via $(\to)$.*

Theorem 4. *Termination: assuming all listeners and handlers terminate, and do not recursively dispatch events, every well-formed event dispatch completes: $((\texttt{pre-dispatch}\ loc_n\ \texttt{()}\ e), H) \to^* (\texttt{skip}, H')$.*

Proof sketch. The propagation path of any node in a well-formed state is finite, because the heap is finite and tree-structured. Every transition from `dispatch-next` either reduces the number of remaining listeners on the current node, or the number of remaining nodes in the propagation path. Every transition out of `dispatch-collect` returns to `dispatch-next` in a single step. By assumption, every listener or handler terminates (in our minimal language this is trivial; in full JavaScript it is not), so every transition to `dispatch` will return to `dispatch-next` in finite time. Finally, again by assumption, the default handlers run by `dispatch-default` terminate. □

Theorem 5. *Double dispatch* (Design Goal 2)*: for every node on a well-formed propagation path that is not the target node, if dispatch is not stopped then that node will be visited exactly twice, once for capture and once for bubbling phases.*

Proof sketch. Because the heap is tree-structured, and by construction, every node in the propagation path is distinct. By construction, `pd->dispatch` collects listeners for the root node; all subsequent transitions from `dispatch-next` to `dispatch-collect` collect listeners for the remaining nodes on the path except the target (`capture->capture`), then the target (`capture->target`), then the path is traversed backward (`target->bubble` and `bubble->bubble`). By immutability of the path, every node visited in capture phase is therefore visited again in bubble phase. □

These properties are intuitively expected by the authors of the dispatch spec, and formally hold of our model; the adequacy of our model (Section 3.2) implies that these properties do hold of the spec itself.

4.2 Finding Real-World Inconsistencies

Our proof of determinism, combined with the model's adequacy, is tantamount to stating that the spec is unambiguous. We have not encountered any obvious ambiguities; our reading of the spec assigned to every claim a specific interpretation (see Section 3.2). But we nevertheless observe differing behavior on real browsers.

Erroneous Treatment of `removeEventListener` We can use our model to randomly construct test cases, or systematically generate a suite of related test cases, observe their behavior in the model, then translate the tests to HTML and JavaScript and test existing browsers to see if they match our expectations. We have been doing so, and continue to test ever larger instances of the model against browsers.

During testing, we have already found several divergences between our model and real browsers, which further revealed inconsistencies between major browsers themselves. The simplest example appears in a systematic suite checking the handling of `removeEventListener`. These tests all use the same $\langle$**div**$\rangle\langle$**p**$\rangle\langle$**span**$/\rangle\langle/$**p**$\rangle\langle/$**div**$\rangle$ document, and install the following listeners:

```
var targetNode, targetCapture;
var triggerNode, triggerCapture;
function g(event) { alert("In g"); }
function f(event) {
  targetNode.removeEventListener("click",
     g, targetCapture);
}
triggerNode.addEventListener("click", f,
                             triggerCapture);
targetNode.addEventListener("click", g,
                            targetCapture);
```

In words, this installs listener `f` on `triggerNode`, either for capture (`triggerCapture` $=$ *true*) or not, that will then remove `g`. It then installs listener `g` on `targetNode` for capture or not (`targetCapture`). A systematic search of all possible values of these four variables reveals that when `targetNode` $=$ `triggerNode` $\neq$ $\langle$**span**$/\rangle$ and `targetCapture` $=$ `triggerCapture`, browser behavior differs. Chrome (v15 and v16), Safari (v5.0.1) and Firefox (v3.6 through v8) will *not* execute `g`, while Internet Explorer (v9) and Opera (v11) *will*.

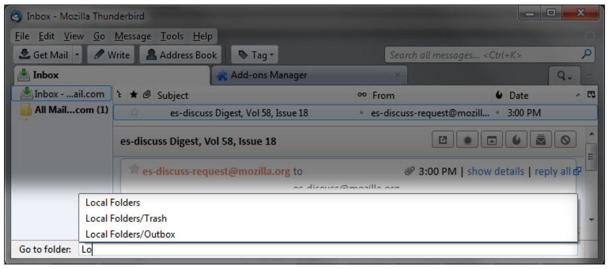
Our model predicts the latter behavior is correct, but in truth one of three situations may hold: IE, Opera and our model may be right, or Chrome, Safari and Firefox may be right, or all six may be wrong. Regardless, our model simplifies making testable predictions about browsers.

Treatment of Legacy Handlers Another example probes the corner cases surrounding setting and clearing legacy handlers. Within a few minutes of generating tests, we found examples where our model disagrees with IE, Chrome and Firefox—and the browsers all disagree with each other, too. Here, the events spec delegates responsibility to the HTML 5 spec itself, which defines how the setting and clearing of handlers interacts with existing listeners. Browsers, however, appear to have implemented variations on the specified behavior. We have identified two variations each for setting and clearing handlers; our model can accommodate them by changing the setting or clearing rule, without needing changes anywhere else. Further testing is needed to decide which of these variations, if any, corresponds to each browser's behavior. More broadly, by continuing to run such tests, we hope to build greater confidence in the quality of the model (and, perhaps, improve the uniformity of browsers, too).

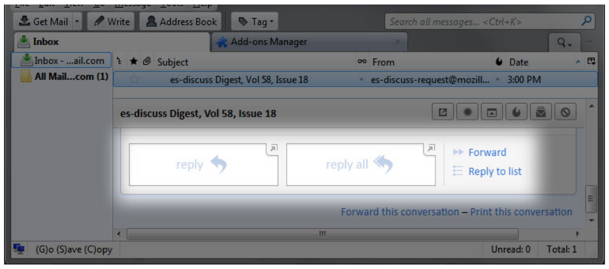
4.3 Detecting Real Extension Conflicts

One of the authors routinely uses extensions to customize Thunderbird. One such extension is Nostalgy[4], which provides several convenient hot keys for archiving messages and navigating among folders. For example, pressing 'S'

[4] `http://http://code.google.com/p/nostalgy/`


Inbox - Mozilla Thunderbird
File Edit View Go Message Tools Help
Get Mail
Write
Address Book
Tag
Search all messages... <Ctrl+K>
Inbox
Add-ons Manager
Subject
From
Date
es-discuss Digest, Vol 58, Issue 18
es-discuss-request@mozilla.org
3:00 PM
show details
reply all
Local Folders
Local Folders/Trash
Local Folders/Outbox
Go to folder: Lo

(a) Nostalgy's main interface: a folder selector in the status bar



(b) Thunderbird Conversation's main interface: text boxes in the conversation view for quick replies

Figure 4: Screenshots of the conflicting extensions' UIs.

will save the current message. This is achieved by two event listeners on the Thunderbird global `window` object:

```
function onNostalgyKeyPressCapture(event) {
  // handle ESC key and cancellation
}
function onNostalgyKeyPress(event) {
  // show folder selector and handle commands
}
window.addEventListener("keypress",
          onNostalgyKeyPress, false);
window.addEventListener("keypress",
          onNostalgyKeyPressCapture, true);
```

Implicit in this code is the assumption that all key presses are intended to control Thunderbird, and not, say, to input text. However, another extension, Thunderbird Conversations[5], redefines the email preview pane to show a Gmail-like conversation view, complete with "quick reply" boxes where the user can compose a response without leaving the main window. This functionality is implemented by the default actions of the quick-reply ⟨**textarea**/⟩ tags, along with a bubble-phase listener on their grandparent ⟨**div**/⟩:

```
quickReplyDiv.addEventListener("keydown",
  function convKeyDown(event) {
    // ENTER=>send message, ESC=>cancel
    event.stopPropagation();
  });
```

[5]`https://github.com/protz/GMail-Conversation-View/`

At first glance, nothing in this code appears problematic; indeed, Conversations and Nostalgy are listening to two different events. However, our model includes the fact that the default action of a "keydown" event is to dispatch a "keypress" event, and while Conversations does `stopPropagation`, it does not `preventDefault`—which means that any key strokes typed into the quick-reply box will effectively call `convKeyDown`, `onNostalgyKeyPressCapture` and finally `onNostalgyKeyPress`. Consequently, typing a word containing an 'S' will steal focus from editing the message, and jump to Nostalgy's "Save Message" UI! And indeed, if we input the Thunderbird DOM and these three event listeners into our model, it confirms that this behavior is correct according to the event dispatch rules.

The author reported this bug to Conversations' developer, who produced the following fix:

```
quickReplyDiv.addEventListener("keypress",
  function convKeyPress(event)
    { event.stopPropagation(); });
quickReplyDiv.addEventListener("keyup",
  function convKeyUp(event)
    { event.stopPropagation(); });
```

Adding these two listeners to our model shows that event dispatch now calls `convKeyDown`, `onNostalgyKeyPressCapture`, and `convKeyPress`, and no longer calls `onNostalgyKeyPress`, thereby avoiding the bug for now. However, some of Nostalgy's code still gets called, leaving open the potential for future bugs. Examining the model, and recalling that Nostalgy never expected to "see" keypress events due to text input, notice that Nostalgy's code is called only because the "keypress" is dispatched, which occurs only because of the "keydown" default action. A simpler, and more robust, fix is therefore available to Conversations: adding a call to `preventDefault` in `convKeyDown` would prevent the dispatch of the "keypress" event in the first place. Implementing this approach in our model confirms that the "keypress" event is never fired. However, implementing it in Conversations does not work, and instead reveals a bug in Thunderbird: "keypress" events appear to be dispatched *regardless* of whether the default has been prevented or not, contrary to the spec.

Generalizing from this example, we can annotate listeners in our model with provenance information, and then query the model for whether there exist any $(eventType, targetNode)$ pairs for which dispatch will cause control to flow from one extension's listeners to another's. We anticipate that such queries will statically yield other pairs of extensions whose behavior might conflict; for example, Conversations is known to be incompatible with other hotkey-related extensions; this analysis can reveal others, then pinpoint where bugfixes are needed.

4.4 Event Propagation and Sandboxes

We have previously discussed the use of sandboxes to protect webapps against invited third-party code. Such sandboxed content, or "widgets" as they are often called, should not be able to suborn the surrounding document, and in fact, some successful efforts have *proven* the effectiveness of these sandboxing techniques [17]. However, there is a serious weakness in these proofs. The authors caveat their results as applying only over the DOM portion they model—which does not include events. While the sandboxes may successfully prevent widgets from calling DOM methods or executing arbitrary code, event dispatch provides an indirect way for widgets to invoke portions of the code of the webapp.

Specifically, because widgets present some form of UI, they can be the target of user-generated events. As a hypothetical example, a malicious widget might display what looks like a typing game. Because of the event dispatch rules, those keystrokes can bubble out of the widget and potentially invoke listeners higher in the webapp's document. The widget could selectively call `stopPropagation`, filtering out unwanted letters, and thereby forge an input to the webapp that the user did not intend. Worse, even if the sandbox stopped *all* propagation, it cannot prevent against listeners being invoked during the capture phase, which means the widget is getting to execute code as the program. To date, we know of no modeling effort that attempts to prove that widgets are sandboxed from conducting such a spoofing attack.

There are two possible approaches to protect against this. First, a conscientious webapp developer can protect his application against such unwanted events with defensive code that checks—in every event handler in the propagation path of a widget—whether the target of the event is in their own content or in the widget. Such coding practices are onerous and fail-open: one missed check could suffice for the attack to proceed. And yet, hardening every event listener would preclude extensions from integrating properly into the webapp, as any events originating from their UI would summarily be ignored! Second, the sandbox could decide that, because it cannot truly protect against a malicious widget due to the capture phase, it might choose to implement its own event dispatch model (as some libraries like jQuery[6] do). In such cases, the sandbox or library developer undertakes the burden of establishing properties of their custom event model. In either case, our DOM model would be useful: in the former case, to determine what propagation cases the surrounding page's listeners are missing, and in the latter case, by being a basis for formalizing and then proving properties about the custom event model.

[6]`http://jquery.com`

5 Related and Future Work

We have already introduced most of the related work in earlier sections of the paper. In particular, Featherweight Firefox [5] and Featherweight DOM [9] present the first formal models of a browser and of DOM tree update, while Akhawe et al. [2] model some security-relevant events but not their dispatch. This work sits between the three and reasons about the reactive behavior of web pages. There are several avenues for future improvement:

Keeping pace with moving standards While building our model, we were made aware of the DOM4 draft spec [21], which will supercede the Level 3 Events spec we modeled. At this time, the draft is not complete enough to model, though it is not intended to introduce substantive changes. Our model should carry over nearly unchanged.

Incorporating JavaScript Fully To focus on the DOM, we have represented JavaScript procedures with a simplified statement language for manipulating listeners and handlers (Fig. 2). This is sufficient for many practical modeling purposes, but it fails to fully capture the effect of JavaScript, which may be needed for some analyses.

Fortunately, this is easy to remedy. Our Redex model is formulated such that it will be straightforward engineering to incorporate the Redex model of λ_{JS}. In particular, though presented as states here, the five steps of event dispatch are modeled as contexts, which provides a great deal of flexibility. λ_{JS} models all of JavaScript with evaluation contexts [8], including one for function calls. In essence, event dispatch is a baroque form of a calling context, namely one that invokes multiple functions in sequence based indirectly on the DOM and the current event, rather than a simple function pointer. Our DOM model will change slightly to incorporate a reified event object, rather than just data carried in the `dispatch-*` contexts; we foresee no technical hurdles here.

We can therefore enhance our model by discarding the simplified statement language in favor of true JavaScript statements. Doing so brings significant benefit to JavaScript analyses as well. Without the structure provided by our model, an analysis of a JavaScript program would necessarily miss many flows that are not caused by explicit function calls in the program text.

Modeling the Document Tree Naturally, the precision of analyses is limited by the precision of modeling the document's structure. We currently model the tree merely as a set of nodes connected by pointers: nothing explicitly records that the structure is a tree rather than an arbitrary graph. We have engineered our model such that it should be possible, though likely not simple, to integrate more powerful tree logics such as separation or context logic [9], and thus improve the model's overall precision.

Iframes and nested documents Our model currently assumes that there is only one document under consideration. Consequently, event propagation (naturally) stops at the document root. A richer model would incorporate a notion of documents and their nesting within ⟨**iframe**/⟩ elements, and explicitly include a rule that terminates the propagation path at the document root.

Additionally, because our model does not fully model JavaScript, we do not model the `window` object, or include it as the first and last targets when constructing propagation paths. This detail does not materially affect our description of event dispatch, and is easy to include.

Non-tree-based dispatch We have focused in our model on how event dispatch proceeds with tree-based sources of events, as they dominate other event sources. However, newer additions to the DOM also supply events that are not dispatched along the tree. For example, `XMLHttpRequest` responses, ⟨**audio**/⟩ and ⟨**video**/⟩ status updates, and web workers all generate events as their states change. To incorporate those into our model, we need only create rules for each of them that construct their specific propagation paths, bypassing `pre-dispatch` and jumping directly to the `dispatch-collect` context. From then on, dispatch proceeds as normal.

Acknowledgements

This work is partially supported by the US National Science Foundation and Google. We are grateful for feedback about this work from Joe Politz and Arjun Guha. We thank Robby Findler for his help getting our model off the ground, and his co-authors and him for their excellent PLT Redex semantics modeling tool.

References

[1] ADsafe. Retrieved Nov. 2009. `http://www.adsafe.org/`.

[2] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. Towards a formal foundation of web security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium* (2010).

[3] Bandhakavi, S., King, S. T., Madhusudan, P., and Winslett, M. VEX: vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium* (Berkeley, CA, USA, Aug. 2010), USENIX Association, pp. 22–22.

[4] Barth, A., Felt, A. P., Saxena, P., and Boodman, A. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)* (2010).

[5] Bohannon, A., and Pierce, B. C. Featherweight Firefox: formalizing the core of a web browser. In *USENIX Conference on Web Application Development (WebApps)* (Berkeley, CA, USA, 2010), USENIX Association, pp. 11–11.

[6] Djeric, V., and Goel, A. Securing script-based extensibility in web browsers. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 23–23.

[7] Felleisen, M., Findler, R. B., and Flatt, M. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[8] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103*, 2 (1992), 235–271.

[9] Gardner, P. A., Smith, G. D., Wheelhouse, M. J., and Zarfaty, U. D. Local Hoare reasoning about DOM. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (New York, NY, USA, 2008), ACM Press, pp. 261–270.

[10] Guha, A., Saftoiu, C., and Krishnamurthi, S. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)* (Berlin, Heidelberg, 2010), Springer-Verlag, pp. 126–150.

[11] Guha, A., Saftoiu, C., and Krishnamurthi, S. Typing local control and state using flow analysis. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software* (Berlin, Heidelberg, 2011), ESOP'11/ETAPS'11, Springer-Verlag, pp. 256–275.

[12] Hors, A. L., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S. Document object model (DOM) level 3 core specification. Written Apr. 2004. `http://www.w3.org/TR/DOM-Level-3-Core/`.

[13] Ian Hickson, E. HTML5: A vocabulary and associated APIs for HTML and XHTML. Retrieved July 6, 2011. `http://dev.w3.org/html5/spec/Overview.html`.

[14] Lerner, B. S. *Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design*. PhD thesis, University of Washington Computer Science & Engineering, Aug. 2011.

[15] Lerner, B. S., Burg, B., Venter, H., and Schulte, W. C3: An experimental, extensible, reconfigurable platform for HTML-based applications. In *USENIX Conference on Web Application Development (WebApps)* (Berkeley, CA, USA, June 2011), USENIX Association.

[16] Maffeis, S., Mitchell, J. C., and Taly, A. An operational semantics for javascript. In *Asian Symposium on Programming Languages and Systems (APLAS)* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 307–325.

[17] Politz, J. G., Eliopoulos, S. A., Guha, A., and Krishnamurthi, S. Adsafety: type-based verification of javascript sandboxing. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 12–12.

[18] Reis, C. *Web Browsers as Operating Systems: Supporting Robust and Secure Web Programs*. PhD thesis, University of Washington, 2009.

[19] Schepers, D., and Rossi, J. Document object model (DOM) level 3 events specification. Written Sept. 2011. `http://dev.w3.org/2006/webapi/DOM-Level-3-Events/html/DOM3-Events.html`.

[20] The Caja Team. Caja. Written Nov. 2009. `http://code.google.com/p/google-caja/`.

[21] van Kesteren, A., Gregor, A., and Ms2ger. Dom4. Written Jan. 2012. `http://dvcs.w3.org/hg/domcore/raw-file/tip/Overview.html`.

Jigsaw: Efficient, Low-effort Mashup Isolation

James Mickens
Microsoft Research
mickens@microsoft.com

Matthew Finifter
University of California, Berkeley
finifter@cs.berkeley.edu

Abstract

A web application often includes content from a variety of origins. Securing such a mashup application is challenging because origins often distrust each other and wish to expose narrow interfaces to their private code and data. Jigsaw is a new framework for isolating these mashup components. Jigsaw is an extension of the JavaScript language that can be run inside standard browsers using a Jigsaw-to-JavaScript compiler. Unlike prior isolation schemes that require developers to specify complex, error-prone policies, Jigsaw leverages the well-understood public/private keywords from traditional object-oriented languages, making it easy for a domain to tag internal data as externally visible. Jigsaw provides strong iframe-like isolation, but unlike previous approaches that use actual iframes as isolation containers, Jigsaw allows mutually distrusting code to run inside the same frame; this allows scripts to share state using synchronous method calls instead of asynchronous message passing. Jigsaw also introduces a novel encapsulation mechanism called surrogates. Surrogates allow domains to safely exchange objects by reference instead of by value. This improves sharing efficiency by eliminating cross-origin marshaling overhead.

1 Introduction

Unlike traditional desktop applications, web applications are often *mashups*: applications that contain code from different principals. These principals often have asymmetrical trust relationships with each other. For example, a page that generates localized news may receive data from a news feed component and a map component; the integrating page may want to isolate both components from each other, and present them with an extremely narrow interface to the integrator's state. As another example, a social networking page might embed a third-party application and an advertisement. The integrating page may expose no interface to the advertisement. However, if the developer of the third-party application has signed a terms-of-use agreement, the integrating page may expose a relatively permissive interface to its local state.

Given the wide range of trust relationships that exist between web principals, it is challenging for developers to create secure mashups. Principals often want to share with each another, but in explicit and controlled ways. Unfortunately, JavaScript (the most popular client-side scripting language) was not designed with mashup security in mind. JavaScript is an extremely permissive language with powerful reflection abilities but only crude support for encapsulation.

1.1 Previous Approaches

Given the increasing popularity of web services (and the deficiencies of JavaScript's isolation mechanisms), a variety of mashup isolation frameworks have emerged from academia and industry. Unfortunately, these frameworks are overly complex and present developers with an unwieldy programming model. Many of these approaches [2, 11] force developers to use asynchronous, pass-by-value channels for cross-principal communication. Asynchronous control flows can be difficult for developers to write and understand, and automated tools that convert synchronous control flows into asynchronous ones can introduce subtle data races (§2.1). Additionally, marshaling data over pass-by-value channels like `postMessage()` can introduce high serialization overheads (§4.2).

Prior mashup frameworks also present developers with complex, overly expansive APIs for policy specification. For example, object views [11] require developers to define policy code that runs during each property access on a shared object. Understanding how these filters compose across large object graphs can be difficult. Similarly, ConScript [12] policy files consist of arbitrary JavaScript code. This allows Conscript policies to be extremely general, but as we demonstrate, such expressive power is often unnecessary. In many cases, secure mashups require only two policy primitives: a simple yes-no mechanism for marking JavaScript state as externally sharable, and a simple grammar based on CSS and regular expressions that constrains how untrusted code can access browser resources like persistent storage and the visual display.

1.2 Our Solution: Jigsaw

In this paper, we introduce Jigsaw, a new framework for mashup isolation. Jigsaw allows JavaScript code from mutually distrusting origins to selectively expose private state. Jigsaw's design was driven by four goals:

Isolation by default: In Jigsaw, an integrating script includes guest code which may hail from a different origin. The integrator has access to browser resources, and the integrator can provide its guests with access to some portion of those resources. However, by default, a guest cannot generate network traffic, update the visual display, receive GUI events, or access local storage. Similarly, each principal's JavaScript namespace is hidden from external code by default, and can be accessed only via public interfaces that are explicitly defined by the owning principal.

Efficient, synchronous sharing: Jigsaw eschews asynchronous, pass-by-value sharing in favor of synchronous, pass-by-reference sharing. Inspired by traditional object-oriented languages like Java and C++, Jigsaw code uses the `public` and `private` keywords to indicate which data can be accessed by external domains. When an object is shared outside its local domain, Jigsaw automatically wraps the object in a *surrogate* object that enforces public/private semantics. By inspecting surrogates as they cross isolation boundaries, Jigsaw can "unwrap" surrogates when they return to their home domain, ensuring that each domain accesses the raw version of a locally created object. Jigsaw also ensures that only one surrogate is created for each raw object. This guarantees that reference-comparison $==$ operations work as expected for surrogates. Using surrogates, Jigsaw can place mutually distrusting principals inside the same iframe while providing iframe-style isolation and pass-by-reference semantics. Since principals reside within the same iframe, no `postMessage()` barrier must be crossed, which allows for true synchronous interfaces.

Simplicity: Using deny-by-default policies for browser resources like network access, and using the `public` and `private` modifiers to govern access to JavaScript namespaces, Jigsaw can express many popular types of mashups—most require only a few lines of policy code and the explicit definition of a few public interface methods. In designing Jigsaw, we consciously avoided more complex isolation schemes like information flow control, object views [11], and ConScript [12]. While these schemes are more expressive than Jigsaw, their interfaces are unnecessarily complex for many of the mashup patterns that are found in the wild.

Fail-safe legacy code: In most cases, regular JavaScript code that has not been adapted for Jigsaw will work as expected when used within a single domain. In all cases, unmodified legacy code will fail safely (i.e., leak no data) when accessed by external domains. Jigsaw prohibits some JavaScript features like dynamic prototype manipulation, but these features are rarely used by benevolent programs (and are potentially exploitable by attackers) [1, 11]. Jigsaw makes few changes to the core JavaScript language, and we believe that these changes will be understandable by the average programmer, since the changes make JavaScript's object model behave more like that of a traditional, class-based OO language like C#. Jigsaw preserves many of the language features that make JavaScript an easy-to-use scripting language. For example, Jigsaw preserves closures, first-class function objects, object literals, an event-driven programming model, and pass-by-reference semantics for all objects, not just those that are shared within the same isolation domain.

2 Design

In Jigsaw, *domains* are entities that provide web content. In the context of the same-origin policy, a Jigsaw domain corresponds to an origin, and we use the terms "domain" and "origin" interchangeably. A *principal* is an instance of web content provided by a particular domain. A principal may contain HTML, CSS, and JavaScript. Note that some principals might contain only JavaScript, e.g., a cryptographic library might only define JavaScript functions to be invoked by external parties.

A user visits top-level web sites; each of these sites can be an *integrator* principal. An integrator may include another principal P_i by explicitly downloading content from P_i's origin. In turn, P_i may include another principal P_j. Figure 1 depicts the relationship between the user and this hierarchy of client-side principals. When there is an edge from P_i to P_j, we refer to P_i as the *including principal* or *parent*, and P_j as the *included principal* or *child*.

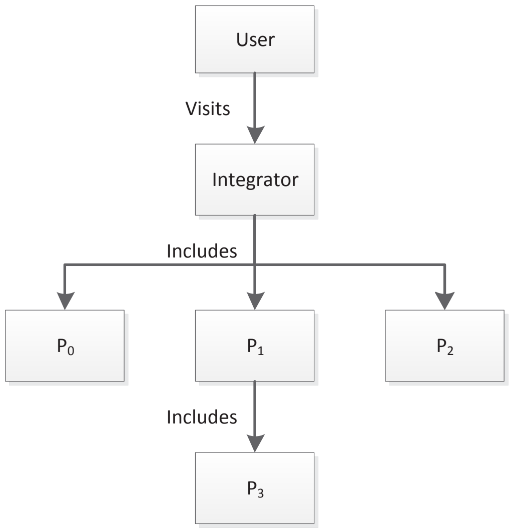


Figure 1: An example of a principal hierarchy. The user visits an integrator site that includes other principals. Each of those principals may include additional principals.

2.1 Boxes

Jigsaw places each principal in an isolation container that we call a *box*. Each box is associated with the following resources:

- A *JavaScript namespace* containing application-defined objects and functions.
- A *DOM tree*, which is a browser-defined data structure representing the HTML and CSS content belonging to the principal.
- An *event loop*, which captures mouse and keyboard activity intended for that box.
- A rectangular *visual region* with a width, a height, a location within the larger browser viewport, and a z-axis value.
- A *network connection*, which allows the principal to issue HTTP fetches for data.
- A *local storage area*, which stores cookies and implements the DOM storage abstraction.

In some ways, a Jigsaw box resembles a traditional iframe. Both provide a principal with a local JavaScript namespace, DOM tree, event loop, and visual field. However, Jigsaw boxes differ from iframes in three important ways.

First, if two iframes share the same origin, they can directly access each other's JavaScript namespaces via frame references like `window.parent` and `window.parent.frames`. A Jigsaw box does not allow such unfettered cross-principal access. By default, two boxes have no way to communicate with each other, even if they belong to the same origin. This enables fault isolation and privilege separation for different principals that originate from the same domain. To allow cross-box communication, each principal must explicitly define public functions on a *principal object*. By exchanging principal objects with each other, boxes define the set of external domains with which they communicate, and the operations that these domains may invoke on private state.

```
//Download a script and place it inside a
//new box. The return value of the script
//is its principal object.
var p = Jigsaw.createBox('http://x.com/box.js',
                  {network: /(x|y)\.com/,
                   storage: true, dom: null});

//Call a public method defined by the box.
//The pass-by-reference of localData and
//result is made safe by the use of
//surrogates.
var result = p.f(localData);
```

Figure 2: An integrator creating and interacting with a box. The box can exchange network traffic with servers from x.com and y.com; it can also access its origin's DOM storage, but it cannot access the integrator's DOM. Surrogates are explained in Section 2.6.2.

A second difference between iframes and Jigsaw boxes is that boxes use nesting relationships to more tightly constrain the resources of children. A page's top-most Jigsaw box is given a full visual field that is equivalent to the entire browser viewport. The root box is also given the maximal network permissions allowed by the same-origin policy. By default, descendant boxes lack access to non-computational browser resources. For example, child boxes cannot issue network requests, and they cannot access the visual field (and thus they cannot receive GUI events from the user). A parent box may delegate a region of its visual field to a child. Similarly, a parent can grant a child box a portion of its network permissions. In both cases, parent-child delegation has monotonically increasing strictness, i.e., a parent can never give a child a larger visual field or more networking permissions than the parent has. Figure 2 shows an example of how an integrator creates a new box. Note that DOM storage ACLs are unique because the browser gives each origin a local storage area. Thus, an integrator can grant a child access to the child domain's local storage, or completely prohibit such accesses. However, the integrator cannot directly expose its own local storage to a child domain (although it can define a method on its principal object that mediates access to its DOM storage).

The final difference between iframes and boxes involves communication channels. Applications in different iframes communicate with the asynchronous

```
var x = 0;  //Global variable

function increment(){
    x++;
}

function f(){
    alert(x);
    increment();
    alert(x);
}

//Create two tasks that fire every 100 ms.
setInterval(f, 100);
setInterval(f, 100);
```

Figure 3: In regular JavaScript, all code is given the illusion of single-threaded execution. Thus, in this example, only one version of `f()` can be executing at any given time. This means that the `alert()` statements from a particular instance of `f()` will always output two consecutive numbers.

```
function f(){
    alert(x);
    increment->(); //This version of f() now
                   //relinquishes the CPU!
    alert(x);
}
```

Figure 4: If increment() is asynchronous and replaced with a pseudo-synchronous continuation, a particular invocation of `f()` may be swapped off of the processor without executing atomically, allowing the other `f()` to execute fully, and causing the first one to output 0 and then 2. This would be impossible if the `->` continuation operator provided true synchronous semantics.

`postMessage()` call. `postMessage()` can only transmit immutable strings; thus, passing large objects across this channel can incur significant marshaling overhead, and explicit message passing is needed to keep mirrored data structures synchronized between iframes. The asynchronous nature of `postMessage()` also makes it difficult to provide true synchronous RPC semantics. Although tools exist to convert asynchronous function calls into continuation-passing style (CPS) [18], CPS can introduce data races that do not exist when function calls are truly synchronous (see Figures 3 and 4). Ensuring that such races do not exist requires the programmer to explicitly reason about synchrony and use application-level coordination mechanisms like locks.

Besides the performance and correctness challenges, asynchronous `iframe/postMessage` architectures are often ill-suited for many mashup designs. For example, consider pure computational libraries. If an integrator has some data that it wants to process using N calls to a cryptographic library or an image manipulation library, it is cumbersome for the integrator to set up a chain of asynchronous callbacks that executes the $(i+1)^{th}$ operation when the i^{th} operation has completed. A synchronous programming model is much more natural.

There are also event-driven mashups that are ill-suited for the asynchronous `iframe/postMessage` model. For example, suppose that an integrator uses an external library to sanitize AJAX data as it streams in. As chunks of data arrive, the browser fires the `XMLHttpRequest` callback multiple times. During each invocation, the integrator must pass the new AJAX data to the sanitization library. However, if the sanitizer lives in a separate `iframe`, the integrator cannot receive the sanitized data immediately—instead, the integrator must buffer data and wait for the sanitizer to asynchronously return the scrubbed results. This introduces two sources of asynchrony (the `XMLHttpRequest` handler and the sanitizer callback) when only one should be necessary (the `XMLHttpRequest` handler).

A Jigsaw application can have multiple boxes, but all of the boxes live in the same frame. As shown in Figure 2, this means that data can be passed synchronously and by reference. As described later, Jigsaw code uses the `public` and `private` modifiers to indicate which methods and variables are accessible to external domains. The Jigsaw runtime uses these modifiers to validate cross-domain operations (§2.6.2).

A single Jigsaw application may contain multiple principals that originate from the same domain. Jigsaw places each of these principals in a separate box. These principals interact with each other using the same public interfaces that are used by principals from different domains.

2.2 Principal Objects

A *principal object* defines the public functions and variables that a domain exposes to untrusted code; to communicate with an external domain, a box must possess a reference to that domain's principal object. A principal can access its parent's principal object by calling `Jigsaw.getParentPrincipal()`. Similarly, the `Jigsaw.principals` array contains principal objects for all of a box's immediate children. `Jigsaw.getRootOriginPrincipal()` returns the principal object belonging to the highest-level box from the caller's origin. This ancestor can act as a coordination point for all principals from that domain, e.g., if the principals want to synchronize their writes to DOM storage. Principal objects can be passed between boxes like any other object.

2.3 The DOM Tree

Each Jigsaw box can potentially contain a DOM tree and an associated visual field. If the box's parent did not delegate a visual field, Jigsaw sets the box's DOM tree to null, and prevents the child principal from adding DOM nodes to it. Otherwise, the child may update its visual field and receive GUI events for that field by modifying the DOM tree in the standard way. Jigsaw ensures that the visual updates respect the constraints defined by the parent.

A visual field consists of a width, a height, a location within the parent's visual field, and a z-order. Parents specify these parameters using CSS-style syntax. Thus, a child can have a static visual geometry, or one that flows in dynamic ways, e.g., to occupy a percentage of the parent's visual field, regardless of how the parent is resized.

Visual parameters are associated with each principal object (e.g., `principal.height`). A parent can dynamically change a child's visual field by writing to these fields. If a child wants to change its visual field, it can also try to write to these fields. However, the changes must be validated by the parent. A child write to a visual field parameter fires a special Jigsaw event in the parent called `childVisualFieldRequest`. If the parent has registered a handler for this event, the handler inspects the changes requested by the child. If the parent approves the change, the handler returns true, otherwise it returns false. Jigsaw will only implement the change if the parent has defined such a handler, the handler returns true, and the change would not place the child's visual field outside the one owned by the parent.

Similar to the Gazelle browser [24], Jigsaw requires visually overlapping boxes to be opaque with respect to each other. In other words, boxes cannot request transparent blending of their overlapping visual region—the box with the higher z-order occludes all others in the stack. This prevents a malicious box from making itself transparent, creating a child box containing a victim page, and then collecting GUI events that the user intended to send to the victim box.

Even with these protections, a principal must still trust its ancestors in the principal hierarchy. This is because a malicious parent can virtualize a child's runtime (§3) in subversive ways, or not create a child at all. Jigsaw can only guarantee that parents are protected from descendants, and that sibling principal hierarchies are protected from each other if the shared parent is non-malicious.

2.4 Network Access

A principal uses HTTP requests to communicate with remote servers. The principal's parent controls the set of servers that are actually reachable. Like visual field permissions, network privileges nest in a monotonically restrictive way. The most expansive privilege is "*", which means that a principal can fetch any resource that is allowed by the same-origin policy. Parents can also restrict children to a limited set of accessible domains. Parents can specify a group of related domains using a straightforward wildcard syntax, e.g., `*.foo.com` or `cache.*.bar.com`. As a syntactic shortcut, a parent can specify the "self" domain to indicate that the child can communicate with servers from the child's origin. Similarly, the "parent" token resolves to the parent's origin.

2.5 Local Storage

In HTML5, the DOM storage abstraction allows each origin to maintain a client-side key/value database. Each database can be accessed only by JavaScript code from the associated origin. Jigsaw partitions DOM storage in the same way. If principals from different origins want to exchange data from their respective DOM storage areas, they must do so via public interface methods.

2.6 The JavaScript Namespace

In traditional JavaScript, objects are dictionaries that map property strings to values. Using JavaScript's extremely permissive reflection interface, a program can dynamically enumerate an object's properties and read, write, or delete those properties. Unlike standard class-based languages like Java and C#, JavaScript uses prototype-based inheritance. A prototype object is an exemplar which defines the property names and default property values for other instances of that object. By setting an object's `__proto__` property to the exemplar, the object becomes an instance of the prototype's class. By setting the `__proto__` fields of prototype objects, one creates inheritance hierarchies. By default, an object's property list is dynamic, so an instance of a particular prototype can dynamically gain additional properties that are not defined by the prototype. An object's `__proto__` field is just another property, meaning that an object's class can dynamically change as well.

These default reflection semantics are obviously unsuited for cross-domain encapsulation. JavaScript does allow a limited form of data hiding using closures, which are functions that can access a hidden, non-reflectable namespace. Unfortunately, closures are an imperfect substrate for cross-domain sharing. They cannot be shared across iframes, and within an iframe, each closure has unfettered access to the DOM tree, event loop, and network resources that belong to the enclosing frame. Furthermore, closures are clumsy to program and maintain, since the hidden closure variables are implicitly obscured via lexical scoping instead of explicitly marked via a special keyword. Jigsaw provides simpler, stronger encapsulation using boxes and the `public`/ `private` keywords.

```
function Ctor(x, y){
    public this.x = -1;
    private this.y = 42;
    this.z = 100;   //By default, a new
                    //field is private.
}

public Ctor.prototype.prop1 = "hi";
private Ctor.prototype.prop2 = "bye";
Ctor.prototype.prop3 = "aloha"; //Private
                                //by default

obj = {};
public obj.a = 0;
private obj.b = 1;
obj.c = 2;  //Private by default.
```

Figure 5: Jigsaw code example (private-by-default property visibility).

```
function Ctor(){}
private Ctor.prototype.x = 0;

obj = new Constructor();
obj.x = 42; //Succeeds: Private properties
            //visible within the creating box.
public obj.x; //Fails: can't override modifier
              //specified by prototype
```

Figure 6: Jigsaw code example (visibility modifiers flow from prototypes to instances).

2.6.1 Visibility modifiers

The `public` and `private` keywords allow a principal to define which object properties are visible when an object is shared across boxes. For example, if a principal from domain X creates the following object ...

```
var obj = {public x:  "foo",
           private y:  "bar"};
```

... and passes it to domain Y, Y can access `obj.x` but not `obj.y`. Note that "access" means the ability to read, write, and delete a property.

The `public`/`private` keywords can be used anywhere a variable is declared. If a variable is declared and no visibility modifier is specified, *it is private by default*, as shown in Figure 5. When code from domain X enumerates the properties of an object from domain Y, private fields do not appear in the enumeration. Within Y, private fields do show up in the enumeration.

As shown in Figure 6, public and private modifiers "flow downward" from prototypes to instances, overriding any attempts by instance objects to reset the modifiers. JavaScript allows object properties to be declared at arbitrary moments, so during program execution, when Jigsaw encounters a `public` or `private` statement, it must dynamically check whether the statement satisfies the visibility settings for the relevant prototype object.

When Jigsaw passes objects between boxes using surrogates (§2.6.2), it never exposes object prototypes or constructor functions. This prevents a wide class of exploits called prototype poisoning [1, 11] in which an attacker dynamically modifies the inheritance chain for an object and subverts the object's intended implementation.

2.6.2 Surrogate Objects

Jigsaw uses surrogate objects to enforce public/private semantics. When an object `obj` is passed between boxes, e.g., when box X invokes a function on Y's principal object and passes a local object `obj` as an argument, Jigsaw wraps `obj` in a surrogate and passes that surrogate, not the original object, to the destination domain Y. To create the surrogate, Jigsaw first creates an initially empty object. Then, for each public property belonging to `obj`, Jigsaw adds a getter/setter pair for a corresponding property on the surrogate object. Getter/setters are a JavaScript feature that allow an object to interpose on reads and writes to a property. For `obj`'s surrogate, the getter for property `p` returns `createSurrogate(obj.p)`. The setter for property `p` executes `obj.p = createSurrogate(newVal);`.

The `createSurrogate()` function has several important features. First, `createSurrogate()` associates at most one surrogate for each "raw" object. Thus, calling `createSurrogate(obj)` multiple times on the same object will always return the same surrogate object. This ensures that the reference-compare $==$ operator has the expected semantics for surrogates. For example, if a box is passed two surrogates from two different domains, and those surrogates refer to the same backing object, then the surrogates will reference-compare as equal.

Another important property of `createSurrogate()` is that it does not always return a surrogate object. For immutable, pass-by-value primitive properties like numbers, `createSurrogate()` returns the primitive value. More interestingly, if a surrogate is being passed to its originating box, `createSurrogate()` returns the backing object. In the previous example, this means that if Y passes `obj`'s surrogate back to X, Jigsaw will "unwrap" the surrogate and hand the raw object back to X. This convenient feature also helps $==$ to work as expected, since boxes do not need to worry about receiving a surrogate for a local object that lacks reference equality with that local object.

A final property of `createSurrogate()` is that surrogate getter/setters invoke it lazily—if a surrogate property is never accessed by external boxes, the surrogate will never call `createSurrogate()` for that property. As we show in Section 4.2, this lazy evaluation is beneficial when boxes share enormous object graphs but only touch a fraction of the objects. Using lazy evaluation, Jigsaw never devotes computational resources to protect objects that are shared but never accessed.

For each public method belonging to `obj`, the surrogate defines a wrapper function whose `this` pointer is bound to `obj`. Thus, even if external code assigns the surrogate method to be a property on another object, the method will always treat `obj` as its `this` pointer. This prevents attacks in which a malicious box subverts a method's intended semantics by supplying an inappropriate `this` object.

When a surrogate function is invoked, it calls `createSurrogate()` on all of its arguments before passing those arguments to the underlying function. The surrogate function also calls `createSurrogate()` on the underlying function's return value, and returns that surrogate object to the original caller of the surrogate function.

In summary, surrogates automatically protect cross-box data exchanges. Since principal objects are surrogates, and boxes can be accessed only via their principal objects, Jigsaw ensures that all cross-box interactions respect public/private semantics.

2.6.3 Predefined JavaScript Objects

The browser predefines a set of JavaScript objects that live in each box's JavaScript namespace. The most important predefined object is the DOM tree; others provide support for regular expressions, mathematical functions, and so on. Jigsaw virtualizes the DOM tree in each box (§3), redirecting the box's DOM operations to a Jigsaw-controlled data structure that performs security checks before reflecting operations into the real DOM. Jigsaw also ensures that constructor functions for globally shared built-in objects like regular expressions are private and immutable. These safeguards prevent a malicious box from arbitrarily manipulating the visual display, or redefining constructor functions that are used by all boxes.

2.6.4 Cross-box Events

Box X may wish to register one of its functions as an event handler in a different box Y. To do so, X simply passes the handler to Y via Y's public interface; Y can then register the handler with the browser's event engine in the standard way. When the relevant event in Y occurs, the browser executes the handler like any other. However, the browser passes a scrubbed event object to the handler. This event does not contain references to Y's private-by-default JavaScript namespace. This prevents information leakage via foreign event handlers. Like all foreign methods, the handler executes in the JavaScript context of the box that created it.

2.7 Client-side Communication Privileges

A parent can restrict the principal objects that are visible to a child. By default, a child can reference its parent's principal object (`Jigsaw.getParentPrincipal()`), the principal objects of its own children (the `Jigsaw.principals` array), and the principal objects of other boxes from its own domain (`Jigsaw.getSameDomainPrincipals()`). Jigsaw associates each principal with a unique id, and a parent can restrict a child's access to a subset of principals ids.

2.8 Dropping Privileges

Jigsaw allows a box to voluntarily restrict the networking privileges that it received from its parent. A box can also relinquish the right to a visual field, or abandon the ability to write to DOM storage. Privilege can drop only in a monotonically decreasing fashion. For example, if a parent gives a child unrestricted network access, the child cannot restrict its privileges to only `foo.com` and then unrestrict itself.

2.9 Summary

Jigsaw provides a robust encapsulation technique for cross-principal sharing. All data is accessible by reference, but all data is implicitly hidden from external parties unless it is explicitly declared as `public` by the owning principal. Parents define resource permissions for the execution contexts of their children. Such permissions become monotonically stricter as the principal nesting depth increases.

Jigsaw does enforce some restrictions on the standard JavaScript language. In particular, a surrogate never exposes the prototype object or constructor function for the underlying object. Jigsaw also prevents box code from tampering with the prototypes of global built-in objects like Array. While this prevents boxes from changing externally defined prototype chains, well-written JavaScript code rarely uses such tricks, and allowing such behavior allows malicious boxes to launch prototype poisoning attacks [1, 11] against other boxes. Despite these restrictions, Jigsaw preserves many features of the standard JavaScript language. For example, Jigsaw supports closures, first-class function objects, object literals, an event-driven programming model, and pass-by-reference semantics for all objects, not just those shared within the same domain.

3 Implementation

Our Jigsaw implementation consists of a Jigsaw-to-JavaScript compiler and a client-side JavaScript library. The compiler parses Jigsaw code using an ANTLR toolchain [20]. A custom C# program adds static security checks to the resulting ASTs, and then translates the modified ASTs to JavaScript code that a browser can execute. The emitted JavaScript code also contains the client-side Jigsaw library, which implements runtime security checks and defines box management interfaces like `Jigsaw.createBox()`.

The rewriter modifies every object creation so that each object receives a unique integer id. This allows the Jigsaw library to maintain a mapping from raw objects to the associated surrogates, ensuring that `createSurrogate()` makes at most one surrogate for each raw object.[1]

The rewriter also tags each object with the id of its creating box (the Jigsaw library defines an internal `getCurrentBoxId()` function to which the rewriter can insert a call). This tag allows the Jigsaw library to determine whether a surrogate is being passed to its original box—if so, Jigsaw "unwraps" the surrogate, returning the backing object instead of the surrogate (§2.6.2). To allow `createSurrogate()` to determine the currently executing box context, the rewriter modifies all function definitions such that on entry, a function pushes its box id onto a stack, and on exit, a function pops the stack.

The rewriter translates `public` and `private` property declarations into operations on a per-object map of visibility metadata. The rewriter assigns such a map to each object at object creation time. Using property descriptors [19], Jigsaw ensures that per-object metadata is immutable and cannot be modified by a malicious or buggy box.

The Jigsaw library is responsible for creating new boxes. To do so, the library uses an `eval()` statement to dynamically load the (rewritten) box code. However, the `eval()` call is invoked within the context of a special Jigsaw function that defines aliasing local variables for standard global properties like `window`, `document`, and so on. The Jigsaw-defined aliases implement a virtualized browser environment that forces a box's communication with the outside world to go through Jigsaw's security validation layer. For example, Jigsaw's virtual `XMLHttpRequest` object ensures that a box's AJAX requests satisfy the security policies defined by the box's parent. Similarly, the virtual DOM tree is backed by a branch of the real DOM tree, but virtual operations are not reflected into the real tree unless they satisfy the parent box's security policy. For dangerous functions like `eval()`, and for sensitive internal Jigsaw functions, Jigsaw creates null virtualizations that do nothing or throw exceptions on access. As with all things JavaScript, there are various subtleties in the virtualization process that we elide due to space constraints.

Our current Jigsaw prototype implements the bulk of the design from Section 2. The primary exception is full DOM tree virtualization. This is still a work-in-progress due to the complexity of the DOM interface.

[1]To prevent this data structure from hindering garbage collection, Jigsaw requires weak maps, whose design is being finalized for the next version of JavaScript [16].

4 Evaluation

Jigsaw's goal is to provide an efficient, developer-friendly isolation framework. In this section, we describe our experiences with porting preexisting JavaScript libraries to Jigsaw; we then evaluate the performance of the modified libraries. We show that porting legacy code to Jigsaw is straightforward, that Jigsaw's pass-by-reference surrogates are much more efficient than pass-by-value marshaling, and that Jigsaw's dynamic security checks are similar in performance to those of other rewriting-based systems like Caja [15].

4.1 Porting Effort

We found that many preexisting JavaScript libraries already had an implicit notion of "public" and "private"; thus, porting these libraries to Jigsaw primarily consisted of making implicit visibility settings explicit via the `public` and `private` keywords. For example, at initialization time, many libraries add a single new object to the global JavaScript namespace, and use that object's properties as the high-level interface to the library code. This object serves as a de facto principal object (although it has none of the security protections afforded by Jigsaw). To port libraries like this to Jigsaw, we first declared the de facto gateway object to be the Jigsaw principal object for the library. We marked that object's properties with the `public` keyword. We then identified the public properties of other library objects with the help of an instrumented version of the Jigsaw runtime. For each surrogate that crossed a box boundary, the instrumented runtime logged the public and private properties for the surrogate's backing object; the runtime also modified each surrogate so that all foreign box accesses to private properties on the backing object threw an immediate exception instead of returning `undefined`.

The surrogate log and the fail-stop exceptions on private property accesses made it easy to identify legacy code properties that needed to be marked as public. Porting was also simplified because we did not have to worry about asynchronous control flows as in PostMash [2].

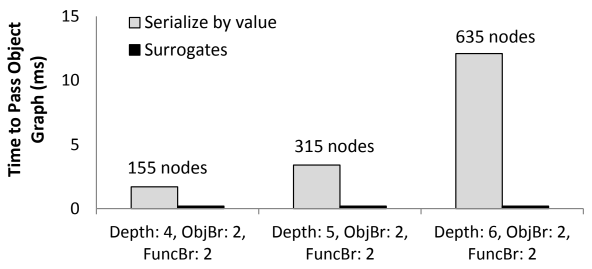


Figure 7: When mutually distrusting domains must share objects with each other, Jigsaw's pass-by-reference surrogate mechanism is much faster than pass-by-value marshaling (`Depth`: depth of shared object tree, `ObjBr`: Number of object properties per object, `FuncBr`: Number of function properties per object).

We also did not need to explicitly insert object sanitization calls—in contrast to Caja [15], which requires developers to invoke a `tame()` sanitization function wherever an object crosses a trust boundary, Jigsaw automatically creates surrogates when "raw" objects travel between boxes. We provide a more detailed comparison of Jigsaw, PostMash, and Caja in Section 5.

4.2 Performance

In this section, we use microbenchmarks and Jigsaw-modified applications to evaluate Jigsaw's performance overheads. All of the results were generated on a Windows 7 PC with 4 GB of RAM and a dual-core 3.2 GHz processor. All web pages were executed in the Firefox 8.0.1 browser.

Sharing overheads: In mashup frameworks that use `postMessage()`, isolation containers share data by asynchronously exchanging immutable strings. If containers wish to share more complex objects, each endpoint must implement a marshaling protocol that serializes objects on sends and deserializes objects on receives. The marshaling costs can be high, particularly if containers wish to share functions, since the recipient has to dynamically compile each shared function's source code using `eval()`.

When a sender passes an object to a recipient, the sender actually shares an object graph that includes all of the objects that are recursively reachable from the root. Figure 7 depicts the sharing cost for synthetic object graphs of various sizes. In these experiments, the graphs were trees. The "Depth" metric corresponds to the height of the tree. The "ObjBr" (object branch) metric indicates how many of an object's properties referenced other objects; similarly, the "FuncBr" (function branch) metric indicates how many properties pointed to functions. Functions had no child objects or functions, i.e., they were leaf nodes in the object tree.

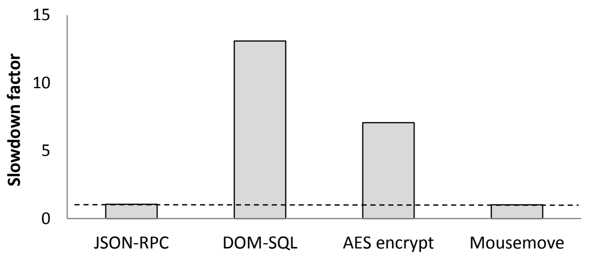


Figure 8: Jigsaw's extra security tests add between 0x–12x performance overhead (the dotted line indicates a slowdown factor of 1, i.e., no slowdown). Jigsaw's performance overheads are similar to those [10, 22] of other rewriting-based mashup frameworks like Caja [15].

As Figure 7 demonstrates, passing an object graph between isolation containers using surrogates has almost zero overhead. When the root of an object graph is passed across a box boundary, Jigsaw must create a new surrogate object for it; however, the only cost is an object creation (for the surrogate object itself) and a function call for each public property on the backing object (to create the getter/setter property descriptors (§2.6.2)). In contrast, marshaling pass-by-value data is much more expensive, since the entire object graph must be traversed, serialized, and then deserialized, with costly `eval()` operations on the receiver-side to recreate shared functions. Note that the results in Figure 7 do not include `postMessage()` overhead, i.e., the sender and the receiver were in the same frame. Thus, these results are a conservative estimate of the marshaling costs in a `postMessage()` system.

In Jigsaw, passing large object trees across isolation barriers is efficient because surrogates are lazily created when a box actually tries to access a foreign object. Thus, at sharing time, initially only one surrogate must be created for the root of the object graph. In pass-by-value systems, once an object graph has been recreated by the receiver, property accesses on the graph are just as cheap as regular property access on locally created objects. In contrast, accessing a property through a Jigsaw surrogate introduces the overhead of invoking a getter or setter. Such accesses are roughly 30 times more expensive than a regular property access. However, we believe that this cost is acceptable for three reasons. First, it is only paid upon accessing external objects; it is not paid for objects that are never accessed by external domains, nor is it incurred when a box accesses its locally declared objects. Second, since Jigsaw's initial sharing cost is *O(1)* in the size of the object graph to share, Jigsaw reduces the initial sharing costs of non-trivial graphs by multiple orders of magnitude compared to a pass-by-value solution. Modern web applications already have object graphs containing hundreds of thousands of objects [14], so for complex mashups exchanging complex object graphs, the initial sharing cost of pass-by-value will be unattractive due to the computational latencies. Third, unlike pass-by-value systems, surrogates allow mashups to communicate synchronously using pass-by-reference semantics. This makes developing mashups much easier, and makes Jigsaw's property access penalties easier to bear.

End-to-end performance: In addition to performing checks during property accesses, Jigsaw must perform a variety of bookkeeping tasks, e.g., assigning box ids and object ids to newly created objects, and interposing on accesses to virtualized browser resources to ensure that boxes adhere to the security policies established by their parents. Figure 8 shows the end-to-end performance slowdown for several Jigsaw-enabled applications. The slowdown is normalized with respect to the performance of the baseline, non-Jigsaw-enabled applications. The dotted line indicates a slowdown of 1, i.e., a situation in which Jigsaw adds no performance overhead. We examined the following applications:

- JSON-RPC [9] is a JavaScript library that layers an RPC protocol atop an AJAX connection. We defined the performance of a JSON-RPC session as the integrator-perceived completion rate of null RPCs that performed no actions at a localhost RPC server. Using a localhost server instead of a remote one eliminated the impact of network delay and allowed us to focus on Jigsaw's CPU overhead.
- The DOM-SQL [3] library provides a SQL interface to DOM storage. We defined the performance of DOM-SQL as the number of rows that the integrator could insert into a table per second. Each insertion caused a synchronous write to DOM storage.
- The AES encryption function belongs to the Stanford JavaScript crypto library [21]. Performance was defined as the throughput with which the integrator could feed plaintext to the library and receive ciphertext.
- Mousemove is a simple benchmark library which registers a handler for mouse movement in the integrator's DOM. A human user moves the mouse back and forth as quickly as possible, and the library increments a counter every time that its handler fires. Performance was defined as the number of times that the handler fired.

In the Jigsaw version of each application, the library code was placed in a separate box from the integrator, and all integrator-integratee communication took place through a principal object or a virtualized DOM resource.

Figure 8 shows that Jigsaw's security checks cause a 0-12x slowdown. The slowdown is application-dependent. For example, in the Mousemove test, the rate at which the browser fired mouse handlers was slow enough that Jigsaw's security overhead was hidden. The JSON-RPC test was similar, since the rate at which AJAX callbacks fired was also slow enough to hide Jigsaw's overhead. In contrast, in the encryption test and the DOM-SQL test, the applications were rarely blocked on external browser activity. Instead, these programs executed many small, application-defined functions. This incurred a lot of Jigsaw bookkeeping overhead, since Jigsaw had to update its internal call stack for each function invocation and return (§3). Nevertheless, Jigsaw's performance overheads were similar to those of other rewriting systems like Caja [10, 22].

5 Related Work

There are many preexisting frameworks for securing mashup applications. As we describe in more detail below, these systems present very different programming models to developers. At a high level, Jigsaw differs from all of these systems due to its focus on providing simple, efficient isolation mechanisms. Jigsaw is simple because it defines a concise ACL language for browser resources, a straightforward `public`/`private` distinction for JavaScript properties, and an automatic surrogate mechanism that transparently protects cross-domain data exchanges while preserving synchronous function call semantics. Jigsaw is efficient because its pass-by-reference surrogates avoid the marshaling overhead that afflicts pass-by-value systems.

ADsafe, FBJS, and Dojo Secure: ADsafe [6] and Dojo Secure [25] use a language subsetting approach, forcing guest code to be written in a restricted portion of the larger JavaScript language. In contrast, Jigsaw allows guest code to be written in a larger, more expressive subset. This makes it easier for developers to port legacy applications to Jigsaw (and write new Jigsaw applications from scratch). However, Jigsaw does pay a performance penalty due to the dynamic security checks that are required to secure the larger language subset.

FBJS [8] uses rewriting to prepend guest code identifiers with a unique random prefix. This effectively isolates the guest code from the integrator. FBJS allows guest code to interact with its parent through a restricted, virtualized API, e.g., through calls to a `VirtDOMnode.getParentNode()` method instead of through direct accesses to the parent's `DOMnode.parentNode` property.

Broadly speaking, FBJS, ADsafe, and Dojo Secure present a similar architectural model: strict guest isolation with a narrow, predefined interface between isolation containers. In contrast, Jigsaw allows principals to define their own public interfaces.

Caja: Like Jigsaw, Caja [15] is a rewriting system that places scripts inside virtualized execution environments. Caja defines a `tame(obj)` function that makes `obj` safe to pass to untrusted JavaScript contexts. `tame()` performs many of the security checks described in Section 2.6.2. For example, it prevents dynamic prototype manipulation, and it ensures that methods cannot be called with arbitrary `this` references.

Jigsaw differs from Caja in several important ways. In Jigsaw, objects and their properties are invisible to external domains by default. Developers use the `public` keyword to mark an interface as externally visible; visibility annotations are defined as part of the interface declaration. In contrast, Caja's visibility metadata is managed at interface *sharing* time instead of interface *declaration* time. In Caja, programmers must remember to invoke `tame(obj)` before `obj` is passed across an isolation barrier. This makes a program's security properties more difficult to understand, since developers can no longer reason about how an object can be accessed without inspecting all of the places at which the object crosses an isolation boundary. In contrast, Jigsaw's declaration-time visibility modifiers provide clearer, more centralized indications of object access rights. Jigsaw's surrogate mechanism also provides automatic "taming" as objects flow between boxes. This eliminates the developer burden of having to manually tame objects at sharing time. It also guarantees that taming takes place all of the time, regardless of whether the developer remembered to tame an object.

Also note that Caja's primary goal is to make it easy for an integrating page to incorporate untrusted scripts—guest-to-guest interactions are of secondary importance. Thus, host-to-guest communication is straightforward, but guest-to-guest interactions must be mediated by the host. This requires the integrator to define and manage a shared communication infrastructure. In contrast, Jigsaw's goal is to make it easy for arbitrary execution contexts to communicate through restricted interfaces. Using principal objects, any two contexts in Jigsaw can exchange information. Furthermore, the integrator is no longer responsible for managing cross-script communication. Instead, each script implements its own cross-box protocols.

Secure ECMAScript (SES): Secure ECMAScript (SES) [17] uses newly standardized features of ECMAScript 5 [7] to provide Caja-like isolation without requiring Caja-like rewriting and runtime virtualization. By pushing dynamic security checks into the JavaScript engine, SES can potentially offer dramatic reductions in the costs of these checks. SES is still being formulated, but once ECMAScript 5 becomes widespread, Jigsaw can use SES techniques to implement its security abstractions.

Pass-by-value systems: Systems like PostMash [2] use iframes as isolation containers, and implement cross-domain communication using the asynchronous, pass-by-value `postMessage()` call. Such isolation frameworks have several drawbacks. First, there is significant marshaling overhead if domains share non-trivial object graphs (§4.2). To avoid this penalty, domains can keep local copies of object graphs and synchronize views across iframes. However, this approach still requires frequent exchanges of synchronization messages. In PostMash, this message traffic induced a 60% performance decrease in a Google Maps mashup [2].

A second drawback of these systems is that they rely on an asynchronous channel for cross-domain communication. Asynchronous communication is an awkward fit for many mashup applications; for example, it is ill-suited for the integration of computationally intensive modules that implement databases, cryptographic operations, input sanitizers, image manipulation routines, and so on. Rewriters can transform asynchronous operations into quasi-synchronous ones using continuation-passing [11]. However, many programmers find it difficult to reason about continuations. Furthermore, continuations can introduce subtle race conditions that are not present in truly synchronous environments (§2.1).

Browsers give each iframe a separate execution thread. Thus, iframe isolation does have the advantage that a parent can make forward progress if a child is hung or computationally intensive. In Jigsaw, each box resides within the same iframe, so a misbehaving child can intentionally or inadvertently perform a denial-of-service attack on its parent. We do not view this as a major disadvantage of Jigsaw, since modern browsers allow users to terminate unresponsive scripts via a pop-up warning dialog.

Object views: Object views [11] let developers specify policy code that controls how objects are shared across isolation boundaries. Policy code is written in the full JavaScript language and is attached to view objects that mediate external access to private backing objects. This security model is very expressive, but we believe that it is unnecessarily complex (and therefore error-prone). Jigsaw's `public` and `private` modifiers present a more intuitive programming model, allowing developers to express simple "yes-no" disclosure policies. In contrast, when a developer writes an object view policy, she must reason about the execution context that initiates an access request, and how context-specific factors should influence data exposure. Jigsaw's visibility identifiers act as explicit, declaration-time indications of visibility policy.

IFC: Information flow control (IFC) systems like Jif [13] assign security labels to variables, allowing developers to precisely specify the data that principals should be allowed to read or write. We eschewed an IFC mashup architecture for two reasons. First, a well-known problem with IFC is that programmers are loath to generate the required security annotations. In contrast, simple visibility modifiers like `public` and `private` have not triggered a similar level of complaint. IFC-style labels are also ill-suited for governing access to browser resources. For example, it is difficult to use labels to express policies like "give a principal update rights to the leftmost 30% of the visual display." Jigsaw can easily express such a policy using a simple CSS-style rule.

ConScript: ConScript [12] uses a modified browser engine to enforce security. Integrators restrict the behavior of guests by attaching policy code to key execution points, e.g., the invocation of a function or the loading of a new script. Like object view policies, ConScript policies are written in arbitrary JavaScript and can be extremely expressive. However, as mentioned above, Jigsaw strives to provide simple, developer-friendly security policies, and we have found that in practice, Jigsaw's simpler policies are sufficient to express many kinds of mashup architectures.

OMash: Like Jigsaw, OMash [5] allows each principal to define a public set of functions that other principals can invoke. However, OMash does not have a private-by-default visibility policy, nor does it wrap objects in proxies before handing them to external domains. Thus, public OMash functions expose an ostensibly narrow interface, but their return values can expose sensitive data. For example, the caller of a public OMash function can perform arbitrary JavaScript reflection on the properties of the returned object (and any other objects reachable from that root). If the caller modifies any of this data, the modifications will be visible in the data's source domain.

MashupOS: MashupOS [23] provides a new set of isolation abstractions for web browsers. In MashupOS, a service instance is a browser-side analogue of a traditional OS process. Each instance gets a partitioned set of hardware resources like CPU and memory, and communicates with other instances using asynchronous, pass-by-value messages. Jigsaw eschews such a communication style in favor of synchronous, pass-by-reference messaging. This necessitates a mechanism like surrogates (§2.6.2) for securely exchanging objects across isolation boundaries.

CommonJS Modules: CommonJS [4] defines a module system for JavaScript. CommonJS gives each library a protected namespace and the ability to define external interfaces. However, CommonJS namespaces are implemented using closures. Thus, unlike Jigsaw boxes, CommonJS namespaces do not protect against attacks like prototype poisoning [1, 11]. CommonJS also does not provide strong notions of public and private data. Thus, as in OMash, the return values from public functions can inadvertently leak private module data.

6 Conclusion

Jigsaw is a new mashup framework for web applications. It allows mutually distrusting content providers to define narrow public interfaces for their private client-side state. Jigsaw strives to be developer-friendly, so it eschews the complicated security policies of prior mashup frameworks; instead, Jigsaw uses the `public` and `private` keywords to mark data as externally visible or domain-private. Jigsaw's security semantics are thus easily understandable to programmers who are familiar with popular languages like Java that also use `public`/`private` distinctions.

Prior mashup frameworks often isolate state using iframes or iframe-like abstractions. These isolation containers force domains to communicate using asynchronous pass-by-value channels. In contrast, Jigsaw's novel surrogate mechanism allows domains to pass objects by reference using synchronous function calls. This makes it easier for developers to reason about cross-origin sharing, since accessing a locally defined object or function looks no different than accessing an object or function that has been shared by an external domain. Pass-by-reference surrogates are also more efficient than pass-by-value approaches because surrogates do not incur marshaling overhead when they travel between domains.

Our evaluation shows that existing web applications are easily ported to the Jigsaw framework. Our evaluation also demonstrates that Jigsaw has similar or better performance than prior mashup schemes.

Acknowledgments

We would like to thank David Wagner and the anonymous reviewers for their comments on earlier versions of this paper. This material is based upon work partially supported by a NSF Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Adida, B., Barth, A., and Jackson, C. Rootkits for JavaScript Environments. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)* (2009), USENIX Association.

[2] Barth, A., Jackson, C., and Li, W. Attacks on JavaScript Mashup Communication. In *Web 2.0 Security and Privacy* (2009).

[3] Boere, P. dom-storage-query-language: A SQL inspired interface for DOM Storage. `http://code.google.com/p/dom-storage-query-language/`.

[4] CommonJS. Modules/1.1.1 Specification, February 2012. `http://wiki.commonjs.org/wiki/Modules/1.1.1`.

[5] Crites, S., Hsu, F., and Chen, H. OMash: Enabling Secure Web Mashups via Object Abstractions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), ACM, pp. 99–108.

[6] Crockford, D. ADsafe. `http://www.adsafe.org`.

[7] ECMA International. Standard ECMA-262: ECMAScript Language Specification, 5.1 Edition, June 2011. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`.

[8] FBJS (Facebook JavaScript). `http://developers.facebook.com/docs/fbjs`.

[9] Gherardi, G. jsonrpcjs. `https://github.com/gimmi/jsonrpcjs`.

[10] Google. google-caja: Performance of cajoled code. `http://code.google.com/p/google-caja/wiki/Performance`.

[11] Meyerovich, L., Felt, A., and Miller, M. Object Views: Fine-grained Sharing in Browsers. In *Proceedings of the 19th International Conference on World Wide Web* (2010), ACM, pp. 721–730.

[12] Meyerovich, L., and Livshits, B. ConScript: Specifying and Enforcing Fine-grained Security Policies for JavaScript in the Browser. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 481–496.

[13] Meyers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. Jif: Java Information Flow, July 2001. `http://www.cs.cornell.edu/jif`.

[14] Mickens, J., Elson, J., Howell, J., and Lorch, J. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of NSDI* (2010).

[15] Miller, M., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized JavaScript. Google white paper. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[16] Miller, M. S. ES Wiki: harmony:weak_maps, May 2011. `http://wiki.ecmascript.org/doku.php?id=harmony:weak_maps`.

[17] Miller, M. S. SES (Secure EcmaScript), May 2011. `https://code.google.com/p/es-lab/wiki/SecureEcmaScript`.

[18] Mix, N. Narrative JavaScript. `http://www.neilmix.com/narrativejs/doc/`.

[19] Mozilla Developer Network. Object.defineProperty(). `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Object/defineProperty`.

[20] Parr, T. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, Raleigh, North Carolina, 2007.

[21] Stark, E., Hamburg, M., and Boneh, D. Symmetric Cryptography in JavaScript. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2009), IEEE, pp. 373–381.

[22] Synodinos, D. ECMAScript 5, Caja and Retrofitting Security: An Interview with Mark S. Miller, February 25 2011. `http://www.infoq.com/interviews/ecmascript-5-caja-retrofitting-security`.

[23] Wang, H., Fan, X., Howell, J., and Jackson, C. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of SOSP* (October 2007).

[24] Wang, H., Grier, C., Moshchuk, A., King, S., Choudhury, P., and Venter, H. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th USENIX Security Symposium* (2009), USENIX Association, pp. 417–432.

[25] Zyp, K. Secure Mashups with dojox.secure, August 2008. `http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/`.

Social Networks Profile Mapping using Games

Mohamed Shehab, Moo Nam Ko, Hakim Touati
Department of Software and Information Systems
University of North Carolina at Charlotte
{*mshehab,mnko,htouati*}@*uncc.edu*

Abstract

Mapping user profiles across social network sites enables sharing and interactions between social networks, which enriches the social networking experience. Manual mapping for user profiles is a time consuming and tedious task. In addition profile mapping algorithms are inaccurate and are usually based on simple name or email string matching. In this paper, we propose a Game With A Purpose (GWAP) approach to solve the profile mapping problem. The proposed approach leverages the game appeal and social community to generate the profile mappings. We designed and implemented an online social networking game (*GameMapping*), the game is fun and is based on human verification. The game presents the players with some profiles information, and uses human computation and knowledge about the information being presented to map similar user profiles. The game was modeled using incomplete information game theory, and a proof of sequential equilibrium was provided. To test the effectiveness of the mapping technique and detection strategies, the game was implemented and deployed on Facebook, MySpace and Twitter and the experiments were performed on the real data collected from users playing the game.

1 Introduction

Social network (SN) services have been one of the main highlights of Web 2.0. Popular SN sites have attracted millions of users, for example Facebook hosts over 500 million users. Different SNs provide users with different sets of services and experiences, for example, Facebook and MySpace allow users to creates photo albums, fan clubs, and post feeds along with sharing all this content with friends, Twitter provides users with the ability to post short messages, and LinkedIn enables users to connect with other users for professional purposes. To enjoy these services, users endup creating accounts on different sites, for example most Twitter users have a Facebook account [14], and 64% of MySpace users have accounts in Facebook [28]. With the increasing popularity of SN connect services [18], this enabled users to connect websites with their SN accounts and to share their opinions and comments across networks. Leading SN sites are moving towards meeting the user's cross site interactions demands [20]. Users are able to connect different SN accounts and to share data across SNs, for instance, a user could connect his Twitter feed to his Facebook status such that his Facebook status will be updated automatically whenever he updates his Twitter feed [3].

When users create new accounts on a site they will spend time trying to rebuild their friendship connections with users they know, to alleviate this task several sites provide users with "import your friends" capability. For example, Bob has an established account in Facebook, and Bob heard from his friends about the video posting services provided by MySpace, so Bob creates a new account in MySpace which offers him to import his friends from Facebook. Using this functionality MySpace imports profile attributes of Bob's Facebook friends, and attempts to locate users who have similar attributes in MySpace (name, location, email hash, etc.) and recommends to Bob to add them as friends in MySpace. This approach is not effective in locating users with popular names, or for users who don't have matching attributes. Studies have shown that users tend to enter false information in their profiles [30], which causes attribute based matching approaches to generate inaccurate results [35, 39]. Furthermore, graph matching solutions are computationally expensive and require the knowledge of the complete graph of both networks [7, 8, 2]. Email based matching is only available when users use a same email across sites. A simple solution would be possible if all sites use a federated identity such as OpenID [31], however this technology is not popular among social network users.

In this paper, we propose a Game With A Purpose approach to solve the profile mapping problem. The proposed approach leverages the game appeal and social community to generate the profile mappings. We designed and implemented an online social networking game (*GameMapping*), the game is fun and is based on human verification. *GameMapping* takes advantage of people's existing perceptual abilities and desire to be entertained. The game will present the player with a user from one social network, and a set of friends from another social network, which represent the *set of mapping recommendations*. The friends' information is summarized in a profile card which includes the profile photo, name, age, location, etc. The player gets a small number of points for choosing one of the provided mappings, this reinforces a sense of *incremental individual success* in the game. The game also rewards *social success* by awarding the player a large number of bonus points when other users or friends agree to the player's provided mappings. This proposed mechanism is similar to social buying, where buyers are offered discounts discount deals (bonus) if they sign up for a deal in large masses [27]. Users will be allowed to invite their friends to play the game in hope of gaining the large bonus points. Similar games with a purpose have been successfully proposed to aid in labeling and tagging images over the web [33]. We also investigated several approaches for generating the set of mapping recommendations. The proposed *GameMapping* game was analyzed using game theory, to identify equilibrium under the current assumptions and point granting scheme to ensure that rational players will provide accurate profile mappings to maximize their game score. We performed several experiments to evaluate our approach on the game results, and we compared it to attribute based mapping which is presented in the experimental section. The main contributions of the paper are summarized as follows:

- We proposed a Game With A Purpose approach for solving the profile mapping problem as a game supported by social verification.
- We proved the equilibrium of the game scoring mechanism using game theory to ensure that rational players will provide accurate profile mappings while playing the game.
- We implemented our game as an online social networking game in Facebook, MySpace and Twitter. This implementation is a proof a concept and was used to collect and perform experimental results.

The rest of this paper is organized as follows. Section 2, provides an overview of Game With a Purpose and social networks. Section 3, defines the problem of profile mapping across sites. Section 4 describes how the proposed game works, and gives game details that include recommendation mechanism and the game theoretic proof. Section 5, describes the implementation of game system and the experimental results. The related work is discussed in Section 6, and Section 7 provides the paper conclusion.

2 Preliminaries

2.1 Game With a Purpose

Games with a Purpose (GWAP) is a form of human computation [33, 34], which gets humans to play enjoyable games that are also productive tools. These games are used in tasks that are hard for computers but easy for humans. For example, the ESP game [33] is a two-player game used for labeling and tagging images over the web, the game is setup to reward players providing the same labels by giving them bonus points if their tags match. Our goal is to design a GWAP to solve the profile mapping problem between social networks, by asking players to map their friends in the different social networks. One of the main challenges is the design of a points system that rewards correctly identified profile mappings and to maximize the reward for truthful rational players, and minimize the reward of irrational players. Gaming on social network platforms is becoming very popular with games such as FarmVille in Facebook [13] hosting over 62 million monthly active users. Our proposed game can easily be deployed on social network sites as an online game, and if it is popular we estimate that most of the account mappings can be properly discovered in a matter of weeks.

2.2 Social Networks

Users and relationships between users are the core components of social networks. Each user manages an online personal profile, which usually includes information such as the user's name, birth date, address, contact information, emails, education, interests, photos, music, videos, blogs, and many other items. Each user $u_i \in V$ maintains a profile P_i, which is composed of N profile attributes, $\{A^i_1, \dots, A^i_N\}$. Each attribute is a name-value pair (an, av), where *an* and *av* represent name and value respectively. For example, a Facebook user profile includes attributes such as birthday, location, gender, religion, etc. Users are also able to post objects such as photos, videos, and notes to their profiles to share with other users.

Users are connected to a set of friends, using this notion a social network can be modeled as an undirected graph $G(V,E)$, where the set of vertices V is the set of users, and the set of edges E is the set of friendship relationships between users. The edge $(u_i, u_j) \in E$ implies that users u_i and u_j are friends. Using the graph based model for social networks, we leverage the node network structural properties to provide additional user attributes. These attributes include several small world network metrics such as: node degree centrality, betweenness, hit rate, eigen values [24]. For a user u_i, we are able to compute M network metrics $B_i = \{B^i_1, \ldots, B^i_M\}$. Each network attribute is similarly represented as a name-value pair (bn, bv) that will be added to the user personal profile attribute previously stated to constitute the user profile P. The neighborhood of user u is the subgraph $\mathcal{N}_u = (V_u, E_u)$, where $V_u = \{v | v \in V, (u, v) \in E\} \cup \{u\}$, $E_u = \{(x, y) | x, y \in V_u, (x, y) \in E\}$.

3 Problem Definition

The global profile mapping is defined as follows:

Definition 1 *(Profile Mapping Problem). Given social networks SN_A and SN_B, with social graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$ respectively, find the set of profile mappings M of the form $(u_i, u_j) \in M$ where $u_i \in V_A$ and $u_j \in V_B$ belonging to the same user in both social graphs G_A and G_B.*

The problem of mapping data concepts between different sites or platforms have been applied to multiple areas, such as: database schema matching [21, 29], web search [10, 5], ontology mapping [9] and visualization [12, 38]. The graph isomorphism is an NP-Complete problem which involves finding one to one mappings between vertices and edges of a pair of graphs [4, 16]. The subgraph isomorphism graph matching problems has been proven to be NP-complete [15]. Furthermore, when $|V_A| \neq |V_B|$ known as the inexact graph matching problem, the complexity is proven in [1] to be NP-complete. In addition, the inexact sub-graph matching problem is NP-complete, and the largest common subgraph problem is also equivalent in complexity to the later which is NP-complete. Several attribute, model, object recognition and network based techniques were proposed to provide heuristic approaches to solving graph matching problems [7, 8, 2], these approaches are computationally expensive, and require the knowledge of the complete graphs G_A and G_B. In this paper, we propose solving the profile mapping problem by using human computation in the form of an online game.This approach has been used in [34, 33] to effectively map tags to images. The main assumption is that with the correct set of incentives, users would enjoy playing a game and at the same time contribute to mapping profiles between users in different networks.

Definition 2 *(Local Profile Mapping Problem) Given a user u who has identities u_i and u_j in social network SN_A and SN_B respectively, and user's local neighborhoods $\mathcal{N}^A_{u_i}$, $\mathcal{N}^B_{u_j}$ find the set of mappings $M_u \subseteq M$ mappings between profiles in $\mathcal{N}_{u_i}$ and $\mathcal{N}_{u_j}$.*

Our proposed approach will leverage the individual and social knowledge of social network users to provide mappings, and to provide mapping verifications which can be then used to solve the local profile mapping problem. The local profile mapping problem does not require knowledge of the whole social network graph, instead it only requires knowledge of the neighborhood network. Providing incentives to ensure the wide spread adoption of the game would allow solving a large number of local profile mappings, which enables the mapping of all similar profiles in large social networks. In fact, this is equivalent to the generalization of the subgraph isomorphism mappings of local networks to the maximum number of common subgraph problem in the global networks [40]. In this paper we are interested in studying mappings between social networks user accounts like Facebook, MySpace, and Twitter. Mapping profiles in social networks is applicable to identity management, and is a step towards enabling cross site interactions between users in different sites.

4 General Game Description

Our proposed game is called *GameMapping*. The basic idea is that players gain points by providing mappings of their friends' profiles in multiple social networks. *GameMapping* allows players to map Facebook and MySpace profiles, or Facebook and Twitter profiles.

In order to play the game the player needs to complete an authentication stage that involves two social network sites. We implement Facebook Connect, MySpaceID and TwitterID to enable the user to authenticate into the corresponding social networks and to authorize the GameMapping site access to the user's profile and friends list. This enables the GameMapping site to retrieve the user's profile and neighborhood social graph data which includes last name, first name, gender, age, country, profile picture, friends list and mutual friendships. This data enables our system to compute the local neighborhood for the current player $(\mathcal{N}^A_u, \mathcal{N}^B_u)$. The user profile referred to as the focus user u_f is presented to the player for mapping. The focus user u_f is selected from the neighborhood with the smaller size (cardinality). Without loss of generality we assume $|\mathcal{N}^A_u| \leq |\mathcal{N}^B_u|$, where the focus user profile u_f is selected from neighborhood $\mathcal{N}^A_u$. The game computes the recommended mappings profiles R from neighborhood $\mathcal{N}^B_u$ based on attribute and network distance metric. The focus user and the computed recommendations are then presented to the player. Figure 1, shows a screen shot of the game, where the focus user is in the center surrounded by his possible best recommended mappings displayed in a random order. The users' profile pictures are shown along with their profile information which include, age, gender, and location. Information about the recommended mappings is presented to the user when the mouse is moved over the photo. The player should decide either to map the focus user to one of the recommended profiles or to skip if no map is present. The player is given 40 seconds to make a decision about the presented game data set, then a new game data set is presented. The game also presents top 10 players ordered by the points earned. To motivate players into making cor-

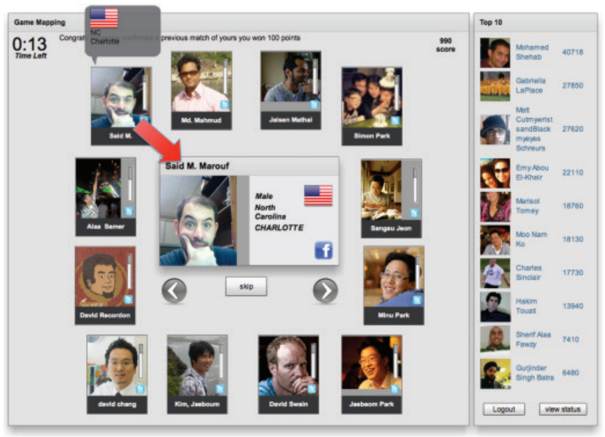


Figure 1: The GameMapping Screen Shot

rect decisions of either mapping or skipping, the game awards the player 10 points for any provided map, 100 bonus points if the provided map is confirmed by another player, and 30 bonus points if a skip is confirmed by another player. In order to maximize the points (reward), a player should focus on providing the mappings that will most probably be confirmed by other players. When a player start the game, the player first plays the game with the player own network data set. In other words, the player maps friend's profiles. After the player is done mapping his local network, the player plays the game with a game dataset that is randomly selected. It ensures that players provide mappings towards multiple local profile mappings and at the same time ensure the game continuity. By motivating players to play multiple data sets enables the game to provide mapping confirmations as will be discussed in the game theoretic proof. In addition, each game dataset represents a local mapping problem, which when combined for multiple data sets results in the global mapping of the overall social network graph.

4.1 Recommendation Generation

Given a player u who owns profiles u_i and u_j, and the neighborhoods $\mathcal{N}_u^A$ and $\mathcal{N}_u^B$ the focus user u_f is selected randomly from the neighborhood that has the smaller number of nodes, which we refer to as the focus network. This design choice was made as the maximum number of possible mappings is equal to $min(|V_u^A|, |V_u^B|)$. Figure 2, shows both neighborhoods and the focus user u_f. Lets

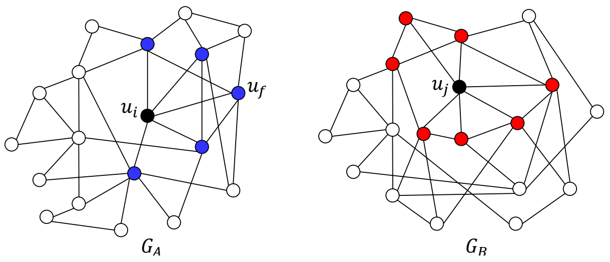


Figure 2: Neighborhood and Focus User Recommendations.

assume the focus user u_f is selected from $\mathcal{N}_u^A$. Given the focus user the mapping recommendation is generated by ranking the user profiles in $\mathcal{N}_u^B$ based on their similarity to the focus user. The similarity between two profiles is computed as a weighted sum of distances between the different user profile and network attributes. The profile attributes include first name, last name, gender, age and address. The network attributes include the centrality, betweenness, hit rate, degree and eigen values [6, 25]. We investigated several vector distances which include the Chebychev and Minkowski distance for numerical attributes, Cosine and Levenshtein distance for nominal attributes, and the Euclidian distance for the numerical attributes (i.e. age) and the Levenshtein distance for nominal attributes (i.e. gender, name) [19]. The weights of each attribute were computed based on a linear regression classifier trained using the knowledge collected from our initial experiments [36, 37]. The recommendation set R is the sorted list of proposed user profiles based on their computed similarity with the focus user. As indicated in Figure 1, the game presents the user with the top 12 recommended mappings select from the recommendation set R following the Top-k Fagin's algorithm [11]. The selected recommendations are shuffled randomly then displayed in a clock-wise fashion around the focus user. This randomization is required to ensure that players put some effort in finding the possible profile mapping among the displayed 12 recommendations. Moreover, by randomizing the recommendation set R this would avoid possible collusion between different players as each player is presented with the same 12 recommendations but not in the same location on the screen.

4.2 Game Theoretic Analysis

In this game the players do not communicate and each player does not know the action taken by other player. The game can be modeled as a two player extensive game with incomplete information. In this game the players are provided with a focus user u_f and a set of recommended mappings $R = \{u_1, \ldots, u_n, \phi\}$. Each player has a set of $n+1$ actions of the form $a_k = \mathbf{map}(u_f, u_k)$ where $u_k \in R$. Note, the action $a_{n+1} = \mathbf{map}(u_f, \phi)$, which is equivalent to the $\mathbf{skip}(u_f)$. The set of actions $A_1 = A_2 = A$, and the utility (δ_i) of player i is selected to satisfy the following conditions:

- $\delta_1 = \delta_2 = \delta$,
- $\delta(a_i, a_j) = \delta(a_j, a_i)$,
- $\delta(a_i, a_i) > \delta(a_i, a_j)$ for all $i \neq j$,
- $\delta(a_i, a_i) > \delta(a_{n+1}, a_{n+1})$ for all $1 \leq i \leq n$

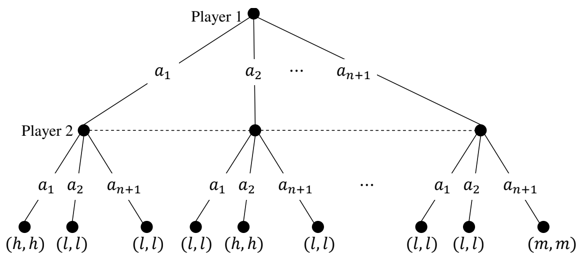


Figure 3: Game tree with imperfect information.

Figure 3, shows the extensive game tree, where each nodes represent players and edges represent player actions. The payoffs for players 1 and 2 are shown at the terminal nodes. The values of h and l are chosen such that $h > l$, this ensures that $u(a_i, a_i) > u(a_i, a_j)$ for all $i \neq j$. This game is a coordination game in which each player is trying to make the same choice as the other player to maximize their utility.

Rational players intend to maximize their expected game payoff. Note that the payoff from agreeing on a map is higher than the payoff from agreeing on a skip $(h > l)$, this motivates rational players to try to find possible maps between the focus user and one of the recommendations and to skip if they can not find a suitable map. The Nash equilibrium is a commonly used equilibrium notion that provides an equilibria such that no player can profitably deviate from and enhance their payoff with the belief that other players will not deviate [26]. Referring to the game representation in table form in Figure 4, The game has $n+1 = |A|$ pure Nash equilibria represented by the set S where $S = \{(a_i, a_i) : a_i \in A\}$, that is strategy that would result in maximizing the user payoff is when both users make the same action.

Player 1

Player 2	a_1	a_2	...	a_n	a_{n+1}
a_1	(h,h)	(l,l)	...	(l,l)	(l,l)
a_2	(l,l)	(h,h)	...	(l,l)	(l,l)
$\vdots$	(l,l)	(l,l)	...	(l,l)	(l,l)
a_n	(l,l)	(l,l)	...	(h,h)	(l,l)
a_{n+1}	(l,l)	(l,l)	...	(l,l)	(m,m)

Figure 4: Game Nash Equilibria Indicated in Grey.

Since the game has multiple equilibria it is still not clear what action strategy with a rational player act upon. Given that each player does not know the action taken by the other player, the question that each player asks themselves is that given $\{u_f, R\}$ "what would other players do if they are presented with the same $\{u_f, R\}$?" and by the theory of focal points [22] players will usually coordinate at points that in some sense stick out from the others (focal points). A player game strategy can be described based on the probability of selecting an action a_i from the action set A given the focus user and recommendation set $\{u_f, R\}$. The probability $p(a_i|\{u_f, R\})$ represents the probability of choosing an action a_i conditioned on the game parameters $\{u_f, R\}$, which can be represented as $p(a_i|\{u_f, R\}) = p(a_i) \times r(a_i, \{u_f, R\})$. Where $r(a_i, \{u_f, R\}) = \frac{p(a_i, \{u_f, R\})}{p(a_i) \times p(\{u_f, R\})}$ is the relevance of action a_i to the set $\{u_f, R\}$. According to focal point analysis, a rational player would choose the action that maximizes the $p(a_i|\{u_f, R\})$ which is the action that is most relevant to the current $\{u_f, R\}$ set, which is described as follows:

$$a^* = \arg\max_{a_i \in A} p(a_i) \times r(a_i, \{u_f, R\})$$

By choosing action a^* players maximize their chance of being matched by other players in the system and ultimately gaining the payoff $\delta(a^*, a^*)$.

Assuming players are rational and they will choose the action that is most relevant for the given focus user and recommendation set, a dominant strategy that ensure that players coordinate and maximize their expected utility is attained when players follow the same actions selection probability $p(a_i|\{u_f, R\})$ [32]. This implies that players will be motivated to provide a map when they recognize a map and will prefer to choose skip if a map does not exist.

5 Experiments and Results

5.1 Implementation Details

We implemented the GameMapping game as an online game. The online game is functional on client browsers supporting Adobe Flash. The game communicates with a centralized GameMapping server to exchange and retrieve data. The game server is responsible for retrieving user profiles from social network sites, generating focus user and recommendation data sets, and storing all the mapping information. To support these features, we implemented social web application tools and APIs in the game server. Figure 5, depicts the architecture of our

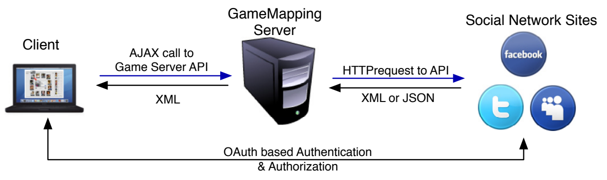


Figure 5: The Architecture of GameMapping

system. The game server connects to the each social network site using social web application tools such as Facebook Connect, MySpaceID, and TwitterID. These tools allow our game server to interact with the APIs of each social network site on behalf of game players. Facebook Connect is based on OAuth 2.0 specification while MySpaceID and TwitterID are based on OAuth 1.0a specification. We also implemented social plugins such as Like Button and Invitation to enhance the popularity and adoption of our game through friend of friend invitations and word of mouth. We implemented a polling mechanism to enable the retrieval of user's profile information, that is based on both server and client technologies (Ajax).

5.2 Collusion and Irrational Behavior

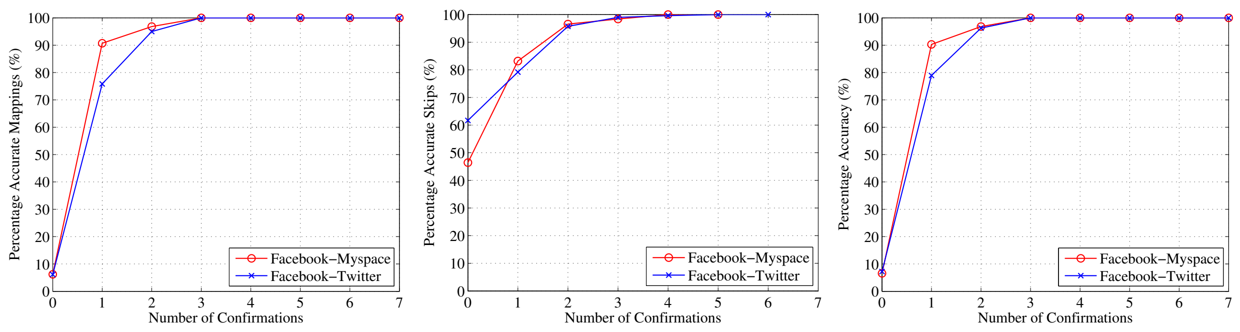
It is possible that some players map non-mapping and incorrect profiles intentionally. Based on the game theoretical discussion in Section 4.2, rational users are able to maximize their payoff by selecting the correct actions (map or skip). Irrational players are players who attempt to play the game and provide inaccurate mappings in hope of gaining high points or simply affecting our mapping accuracy. Although our game system does not provide a chatting feature, players might collude using another communication channel such as AIM or MSN chat, in order to provide the same inaccurate mappings to the game. To prevent collusion among players, our game displays randomly selected data sets to different players, who are allowed to play each game data set only once. Another irrational behavior is a player providing inaccurate mappings continuously by guessing, and getting l points for each provided map or skip. The game scoring mechanism ensures that rational players converge to a high score faster than guessing players.

In addition, we insert detection datasets into the normal game datasets to detect the irrational players. The detection game datasets are normal dataset that do not contain any correct mapping. If a player provides many mappings for the detection game dataset, there is high probability the player is an irrational player. We also recorded the amount of time taken by players in making each mapping to detect the irrational players and robots. If a player is an irrational player or a robot, the player might spend less time in each single mapping than rational players since the irrational players might provide mappings without comparing profiles. The game provides a CAPTCHA if the response rate is above the normal rate to prevent robots from playing the game. Finally we applied mapping confirmation strategy. If an irrational player provides inaccurate mappings, there is a low chance the inaccurate mapping gets a confirming map from other rational players.

5.3 Experiments

To evaluate our approach, we recruited participants which have accounts in multiple social networks by inviting users from MySpace and Twitter groups and apps on Facebook. As an incentive to play the game, we held a two week game competition to encourage users to participate in our research and distributed 10 iTunes gift cards to the top 10 players and an iPod Nano to the top player. One hundred and twenty-four players agreed to play the game, of which 80 where male, 32 female and 12 did not indicate their gender. There were two kinds of game the Facebook-MySpace (FB-MS) game for mapping user profiles between Facebook and MySpace and the Facebook-Twitter (FB-TW) game to map Facebook to Twitter. The FB-MS game was played by 30 players, and 94 players registered and played the FB-TW game. Perhaps users favored playing the FB-TW game due to the increasing popularity of both Facebook and Twitter. During the two weeks game competition, we collected 38,532 Facebook profiles, 8,452 MySpace profiles, 11,775 Twitter profiles and 7,411 profile mappings between user profiles. The collected profiles were used to generate the game datasets which were presented to the players to provide mappings between profiles in different networks. The game presented the players with a privacy consent that indicated that only the public information will be shared with other players which included the user's first name, last name, and location.

For verification and experimental purposes we manually verified all the provided profile mappings provided by the players using a simple verification web tool that shows details of mapped user's profiles with an inspection form. We designed the tool to generate comparison results of last name, first name, age, and gender automat-



(a) Mapping Accuracy (b) Skip Accuracy (c) Overall Accuracy

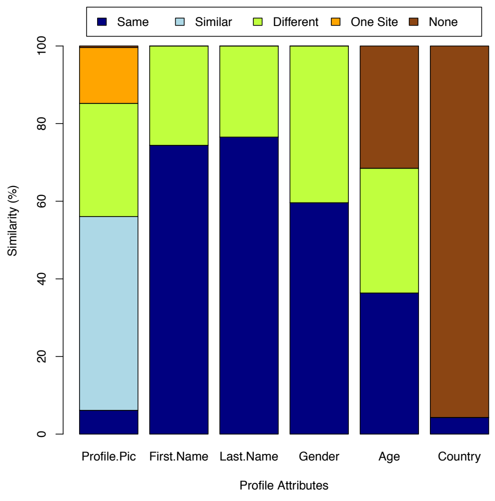
Figure 6: GameMapping Experimental Accuracy Results.

ically and we manually input the comparison result for profile pictures and countries. For each profile mapping we compared the profile pictures and categorized them into one of 5 types which include, Same, Similar, Different, Picture present only in one site, and None (picture is not present). In case of address and location information, geocoding distances were used to compare both profiles. If the profile information was not enough to make a decision, the inspectors visited profile page in each social network site to compare both profiles.

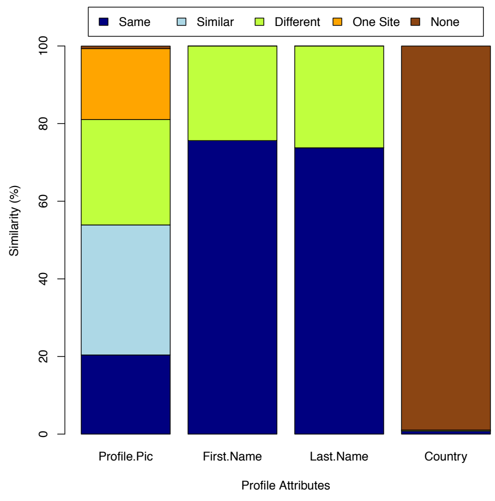
5.4 Evaluation of Mapping results

We analyzed the number of player confirmations required for accurate profile mappings and skippings by comparing the mappings provided by the players with the mappings verified manually. Figure 6(a) presents the mapping accuracy for different number of confirmations for both kinds of games (FB-MS and FB-TW), as shown the mapping accuracy increases as the number of confirmations increase. Note that, the mapping confirmation plateau's at 100% after 3 confirmations, which indicates that we need at least 3 confirmations to support 100% accuracy and 2 confirmations for 95% mapping accuracy. Figure 6(b) presents the skipping accuracy, which follows a similar pattern as the mapping accuracy as it also plateau's at 100% accuracy after 3 player confirmations for both FB-MS and FB-TW games. The FB-MS mapping and skipping results show a higher accuracy when compared to the FB-TW case, this is because the FB-MS dataset provides more user profile information to the player such as gender, age, address and other attributes that may help players in easily locating similar profiles accurately. Further, the friend relationship of Facebook and MySpace is based on mutual agreement and following relationship of Twitter is not based on mutual agreement. Therefore, the mutual agreement based relationship provides more knowledge for friends and higher accuracy. Figure 6(c) shows the over all confirmation accuracy for both the map and skip cases, which also plateau's at 3 confirmations.

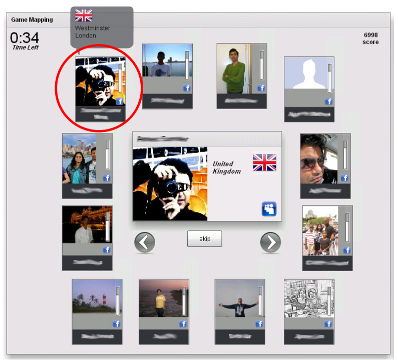
Figure 7(a) depicts the contribution of each profile attribute in verified FB-MS mapping results. Six attributes such as profile picture, first name, last name, gender, age, and country were used in comparing the profiles in the game. Note that, only 5.6% of users post exactly the same profile picture and 96.4% of users do not use a same profile picture (48.7% use similar pictures, 31.6% use different pictures, 13.7% have a profile picture in only one site, and 0.4% of the users do not have profile pictures). This shows that players mapped the same profiles based on other knowledge such as friendship information even if the two profiles did not use the same profile pictures. Last name and first name are important attributes in attribute based mapping, our results show that 74.4% of the users have the same last name and 72.8% users have the same first name. Which indicates that if the profile mapping is performed by comparing the name attributes, we expect about 73% matching accuracy. In other words, our game based mapping approach with confirmation is able to detect profile mappings for none matching profile names and provide a 27% improvement over the name based mapping. If gender and age are considered in attribute based mapping, the mapping result is not expected to increase as this usually missing or is low quality. Figure 7(b) depicts the contribution of each attribute in the verified FB-TW profile mapping results. In Twitter, only four attributes are used to compare the profiles in the game which include, profile picture, first name, last name, and country. The FB-TW attributes show a pattern similar to the FB-MS attributes. The minor difference is in the percentage of profiles that use the same profile pictures, last name and first name, where FB-TW shows higher percentages of similar profile attributes. The reason might be Facebook and Twitter are currently very popular sites. It makes many users to



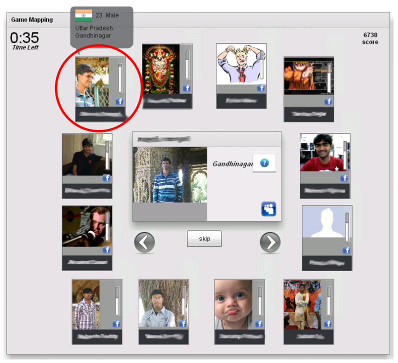
(a) Attribute Similarity in FB-MS



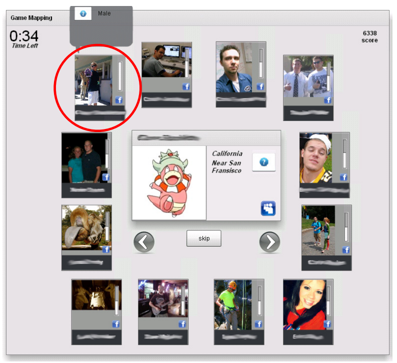
(b) Attribute Similarity in FB-TW



(c) Same Profile Photo



(d) Similar Profile Photo



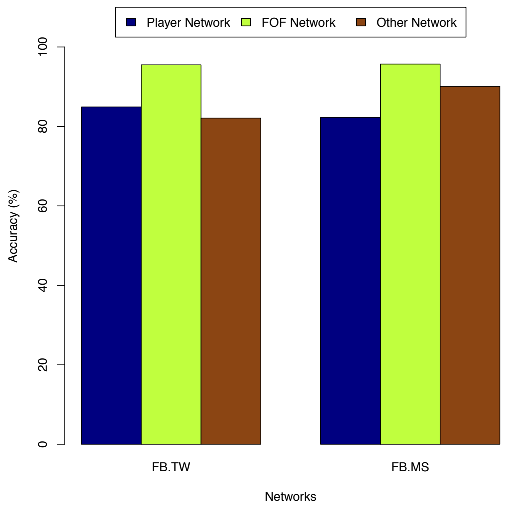
(e) Different Profile Photo

Figure 7: GameMapping Attribute and Photo Statistics

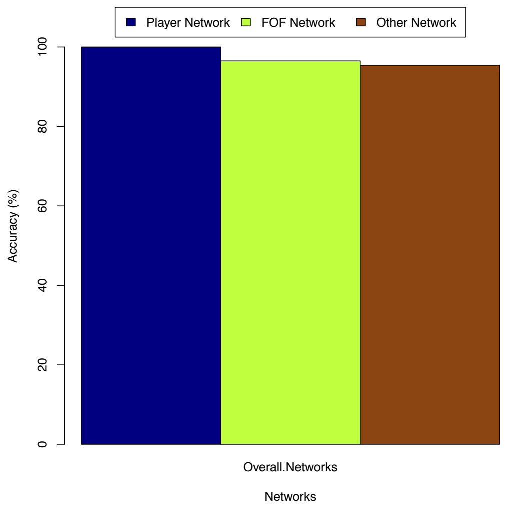
keep their profiles consistently up to date. In comparison to the name based attribute mapping, the FB-TW shows a 25% improvement in mapping accuracy. Figures 7(c)-7(e), show the possible profile mappings with respect to the same, similar and different profile photos, note that some users had the same, similar and different profile photos. The mapped user is indicated by the red circle.

To better understand how other network based approaches perform in matching the collected profile data. We used the similarity flooding graph matching approach [23], which matches profiles based on both profile attributes and network neighborhood similarity. The algorithm takes two labeled graphs (game data sets) as input and produces as output a mapping between matching profiles. We applied the collected game datasets to the similarity flooding algorithm and the generated an average matching accuracy of 74%. This result is far less than our proposed game mapping approach. The low accuracy generated by the similarity flooding approach could be attributed to the low similarity between the mapping neighborhoods which reduces the effectiveness of flooding algorithm. As indicated in Figure 7(a) and 7(b) profile attributes used in different social networks have a low degree of similarity, users do not always provide correct data or data is missing, attribute similarity is important in similarity flooding as it is used in initialization and flooding phases of the similarity flooding algorithm. In addition the neighborhood graph information for users in different social networks do not have considerable similarity in friendship connections and neighborhoods which tends to reduce the effectiveness of the flooding based similarity. On the other hand, our proposed approach provides higher accuracy due to the fact that player's map profiles not only based on the profile attributes but also based on the player's implicit knowledge about the profiles and on the reasoning behind of likelihood of mapping confirmation.

The game datasets are generated from the player's net-



(a) Accuracy and Network



(b) Accuracy of knowledgeable players

Figure 8: GameMapping Experimental Results.

work, Friend of Friend (FOF) network, and other user's network data. Figure 8(a) depicts the average accuracy of mapping results for different network types. For both FB-MS and FB-TW games, the results show that the accuracy of player network is lower than the accuracy of FOF network. The results did not meet our expectation that the accuracy of player network is higher than the accuracy of FOF network, which would be in turn higher than the accuracy of other network, since the players have more knowledge about their friends. We investigated the whole process of the game to answer the question why the accuracy of player network is lower than the accuracy of FOF network. First, we found that most players did not watch the video tutorial that is on the game homepage before they started the game. It made the players start the game without the knowledge about the game. Second, the players first played the game for their network dataset. Therefore, the players learned how to play the game while they were making incorrect or correct mappings on their network dataset. Then, they were able to play better when they played on the FOF network or other user's network game datasets. To confirm our discovered cause, we also investigated the mapping data. Figure 8(b) depicts the accuracy of knowledgeable players who knew how to play the game before starting the game. The knowledgeable players provided 100% accuracy on their network, 96.5% accuracy on FOF network, and 95.5% accuracy on other networks. It shows the players' friend relation influence on the accuracy of mapping results. The players provided higher accuracy on their friend profile mappings than unknown people's profile mappings. In summary, the game based profile mapping approach with confirmation provides better mapping results when compared to simple attribute mapping approaches. It is able to generate 100% accurate profile mappings with 3 or more mapping confirmations. Friend relation knowledge influences on the accuracy of mappings for different network types.

5.5 Irrational Player Detection Evaluation

In the initial stage of game design, we considered the irrational players and designed prevention and detection strategies as described in Section 5.2. To identify the irrational players, we calculated the mapping accuracy distribution of players as presented in Figure 9.

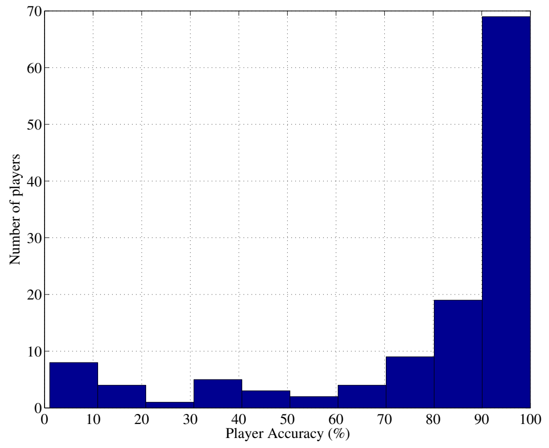


Figure 9: Accuracy Distribution of Players.

In our game period, 69 players provide over 90% mapping accuracy (18 players provided 100% mapping accuracy), and 8 players provided less than 10% mapping accuracy. We classify irrational players as either passive or active irrational players. A passive irrational player is a player that provides a small number of mapping which is lower than the average mapping of all the game players (105 mappings), and has an accuracy of 20% or less. On the other hand, an irrational player is considered active if he provides more than the average number of mappings and has an accuracy of 20% or less. Based on this classification, we discovered 12 irrational players, with 9 passive and 3 active irrational players. The passive irrational players provided 14 mappings on average, which implies that most passive irrational players did not spend much time in playing the game and left it shortly after their registration stage. There might be several reasons behind the reason for their low accuracy, one possible reason is that they did not understand the game and decided to test it out by providing random mappings. Table 1 shows a summary of the results extracted from the 3 active irrational players. The player 1 spent on average 7 seconds to map each profile and provided 130 mappings with 6.15% accuracy and the player 2 spent 0.55 seconds to map each profile and provided 551 mappings with 3.62% accuracy. Both players have low accuracy but it is evident that player 2 did not review the focus user data or the recommend user profiles instead he preferred to randomly map or skip the presented user. All the three players did play the detection game, and all of them provided 0% mapping and skipping accuracy for the detection game. Therefore, all the above 3 players were detected by the detection game strategy. Another detection strategy was based on comparing the average mapping time, where the average mapping time of the players who have accuracy above 90% was 6.7 seconds. On the other hand, the average mapping time for the irrational players was 3 seconds. This implies that rational players spend about twice the time to map profiles when compared to irrational players. Moreover, most mapping results from the irrational players did not get a confirmation, and they were not in the top 10 players.

Irrational Player	Mapping	Accuracy	Average Time
Player 1	130	6,15%	7.00 sec
Player 2	551	3.62%	0.55 sec
Player 3	2643	1.05%	1.65 sec

Table 1: Active Attackers

6 Related Work

Mapping users account across social networks is an important task that will allow users and third party applications to interact across social networks. In this paper, we divided our literature review to the following areas: attribute matching, graph matching, and human computation using games.

Without a common identity management system between different sites, attribute matching techniques are used to detect the same user in different sites by utilizing user's information. Wang et al. [35] proposed a record comparison algorithm that detects deceptive criminal identities using four personal attributes: name, date of birth, social security number and address. It calculates the overall similarity score of personal attributes. If the overall similarity score is higher than a pre-defined threshold, two people are considered a matched people. The authors also revealed that incomplete records with many missing data could significantly increase the error rate of the record comparison algorithm that is a common limitation of many identity matching techniques using only personal attributes. Jennifer et al. [39] showed that combining social features with personal features could improve the performance of criminal identity matching. They artificially constructed incomplete datasets from a complete datasets by randomly choosing a percentage of person's records and removing their data of birthday or address values. Using this incomplete dataset with a decision tree classification method, they found out if the dataset had more missing values in personal identity attribute, the social contextual features significantly increase the matching performance. This paper showed how personal attributes and social features affect the performance of the identity matching.

The graph matching problem was classified as one of the most difficult problems. In fact, many categories of graphs were classified as NP-compete problem in [16]. Exact subgraph matching problem, for example, where the number of vertices in each subgraph is the same was proven to be NP-complete by [15], however under certain constraints, where the subgraph is a tree in the big forest graph, it was proven to be resolved in polynomial time. In our paper, we consider the inexact subgraph matching problem, where the number of vertices (nodes) in each network subgraph is different, and this problem was also proved to be NP-complete by [1]. In [23] the other use a directed graph matching approach for database schema matching consisting of similarity flooding with a fixed point computation of similarity. In our paper we represent the social networks with undirected graphs.

Using human knowledge for computation while entertaining them is one of the increasing trends in the recent years. Most of the research applications of this technique is in the image labeling problem that is described in [33], where the authors created an image labeling game called the ESP Game to take advantage of the powerful vision sense and common knowledge of humans to achieve the labeling. The ESP game is played by two players without any information or link between each other but the image being labeled, and they are asked to label objects that are present in the image. Once the players agree on an object that is present in the image they will be introduced with another image and so on. Another good game that used human common knowledge for semantic annotation is PhotoSlap [17]. In PhotoSlap, the authors based their idea on the ESP game and the popular Snap card game, where the players flip cards containing random images, and *slap* each time they identify two consecutive images of the same person. In addition, the game supports the *objection* and *trap* actions to enforce truthfulness, where the players are presented with a set of images that they can set as *traps* (i.e. photos containing similar faces/heads) at the beginning of the game. Once a player *slaps*, the other players may *object* to the truthfulness of the *slap*, which is verified by the *traps* defined earlier in the game. Our idea is similar to the ESP and PhotoSlap games in the way of using human knowledge to map between user accounts using not only images, but also profile attributes, such as: age, gender, first name, last name and other attributes that might be helpful for a human to make a mapping decision.

7 Conclusion

In this paper, we presented the Game With A Purpose (GWAP) approach that solves the profile mapping problem. We provide two type of games: Facebook-MySpace (FB-MS) game and Facebook-Twitter (FB-TW) game. To detect irrational player who provide incorrect mapping intensionally, we also designed and applied an irrational player detection strategies to our game system. In our experiments, the proposed detection strategies detected irrational players effectively. It discovers the active irrational player spent 50% less time than rational players for mapping and their most mapping results did not get the agreement from other players. The evaluation of mapping results show our proposed mapping approach generate higher mapping accuracy (FB-MS: 27% improvement, FB-TW: 25% improvement) than the name based mapping results. We also observed that users are able to accurately map their friends, friend of friend and other network profiles. Finally, we showed that accurate mappings can be concluded if 3 or more rational players agree on it. In the future, we will extend this work to support other social networking sites, and to deploy the game on these sites.

8 Acknowledgments

This research was partially supported by grants from the National Science Foundation (NSF-CNS-0831360, NSF-CNS-1117411) and a Google Research Award.

References

[1] ABDULKADER, A. M. Parallel algorithms for labelled graph matching. *PhD thesis, Colorado School of Mines.* (1998).

[2] AUWATANAMONGKOL, S. Inexact graph matching using a genetic algorithm for image recognition. *Pattern Recognition Letters 28*, 12 (2007), 1428 – 1437.

[3] B. SCHWARTZ. How to connect twitter to facebook status updates, `http://www.ehow.com/how_4668396_connect-twitter-facebook-status-updates.html`, 2010.

[4] BASIN, D. A. A term equality problem equivalent to graph isomorphism. *Inf. Process. Lett. 51*, 2 (1994).

[5] BLONDEL, V. D., GAJARDO, A., HEYMANS, M., SENELLART, P., AND DOOREN, P. V. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM* (2004).

[6] BORGATTI, S. P., AND EVERETT, M. G. A graph-theoretic perspective on centrality. *Social Networks 28*, 4 (October 2006), 466–484.

[7] CESAR, R., BENGOETXEA, E., AND BLOCH, I. Inexact graph matching using stochastic optimization techniques for facial feature recognition. *Pattern Recognition, International Conference on 2* (2002), 20465.

[8] CESAR, JR., R. M., BENGOETXEA, E., BLOCH, I., AND LARRA NAGA, P. Inexact graph matching for model-based recognition: Evaluation and comparison of optimization algorithms. *Pattern Recogn. 38*, 11 (2005), 2099–2113.

[9] DOAN, A., MADHAVAN, J., DOMINGOS, P., AND HALEVY, A. Ontology matching: A machine learning approach. In *Handbook on Ontologies in Information Systems* (2004), Springer-Verlag.

[10] DONG, X., HALEVY, A., MADHAVAN, J., NEMES, E., AND ZHANG, J. Similarity search for web services. *VLDB* (2004).

[11] FAGIN, R., LOTEM, A., AND NAOR, M. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences 66* (2002).

[12] FAN, K.-C., LU, J.-M., AND CHEN, G.-D. A feature point clustering approach to the recognition of form documents. *Pattern Recognition 31*, 9 (1998).

[13] FARMVILLE GAME. Zynga game network inc., `http://www.facebook.com/FarmVille`, 2010.

[14] FRANKY BRANCKAUTE. Twitter's Meteoric Rise Compared to Facebook [Infographic]. The Blog Herald, June 2010.

[15] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, January 1979.

[16] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[17] HO, C., HSIANG, T., AND HSU, J. Photoslap: A multi-player online game for semantic anotation. *AAAI'07* (2007).

[18] KO, M. N., CHEEK, G., SHEHAB, M., AND SANDHU., R. Social-networks connect services. *IEEE Computer 43*, 8 (aug. 2010), 37 –43.

[19] LIU, B. *Web DataMining Exploring Hyperlinks, Contents, and Usage Data*. Springer-Verlag, 2007.

[20] M. GUMMELT. Publishing to twitter from facebook pages, `http://blog.facebook.com/blog.php?post=123006872130`, 2010.

[21] MADHAVAN, J., BERNSTEIN, P., CHEN, K., HALEVY, A., AND SHENOY, P. Corpus-based schema matching. In *In ICDE* (2003), pp. 57–68.

[22] MCADAMS, R. H. A focal point theory of expressive law. *Virginia Law Review 86*, 8 (2000), pp. 1649–1729.

[23] MELNIK, S., GARCIA-MOLINA, H., AND RAHM, E. Similarity flooding: A versatile graph matching algorithm. In *Ontology handbook* (2002), pp. 117–128.

[24] NEWMAN, M. E. J. Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. *Physical Review E 64*, 1 (June 2001).

[25] NEWMAN, M. E. J. The structure and function of complex networks. *SIAM Review 45*, 2 (2003), 167–256.

[26] OSBORNE, M. J., AND RUBINSTEIN, A. *A Course in Game Theory*. The MIT Press, July 1994.

[27] P. BOUTIN. Investors hot on social shopping: First groupon, now livingsocial, `http://venturebeat.com/2010/04/29/livingsocial-funding`, 2007.

[28] PATRIQUIN, A. Connecting the social graph: Member overlap at opensocial and facebook, . `http://blog.compete.com/2007/11/12/`, Nov 2007.

[29] RAHM, E., AND BERNSTEIN, P. A. A survey of approaches to automatic schema matching. *VLDB* (2001).

[30] REALWIRE. Social networking sites: Almost two thirds of users enter false information to protect identity, `http://www.realwire.com/release_detail.asp?ReleaseID=6671`, 2007.

[31] RECORDON, D., AND REED, D. Openid 2.0: a platform for user-centric identity management. In *DIM '06: Proceedings of the second ACM workshop on Digital identity management* (NY, USA, 2006), pp. 11–16.

[32] SUGDEN, R. A theory of focal points. *Economic Journal 105*, 430 (May 1995), 533–50.

[33] VON AHN, L., AND DABBISH, L. Labeling images with a computer game. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems* (NY, USA, 2004), pp. 319–326.

[34] VON AHN, L., LIU, R., AND BLUM, M. Peekaboom: a game for locating objects in images. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems* (NY, USA, 2006).

[35] WANG, G. A., CHEN, H., XU, J. J., AND ATABAKHSH, H. Automatically detecting criminal identity deception: An adaptive detection algorithm. *IEEE Trans. on Systems, Man and Cybernetics, Part A* (2005).

[36] WITTEN, I. H., AND FRANK, E. *Data Mining: Practical Machine Learning Tools and Techniques*, 2 ed. Morgan Kaufmann, 2005.

[37] WITTEN, I. H., AND FRANK, E. *Data Mining: Practical Machine Learning Tools and Techniques*, second ed. Morgan Kaufmann, 2005.

[38] WU, H., CHEN, Q., AND YACHIDA, M. Face detection from color images using a fuzzy pattern matching method. *IEEE Transactions on Pattern Analysis and Machine Intelligence 21*, 6 (1999), 557–563.

[39] XU, J., WANG, G. A., LI, J., AND CHAU, M. Complex problem solving: identity matching based on social contextual information. *Journal of the Association for Information Systems, Vol 8, Issue 10* (Oct 2007).

[40] ZAGER, L. *Graph Similarity and Matching*. Master dissertation, MIT, 2005.

Hybrid Cloud Support for Large Scale Analytics and Web Processing

Navraj Chohan Anand Gupta Chris Bunch Kowshik Prakasam
Chandra Krintz
Computer Science Department
University of California, Santa Barbara, CA

1 Abstract

Platform-as-a-service (PaaS) systems, such as Google App Engine (GAE), simplify web application development and cloud deployment by providing developers with complete software stacks: runtime systems and scalable services accessible from well-defined APIs. Extant PaaS offerings are designed and specialized to support large numbers of concurrently executing web applications (multi-tier programs that encapsulate and integrate business logic, user interface, and data persistence). To enable this, PaaS systems impose a programming model that places limits on available library support, execution duration, data access, and data persistence. Although successful and scalable for web services, such support is not as amenable to online analytical processing (OLAP), which have variable resource requirements and require greater flexibility for ad-hoc query and data analysis. OLAP of web applications is key to understanding how programs are used in live settings.

In this work, we empirically evaluate OLAP support in the GAE public cloud, discuss its benefits, and limitations. We then present an alternate approach, which combines the scale of GAE with the flexibility of customizable offline data analytics. To enable this, we build upon and extend the AppScale PaaS – an open source private cloud platform that is API-compatible with GAE. Our approach couples GAE and AppScale to provide a hybrid cloud that transparently shares data between public and private platforms, and decouples public application execution from private analytics over the same datasets. Our extensions to AppScale eliminate the restrictions GAE imposes and integrates popular data analytic programming models to provide a framework for complex analytics, testing, and debugging of live GAE applications with low overhead and cost.

2 Introduction

Cloud computing has revolutionized how corporations and consumers obtain compute and storage resources. Infrastructure-as-a-service (IaaS) facilitates the rental of virtually unlimited IT infrastructure on-demand with high availability. Service providers, such as Amazon AWS [1] and Rackspace [28], consolidate and share vast resource pools across large numbers of users, who employ these resources on demand on a pay-per-use basis. Customers provision virtual machines (VMs) via API calls or browser portals, which they then configure, connect, monitor, and manage manually according to their software deployment needs.

Platform-as-a-service (PaaS) offerings, such as Microsoft Azure [2] and Google App Engine [16], automate configuration, deployment, monitoring, and elasticity by abstracting away the infrastructure through well-defined APIs and a higher-level programming model. PaaS providers restrict the behavior and operations (libraries, functionality, and quota-limit execution) of hosted applications, both to simplify cloud application deployment, and to facilitate scalable use of the platform by very large numbers of concurrent users and applications. Google App Engine (GAE), the system we focus on herein, currently supports over 7.5 billion page views per day across over 500,000 active applications [15] as a result of their platform's design. As is the case for public IaaS systems, public PaaS users pay only for the resources and services they use.

A key functionality lacking from the original design of PaaS systems is online analytics processing (OLAP). OLAP enables application developers to model, analyze, and identify patterns in their online web applications as users access them. Such analysis helps developers target specific user behavior with software enhancements (code/data optimization, improved user interfaces, bug fixes, etc.) as well as applying said analysis for commercial purposes (e.g. marketing and advertising). These improvements and adaptations are crucial to building a customer base, facilitating application longevity, and ultimately commercial success for a wide range of companies. In recognition of this need, PaaS systems are increasingly offering new services that facilitate OLAP execution models by and for applications that execute over them [17, 26, 3]. However, such support is still in its infancy and is limited in flexibility, posing questions as to what can be done within quota limits and how the service connects with the online applications they analyze.

In this paper, we investigate the emerging support of OLAP for GAE, identify its limitations, and its impact on the cost and performance of applications in this setting. We propose an alternate approach to OLAP, in the form of a hybrid cloud consisting of a public cloud executing the live web application or service and a remote analytics cloud which shares application data. To enable this, we build upon and extend AppScale, an open source PaaS offering that is API-compatible with GAE. AppScale executes over a variety of infrastructures using VM-based application and component isolation. This portability gives developers the freedom and flexibility to explore, research, and tinker with the system level details of cloud platforms [9, 10, 22]. Our hybrid OLAP solution provides multiple options for data transfer between the two clouds, facilitates deployment of the analytics cloud over Amazon's EC2 public cloud or an on-premise cluster, and integrates the popular Hive distributed data warehousing technology to enable a wide range of complex analytics applications to be performed over live GAE datasets. By using a remote AppScale cloud for analytics of live data, we are able to specialize it for this execution model and avoid the quotas and restrictions of GAE, while maintaining the ease of use and familiarity of the GAE platform.

In the sections that follow, we first provide background on GAE and AppScale. We then describe the design and implementation of our hybrid OLAP system. We follow this with an evaluation of existing solutions for analytics, our Hive processing, and an analysis of the cost and overhead of cross-cloud data synchronization. Finally, we present related work and conclude.

3 Background

Google App Engine was released in 2008, with the goal of allowing developers to run applications on Google's infrastructure via a fully managed and automatically scaled system. While the first release only supported the Python programming language, the GAE team has since introduced support for the Java and Go languages. Application developers can access a variety of different services (cf., Table 1) via a set of well-defined APIs. The API implementations in the GAE public cloud are optimized for scalability, shared use, and fault tolerance. The APIs that we focus on in this paper are the Datastore (for data persistence), URL Fetch (for communication), and Task Queues (for background processing).

Table 1: Google App Engine APIs.

Name	Description
Datastore	Schemaless object storage
Memcache	Distributed caching service
Blobstore	Storage of large files
Channel	Long lived JavaScript connections
Images	Simple image manipulation
Mail	Receiving and sending email
Users	Login services with Google accounts
Task Queues	Background tasks
URL Fetch	Resource fetching with HTTP request
XMPP	XMPP-compatible messaging service

AppScale is an open source implementation of the GAE APIs that was released in early 2009, enabling users to run GAE applications on their local cluster or over the Amazon EC2 public IaaS cloud. AppScale implements the APIs in Table 1 using a combination of open source technologies and custom software. It provides a database-agnostic layer, which multiple disparate database/datastore technologies (e.g. Cassandra, HBase, Hypertable, MySQL cluster, and others) can plug into [6]. It implements the Task Queue API by executing a task on a background thread in the same application server as the application instance that makes the request. This support, though simple, is inherently inefficient and not scalable, because it is neither distributed nor load-balanced. Moreover, it does not share state between application servers, which leads to incorrect application behavior when more than one application server is present. We replace this API implementation as part of this work, addressing this limitation.

3.1 App Engine Analytics Libraries

The Task Queue API facilitates the use of multiple, independent user-defined queues, each with a rate limit of 100 tasks per second (which can be increased in some cases [16]) in GAE. A task consists of an application URL, which is called by the system upon task dequeue. A 200 HTTP response code (OK) indicates that the task completes successfully. Other HTTP codes cause reenqueuing of the task for additional execution attempts. The number of retries, a time delay, and a task name can be optionally specified by developers as part of the task when it is enqueued. Use of task names is important to prevent the same task from being enqueued multiple times (the lack of such measures can result in a task fork bomb, in which a task is infinitely enqueued). One way to circumvent the 10 minute time limit for a task is to chain tasks, in which the initial task performs a portion of the work, and enqueuing another task to resume where

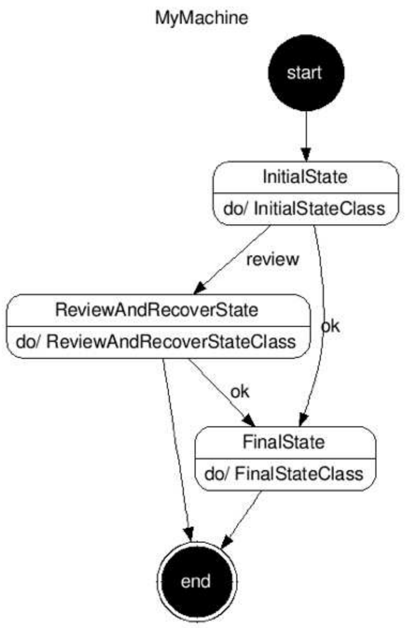


Figure 1: An example state machine in Fantasm.

it has left off. Tasks should be idempotent, or only perform side effects (e.g., updating shared, persistent data) as the final operation – since any failure of a previous statement will cause the task to be re-enqueued (potentially updating shared state incorrectly).

GAE application developers are responsible for program/task correctness when failures occur. This requires that developers make proper use of task names and chaining, and implement tasks that are idempotent. Doing so for all but the most trivial of applications can be a challenging undertaking for all but expert developers. To address this limitation, there are libraries that provide a layer of abstraction over the GAE task queue interface and implementation. These libraries are Fantasm [14], GAE Pipeline [26], and GAE MapReduce [17]. Each automates naming and failure handling by saving intermediate state via the Memcache and the Datastore APIs.

Fantasm, based on [18], employs a programming model that is based on finite state machines (FSM). A programmer describes a state machine via the YAML markup language by identifying states, events, and actions. The initial state typically starts with a query to the datastore, to gather input data for analysis. Fantasm steps through the query and constructs a task for each entity (datastore element) that the query processes in each state. Optionally, there can be a fan-in state, which takes multiple previous states and combines them via a reduction method. Figure 1 shows an example FSM. A limitation of Fantasm is how it iterates through data. It does not shard datasets, but instead, pages through a query serially, leading to inefficient execution of state machines.

```
class WCUrl(pipeline.Pipeline):
  def run(self, url):
    r = urlfetch.fetch(url)
    return len(r.data.split())

class Sum(pipeline.Pipeline):
  def run(self, *values):
    return sum(values)

class MySearchEngine(pipeline.Pipeline):
  def run(self, *urls):
    results = []
    for u in urls:
      # Do word count on each URL
      results.append((yield WCUrl(u)))
    yield Sum(*results) # Barrier waits
```

Figure 2: Code example of Pipeline parallellizing work.

The GAE Pipeline library facilitates chaining of tasks into a workflow. Pipeline stages (tasks) yield for barrier synchronization, at which point the output is unioned and passed onto the next stage in the pipeline. Figure 2 shows an example of parallel processing via Pipeline that counts the number of unique words on multiple web pages. The *yield* operator spawns background tasks, whose results are combined and passed to the *Sum* operation. Implementing similar code via just the Task Queue API is possible, but is more complicated for users.

The GAE MapReduce library performs parallel processing and reductions across datasets. Mapper functions operate on a particular kind of entity and reducer functions operate on the output of mappers. Alternative input readers (e.g. for use of Blobstore files) and sharding support is also available. The GAE MapReduce library uses the Task Queue API for its implementation, as opposed to using Google's internal MapReduce infrastructure or Hadoop, an open source implementation. Both are more flexible than GAE MapReduce, and allow for a wider range of analytics processing than this library. Currently, a key limitation of GAE MapReduce is that all entities in the Datastore are processed, even when they are not of interest to the analysis.

Each of these abstractions for background processing and data analytics in GAE introduce a new programming model with its own learning curve. Moreover, analytics processing on the dataset is intertwined with the application, (that users use to produce/access the dataset) which combines concerns, can introduce bugs, and can have adverse affects on programmer productivity, user experience, and monetary cost of public cloud use. To address these limitations, we investigate an alternate approach to performing online data analytics for applications executing within GAE that employs a combination of GAE and AppScale concurrently.

4 Hybrid PaaS Support for Web Application Data Analysis

In this work, we investigate how to combine two PaaS systems together into a hybrid cloud platform that facilitates the simple and efficient execution of large-scale analysis of live web application data. Our hybrid model executes the web application on the GAE public cloud platform, synchronizes the data between this application/platform and a remote AppScale cloud, and facilitates analysis of the live application data using the GAE analytics libraries, as well as other popular data processing engines (e.g. Hadoop/Hive) using AppScale. Users can deploy AppScale on a local, on-premise cluster, or over Amazon EC2. In this section, we overview the two primary components of our hybrid cloud system: the data synchronization support and the analytics processing engine. We then discuss our design decisions and how our solution works within the restrictions of the GAE platform.

4.1 Cross-Cloud Data Synchronization

The key to our approach to analytics of live web applications is the combined use of GAE and AppScale. Since the two cloud platforms share a common API, applications that execute on one can also do so on the other, without modification. This portability also extends to the data model. That is, given the compatibility between AppScale and GAE, we can move data between the two different platforms for the same application. We note that for vast datasets such an approach may not be feasible. However, it is feasible for a large number of GAE applications today. The cross-platform portability facilitates and simplifies our data synchronization support, and makes it easier for developers to write application and analytics code, because the runtime, APIs, and code deployment process is similar and familiar.

We consider two approaches to data synchronization: bulk and incremental data transfer. For bulk transfer, GAE currently provides tools as part of its software development kit (SDK) to upload and download data into and out of the GAE datastore en masse. We have extended AppScale with similar functionality. Our extensions provide the necessary authentication and data ingress/egress support, as well as support for the GAE Remote API [16], which enables remote access to an application's data in the datastore. The latter must be employed by any application for which hybrid analytics will be used. Using the Remote API, a developer can specify what data can be downloaded (the default is all). Bulk download from, and upload to, is subject to GAE monetary charges for public cloud use.

There are several limitations to bulk data transfer as a mechanism for data synchronization between the two application instances. First, in its current incarnation, transfer is all or nothing (of the entities specified). As such, we are able to only perform analytics off-line or postmortem if we are to copy the dataset once (the most inexpensive approach). To perform analytics concurrently with web application execution, we are forced to download the same data repeatedly over time (as the application changes it). This can be both costly and slow. Finally, the data upload/download tools from GAE are slow and error prone, with frequent interruptions and data loss.

To address these limitations, we investigate an alternative approach to synchronizing data between GAE and AppScale: incremental data transfer. To enable this, we have developed a library for GAE applications that runs transparently in both GAE and AppScale. Our incremental data transfer library intercepts all destructive operations (writes and deletes) and communicates them to the AppScale analytics cloud. In our current prototype, we do not support the limited form of transactions that GAE applications can perform [13]. As part of our ongoing and future work, we are considering how to reflect committed transactional updates in the AppScale analytics cloud. Developers specify the location of the AppScale analytics cloud as part of their GAE application configuration file. Since the library code executes as part of the application in GAE, it must adhere to all of the GAE platform restrictions. Furthermore, communication to the AppScale analytics cloud is subject to GAE charges for public cloud use.

Our goal with this library is to avoid interruption or impact on GAE web application performance and scale, from the users' perspective. We consider two forms of synchronization with different consistency guarantees: eventual consistency (EC) and best effort (BE). EC incremental transfer uses the Task Queue API to update the AppScale analytics cloud. Using this approach, the library enqueues a background task in GAE upon each destructive datastore operation. The task then uses the URL Fetch library to synchronously transmit the updated entity. In GAE, tasks are retried until they complete without error. Thus, GAE and AppScale data replicas for the application are eventually consistent, assuming that both the GAE and AppScale platforms are available.

Our second approach, best effort (BE), for incremental transfer implements an asynchronous URL Fetch call to the AppScale analytics cloud for the application upon each destructive update. If this call fails, the GAE and AppScale replicas will be inconsistent until the next time the same entity is updated. The BE approach can implement potentially fewer transfers since failed transfers are not retried. This may impact the cost of hybrid cloud analytics using our system. BE is useful for settings in which perfect consistency is not needed.

To maintain causal ordering across updates we employ a logical clock (a Lamport clock [23]), ensuring that only the latest value is reflected in the replicated dataset for each entity. Using this approach, it is possible that at any single point in time there may be an update missing (still in flight due to retries in EC or failed in BE) in the replicated dataset. We transmit entity updates as Protocol Buffers, the GAE transfer format of Datastore entities.

4.2 Analytics Processing Engine within AppScale

We next consider different implementations of the AppScale analytics processing engine. We first extend AppScale to support each of the three analytics libraries that GAE supports, described in Section 3.1. We start by replacing the TaskQueue API implementation in AppScale, from a simple, imbalanced approach, to a new software layer, similar to that for the Datastore API implementation and transaction support [9], that is implementation-agnostic and allows different task queue implementations to be plugged in and experimented with.

The GAE Task Queue API includes the functions:

```
AddTask(name, url, parameters)
DeleteTask(name)
PurgeQueue()
```

We emulate the GAE behavior of this API (that we infer using the GAE SDK and by observing the behavior of GAE applications) in our task queue software layer within AppScale. Each task that is added to the queue specifies a *url* that is a valid path (URL route) defined in the application, to which a POST request can be made using the *parameters*. The *name* argument ensures that a task is only enqueued once given a unique identifier. If a name is not supplied, a unique name is assigned to it. The *PurgeQueue* operation will remove all tasks from a queue, resetting it to an initial, empty state, whereas *DeleteTask* will remove a named task if it is still enqueued. Task execution code is within the application itself (a relative path), or can be a fully remote location (a full path). Successful execution of a task is indicated by a HTTP 200 response code. The task queue implementation retries failed tasks up to a configurable number of times, defaulting to ten attempts.

The AppScale Task Queue interface for plugging in new messaging systems is as follows: This API includes the functions:

```
EnqueueTask(app_name, url, parameters)
LocateTask(app_name, task_name)
AddTask(app_name, task_name)
AckTask(app_name, task_name, reenqueue)
PurgeQueue(app_name)
```

The *AddTask* function stores the given task name and state in the system-wide datastore. Possible task states are 'running', 'completed', or 'failed', and states can be retrieved via *LocateTask*). *AckTask* tells the messaging system whether the task should be re-enqueued, and if it should be, the messaging system increments the retry count associated with that task. Each function requires the application name because AppScale supports multiple applications per cloud deployment, isolating such communications.

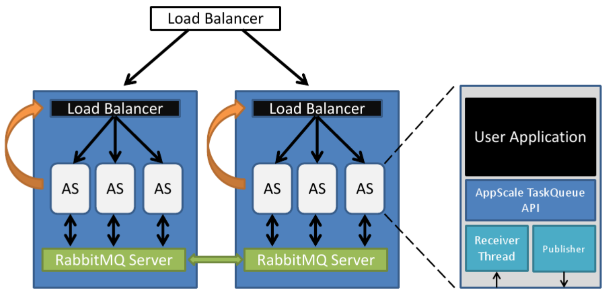


Figure 3: Overview of RabbitMQ implementation in AppScale.

Using the AppScale task queue software layer, we plug-in the VMWare RabbitMQ [27] technology and implement support for each of the GAE analytics libraries (GAE MapReduce, GAE Pipeline, and Fantasm) described in Section 3.1 on top of the Task Queue API. We have chosen to integrate RabbitMQ due to its widespread use and multiple useful features within a distributed task queue implementation, including clustering, high availability, durability, and elasticity. Figure 3 shows the software architecture of RabbitMQ as a task queue within AppScale (two nodes run a given application in this figure). Each AppScale node that runs the application (load-balanced application servers) runs a RabbitMQ server. Each application server has a client that can enqueue tasks or listen for assigned tasks (a callback thread) to or from the RabbitMQ server. We store metadata about each task (name, state, etc.) in the system in the cloud datastore. A worker thread consumes tasks from the server. Upon doing so, it issues a POST request to its localhost or full path/route (if specified), which gets load-balanced across application servers running on the nodes. Tasks are distributed to workers in a round-robin basis, and are retried upon failure. RabbitMQ re-enqueues failed tasks and is fault tolerant.

In addition to the Task Queue, MapReduce, Pipeline, and Fantasm APIs, we also consider a processing engine that is popular for large-scale data analytics yet that is not available in GAE. This processing engine employs a combination of MapReduce [12] (not to be confused with GAE MapReduce, which exports different semantics and behavioral restrictions) and a query processing engine that maps SQL statements to a workflow of MapReduce operations. In this work, we employ Hadoop, an open source implementation of a fully featured MapReduce system, and Hive [29, 25, 20], an open source query processing engine, similar in spirit to Pig and Sawzall. This processing engine (Hive/Hadoop) provides users with ad-hoc data querying capabilities that are processed using Hadoop, without requiring any knowledge about how to write or chain MapReduce jobs. Moreover, using this AppScale service, users can operate on data using the familiar syntax of SQL and perform large-scale, complex data queries using Hadoop.

AppScale integrates multiple datastore technologies, including Cassandra, Hypertable, and HBase [6, 7]. All of these datastores are distributed, scalable, fault-tolerant, and provide column-oriented storage. Each datastore provides a limited query language, with capabilities similar to the GAE Datastore access model: entities, stored as Protocol Buffers, are accessed via keys and key ranges. We focus on the currently best performing datastore in this work, Cassandra [9].

Our extensions swap out the Hadoop File System (HDFS) in AppScale and replace it with CassandraFS [5], an HDFS-compatible storage layer, that interoperates directly with Cassandra, with the added benefit of having no single points of failure within its NameNode process. Above CassandraFS, we deploy Hadoop; above Hadoop, we deploy Hive. Developers can issue Hive queries from the command line, a script issued on any AppScale DB node [22], or via their applications through a library, similar to the GAE MapReduce library implementation in AppScale.

To enable this, we modified the datastore layout of entities in the AppScale datastore. Previously, we employed a single column-family (table) for all kinds of entities in an applications dataset. We shared tables across multiple applications and we isolated datasets using namespaces prepended to the key names. In this work, we store column-families for each kind of entity. The serialization and deserialization between Hadoop, CassandraFS, and Cassandra happens through a custom interface, which enables Hadoop mappers and reducers to read and write data from Cassandra. We extended the AppScale Datastore API with a layer that translates entities to/from Protocol Buffers. Our extensions eliminate the extract-transform-load step of query processing so that entities can be processed in place.

This support enables Hive queries to run SQL statements which are partitioned into multiple mapper and reducer phases. Hive compiles SQL statements into a series of connected map and reduce jobs. Analysts can perform queries that are automatically translated to mappers and reducers, rather than manually writing these functions and chaining them together. Take for example the task of getting the total count of entities of a certain kind. A Hive query is as simple as:

us-east-1	Northern Virginia, USA
eu-west-1	Dublin, Ireland
ap-southeast-1	Singapore
ap-northeast-1	Tokyo, Japan
sa-east-1	Sao Paulo, Brazil
us-west-1	Oregon, USA
us-west-2	California, USA

Table 2: EC2 Regions for Amazon Web Services.

```
SELECT COUNT(*) FROM appid_kind;
```

To to the same thing in GAE, the entities are paged through and a counter incremented. Note that the Google Query Language for GAE applications limits the number of entities in a single fetch operation to 1000. If the dataset is large enough, then the developer must use a background task or manually implement task queue chaining. Another alternative approach is to use sharded counters to keep a live count; multiple counter entities are required if the increment must happen at a rate faster than once per second. Both methods are foreign to many developers and are far more complex and non-intuitive than simple SQL Hive statements.

5 Evaluation

In this section, we evaluate multiple components of our hybrid web application and analytics system. We first start with an evaluation of the cross-cloud connectivity within a hybrid cloud deployment. For this, we analyze the round-trip time (RTT) between a deployed GAE application in Google datacenters and virtual machines deployed globally across multiple regions and availability zones of Amazon EC2. We next evaluate the performance of the GAE libraries for analytics using the GAE public cloud. We then evaluate the efficacy of our extensions to the AppScale TaskQueue implementation. Lastly, we show the efficiency of using the AppScale analytic solution running Hive over Cassandra.

5.1 Cross Cloud Data Transfer

To evaluate the performance of cross-cloud data synchronization between GAE and AppScale, we must first understand the connectivity rate between them for incremental data transfer (cf Section 4.1). To measure this, we deploy an application in the GAE public cloud that we access remotely from multiple Amazon EC2 micro instances in 16 different availability zones, spanning seven

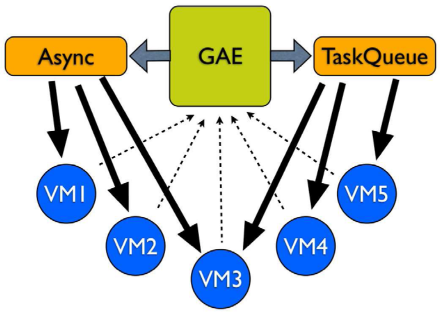


Figure 4: Experimental Setup for Measuring Round-trip Time and Bandwidth Between a GAE Application and VMs in Multiple EC2 Regions.

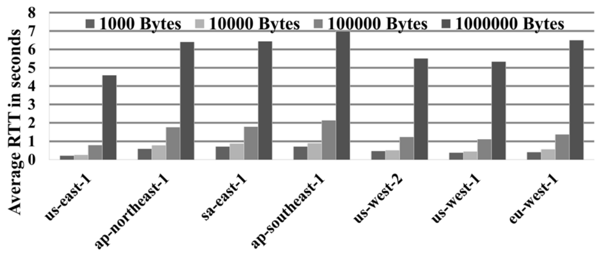


Figure 5: Round-trip Time Per Different Packet Size.

regions. Figure 2 shows the regions we consider, and Figure 4 depicts our experimental setup.

Our experiment issues a HTTP POST request from the EC2 instances, each with a data payload of a particular size, a destination URL location, a unique identifier, and the type of hybrid data synchronization to employ: eventually consistent (EC) or best effort (BE). The sizes we consider are 1KB, 10KB, 100KB, and 1MB (the maximum allowed for GAE's Datastore API). The EC2 instances host a web server, which receives the data from the GAE application (either from a task via EC or from the application itself via BE) and records the current time and request identifier. Figure 5 shows the average RTT for different packet sizes, for each availability zone. The data indicates that it is advantageous to batch updates when possible since there is not a linear relationship between size and RTT, as sizes grow.

We next consider whether the geographical location of the AppScale cloud (different EC2 regions) makes a significant difference in the communication overhead on data synchronization. To evaluate this, we consider the average round-trip time (RTT) and bandwidth across payload sizes to the GAE application for the different regions (Figure 6). The US East region had the RTT

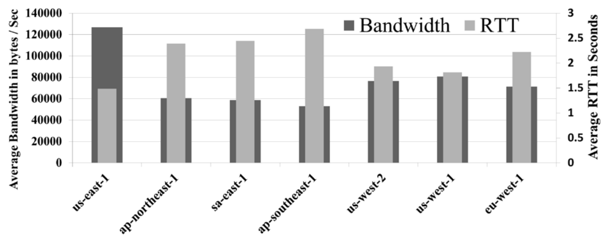


Figure 6: Round-trip Time and Bandwidth Between a GAE Application and Different EC2 Regions.

with the highest bandwidth, by a factor of two. Both US regions have the next best performing communication behavior. This data suggests that our GAE application is hosted (geographically) in GAE in the Eastern US. Locality to the application shows more than 2x the bandwidth for the US East availability zone than other zones (130KB versus 50KB to 80KB for other zones). We investigated this further and found via traceroutes and pings that the application was located near or around New York. We also found with this experiment that bandwidth over time is generally steady, with the exception of between the hours of 16:00 and 22:00 (figure not shown). It may be possible to take advantage of such information to place the AppScale cloud to enable more efficient data synchronization.

We next investigated the task queue delay in GAE. We are interested in whether the delay changes over time or remains relatively consistent. We present this data in Figure 7, as points at each hour in the day (normalized to Eastern Standard Time) that we connect using lines to help visualize the trends. The left x-axis is RTT in seconds for the region, and the right x-axis is the average queue delay (in seconds) for the region. Queue delays do vary but this variance (impact on RTT) is most perceptible during the early evening hours in all regions.

Finally, we compare our two methods for synchronization: EC and BE. EC uses a combination of the Task Queue API and synchronous URLFetch API; the use of the former ensures that all failed tasks are retried until they are successful. BE uses asynchronous URLFetch for all destructive updates and does not retry upon failures.

We ran the experiment for seven days and sent a total of 1195288 requests. Out of the 597644 packets (half of the total packets) sent via the TaskQueue option, 11679 were duplicates (unnecessary transfers). The asynchronous URLFetch experienced 10 duplicate packets suggesting the URLFetch API will retry in some cases from within the lower layers of the API implementation as needed. We experienced no update loss using EC and 5 updates lost for BE.

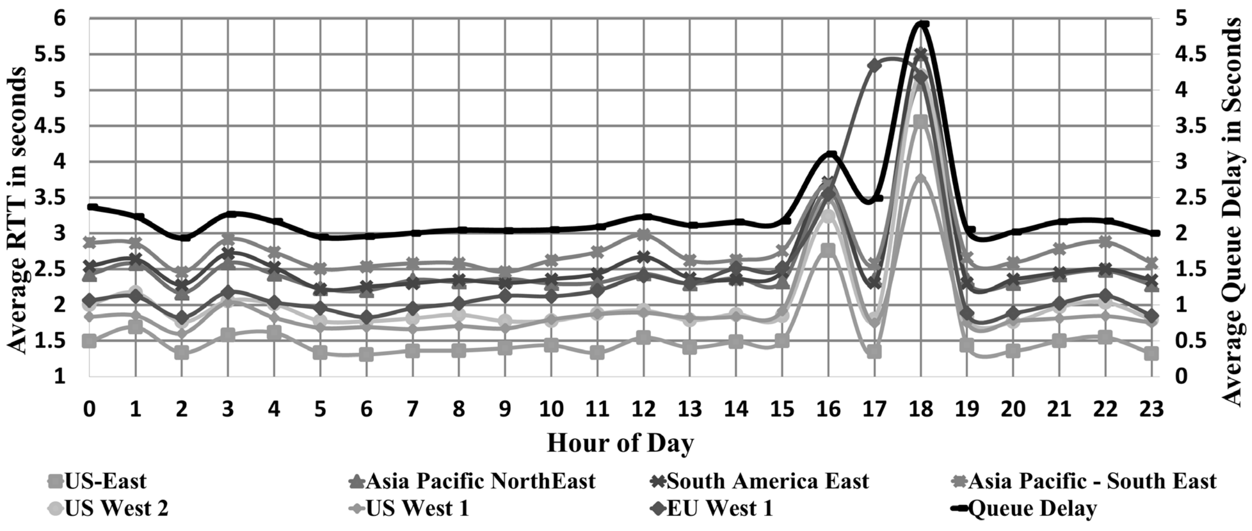


Figure 7: Round-trip time from multiple regions to a deployed GAE application with task queue delay.

5.2 Benchmarks

We next consider the performance of five different and popular analytics benchmarks: wordcount, join, grep, aggregate, and subset aggregate. Wordcount counts the number of times a unique word appears. Join takes two separate tables and combines them based on a shared field. Grep searches for a unique string for a particular substring. Aggregate gives the summation of a field across a kind of entity, while subset aggregate does the same, but for a portion of the entire dataset (one percent for this benchmark). We implemented each benchmark using the Fantasm, Pipeline, and MapReduce GAE libraries, as well as a Hive query.

5.3 Google App Engine Analytics

For the experiments in this section, we execute each benchmark five times and present the average execution time and standard deviation. We use the automatic GAE scaling thresholds, and had billing enabled. We considered experiments with 100, 1000, 10000, and 100000 entities in the datastore. We attempted even higher numbers of entities, but the running time for each trial became infeasible to get complete results.

The tables in 3 shows the results for all of the benchmarks. The Fantasm implementation shows a large latency for a significant numbers of entities, and compared to Pipeline, is 6X to 30X slower. This is due to the fact that Fantasm's execution model has a task for each entity, so it must do paging through the query[1]. Pipeline, by comparison, retrieves a maximum of 1000 entities at a time from the datastore, reducing the amount of time spent querying the database. Pipeline does not see much latency increases from 100 to 1000 entities, because both require only a single fetch from the datastore, and the difference lays in the summation. MapReduce also deals in batches, but the size of the batch depends on the number of shards. When the number of entities went from 100 to 1000 for MapReduce, the growth in latency was over 5X because the number of shards was one. 10000 entities, on the other hand, had 10 shards, and therefore did more work in parallel, seeing an increase in less than half the time. Pipeline has an advantage because of its ability to combine multiple entity values before doing a transactional update to the datastore, whereas both MapReduce and Fantasm are incrementing the datastore transactionally for each entity. For the implementation, the counter was sharded to ensure that there was high write throughput for increments.

Pipeline shows less overhead for Grep as compared to Aggregate (100-1000) because it uses half as many Pipeline stages. In the aggregate Pipeline implementation, there was an initial Pipeline which does the query fetches to the datastore, and another for incrementing the datastore in parallel after combining values. Grep, by comparison, does not need require combining or transactional updates, as required for the counter update in aggregate. Counter updates require reading the current value, incrementing it, and storing it back. Aggregate vs Grep MapReduce has a similar behavior to Pipeline because each mapper does not require transactional updates.

The Join benchmark combines two different entity kinds to create a new table. The Join results show similar trends as Aggregate and Grep. During the exper-

[1]The Fantasm library, since the writing of this paper, has added the ability to do batch fetches for better performance.

	100	1000	10000	100000
Fantasm	13.80 ± 1.61	110.29 ± 4.70	1148.24 ± 86.20	11334.59 ± 1047.57
Pipeline	2.46 ± 0.86	3.05 ± 0.32	11.08 ± 0.50	98.34 ± 3.82
MapReduce	9.34 ± 0.35	57.36 ± 8.96	104.56 ± 17.83	377.70 ± 63.35

Aggregate

	100	1000	10000	100000
Fantasm	10.85 ± 0.77	121.21 ± 21.07	1819.86 ± 1175.19	10360.40± 396.56
Pipeline	2.40 ± 1.26	2.663 ± 0.51	9.77 ± 0.72	98.89 ± 13.76
MapReduce	2.73 ± 0.30	4.56 ± 0.09	24.05 ± 0.30	227.57 ± 20.76

Grep

	100	1000	10000	100000
Fantasm	10.71 ± 1.22	109.83 ± 4.90	977.23 ± 80.34	10147.75 ± 1106.15
Pipeline	4.54 ± 2.34	14.48 ± 5.22	44.11 ± 12.57	159.96 ± 73.30
MapReduce	6.28 ± 1.43	40.18 ± 1.66	66.76 ± 10.92	256.40 ± 11.16

Join

	100	1000	10000	100000
Fantasm	0.58 ± 0.30	3.54 ± 0.28	16.95 ± 1.34	78.28 ± 10.62
Pipeline	1.97 ± 0.05	2.04 ± 0.20	2.01 ± 0.09	3.81 ± 1.60
MapReduce	2.67 ± 0.24	5.42 ± 0.45	27.66 ± 1.74	237.75 ± 12.00

Subset

	100	1000	10000	100000
Fantasm	12.22 ± 3.20	105.82 ± 8.45	1022.96 ± 72.85	10977.50 ± 1258.76
Pipeline	3.63 ± 0.74	4.97 ± 0.92	25.89 ± 8.92	222.14 ± 9.02
MapReduce	6.40 ± 0.96	42.70 ± 0.72	134.88 ± 9.59	840.71 ± 125.15

Wordcount

Table 3: Execution time in seconds for the benchmarks in GAE.

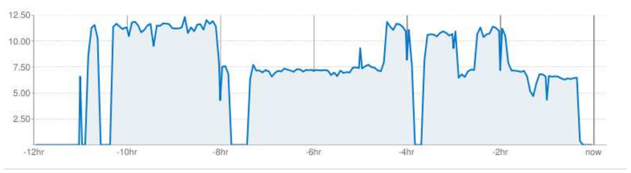


Figure 8: An identical benchmark run three times showing variability in run time.

iments for Join, we experienced high variability in the performance of both the Pipeline and Fantasm libraries. Figure 8 shows a snapshot of three separate trials for Fantasm, in which noticeable differences in processing times occur. Multitenacy could be a primary reason for the fluctuations, yet the exact reasons are unknown and requires further study.

The Subset benchmarks queries a Subset of the entities rather than the entire dataset. Here we see that Fantasm does well, as this scenario was the primary reason for developing the library according to its developers [14]. Pipeline performs best, once again, because of its ability to batch the separate entities, and to not require separate web requests to process individual entities as Fantasm does. MapReduce suffers the most because it must map the entire dataset even though only a Subset is of interest.

For wordcount, MapReduce experiences its largest increase from 10000 to 100000 in this benchmark, which was due to several retries because of transaction collisions. The optimistic transaction support in GAE allows for transactions to rollback if a newer transaction begins before the previous one finishes. This is ideal for very large scale deployments, where failures can happen and locks could be left behind to be cleaned up after a timeout has occurred. Yet it is also possible to bring the throughput of a single entity to zero if there is too much contention. The performance of the wordcount benchmark can be improved by using sharded counters per word as opposed to the simple non-shared counter per word in our implementation. Built-in backoff mechanisms in the MapReduce library alleviates the initial contention, allowing the job to complete.

5.4 AppScale Library Support

We next investigate the use of the GAE analytics libraries over AppScale using the original Task Queue implementation in the GAE software development kit (SDK) and our new implementation based on the RabbitMQ (RMQ) distributed messaging system. We present only Pipeline results here for brevity (the relative differences between GAE and AppScale are similar). Table 4 shows the average time in seconds for the GAE applications executing over a 3 node Xen VM AppScale deployment. Each VM had 7.5GBs of RAM and 4 cores, each clocked at 2.7GHz. Note, that for the GAE numbers, we do not know the number of nodes/instances or the capability of the underlying physical machines employed.

The left portion of the table shows the RMQ execution time in seconds for each message size. The right portion of the table shows the SDK execution time in seconds for each message size. The SDK implementation enqueues the tasks as a thread locally rather than spreading out load between nodes. In addition, the SDK spawns a thread for each task which posts its request to the localhost. Tasks which originate from the local host will never be run on another node. RabbitMQ, on the other hand, spreads load between nodes, preventing any single node from performing all tasks. We are unable to run the 100K jobs using the SDK because the job fails each time from a lack of fault tolerance. If for any reason the node which enqueues the task fails, that task is lost and not rerun again. RabbitMQ, however, will assign a new client to handle the message, continuing on in the face of client failures. For larger sized datasets we also see a speedup because of the load distribution of tasks.

5.5 AppScale Hive Analytics

We next investigate the execution time of the GAE benchmarks using the Hive/Hadoop system. Figure 5 presents the execution time for the previous benchmarks using the Hive query language on a AppScale Cassandra deployment. There was no discernible difference between the sizes of the datasets, but rather the number of stages, where grep only needed a single mapper phase, while the rest had both mapper and reducer phases. While slower for smaller sizes than the GAE library solutions, the Hive solution is consistently faster when dealing with larger quantities of entities (although it has the same issue as the MapReduce library when dealing with data subsets).

The Hive/Hadoop system in AppScale introduces a constant startup overhead for each phase (map or reduce) of approximately 10s. This overhead is the dominant factor in the performance. Once the startup has occurred, each benchmark completes very quickly. The numbers in the table include this overhead. Each of the benchmarks use a single mapper and reducer phase except for Grep. Our approach is significantly more efficient (enabling much larger and more complex queries) than performing analytics using GAE. Moreover, our approach significantly simplifies analytics program development. Each of our GAE benchmarks requires approximately 100 lines each to implement their functionality. Using our system, a developer can implement each of these benchmarks using a single line with fewer than 50 characters.

5.6 Monetary Cost

The cost of transferring data in GAE is dependent on two primary metrics: bandwidth out which is billed at .12 USD per gigabyte, and frontend instances, at .08 USD per hour. For low traffic applications, these costs can be covered by the free quota. For higher traffic, it is possible to adjust two metrics to keep cost down; the first is the maximum amount of time waiting before a new application server is started (where it will be billed for a minimum of 15 minutes), and the second is the number of idle instances that can exist (lowers latency to new requests in exchange for higher frontend cost).

We can compress data and work in batches to lower the bandwidth cost, seeing as how the additional latency for sending updates is between 4 and 7 seconds on average for the largest possible entity of 1MB. The compression execution time is added to frontend hour cost, and the level of compression is very dependent on the application's data (images, for example, may already be highly compressed). The average daily cost of the data transfer was 12.41 USD for frontend hours, 1.03 USD for datastore storage (went up over time), 2.55 USD for bandwidth, and 15.63 USD for datastore access. As future work, we are leveraging our findings to improve our datastore wrapper to minimize cost while still maintaining low latency overhead.

The cost for on-site analytics such as Fantasm and Pipeline is based on datastore access, both for reading the data which is needed for operation, and metadata for tracking the current progress of a job. The other cost associated is the frontend instance hours. The cost for running Pipeline for wordcount on 100000 entities was 0.34 USD (not accounting for the free quota), where 0.056 USD was frontend hours, 0.13 USD was datastore writes, and 0.154 USD on datastore reads. The cost of datastore writes is highly dependent on the number of indexed en-

	100 RMQ	1000 RMQ	10000 RMQ	100000 RMQ	100 SDK	1000 SDK	10000 SDK
Aggregate	3.02	5.72	183.93	610.12	3.77	6.14	N/A
Grep	5.37	16.90	205.53	862.36	6.11	28.88	260.03
Join	2.72	5.16	165.03	455.31	3.78	5.90	305.82
Subset	2.45	3.12	12.61	786.53	2.55	3.20	12.11
Wordcount	7.41	11.43	311.52	635.28	8.38	17.40	411.12

Table 4: Execution time in seconds for benchmarks using the Pipeline library on AppScale with RabbitMQ (RMQ) and the SDK implementation.

	100	1000	10000	100000
Aggregate	20.59 ± 1.41	21.14 ± 0.55	20.30 ± 0.88	20.94 ± 0.59
Grep	11.90 ± 1.32	11.00 ± 0.58	11.17 ± 1.30	10.69 ± 0.44
Join	20.52 ± 1.01	20.71 ± 0.84	20.43 ± 0.57	23.41 ± 0.64
Subset	19.93 ± 0.54	20.07 ± 1.34	20.26 ± 0.86	20.66 ± 0.45
Wordcount	21.73 ± 1.50	22.13 ± 1.51	22.19 ± 0.96	21.54 ± 0.95

Table 5: Execution time in seconds for benchmarks using Hive.

tities, and therefore if the entities have more properties, the writes can multiply quickly as would cost (each index write counts as a datastore write). In general, it is difficult to predict the cost of GAE analytics. Our approach allows developers to perform analytics repeatedly without being charged at the cost of data transfer.

Our other option for downloading the data is via bulk transfer using tools provided by the SDK. We investigated the use of such tools but we ran into difficulties where exceptions arose and the connection would drop. Multiple attempts were needed, driving cost up as much to 5 to 6 times the cost of a daily experimental run (from 15 USD to 86 USD) before being able to complete a full download of the data. It took 9520 seconds on average for the three successful downloads of a dataset of 202MB. This option is clearly not acceptable for hybrid analytic clouds.

6 Related Work

OLAP and data warehousing systems have been around since the 1970s [8], yet there is no system available for GAE which is currently focused providing OLAP for executing web applications. AppScale, with its API compatibility and our extensions herein, brings OLAP capabilities (as well as its testing and debugging) to this domain.

TyphoonAE is the only other framework which is capable of running GAE applications outside of GAE. TyphoonAE however is a more efficient version of the SDK (executes the system serially) and only supports the Python language. AppScale and our work supports Python, Java, and Go languages and is distributed and scalable. TyphoonAE does not have the same facility as AppScale to run analytics, as it does not support datastores capable of Hive support. Private PaaS offerings such as Cloud Foundry [11] offer an open source alternative to many proprietary products and offer automatic deployment and scaling of applications, yet do not support GAE APIs.

There are many cloud platforms which allows for analytics to be run on large scale datasets. Amazon's Elastic MapReduce is one such service, where machines are automatically setup to run jobs, along with customized interfaces for tracking jobs [24]. The Mesos framework is another cloud platform which can run a variety of processing tools such as Hadoop and MPI, and does so with a dynamically shared set of nodes [19]. Helios is yet another framework that simplifies the application deployment process.

In [21], the authors measured data-intensive applications in multiple clouds including GAE, AWS, and Azure. Their application was a variant of the TPC-W benchmark, similar to an online bookstore. Our benchmarks, by comparison, are analytics driven rather than online processing. Furthermore, since the time of publication Google–as well as the other cloud providers–have continuously improved functionality and added features. Our work provides a new snapshot in time of the current system, which has since come out of preview and become a fully supported service.

Data replication across datacenters is a common method for prevention of data loss and to enable disaster recovery if needed. Currently GAE implements threeplus times replication across datacenters using a variant of the Paxos algorithm [4]. Extant solutions, such as [30], however, are not applicable because of the restrictions imposed by the GAE runtime. To overcome this limitation, we provide a library wrapper around destructive datastore operations, to asynchronously update our remote AppScale analytic platform. As part of future work, we are investigating how to provide disaster recovery using our hybrid system.

7 Conclusion

Cloud computing has seen tremendous growth and wide spread use recently. With such growth comes the need to innovate new methods and techniques for which extant solutions do not exist. Online analytics processing systems are such an offering for Google App Engine, where current technology has focused on web application execution at scale and with isolation, and existing solutions have operated within the restrictions imposed.

In this paper we have described, implemented, and evaluated two systems for running analytics on GAE application, running current libraries in AppScale through the implementation of a distributed task queue, and the ability to run SQL statements on cross-cloud replicated data. Future work will carry forward our findings to optimize cross-cloud data synchronization as well apply our system to another use case: disaster recovery.

References

[1] Amazon Web Services. http://aws.amazon.com/.

[2] Microsoft Azure Service Platform. http://www.microsoft.com/azure/.

[3] AZURE, M. Business Analytics, 2011. http://www.windowsazure.com/en-us/home/tour/business-analytics/.

[4] BAKER, J., BOND, C., CORBETT, J., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *5th Biennial Conference for Innovative Data Systems Research* (2011).

[5] Brisk Datastax. http://www.datastax.com.

[6] BUNCH, C., CHOHAN, N., KRINTZ, C., CHOHAN, J., KUPFERMAN, J., LAKHINA, P., LI, Y., AND NOMURA, Y. An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. In *IEEE International Conference on Cloud Computing* (Jul. 2010).

[7] BUNCH, C., KUPFERMAN, J., AND KRINTZ, C. Active Cloud DB: A RESTful Software-as-a-Service for Language Agnostic Access to Distributed Datastores. In *ICST International Conference on Cloud Computing* (2010).

[8] CHAUDHURI, S., AND DAYAL, U. An overview of data warehousing and olap technology. *SIGMOD Rec. 26* (March 1997), 65–74.

[9] CHOHAN, N., BUNCH, C., KRINTZ, C., AND NOMURA, Y. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing* (July 2011).

[10] CHOHAN, N., BUNCH, C., PANG, S., KRINTZ, C., MOSTAFA, N., SOMAN, S., AND WOLSKI, R. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *ICST International Conference on Cloud Computing* (Oct. 2009).

[11] Cloud Foundry. http://cloudfoundry.com/.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI)* (2004), 137–150.

[13] ENGINE, G. A. App Engine Transaction Semantics, 2010. http://code.google.com/appengine/docs/python/datastore/transactions.html.

[14] Fantasm. http://code.google.com/p/fantasm/.

[15] Google app engine blog. http://googleappengine.blogspot.com.

[16] Google App Engine. http://code.google.com/appengine/.

[17] Google App Engine MapReduce. http://code.google.com/p/appengine-mapreduce/.

[18] GURP, J. V., AND BOSCH, J. On the implementation of finite state machines. In *in Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta* (1999), Press, pp. 172–178.

[19] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Networked Systems Design and Implementation* (2011).

[20] HIVE. Hive Query Processing Engine, 2010. https://cwiki.apache.org/confluence/display/Hive/Home.

[21] KOSSMANN, D., KRASKA, T., AND LOESING, S. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 579–590.

[22] KRINTZ, C., BUNCH, C., AND CHOHAN, N. AppScale: Open-Source Platform-A s-A-Service. Tech. Rep. 2011-01, University of California, Santa Barbara, Jan. 2011.

[23] LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (1978).

[24] MAPREDUCE, A. E. Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce.

[25] MURTHY, R., AND JAIN, N. Talk at ICDE 2010. Hive–A Petabyte Scale Data Warehouse Using Hadoop., Mar. 2010.

[26] Google App Engine Pipeline. http://code.google.com/p/appengine-pipeline/.

[27] RabbitMQ. http://www.rabbitmq.com.

[28] Rackspace Hosting. http://www.rackspace.com.

[29] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive- a warehousing solution over a map-reduce framework. In *VLDB* (2009), pp. 1626–1629.

[30] WOOD, T., LAGAR-CAVILLA, H. A., RAMAKRISHNAN, K. K., SHENOY, P., AND VAN DER MERWE, J. Pipecloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 17:1–17:13.

Poor Man's Social Network: Consistently Trade Freshness For Scalability

Zhiwu Xie
Virginia Polytechnic Institute and State University
Jinyang Liu
Howard Hughes Medical Institute
Herbert Van de Sompel
Los Alamos National Laboratory
Johann van Reenen and Ramiro Jordan
University of New Mexico

Abstract

Typical social networking functionalities such as feed following are known to be hard to scale. Different from the popular approach that sacrifices consistency for scalability, in this paper we describe, implement, and evaluate a method that can simultaneously achieve scalability and consistency in feed following applications built on shared-nothing distributed systems. Timing and client-side processing are the keys to this approach. Assuming global time is available at all the clients and servers, the distributed servers publish a pre-agreed upon schedule based on which the continuously committed updates are periodically released for read. This opens up opportunities for caching and client-side processing, and leads to scalability improvements. This approach trades freshness for scalability.

Following this approach, we build a twitter-style feed following application and evaluate it on a following network with about 200,000 users under synthetic workloads. The resulting system exhibits linear scalability in our experiment. With 6 low-end cloud instances costing a total of no more than $1.2 per hour, we recorded a peak timeline query rate at about 10 million requests per day, under a fixed update rate of 1.6 million new tweets per day. The maximum staleness of the responses is 5 seconds. The performance achieved sufficiently verifies the feasibility of this approach, and provides an alternative to build small to medium size social networking applications on the cheap.

1. Introduction

Scalability has emerged as a significant challenge for the social networking web applications. If built with the traditional web application framework, even a small social network can be easily strained by a modest level of user interaction. The performance bottleneck usually locates at the back-end persistent data store, which is typically a relational database. Following the examples of Twitter and Facebook, many social networking web sites start to migrate their relational databases to various key-value stores, collectively branded as being "NoSQL". This approach indeed can scale to a larger workload, but always at the expense of a deliberate void of the consistency guarantee.

What does consistency mean and entail in this context? Since many NoSQL advocates cite the Brewer's Conjecture [8], also known as the CAP theorem, as the theoretical foundation to justify this trade-off, we naturally adopt the consistency definition used in its formal proof [18]. Used elsewhere, this type of consistency is also known as Atomicity [23], Linearizability [20], or One-copy Serializability (1SR) [6]. This is different from the consistency as referred in the ACID properties of the database. To avoid further ambiguity, in this paper we regularly use the abbreviation 1SR to denote such consistency, and unless noted otherwise, consistency always refers to 1SR in this paper.

1SR provides the clients of a distributed system with an equivalent single processor view that allows them to reason the system behavior regardless of how many distributed servers are used to run the service, how geographically far apart they are from each other and from the clients, and how they are synchronized. Without 1SR, the distributed system may exhibit odd behaviors that confuse the users. We therefore are interested in exploring the possibility of scaling the social networking functionalities, especially the feed following applications, without violating 1SR.

Given the formal CAP theorem proof this may seem impossible. But the proof itself is strictly preconditioned on the asynchronous network model, where the only way to coordinate the distributed nodes is to pass messages across the network. Practical distributed systems usually have more tools in hand, and one of the tools is a reasonably synchronized and approximated global time. Indeed, the authors of the CAP theorem proof used the second half of their paper to show that under a partially synchronous, or timing-based, distributed model [24], where global time is assumed to be available, CAP may indeed be simultaneously achievable most of the time, although in return we may have to give up some freshness, but not necessarily the latency.

Unfortunately, this aspect of the CAP has not attracted sufficient attention from the industry nor from academic researchers. As early as 2008, Roy Fielding proposed a RESTful approach [15], which is distinctively timing-based, for the known hard-to-scale feed following problem. But we are not aware of many real-world social networking web applications that are built this way, and to the best of our knowledge, no empirical or experimental data are publicly available to verify its scalability properties.

In this paper we take Roy Fielding's proposal as a starting point, extrapolate it to shared-nothing distributed systems, and fine-tune its timing method for replication control. We then provide a formal description of the algorithm, prove its consistency property, and analyze its trade-offs. In order to gain insights on how much freshness we must give up to gain the level of scalability currently provided by the NoSQL approach, we build a system and test its performance with the workloads similar to that used in a Yahoo! PNUTS based Twitter-like feed following experiment [27]. The server side of the system is fully implemented, but instead of writing and delivering client-side code to real browsers, we implement an emulated browser on the client side to facilitate performance testing. In our experiment, we set a fixed staleness limit of 5 seconds and an update rate of 19 new tweets per second. Due to the limitation of the test facility we used, we were not able to generate a query workload exceeding 40% of the PNUTS experiment. But within this limitation our system exhibits linear scalability, up to 6 servers.

The main contributions of this paper include:

- A formal description of a timing-based replication control algorithm for the feed following problem
- A proof of its consistency property
- A working implementation built with lower performing commodity virtual machines in the cloud
- An experiment to thoroughly test its performance and trade-offs
- A working example that demonstrates how freshness can be exchanged for better scalability

With our approach, social networking applications are able to scale linearly to fairly high workloads with low-end machines. In our implementation we take advantage of the relatively stable and familiar commodity web server hardware and software stacks, which over the years have gained wide adoption and the prices have dropped significantly. No additional expertise and training are required to operate these stacks, potentially saving the operating costs as well. We therefore dub this approach "the poor man's social network." Nevertheless, there is no real obstacle to apply the timing-based approach to various NoSQL data stores. In our approach the relational databases are used mostly as simple key-value stores. We chose PostgreSQL in particular only to take advantage of its built-in triggers, procedural languages, and the transactional features not readily available from many key-value stores. But we anticipate that the same idea may be used by key-value stores as a base to develop more consistency features in addition to the eventual consistency.

This paper consists of the following sections: we first introduce the feed following problem and argue for a different tradeoff strategy to the popular method. We then provide a formal algorithm to implement this alternative in shared-nothing distributed systems and prove its correctness. After describing our implementation and experimental configurations, we present the results and the related work, then conclude the paper with discussions and future work.

2. Feed Following

2.1. Problem Statement

Feed following is the type of social networking functionality that layers on top of a following network consisting of large numbers of feed consumers and feed producers. Each feed consumer follows a usually large and distinctive group of feed producers, and each producer independently produces event items over the time. Now each of the consumers wants to query the *n* most recent event items produced by all the producers this particular consumer follows. Silberstein et al. give a more formal definition of the problem [28].

Twitter's timeline application is a typical feed following problem, where each event item is called a tweet. Many other social networking features may be modeled as variations of the feed following, and the "*n* most recent" predicate may also have many other flavors. But the common theme is that each feed following query can be quite personalized and distinctive from the others such that the query results for one consumer are of little or no use to another for the purpose of directly reusing the results to reduce the overall query load.

Moreover, even the newly produced query results may quickly become outdated for the same consumer. In order to provide the freshest possible response, a consumer's feed query result must be invalidated as soon as any of the followed producers posts a new event, and each producer's new event must also invalidate all the current feed following query results for each of the consumers that follow this producer. The large number of consumers also make it very expensive to maintain the materialized views for each of them. It is easy to see why the traditional scalability tools such as data partition, query and web caching, materialized view, and replication perform poorly in this type of problem. Silberstein et al. provide a more thorough analysis on the difficulties to scale the feed following problem [27].

2.2. Trade Consistency For Scalability

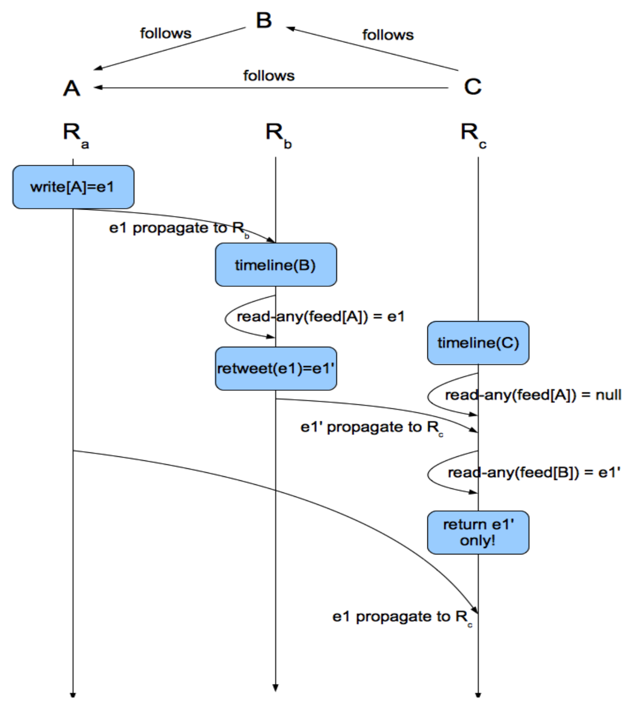
Popular NoSQL style applications are built under the assumption that consistency should be sacrificed to gain better scalability. To relax 1SR for feed following in a shared nothing, fully or partially replicated distributed environment, we may declare an event update successful as soon as one of the replicas commits it locally and before this update finishes propagating to most or all the other replicas. By eliminating the consistency locks, the replicas become more independent and can work in a more paralleled manner. But as a result, the follower's view becomes rather unpredictable. Even if a producer receives the confirmation of a successfully committed update, there is no guarantee when her followers can see this event. Some may see it shortly after, some others may need to wait for an extended period of time before this new event shows up, and even those who have already seen it once may not see it in the subsequent queries. We want to emphasize that waiting to see the most recent updates is not the real issue here. The real problem is the unpredictable nature of the wait.

Despite this, the consensus among the industry is that the users should be able to tolerate some lost events as long as they eventually show up. After all, unless the feed producers and the consumers are actively tracking and comparing their timelines, the temporarily lost events are not particularly noticeable. Moreover, many feed following applications such as Twitter do not allow editing; therefore eliminate the needs to reconcile the conflicting updates, as normally seen in an optimistic approach like this. However the inconsistency becomes apparent when the following network starts to change, e.g., a consumer decides to stop following a producer. If this relationship update is to be committed the same way as above, then it is possible for the consumer to continue receiving the event items from this producer even if this consumer has been notified of the successful unfollowing, simply because the unfollow update has not been propagated to the replica that processes the feed following query.

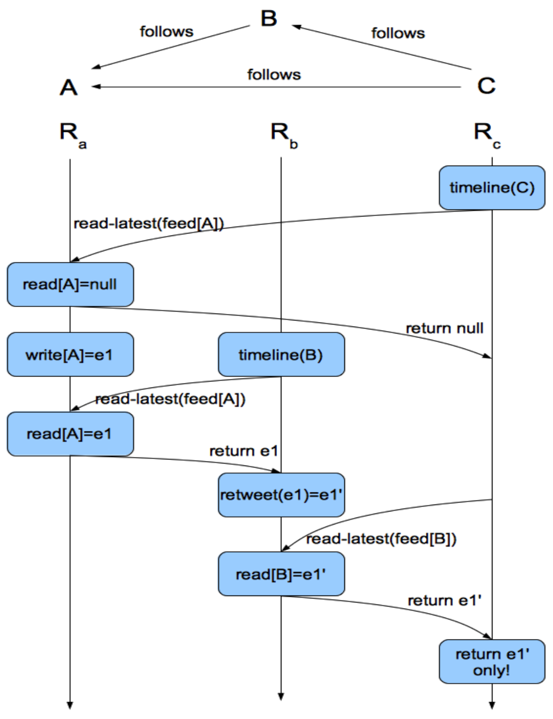
One approach to mitigate this problem is to slightly tighten up the eventual consistency model described above, to the "per-record timeline consistency", as exemplified by Yahoo!'s PNUTS [28]. This approach assigns a master replica for each record, then allows the application developer to specify what type of query to use: i.e., Read-any, Read-critical, or Read-latest. An update is not successful unless it is committed to the master copy, and a Read-latest query will always validate against that record's master copy to retrieve the most recently committed change, although a Read-any query will return any locally available data regardless of their validity. The assumption is that for those critical and consistency sensitive queries we should use Read-latest, which, although more expensive than the other two options, can at least provide some record-level consistency guarantees.

Nonetheless, this approach cannot prevent the inconsistency exposed by the retweets. A retweet is a standalone new tweet produced by a user who follows the original tweet producer. The retweet either includes the original tweet content by value or links to the original by reference. By logic a retweet can only commit after the original tweet is committed, because there exists a conflicting feed query between these two updates. It is therefore rather confusing for a feed consumer to observe only the retweet but not the original if this consumer follows both the original producer and the retweeter, which is fairly common in closely knit social groups. This arouses suspicion of either voluntary retraction or censorship, although in fact is merely a symptom of the distributed inconsistency.

As illustrated in figure 1, the retweet inconsistency may still happen in systems like PNUTS using either the Read-any or the Read-latest, as long as 1SR is not guaranteed. In the figure we assume Replica Ra is the master copy for all tweets produced by user A, et al. When using Read-any to retrieve the feeds from Replica Rc, if the retweet from Rb propagates to Rc faster than the original tweet from Ra, we may observe the inconsistency at Rc. The probability of inconsistency may be lower if we use Read-latest, in which case all the feed queries must be routed to their master replicas. But in between these large number of independent and usually



(a) Read-any



(b) Read-latest

Figure 1. Retweet Inconsistency

remote queries to the master copy, the logical sequence between the original tweet and its retweets may still be reversed, as shown in Figure 1(b). Furthermore, using Read-latest on all feed queries carries a steep performance penalty, because we lose the benefits of replication and caching, leaving data partition the only performance booster.

The Read-critical query, which "returns a version of the record that is strictly newer than, or the same as the required version" [28], provides little help in the above scenario. This is because in the per-record timeline consistency, the record's version is specified locally by its master copy. It is not a global version, therefore affords no meaningfully comparison between the versions of two different records, e.g., a tweet and its retweet. It is tempting to devise a sophisticated global versioning scheme to determine the sequence of all the updates, but this already implies 1SR. From the CAP theorem proof we already know that if a distributed system relies solely on the message passing to implement 1SR, then it is hard bounded by the CAP compromise and cannot scale well reliably. A more promising approach would be to explore beyond the asynchronous network model and more specifically, to exploit global time, which does not depend on the message passing exclusively. In the next section we discuss how this approach trades freshness for scalability.

2.3. Trade Freshness For Scalability

Freshness is oftentimes unnecessarily entangled with 1SR. For example, Vogels defines the strong consistency as being always guaranteeing the freshness and 1SR. But in its first degree of relaxation, the weak consistency allows a period of "inconsistency window" during which an update is not guaranteed to be always available to all queries [30], "not always" being the keyword. During the "inconsistency window" such weak consistency fails not only the freshness test but also the 1SR test. Such a categorization overlooks an intermediate level of distributed database system behavior which guarantees to return the 1SR responses, although they may be stale. Such systems indeed exist, e.g., a log-shipping based master-slave replication system where all the updates are processed at the master but all the queries go to the slave. The query results always lag behind the freshest state at the master, but the system is nevertheless 1SR.

We further argue that absolute freshness is not even worth pursuing in a web based system, because even the freshest query results still need to be transported across the web to the clients, yet the web latency is not negligible. When the clients receive the results, they may have already turned stale, therefore from the holistic system view it seems rather unnecessary to guarantee the absolute freshness within the boundary of the database servers. Indeed, users of the web applications intuitively feel the latency and understand its effects. When using high-volume transactional systems such as the online bidding or stock exchange web applications, the users are acutely aware that the quote prices shown on the screens are not real-time but with delays built in. Moreover, not every system demands high levels of freshness anyway. Most social networking applications are not meant to be real-time point-to-point messaging systems, therefore some delays is tolerable and even expected.

We also note the differences between staleness and latency. Latency characterizes the speed of the request-response process, while staleness characterizes the recency quality of the data carried by the response. From the end user's point of view, latency always adds further staleness to the response, but not vice versa. Web users with short attention spans have fairly low tolerance for unresponsive web services, but not necessarily for staleness. Given a choice, faster responses carrying slightly more stale information should be much preferred than the opposite.

We now explain why freshness may be traded for scalability. We draw an analogy between this tradeoff and the mass transit system. When driving our own cars, we can freely choose the departure times and the destinations. But when using the mass transit systems, we must time our activities according to the published schedules, travel only to the vicinities of the bus stops, and make transfers between different transit lines by ourselves. Bus riders lose the flexibility to travel at-will, but gain overall efficiency and economic benefits by sharing resources. Such benefits are especially significant in metropolitan areas where not only the opportunity for sharing is higher but also the transportation resources are under much heavier loads and are much more congested.

Caching is the web's way to share resources. The web is built with the caching facilities at its core to address the scalability issues. But as explained in section 2.1, the current way of building feed following applications is not attuned for taking full advantage of the web caches. This is because such a system is built to accommodate the private-car style of usage, striving to provide the personalized response accurately and consistently at the time when the system executes that particular request. Caching is less effective because the queries are not only highly personalized, but also extremely ephemeral.

A mass transit style of feed-following system may improve the situation on two fronts. First, it may address the ephemeral issue by only executing queries with accuracy and consistency guarantee by a pre-agreed upon schedule, e.g., every 5 seconds. In a mass transit system all the passengers arriving at the station before the scheduled departure time must wait for the next bus. By the same logic, if enhanced with this improvement, all queries submitted to the servers between 1:05:30PM and 1:05:35PM will be immediately responded, but with the results that are accurate and consistent only as of 1:05:30PM. Conceptually this enhancement allows the queries received within this period all be executed against the same database snapshot taken at 1:05:30PM rather than against a moving target. We have built in no more than 5 seconds' staleness in all responses, yet the latencies are not necessarily higher. Due to effective caching and reusing, this approach may even significantly lower the response latency. However, the system works differently on updates. For example, if an update is received at 1:05:33PM, it will be committed as soon as the system permits, but the committed result will not be available for queries until the next scheduled time point, e.g., 1:05:35PM.

On the other hand, much like the mass transit system that will not board and drop off riders at any location, the server may also decline to execute personalized feed following queries. Instead, it may analyze and reorganize these queries, break them down into multiple steps, and only execute the commonly shared queries on the server. In case of the feed following, one exemplary common query useful for all users is the "time map" query, which tells us which feed producers have created new events during the past scheduled intervals. With such information at hand, the feed followers themselves can combine and match to produce their own personalized event lists on the client side. This is analogous to bus riders making transfers by themselves. Note that such an approach is only feasible when the queries are against the same database snapshot. Querying against changing database states cannot be reorganized correctly in this manner. In other words, this enhancement is preconditioned on the prior one.

In our approach a large portion of the processing is therefore offloaded to the clients, shifting the system from a thin client system to a fat client system. This is quite different from the NoSQL approach where the distributed servers still attempt to process all query loads from the start to the end albeit abandoning the 1SR guarantee. In the next section we will formalize the algorithm and prove its correctness with respect to 1SR.

3. Replication Control and Its Correctness

We adopt the lazy-master style partial replication strategy for this distributed system. In particular, a conceptual centralized master database is naively and horizontally partitioned into multiple smaller physical database servers based on the producer id. A new tweet is routed to its own partition server according to the producer id and then committed there. Each partition server maintains a "time map". This is a materialized view that documents which producers allocated to this server have tweeted in the current scheduled interval. This view must be synchronously maintained within the same atomic update transaction boundary for a new tweet. In practice we also maintain multiple combined views that cover larger granularity of the time intervals.

A client, upon receiving a timeline query, first checks global time and then determines by itself the most recently scheduled release time. Imagine we have taken a conceptual full database snapshot at this particular release time. This is the database state against which this particular timeline query needs to be executed. The client then must make sure it has synchronized its local partial data with this snapshot before executing the timeline queries. Note the web architecture mandates that a server cannot initiate connection to the clients. This determines the lazy nature of our replication strategy. That is, an update is committed at the master copy but not atomically propagated to all the caches and the clients. We allow the partial database replicas to lag behind the master until they are used for queries, by which time they must catch up to the scheduled snapshot.

We now give the formal definition of the scheduled releasing mechanism. For any given time $t > t_0$, where t_0 is the initial database time, there exists one and only one time period $[t_i, t_i+\Delta t_i)$, such that for $t \in [t_i, t_i+\Delta t_i)$ and $\Delta t_i = O(\Delta t_L)$, where Δt_L is the network latency or a tolerable time interval, any query issued at t will be responded with the same result as if the query is executed at t_i. We require the staleness limit to be much larger than the network latency, because from the user's point of view the network latency is automatically added to the staleness of every response, therefore we cannot promise the staleness to be less than that. Larger staleness limit will also have positive effects on the scalability.

We require the releasing schedules be defined a priori. At any given time after a web client initializes itself, it should already know the corresponding time intervals without having to contact the server again to find out. We also require the time intervals defined in absolute time and all the web clients reasonably synchronized to a NTP server to guarantee limited time skews among the replicas and the server. This is in line with the partially synchronous distributed model and eliminates the unnecessary web and database operations.

The replication control algorithm is described in the following. The pseudo code is depicted in Figure 2. In our algorithm, an update is routed to the corresponding master database partition server allocated for that producer, and executed in an ACID manner, e.g., using the strict two-phase locking (2PL) protocol. The queries executed at the clients are against their explicitly scheduled snapshot, and we require the clients to synchronize to that snapshot with the master database before the execution. This is similar to the multi-version mixed method described in [6] or the snapshot isolation protocols [4] with one important distinction. In our method the snapshots are chosen a priori and independently from the database states and the timing of the queries.

```
**Upon:** submit of a read-only transaction T to client at time t
1:  assign T the timestamp $t_i$, the starting time of its sched-
uled time interval
2:  **if** local database is not synced to the snapshot at $t_i$:
3:    request from all the master partitions the writesets up
to $[t_{i-1}, t_{i-1}+\Delta t_{i-1})$
4:    sync local database the snapshot at $t_i$
5:  execute T at the local database
6:  **return** result

**Upon:** submit of an update transaction T to client
7:  forward T to the master

**Upon:** submit of an update transaction T to a master partition
8:  atomically request necessary shared and exclusive locks
9:  wait until all locks are granted
10: execute T at master partition, record commit time t
11: **for** all the materialized views $V_i$ covering t:
12:   update $V_i$ to reflect the writeset of T
13: release locks of T
14: **return** ok

**Upon:** submit of a request to a master partition for writeset
for time interval $[t_n, t_n+\Delta t_n)$
15: **for** all the materialized views $V_i$ on this partition:
12:   if $V_i$ is for $[t_n, t_n+\Delta t_n)$:
13:     **return** writeset in $V_i$
```

Figure 2. Replication control algorithm.

Conceptually the scheduled releasing introduced here enforces a new transactional state to the database. Traditionally we assume once an update is committed its changes are immediately visible to all the other active transactions. We revoke this assumption and define a "QUERY VISIBLE" state after "COMMITTED", as shown in Figure 3. The changes made by an update are still immediately visible to other updates once committed, but are visible to active queries only when they reach the “QUERY VISIBLE” state.

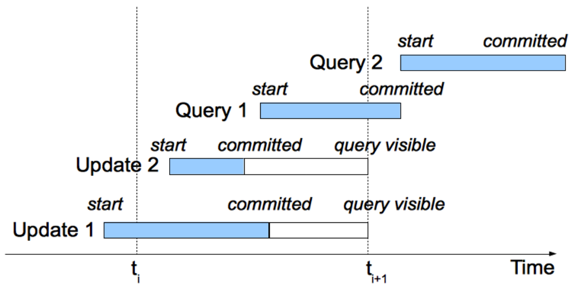


Figure 3. Update visibility and serial execution.

In comparison, if we use the mixed protocol or snapshot isolation, Query 1 in Figure 3 would be able to see Update 2 because it starts after Update 2 commits. But this would require a new version or snapshot being created for each committed update, and each of them may need to be individually propagated to all the replicas. In our protocol, Update 2 is invisible to Query 1, because it becomes visible to queries after the latter starts. Query 2, on the other hand, can see both updates. The scheduled release time interval is independent from the query load. When the query load becomes heavier, more updates become visible at each scheduled release, but the number of snapshots to be propagated per time unit remains the same. On the other hand, if both updates in Figure 3 write to the same data item, the data written by Update 2 is lost in our protocol because no query ever sees it. We simply assume these updates represent the intermediate states, which despite being committed, the clients don’t care to know. Further, since we did not relax the irrevocability aspect of the COMMITTED state, the ACID property of the database still holds.

Proving the 1SR property of this protocol is a two-step process. We first show that there exists a one-copy equivalence of this protocol. We then show this one-copy equivalence is serializable. Due to the scheduled release and the master-replica differentiation between the update-query transactions, executing queries on the client-side replicas introduces no data contention, and the network latency is masked by the timestamps. We can easily see that any query executed on the client is equal to the execution of the same transactions on a single copy master database. As for the single copy execution equivalence, since the updates are executed in strict ACID manner on each partition, there exists a serial execution of the updates on the same partition, and they are serialized by the sequence of their commit times. Since global time exists across all partitions, there also exists a global serialization, ordered by the global time and segmented by the scheduled releasing. By definition all queries can be moved to the start of their time intervals, and their relative ordering does not matter because there’s no update transaction in between. Therefore the single copy execution equivalence is serializable, and the replication control protocol is 1-copy serializable. As an example, the transactions shown in Figure 2 can be serialized in this order: Query 1 $<_t$ Update 2 $<_t$ Update 1 $<_t$ Query2.

4. Implementation

We implemented a twitter-like feed following application prototype. The server side was fully implemented with Python/Django and PostgreSQL. We chose PostgreSQL as the backend database because of its mature support for the time travel functionalities, which goes back to its origin. The scheduled releasing, time map, and related functionalities were implemented with triggers and programmed in PL/pgSQL. Nonetheless, the database was queried primarily as a key-value store.

The client-side functionalities could have been fully implemented in Javascript and client side database, but we were concerned about how to evaluate the system performance. At the time of the experiment, our Amazon account only allowed up to 100 instances running at the same time, but we potentially needed thousands of real browsers running in parallel to generate the desired workload. We eventually decided to first implement a simplified emulated browser in Python/Django and PostgreSQL.

We picked the maximum staleness level at 5 seconds. But for those followers who didn’t follow many active producers, it would have taken them numerous 5-second time map queries to gather the 20 most recent tweets. We therefore also implemented time maps for the following larger granularities to speed up the time map query: 30 seconds, 3 minutes, 15 minutes, 1 hour, 4 hours. Accordingly, we also added slightly more materialized view maintenance work when committing each new tweet. If a client is not able to gather sufficient tweets from a time map query, it will attempt an earlier time map at either the same or larger granularity level if it’s available.

5. Evaluation

To better evaluate our implementation we must generate more realistic social networking workloads. Attempting to compare our implementation with the Yahoo! PNUTS system, we used the same zipfan parameters as used in that experiment [28]. Both the social network and the synthetic workload were generated with Yahoo! Cloud Serving Benchmark [10], and Table 1 lists the parameters used in both experiments.

Since our implementation relies heavily on the clients being able to programmatically interpret the workload and the server responses and do local calculations as well as database operations, we need large number of machines and computing power to emulate the client-side processing. The only viable test environment seemed to be the computing cloud. We therefore set up our test environment in Amazon EC2.

Table 1. Comparing workload parameters with the Yahoo! PNUTS experiment [25]

		PNUTS	This
Number of producers		67,921	67882
Number of consumers		200,000	196,283
Consumers per producer	Average	15.0	13.38
	Zipf parameter	0.39	0.39
Producers per consumer	Average	5.1	4.63
	Zipf parameter	0.62	0.62
Per-producer rate	Average	1/hour	1/hour
	Zipf parameter	0.57	0.57
Per-consumer rate	Average	5.8/hour	varied
	Zipf parameter	0.62	0.62

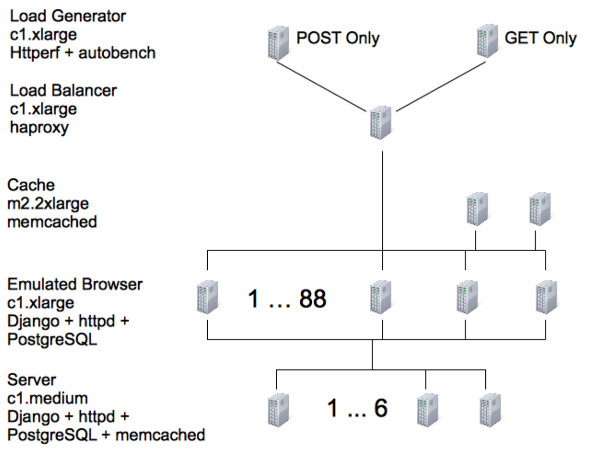


Figure 4. Experiment configuration

Figure 4 schematically shows the multi-layer server configuration deployed to conduct the experiment. On the bottom layer we deployed a small number of low-end servers, initially up to 20 small instances (m1.small) then upgraded to 3 to 6 high-CPU medium instances (c1.medium). The reason we upgraded was because even at the maximum number (100) of total instances we still were not able to generate sufficient client-side processing power to drive up to 40% of the query load in the Yahoo! PNUTS experiment. We therefore decided to move more virtual machines to simulate the clients rather than further increasing the server numbers. These low-end servers also run their local memcached service. Directly above the servers were up to 88 high performance instances (high-CPU extra large instance, or c1.xlarge) used to simulate the client-side processing. These emulated browser machines then shared up to 2 high-performance instances (m2.2xlarge, or high-memory double extra large instance) running standalone memcached server to simulate the web caching. We then ran HAProxy on one c1.xlarge instance to evenly distribute the workloads to these emulated browsers, and had two c1.xlarge instances, both running httperf and autobench, to drive the workload and run benchmarking, one for the update load and another for the query load. All these instances also had collectd installed and had various statistics reported back to our cloud service control panel.

Much like the other web applications, social networking applications' query load vastly dominates their update load. If the Yahoo! PNUTS experiment workload is any indication, the query load exceeds 99% of the total requests.

In our experiment, however, we decided to use a fixed update load at 19 requests per second, only slightly higher than the average update load in the PNUTS experiment. We then slowly drove up the query load until any server returned a 500 code. Figure 5 shows the linear scalability observed when the number of the servers was increased from 3 to 6. Figure 6 depicts the latency-load relations under different server configurations. Beyond 6 servers, the client-side simulation became the bottleneck and no meaningful data could be obtained. The linear scalability property is further supported by the following observations:

First, we observed extremely high cache hit rates, oftentimes exceeded several thousands to one, at the standalone memcached servers deployed to simulate the web caches, as shown in Figure 7. This indicated the success of the mass transit style approach we employed to build the feed following applications. The changes of the cache hit rate corresponded nicely to the scheduled releasing times, increased during the intervals, and had sudden dips at the release time points when newly released data caused cache misses.

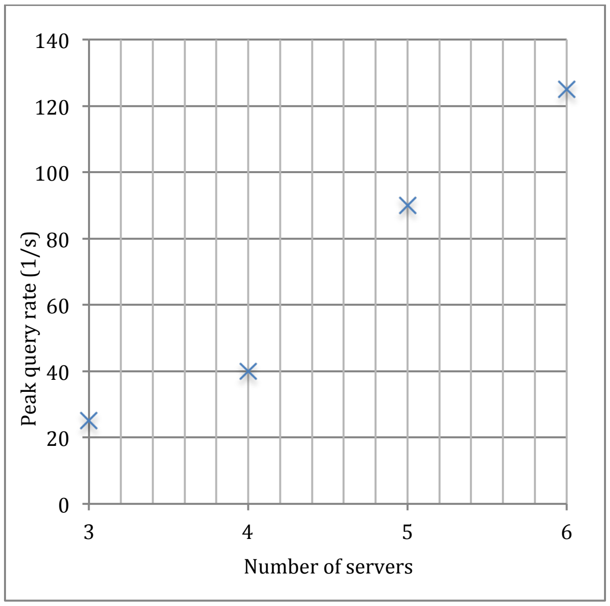


Figure 5. Peak query rates vs. number of servers under the query/update combined workload

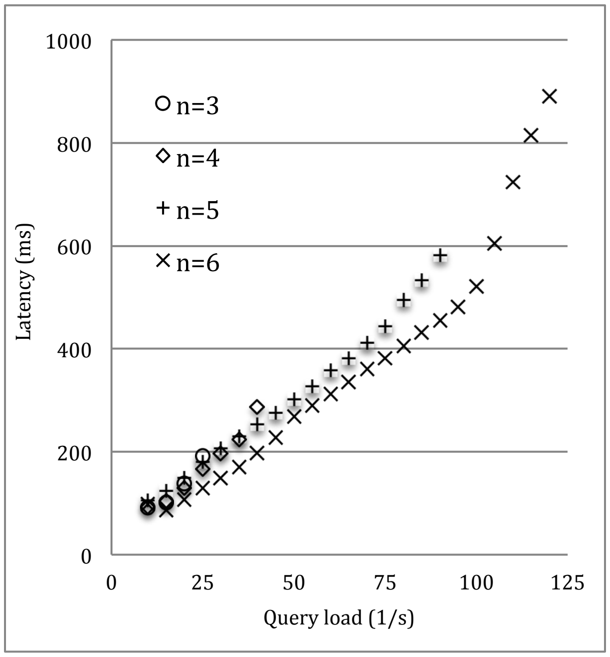


Figure 6. Latency vs. query load, with constant update load at 19 updates per second.

Secondly, at the local memcached service inside the server, we also observed fairly good, but not much as high, cache hit rates, at approximately 2 to 1 to 4 to 1. This indicated that the distributed system overall fared pretty well in not bothering to contact the origin server unless absolutely necessary. This is in line with the principle of the web architecture and the web caching.

Finally, we observed that when the query load increased, the CPU load increased much faster at the emulated browsers than at the servers, as shown in Figure 8 and 9. This illustrated the strength of this approach, which distributed a larger portion of the increasing processing load to the clients than absorbed by the servers. This was the source of the favorable scalability properties at display in this experiment.

6. Related Work

Partitioning, replication, and caching are time-tested and battle-hardened strategies to scale web applications as well as the databases that drive these applications. Unwilling to voluntarily give up on consistency, researchers typically rely on consistency guaranteeing network communication protocols to implement these strategies. Examples include the multicast total ordering as used in Postgres-R [21], RSI-PC as used in Ganymed [26], the total ordering certifier as used in Tashkent and related designs [13][22][7], Pub/Sub as used in Ferdinand [17], and the deterministic total pre-ordering [29]. Their performance is in theory upper-bounded by the centralized service implementing these protocols, and in some cases also by that of the snapshot isolation [4]. In contrast, our design differs fundamentally from these systems in terms of the distributed system model. Relying on global time, we managed to eliminate all centralized services and more efficiently implement the partitioning, replication, and caching. At the partition level, our design may also be considered a much simplified special case of the snapshot isolation, where not matter how high the query load is, only one snapshot is taken for each scheduled release interval.

Recognizing the CAP tradeoff [18], NoSQL systems like Dynamo [12], MongoDB [25], CouchDB [11], and Cassandra [2] etc. conscientiously sacrifice 1SR for better scalability, but the inconsistency exposed thereafter is undesirable even for non-critical web services such as social networks. More recent NoSQL systems such as PNUTS [9] slightly tighten this up but are still lacking. Different from these systems, our design guarantees 1SR by design. Performance wise, since our design pushes a larger portion of the increased load to the clients, we anticipate performance advantages under higher load. Such advantages are inherently true even compared to much faster systems that move all data to the in-memory distributed cache, for the same reasons that more highways and faster cars do not diminish the advantages of the mass transit systems.

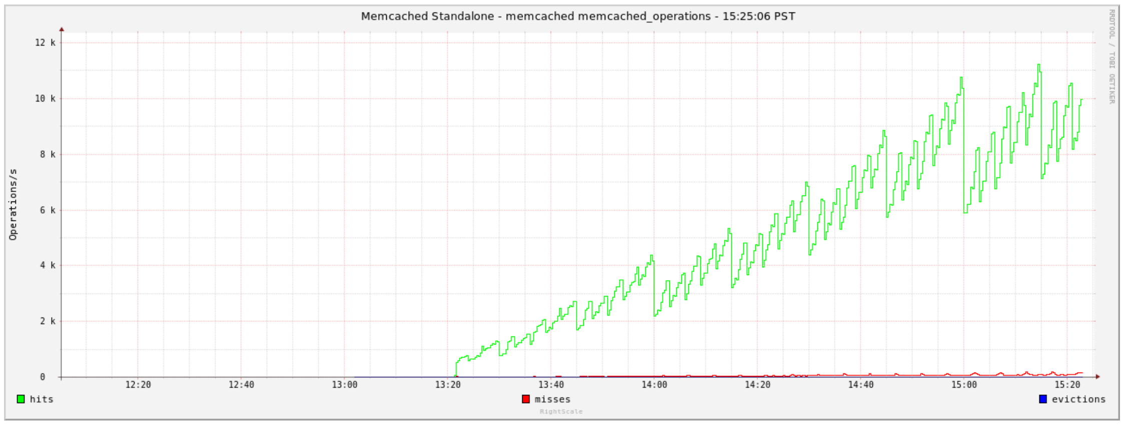


Figure 7. Cache operations at the standalone memcached server

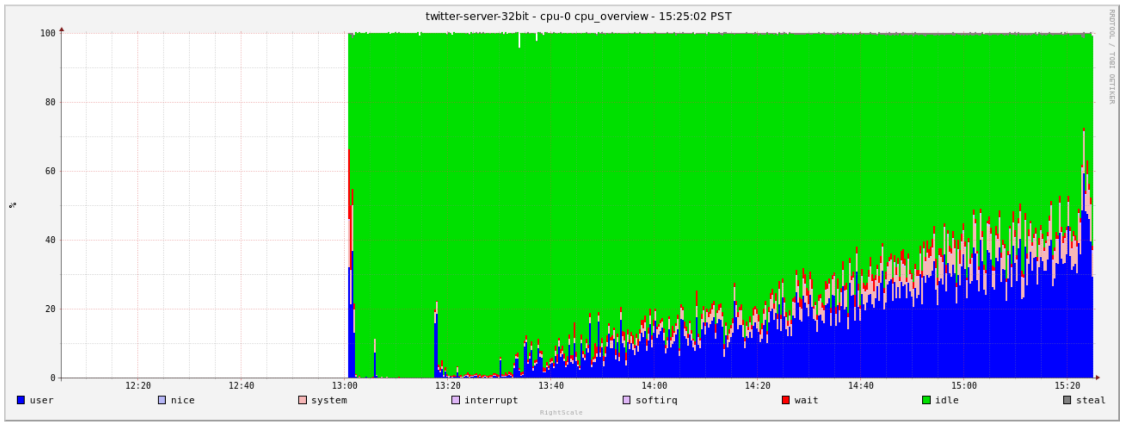


Figure 8. CPU load of a server

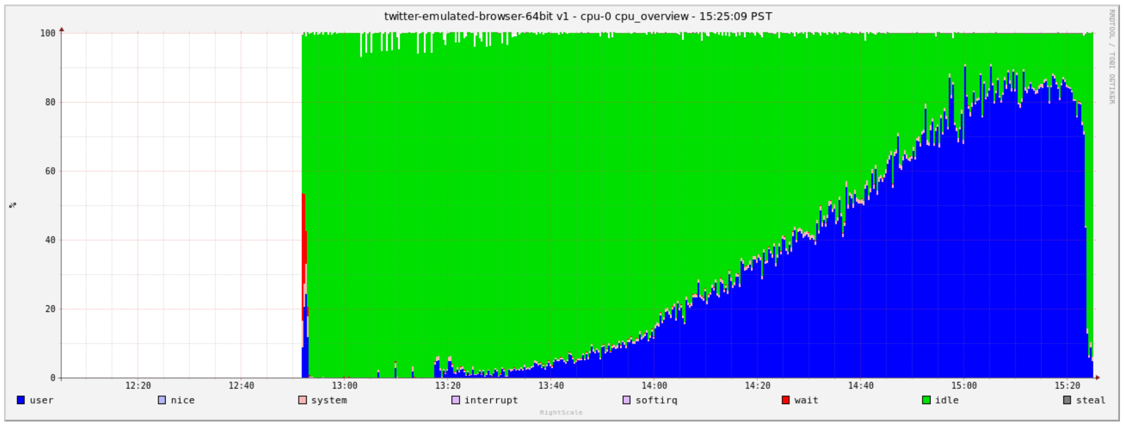


Figure 9. CPU load of an emulated browser

Trading staleness for scalability isn't a new idea, but the previous systems didn't attempt to preserve 1SR [1][16][19][5]. Our approach is the first known to us that guarantees 1SR under this tradeoff.

This paper is primarily inspired by Roy Fielding's blog post on RESTful feed following [15], and much of the experimental verification is adopted from the Yahoo! PNUTS experiment [27]. Fielding's approach is distinctively timing-based, but he did not elaborate on the theoretical foundation, the replication control algorithm, the freshness-scalability tradeoff involved, and the database partitioning, replication, and caching details. His implementation is based on a single centralized server, while ours is based on shared-nothing distributed systems. Nevertheless, his emphasis on the RESTful design [14] and the web architecture [3] reminds us that scalability is not a pure database endeavor but requires a holistic view of the system.

7. Discussions and Future Work

Because the server side scalability is known to be the primary bottleneck for the current social networking applications, in this paper we evaluate this new approach mainly from the server's perspective. We have not delved into its detailed effects on the real web browsers. But as an indicator, we have observed a peak client load of approximately 2 queries per second per client with cloud instances running the emulated browsers. When running a real browser on a physical machine the result may be slightly different due to various factors including the absence of the hypervisor isolation and the cloud CPU throttling, the real browsers' performance differences, and their implementations of the client-side database. This will be the topic for further research, but in general we are optimistic that the clients' processing capability should not pose an intractable bottleneck. This is because in our approach the same-client queries beyond the rate of once per 5 seconds do not require additional client and server processing. The browser will simply respond with the same, client cached response generated for an earlier request. If the higher query rate is caused by more clients initiating queries concurrently, these additional clients also bring in more processing power to counterbalance the increased client side processing needs.

Another practical concern for this approach is the wide adoption of the mobile devices as the social networking clients. While the processing capabilities of these devices may be continuously improving, the network bandwidth provided for these devices is harder to reach a satisfying level. For these non-performing clients, it is still possible to tier the web services such that many emulated browsers we used in the experiment can be deployed at the edge of the web (e.g., CDN) and repurposed as proxy servers for them. At least in theory this does not alter the linear scalability property of this approach.

The third practical concern is the implementation of global time. In our experiment all clients and servers frequently synchronized their local times with authoritative time sources using ntpd. The small time skews resulting in this approach did not pose a problem for the duration of our experiment. Even if the clients lag far behind the servers, the system performance should not decline significantly if sufficient web cache is provided. Nevertheless, in practice the assumption may be too strong for all the clients and servers to always maintain global time. In a follow-up research we relax this condition to only requiring all the servers to be properly synchronized. The client must send one extra request to detect the global server time before any timeline request can be processed. Under such relaxation, we can still show that the consistency guarantee and the linear scalability property are largely maintained.

Finally, we also noticed a potential problem with the load balancing. We employed a naïve database partitioning strategy but have not built any load elasticity and protection. Under such circumstances, even if all the other servers were way below their capacities, if one partition server encountered aberrant load spike and crashed, the whole system would crash. This issue may be addressed in the future work. We also expect to further this research by experimenting with different staleness levels to investigate their impacts on scalability, and extending load generation capabilities to further verify the linear scalability property.

To summarize, in this paper we describe, implement, and evaluate a novel method that can simultaneously achieve scalability and consistency in feed following applications built on shared-nothing distributed systems. In our experiments the servers scaled linearly, and sustained sufficiently high workloads to be of practical use for small to medium size social networks. The cost of running 6 low-end servers was fairly reasonable for the performance and the capacity they delivered.

We also demonstrate for the first time the feasibility of a new design pattern that consistently trades freshness for better scalability. This is achieved by assuming the availability of global time in a shared nothing distributed system, timing the queries with a pre-published release schedule, pushing much of the personalized query workload to the clients, and more efficiently partitioning, replicating, and caching.

References

[1] Alonso, R., Barbara, D. and Garcia-Molina, H. 1990. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*. 15, 3 (Sep. 1990), 359-384.

[2] Apache Cassandra, http://cassandra.apache.org/.

[3] Architecture of the World Wide Web, Volume One. http://www.w3.org/TR/webarch/.

[4] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* (May 1995). 1995, 1–10.

[5] Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R. and Tamma, P. 2006. Relaxed-currency serializability for middle-tier caching and replication. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (Chicago, IL, USA, 2006), 599–610.

[6] Bernstein, P.A., Hadzilacos, V. and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company.

[7] Bornea, M.A., Hodson, O., Elnikety, S. and Fekete, A. 2011. One-copy serializability with snapshot isolation under the hood. *2011 IEEE 27th International Conference on Data Engineering (ICDE)* (Apr. 2011), 625–636.

[8] Brewer, E.A. 2000. Towards robust distributed systems (abstract). *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2000), 7–.

[9] Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D. and Yerneni, R. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow*. 1, 2 (2008), 1277–1288.

[10] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing*. (2010), 143–154.

[11] CouchDB, http://couchdb.apacheorg/.

[12] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. 2007. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev*. 41, 6 (Oct. 2007), 205–220.

[13] Elnikety, S., Zwaenepoel, W. and Pedone, F. 2005. Database Replication Using Generalized Snapshot Isolation. *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems* (2005), 73–84.

[14] Fielding, R.T. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. University of California.

[15] Fielding, R.T. 2008. Paper tigers and hidden dragons. http://roy.gbiv.com/untangled/2008/paper-tigers-and-hidden-dragons.

[16] Gallersdörfer, R. and Nicola, M. 1995. Improving performance in replicated databases through relaxed coherency. *Proceedings of the 21th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1995). 1995, 445–456.

[17] Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C. and Tomasic, A. 2008. Scalable query result caching for web applications. *Proc. VLDB Endow*. 1, 1 (2008), 550–561.

[18] Gilbert, S. and Lynch, N. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*. 33, 2 (2002), 51-59.

[19] Guo, H., Larson, P. and Ramakrishnan, R. 2005. Caching with "good enough" currency, consistency, and completeness. *Proceedings of the 31st International Conference on Very Large Data Bases* (2005), 457–468.

[20] Herlihy, M.P. and Wing, J.M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst*. 12, 3 (Jul. 1990), 463–492.

[21] Kemme, B. and Alonso, G. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. *Proceedings of the 26th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 2000), 134–143.

[22] Krikellas, K., Elnikety, S., Vagena, Z. and Hodson, O. 2010. Strongly consistent replication for a bargain. *2010 IEEE 26th International Conference on Data Engineering (ICDE)* (Mar. 2010), 52–63.

[23] Lamport, L. 1986. On interprocess communication. *Distributed Computing*. 1, 2 (Jun. 1986), 86-101.

[24] Lynch, N.A. 1996. *Distributed Algorithms*. Morgan Kaufmann.

[25] MongoDB, http://www.mongodb.org/.

[26] Plattner, C. and Alonso, G. 2004. Ganymed: scalable replication for transactional web applications. *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware* (Toronto, Canada, 2004), 155–174.

[27] Silberstein, A., Machanavajjhala, A. and Ramakrishnan, R. 2011. Feed following: the big data challenge in social applications. *Databases and Social Networks* (New York, NY, USA, 2011), 1–6.

[28] Silberstein, A., Terrace, J., Cooper, B.F. and Ramakrishnan, R. 2010. Feeding frenzy: selectively materializing users' event feeds. *2010 International Conference on Management of Data* (Indianapolis, IN, United states, 2010), 831-842.

[29] Thomson, A. and Abadi, D.J. 2010. The case for determinism in database systems. *Proceedings of the VLDB Endowment*. 3, (Sep. 2010), 70–80.

[30] Vogels, W. 2009. Eventually consistent. *Communications of the ACM*. 52, (Jan. 2009), 40–44.

Executing Web Application Queries on a Partitioned Database

Neha Narula
MIT CSAIL

Robert Morris
MIT CSAIL

Abstract

Partitioning data over multiple storage servers is an attractive way to increase throughput for web-like workloads. However, there is often no one partitioning that yields good performance for all queries, and it can be challenging for the web developer to determine how best to execute queries over partitioned data.

This paper presents DIXIE, a SQL query planner, optimizer, and executor for databases horizontally partitioned over multiple servers. DIXIE focuses on increasing inter-query parallel speedup by involving as few servers as possible in each query. One way it does this is by supporting tables with multiple copies partitioned on different columns, in order to expand the set of queries that can be satisified from a single server. DIXIE automatically transforms SQL queries to execute over a partitioned database, using a cost model and plan generator that exploit multiple table copies.

We evaluate DIXIE on a database and query stream taken from Wikipedia, partitioned across ten MySQL servers. By adding one copy of a 13 MB table and using DIXIE's query optimizer, we achieve a throughput improvement of 3.2X over a single optimized partitioning of each table and 8.5X over the same data on a single server. On specific queries DIXIE with table copies increases throughput linearly with the number of servers, while the best single-table-copy partitioning achieves little scaling. For a large class of joins, which traditional wisdom suggests requires tables partitioned on the join keys, DIXIE can find higher-performance plans using other partitionings.

1 Introduction

High-traffic web sites are typically built from multiple web servers which store state in a shared database. This architecture places the performance bottleneck at the database. When a single database server's performance is not sufficient, web sites typically partition data tables horizontally over a cluster of servers.

There are two main approaches to executing queries on partitioned databases. For large analytic workloads (OLAP), the approach is to maximize parallelism within each query by spreading the query's work over all servers [11, 17]. In contrast, the typical goal for workloads with many small queries (OLTP) is to choose a partitioning that allows most queries to execute at just a single server; the result is parallelism among a large set of concurrent queries. Queries that do not align well with the partitioning must be sent to all servers. Many systems have addressed the problem of how to choose good partitionings for these workloads [2, 4, 7, 19].

Some workloads, however, execute small queries but do not partition cleanly, as different queries access the same table on different columns. For example, one query in a workload may access a `users` table using the `id` column, while another accesses the table with the `username` column. No single partitioning will allow both queries to be sent to just one server; as a result the workload does not *cleanly partition*. Workloads that cleanly partition allow capacity to scale as servers are added. In contrast, queries that restrict on columns other than the partition column do not scale well, since each such query must be sent to all servers.

This paper suggests the use of *table copies* partitioned on different columns, in order to allow more queries in a workload to partition cleanly. This idea is related to the pre-joined projections of tables used in column stores [20, 23, 29] to increase intra-query parallelism, but here our goal is to increase inter-query parallelism.

In order to exploit partitioned data, including table copies, this paper presents the DIXIE query planner. DIXIE focuses on small queries that can execute on a subset of the servers if the right table copies are available. It addresses an intermediate ground between the whole-table queries of OLAP workloads and the straightforward clean partitioning of some OLTP workloads.

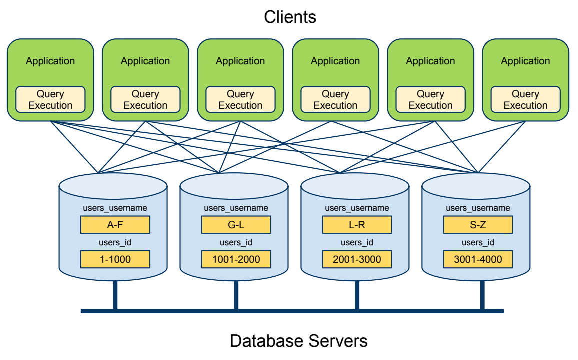


Figure 1: Web site architecture. Front-end web servers running application code issue queries to a cluster of database servers. The users table is copied and partitioned, once by user name and once by ID.

DIXIE uses two key ideas. First, for small queries the overhead at the server may be larger than the data handling cost; these queries do not benefit from distributing the work of an individual query among many servers. As an example, on our experimental MySQL setup, a simple query that retrieves no rows consumes almost as much server CPU as a query that retrieves one row. As a result, cluster throughput can be dominated by the resources wasted due to overhead if small queries are sent to all servers. In the extreme, sending each query to all servers in an N-server cluster may result in $1/N$ the throughput of sending each query to one server.

Second, DIXIE's optimizer uses a novel cost model which appropriately weights query overhead against the costs of data retrieval. For example, DIXIE may prefer a plan that retrieves more rows than another plan, but from fewer servers, if it predicts that the the reduction in overhead from the latter plan outweighs the per-row cost of the former.

We have implemented DIXIE as a layer that runs on each client and intercepts SQL queries; see Figure 1. Applications which use DIXIE can be written as if for one database with a single copy of each table. We have evaluated it on a cluster of ten servers with a traced Wikipedia workload and with synthetic benchmarks. With appropriately chosen table copies, DIXIE provides 3.2X higher throughput for the Wikipedia workload than the best single-table-copy partitioning.

2 Problem

The following example illustrates costs when executing queries over a partitioned database. Consider a simple case with a users table, containing `id`, `group`, `name`, and `address` columns. The `id` column is unique, and the table will be range partitioned over ten servers using some column which we will choose. Assume we would like to issue the following query:

```
Q1: SELECT * FROM users WHERE id = ?
```

Executing many concurrent queries choosing `id` values randomly and uniformly, if we send each query to only one of the ten servers one would certainly expect a throughput higher than if the query was sent to all servers. However, it is unclear exactly how much these costs would differ, since in the ten server case nine of the servers do not need to retrieve or send back any rows.

The cost on the server of executing a simple query like the one above that returns zero data is 90% of the server CPU time of executing a query which returns a small amount of data: 0.36 ms vs. 0.4 ms (these numbers are server processing costs only, they do not include client or network transit time). Section 8 describes the experimental setup. This shows that requesting a row, even if there is no data to read and transfer back to the client, incurs a very significant cost. A profile of the MySQL server shows that the cost consists of optimizing the query, doing a lookup in the btree index, and preparing the response and sending it to the client. On a system executing many concurrent queries, we measure a 9.1X increase in overall throughput if each query is sent to one server instead of ten.

Thus this query would would incur much less cost if the table were range partitioned by `id`, and requests could be sent to one server, as opposed to partitioning on some other column, requiring requests to be sent to all ten servers.

Unfortunately, a single partitioning of a table does not always suffice. Many applications issue queries which access the same table restricting on different columns. Analysis of Wikipedia shows that for a table which comprised 50% of the overall workload, half the queries on that table restricted on one column, half on another. This pattern also occurs in social networking applications with many-to-many relationships [19]. Consider a different query:

```
Q2: SELECT * FROM users WHERE group = ?
```

Partitioning on the `id` column would cause Q2 to go to all servers, while partitioning on the `group` column would cause Q1 to go to all servers. There is no way to cleanly partition this table for both queries.

Storing multiple copies of the `users` table partitioned in different ways can solve this problem. If we create two copies of the `users` table, one partitioned on `id` and the other on `group`, we can efficiently execute both Q1 and Q2. The cost is a doubling of storage space and a doubling in the cost of updates. For workloads dominated by reads the tradeoff may be worthwhile.

With more ways to access a table, query planning becomes more complicated. A smart query optimizer should choose plans which avoid unnecessary lookups, given the appropriate table copies. Properly optimizing these workloads is not just a matter of directing a single query to

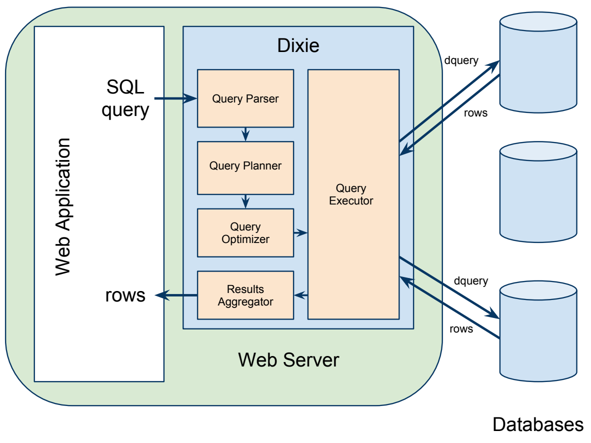


Figure 2: DIXIE's design. There may be many web servers talking to the database server cluster, and thus many instances of DIXIE.

the single appropriate partition. Web applications issue complex queries which often have many potential plans and require accessing multiple servers. For instance, the existence of table copies might affect the table order used in a join, or whether to execute a join by pushing down the query into the servers. The problem DIXIE solves is the selection of plans for executing queries on databases with multiple partitioned copies of tables, with a goal of increasing total multi-client throughput by (among other considerations) decreasing query overhead.

3 Overview and System Model

DIXIE is a query planner intended to operate within a clustered database system. DIXIE takes as input SQL queries written for a single database, plans and executes them on the cluster, and returns rows to the client. Some other mechanism in the overall clustered database handles transactions, if necessary. DIXIE relies on each server in the cluster taking care of local planning and optimization, leaving DIXIE with the task of deciding how to divide the work of each query among the servers.

The key insight behind DIXIE is to reduce the number of servers involved in executing a query by using copies of tables partitioned on different columns, thus improving throughput when there are many concurrent queries. The challenge lies in a choosing a good plan to efficiently use server CPU resources.

Figure 2 shows the architecture of DIXIE. The web application on each front-end web server generates SQL queries intended for a single database server and passes them to the local DIXIE library. DIXIE parses each query, and then the planner generates a set of candidate plans for the query. The query optimizer evaluates the cost of each plan, chooses the plan with the minimum predicted cost, and sends this plan to the executor. The executor follows the plan by sending requests to the cluster of database servers (perhaps in multiple rounds), filtering and aggregating the retrieved rows, and returning the results to the application. Queries generated by the web application are *queries* and the requests generated by DIXIE to the backend database servers are *dqueries*.

The developer provides DIXIE with a *partitioning schema*. This schema identifies the copies of each table and the column and set of ranges by which each copy is partitioned. All copies use range partitioning. DIXIE adds measured selectivity estimates for each column in each table to the schema.

Table copies are referred to by the table name and partitioning key. In the application shown in Figure 1 in Section 1, the `users` table has two copies, one partitioned on `username` and one partitioned on `id`.

DIXIE's goal is to increase total throughput for workloads with many independent concurrent queries, each of which involves only a small fraction on the database. This goal is consistent with the needs of many web applications. Web applications must retrieve data quickly to render HTML pages for a user, and so developers often expect to serve the working set of data from memory. DIXIE's cost estimates assume that most rows are retrieved from memory instead of disk, and that each column used for partitioning has a local index on each server, so that the cost of looking up any row is roughly the same. DIXIE also assumes that applications only issue equality join queries on a small number of tables. These assumptions are consistent with the design of Wikipedia, for example, which does not execute joins on more than three tables, and of Pinax [24], an open source suite of social network applications. Extending DIXIE's cost model to handle workloads which frequently retrieve data on disk is future work. DIXIE handles a select-project-join subset of SQL.

Adding copies of tables can reduce the costs of read queries at the expense of increasing the cost of write queries – writes must be executed on all table copies. Fortunately the applications we examined have very low write rates – by one account Wikipedia's write rate is 8% [6] and based on a snapshot of MySQL global variables provided by the Wikipedia database administrator on 7/11/2011, the write rate was as low as 1.7% (including all `INSERT`, `UPDATE`, and `DELETE` statements). DIXIE is a read query optimizer; but we examine the throughput effect on the server of writing to multiple table copies in Section 8, and show that in a synthetic workload on ten servers with write rates as high as 80% the benefit to reads outweighs the added expense of writing to two copies.

Choosing an appropriate set of table copies and partitions is important for good performance of a partitioned

```
SELECT *
FROM   blogs, comments
WHERE  blogs.author = 'Bob'
AND    comments.user = 'Alice'
AND    blogs.id = comments.object
```

Figure 3: Q3, Alice's comments on Bob's blog posts.

database, but is outside the scope of this work; DIXIE requires that the developer establish a range partitioning beforehand. Our experience shows that potential partitioning keys are columns frequently mentioned in the `WHERE` clause of queries. A subset of DIXIE's techniques would work with another partitioning method like hash partitioning, but in that case DIXIE would not be able to optimize range queries.

4 Query Planning

DIXIE generates a set of plans corresponding to different ways to execute the original query. It estimates the cost of each plan using a cost model, and then executes the lowest cost plan. DIXIE generates plans similar in style to a traditional distributed query optimizer such as R* [17], with a few key differences. First, DIXIE rewrites the application SQL query into many SQL dqueries; essentially it transforms nodes and subtrees in the query plan into SQL statements which can be issued to the RDBMS backend servers. A DIXIE *query plan* is a description of steps to take to execute a query. Second, DIXIE incorporates partitionings of table copies, so it generates plans with different table copies that issue dqueries to subsets of the servers.

As an example, Figure 3 shows a query which retrieves all of Bob's blog posts on which Alice has written a comment. DIXIE will decompose this SQL query into smaller dqueries for each table (or combination of tables) in the query. We assume a partitioned database over N servers with the `blogs` table partitioned on the `author` column and the `comments` table partitioned on the `user` column. One of the plans DIXIE generates for the example query retrieves all of Bob's blogs using the `author` table copy, then retrieves all of Alice's comments on Bob's blogs using the `user` table copy restricting by the blog ids returned in the first step, and finally assembles the results in the client to return to the application. If the tables had copies partitioned on the join keys, it would also generate a plan which sent the join to every server and unioned the results (a *pushdown join*).

DIXIE goes through four stages to generate a set of plans: rewriting the query to separate and group the clauses in the predicate by table, creating different join orders, assigning table copies, and narrowing the set of partitions. The planner generates a *step*, which explains how to access a table, for each table in the query. Each step has a SQL statement to execute on the table, a specific table copy for the table in the step, a list of partitions to which to send the request, a description of what values need to be filled in from previous steps, and what values to save from this step to fill in the next one. A pushdown join step will mention multiple tables.

After DIXIE chooses the generated plan with the lowest estimated cost, its executor saves the results of each step in a temporary table both to fill in the next step and to compute the final result. The query plan specifies which dqueries the executor should send in what order (or in parallel), what data the executor should save, how it should substitute data into the next query, and how to reconstruct the results at the end.

4.1 Query Rewriting

DIXIE seeks to create plans which push down projections and filters into the servers to reduce the amount of data returned to the client. To do this it rewrites each query into queries on each table. If tables have partitionings on join keys, DIXIE will also generate plans that execute the entire join in the database servers (or parts of the join tree in the servers), which we describe in the next section.

Consider a `SELECT` query which uses one table:

```
Q4: SELECT * FROM T WHERE T.a=X
    AND (T.b=Y OR T.c=Z)
```

DIXIE needs to generate a set of dqueries for this query, with each dquery accessing a single table, perhaps on a single partition. To decide what partition(s) a query must use, DIXIE observes that an AND must run on the intersection of the sets of partitions needed by the ANDed expressions, and that an OR must run on the union. To ease this analysis, DIXIE flattens a query's predicate into disjunctive normal form, an OR of ANDs. DIXIE would rearrange Q4's predicate thus:

```
WHERE (T.a=X AND T.b=Y)
   OR (T.a=X AND T.c=Z)
```

DIXIE will create one dquery which retrieves X and Y, and another which can be executed in parallel retrieving X and Z. The results must be unioned together. DIXIE will use the partitioning scheme to determine which partitions should execute each part of the query.

4.2 Join Orderings

DIXIE's current implementation considers all possible join orderings, and creates a plan for each, with each step accessing one table. DIXIE also generates pushdown joins by combining every prefix of the sequence of tables in a join ordering, and creating a plan where the first step of the plan is a pushdown join dquery on the tables in the

```
1  SELECT * FROM blogs WHERE author='Bob'
2  SELECT * FROM comments
    WHERE user='Alice' AND object=?

1  SELECT * FROM comments
    WHERE user='Alice'
2  SELECT * FROM blogs
    WHERE author='Bob' AND id=?
```

Table 1: Set of plans for the query in Figure 3.

```
1  SELECT * FROM blogs,comments
    WHERE blogs.id=comments.object
    AND author='Bob' AND user='Alice'
```

Table 2: Additional pushdown join plan for the query in Figure 3, with blogs.id and comments.object table copies.

prefix. DIXIE also considers all possible combinations of table copies. If T is the set of tables in the query and there are c_t table copies for table $t \in T$, the original size of the set of plans is:

$$|T| * |T|! * \prod_{t \in T} c_t$$

DIXIE discards any plan with a pushdown join step where there are not matching table copies partitioned on the join keys. For the query in Figure 3 the planner would create the following set of table orderings:

(`blogs`,`comments`),(`comments`,`blogs`)

From that it would generate the two plans shown in Table 1, requesting rows from `blogs` and `comments` in different orders. The pushdown join plan is invalid with the current set of table copies, `blogs.author` and `comments.user`, so DIXIE would prune this plan.

If we had `blogs.id` and `comments.object` table copies, DIXIE would generate sixteen plans, using all combinations of the new table copies, join orderings, and prefixes. In particular, it would generate the pushdown join plan shown in Table 2.

4.3 Assigning Partitions

Every plan is a sequence of steps, one for each table or combination of tables. DIXIE converts this into a sequence of *execution steps*. An execution step is a SQL query, a table copy for each table in the query, and a set of partitions. The planner can narrow the set of partitions based on the table copies and expressions in each step; for example, in the plan where the planner used a copy of table `blogs` partitioned on `author`, the first step of the first plan in Table 1 could send a dquery only to the partition where `blogs.author = 'Bob'`. The set of partitions for each step might be further narrowed in the executor, depending on values that are retrieved and substituted from previous steps. Each execution step also contains instructions on what column values from the results of the previous dqueries to substitute into this step's dqueries, by storing expressions which refer to another table. For example, step two of the first plan in Table 1 would store an expression indicating that the `comments.object` clause required data from the `blogs.id` values that are retrieved in the first step. DIXIE would then convert this into an `IN` expression. Substitution is done during execution. If a step does not require data from any other step, it can be executed in parallel with other steps.

Table 3 shows a set of three plans that DIXIE would generate for Q3 in Figure 3. This table shows the SQL to be issued in the dquery in each step of each plan in the left column. Plans 1 and 2 have two steps each, Plan 3 has one. None of these plans have steps which can be executed in parallel. The middle column uses B, C, and R to represent the intermediate storage for the results as the steps are executed. The second steps of Plans 1 and 2 use `B.id` and `C.object` as placeholders for values returned in the previous steps, which it substitutes into these dqueries during execution. The right column has four parts for each step, and for the purposes of planning we limit our explanation to the first two: the table copy (or copies) used in the step and the partitions to which to send the dquery in the step. We will discuss n_r and n_s in Section 5 when explaining optimization.

Step one of Plan 1 can be sent to just the partition with Bob's blogs, p_1, and step two can go just to the partition with Alice's comments, p_0, independent of the results returned in step one since we intersect the sets of partitions for ANDs. Plan 3 must execute a dquery for every partition because it is not using the table copies which partition on columns used in the most restrictive clauses. DIXIE generates more plans for this query, but these are the three most likely to have the smallest cost, because they use table copies partitioned on columns mentioned in the query.

All of the queries Wikipedia and Pinax issued are simple enough that DIXIE's query planner can efficiently generate a plan for every combination of order of tables in the join and possible table copy. In the applications we examined no query ever joined across more than three tables and no table had more than three copies. However, the number of plans generated is exponential in the size of the number of tables in the query, and existing pruning techniques could be used to reduce the number of plans considered [21].

5 Cost Model

DIXIE predicts the cost of each generated plan using a cost model designed to estimate server CPU time, and chooses the lowest cost plan for execution. DIXIE models the cost

Query Steps		Table Copy Partitions n_r, n_s

Plan 1:
`SELECT *`
`FROM blogs`
`WHERE author = 'Bob'`
	$\rightarrow$ B	`author`
		p_1
		20, 1

`SELECT *`
`FROM comments`
`WHERE user = 'Alice'`
`AND object IN (B.id)`
	$\rightarrow$ C	`user`
		p_0
		1, 1

Plan 2:
`SELECT *`
`FROM comments`
`WHERE user = 'Alice'`
	$\rightarrow$ C	`user`
		p_0
		10, 1

`SELECT *`
`FROM blogs`
`WHERE author = 'Bob'`
`AND id IN (C.object)`
	$\rightarrow$ B	`author`
		p_1
		1, 1

Plan 3:
`SELECT *`
`FROM blogs,comments`
`WHERE author = 'Bob'`
`AND user = 'Alice'`
`AND blogs.id`
`=comments.object`
	$\rightarrow$ R	`id,`
		`object`
		$p_0,...,p_N$
		1, N

Table 3: Candidate query plans generated by DIXIE for the query in Figure 3

of a plan by summing the costs of each step in the plan, and estimates the cost of a step by summing the query overhead and the row retrieval costs in that step. Minimizing CPU time, rather than elapsed time, is appropriate to the goal of inter-query parallelism.

$$cost_{step} = cost_r * n_r + cost_s * n_s$$

Query overhead, $cost_s$, is the cost of sending one dquery to one server. $cost_r$ is the cost of data retrieval for one row, which includes reading data from memory and sending it over the network. n_r is the number of rows sent over the network to the client, which we assume is close to the number of rows read in the server since most data retrieval is index lookups. Since all rows are indexed and in memory, row retrieval costs do not include any disk I/O costs. DIXIE's optimizer computes cost per step as the sum of the row retrieval cost per row times the number of rows read in the step and the cost of receiving a query at the server times the number of dqueries sent in the step, and the total query cost as the sum of the cost of its steps. It is irrelevant to cost estimation whether the steps in the plan were executed in parallel or sequentially, since we are interested in minimizing overall server CPU time.

Costs are only used to compare one plan against another, so DIXIE's actual formula assumes $cost_s$ is 1 and scales $cost_r$. Section 8 shows how to determine $cost_r$ and $cost_s$. DIXIE uses table size and selectivity of the expressions in the query to estimate n_r, a proxy for the number of rows that might be read in the server. It uses the cost functions below to estimate n_r, the number of rows retrieved, and n_s, the number of servers queried, for each step $step$.

$$selectivity(s) = \prod_{c \in s} \frac{1}{|dk_c|}$$

$$n_r = table\_size_s * selectivity(step)$$
$$n_s = |partitions_{step}|$$

To compute selectivity, DIXIE stores the number of rows in each table and dk_c, the number of distinct values in each column. DIXIE could be extended to support histograms of values and dynamically updating selectivity statistics, by periodically querying the tables and rewriting the partitioning plan. We leave this to future work. The selectivity function shown assumes a `WHERE` clause with only ANDs, so it can multiply the selectivity of the different columns mentioned in the query.

Table 3 shows three plans for the query shown in Figure 3. Assume the statistics in the partitioning plan predict that Bob has authored twenty blog posts, Alice has written ten comments, and Alice has commented once on one of Bob's blogs. Then the optimizer would assign costs according to the formula described above, as shown in Table 3. The total costs for Plans 1, 2, and 3 are as follows:

$$cost(\text{Plan 1}) = 21 * cost_r + 2 * cost_s$$
$$cost(\text{Plan 2}) = 11 * cost_r + 2 * cost_s$$
$$cost(\text{Plan 3}) = cost_r + N * cost_s$$

6 Query Executor

The executor takes a query plan as input and sends dqueries for each step in the plan to a set of backend database servers. The executor executes independent steps in parallel, and steps which require data from another step in sequence. DIXIE assumes that dqueries request small enough amounts of data that the executor can temporarily store the results from each step in the client. The executor substitutes results to fill in the next step of the plan with values retrieved from the previous steps' dqueries. For example, in Plan 1 in Table 3, the executor would insert `blog_post.id` values from step one into the `comments.object` clause in step two.

The executor can often reduce the number of dqueries it issues by further narrowing the set of servers required to satisfy a step's request. This means that the cost initially assigned to a step by the optimizer may not be correct. For example, the returned values from the first step of a join may all be on one partition, meaning the executor will only need to send one dquery for the second step to one server, reducing the query overhead and thus reducing the total cost. The optimizer has no way of knowing this at the time when it chooses a plan for execution, and so there are cases where it will not select the optimal plan.

The executor uses an in-memory database to store the intermediate results and to combine them to return the final result to the client. This produces correct results because DIXIE will always obtain a superset of the results required from a table in the join. As it executes dqueries, the executor populates subtables for every logical table in the dquery (not one per table copy). After completion, it uses the in-memory database to execute the original query on the subtables and return the results to the client.

7 Implementation

We have implemented a prototype of DIXIE in Java. It accepts SQL queries and produces dqueries which it executes on a cluster of MySQL databases. DIXIE expects table copies to be stored as different tables in the MySQL databases. The prototype uses JSQLParser [14] to create an intermediate representation of each SQL query. JSQLParser is incomplete, so we altered JSQLParser to handle `IN`, `INSERT`, `DELETE`, and `UPDATE` queries.

We tested the effectiveness of DIXIE using queries generated by Wikipedia and an open source suite of social web applications called Pinax [24], including profiles, friends, blogs, microblogging, comments, and bookmarks.

We implemented a simple partitioner which recommends partitions by parsing a log of application queries and counting columns and values mentioned in `WHERE` clauses. The simple partitioner then generates a partitioning plan with those columns split on ranges to evenly distribute query traffic to each partition. We found DIXIE to be useful in testing different partitioning schemes without changing application code.

DIXIE keeps static counts of number of rows, partitioning plans, table copies, and distinct key counts for each table, for use by its query optimizer. These are stored in configuration files which are read on start up once and not updated. A mechanism to update these configuration files on the fly as table copies are added and deleted or as table counts change could be implemented by regularly re-reading the files and updating in-memory data structures to use the new configuration and statistics.

DIXIE's executor saves intermediate results in a per-thread in-memory database, HSQLDB [13]. DIXIE then executes the original query against this in-memory database. An alternate implementation would have been to construct the response on the fly as results are returned from each partition and each step, but using an in-memory database allowed us to handle a useful subset of SQL without having to write optmized code to iterate over and reconstruct results. In the applications we examined, DIXIE never needed to execute a plan which read a large portion of a table into the client, but in a future version DIXIE will do so by requesting tables in chunks.

DIXIE is designed to address the problem of scaling reads. To simulate the costs added by writes, we execute writes sequentially in the client to each table copy, without any serialization between clients. In our experiments, concurrent writes to the same row could cause table copies to become out of sync. We believe this is acceptable since the purpose of this work is to measure the performance impact of added writes in the database servers. The application developer can use existing mechanisms for distributed transactions to manage writes to table copies; this could change what might have been single partition write transactions into distributed write transactions with DIXIE.

8 Evaluation

This section demonstrates DIXIE's ability to automatically exploit table copies to improve database throughput on a realistic web workload. The improvement increases with the number of servers, and is a factor of three compared to the best single-table-copy performance on ten servers and a factor of 8.5 over a single server. This section also explores the factors that DIXIE weighs when choosing among query plans and examines the accuracy of its cost prediction model.

8.1 Workloads and Experimental Setup

Database Workloads. The workload used in Section 8.2 models the Wikipedia web site: it uses a subset of a Wikipedia English database dump from 2008 [1] partitioned across 10 servers (the original Wikipedia database is not partitioned), a trace of HTTP Wikipedia requests [27], and a simulator which generates SQL queries from those requests [6]. The simulator uses the Wikipedia data, published statistics about caching in the application layers, and information from the Wikipedia database administrator to generate an accurate workload. The total database including indexes is 36 GB in size, but the 2008 workload only ends up using a subset of the data that fits in memory. We verify on read workloads that the servers are not using the disk. There are 100K rows in the `page` table, 1.5M rows in the `text` table, and 1.5M rows in the `revision` table (the most heavily queried tables). A majority of Wikipedia queries use the `page` table, restricting on the `page.title` or `page.id` columns. Figure 4 shows the schema of the `page` and `revision` tables.

`INSERT`, `UPDATE`, and `DELETE` queries are 5% of the overall workload, consistent with information provided by

```
page (id, namespace, title, ...)
revision (id, page, text_id, ...)
```

Figure 4: Partial Wikipedia page and revision table schemas.

the current Wikipedia database administrator. Writes are not transactional: the client sends independent writes to each server that needs to be updated. For example, when writing a table with multiple copies, the client sends a separate write to update each of the copies. The rest of the evaluation uses synthetic queries and data constructed to explore specific questions. Every column is indexed.

Hardware. All experiments run on Amazon EC2. Each database server is a "small instance" (one CPU core and 1.7GB of RAM). Experiments use ten servers unless indicated otherwise. Three "large" instances (each with five CPU cores and 12GB of RAM) generate client requests. The clients are fast enough that throughput is limited by the database servers in all experiments. All machines are in the same availability zone, so they are geographically close to each other. Each database server is running MySQL 5.1.44 on GNU/Linux, and all data is stored using the InnoDB storage engine. MySQL is set up with a 50MB query cache, 8 threads, and a 700 MB InnoDB buffer pool. This is sufficient for the working set of Wikipedia data in the workload, because the majority of the database is the text of Wikipedia pages, many of them not accessed.

Runtime Measurement. Before each experiment, DIXIE's planner and optimizer generate traces of plans from application SQL traces. During each experiment, multiple client threads run DIXIE's executor with a plan trace as input; the executor sends dqueries to servers and performs post-processing on the client. We pre-generate plans in order to reduce the client resources needed at experiment time to saturate the database servers. Planning and optimization take an average of 0.17ms per query.

Throughput is measured as the total number of application queries per second completed by all clients for a time period of 300 seconds, beginning 30 seconds after the clients start, and ending 30 seconds before stopping the clients. The traces are long enough that the clients are busy during the duration of measurement. Before measurement, a read-only version of each trace file is run all the way through the system to warm up the database cache and the operating system file cache, so that during the experiment the databases minimally use the disk.

8.2 Wikipedia

Wikipedia's 2008 workload benefits from both partitioning the data over multiple servers and from using more than one table partioning. Figure 5 shows the change in overall throughput for the same workload over 1, 2, 5, and 10 partitions, with and without an extra table copy. The best partitioning with only one copy per table, which partitions the `page` table by `page.id`, yields a total of 7400 QPS (application queries per second) across ten partitions (see Figure 5, 10P). In this partitioning, 25% of all Wikipedia queries generate dqueries to all partitions. Adding a copy of the `page` table partitioned on `page.title` reduces this number to close to zero, and increases overall throughput to 23347 QPS, a 3.2X increase. DIXIE automatically exploits the table copy to achieve this increase.

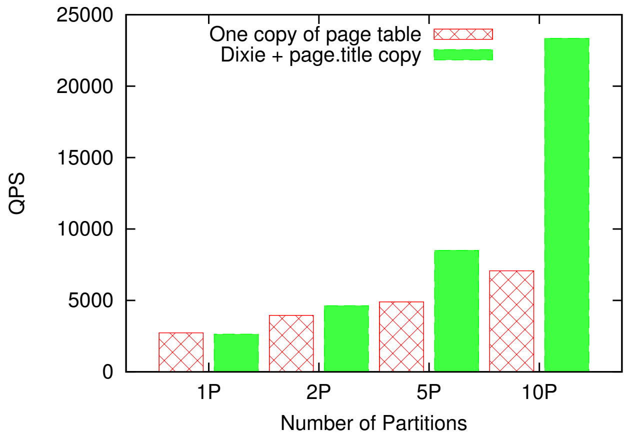


Figure 5: Throughput (queries per second) of the Wikipedia workload with 1, 2, 5, and 10 servers. The patterned red bars correspond to the best setup with only one copy of each table (this setup partitions the page table by page.id). The solid green bars correspond to a setup with an extra copy of the page table, partitioned by page.title.

In order that the data fit in server memory, the one and two partition cases use a dataset with only 10K rows in the `page` table. With one partition, it is better to have only one copy of each table to avoid extra writes. The benefit of the extra table copy increases with the number of partitions because sending a query to one partition instead of N frees $N-1$ query overhead's worth of CPU time for use by other queries.

To help explain the details of how the added table copy helps, we can divide the read-only portion of the Wikipedia query workload into three parts:

Single table, single-partition queries. These queries access a single table restricting on a column used as a partitioning key. These queries can be sent to one server.

Single table, multi-partition queries. These queries only access one table, but not on any partition key, so they must be sent to all partitions.

Multi-step queries. These are join queries which require accessing two or more tables. They can be executed either as a single pushdown join (which in this workload must be sent to all servers) or in multiple steps. Each step

A	`SELECT * FROM page` `WHERE page.id =` *i*
B	`SELECT * FROM page` `WHERE page.title =` *t*

C	`SELECT * FROM page,revision`
	`WHERE page.namespace = 0`
	`AND page.title =` *t*
	`AND revision.id = page.latest`
	`AND page.id = revision.page`

D	`INSERT INTO page`
	`VALUES (`c_o`,` c_1`, ...` c_n`)`

Table 4: Examples of Wikipedia page table queries.

is either a pushdown join on a subset of the tables in the join or a set of lookups on a single table, which can be modeled as a single-table query.

We will focus on queries that use the `page` table, which are affected by the addition of the `page.title` table copy. 50% of queries use the `page` table; examples of each type of these queries are shown in Table 4. Queries A and B are single table queries. Query A is a single-partition query, since it can be sent to the single partition containing the appropriate `page.id` value. Query B is originally a multi-partition query. Query C is a multi-step query which is executed as a pushdown join query and must be sent to all servers, and Query D inserts a row into the `page` table.

With the addition of a `page.title` table copy, Query B generates a dquery to one server instead of dqueries to all, reducing query overhead by a factor of ten. Query C requires DIXIE to choose between a pushdown join and joining in the client. In this workload DIXIE always chooses the latter. The next section shows how DIXIE makes that choice depending on the selectivity of the columns mentioned in the query.

Adding table copies can impose a penalty: writes must update all copies. Figure 6 shows the difference in throughput when adding a `page.title` table copy and increasing the percentage of writes. When writes are at 0%, using the `page.title` table copy to direct each query to one partition instead of all partitions achieves a 9.1X improvement in throughput. Throughput improvement decreases as the workload contains a higher percentage of writes. The table shows results from a benchmark with a mix of Queries B and D on the `page` table partitioned over ten servers, randomly and uniformly reading and inserting page rows.

8.3 Query Planning

For some join queries, DIXIE must choose between two plans: a two-step join and a pushdown join. To illustrate this choice, we examine Plans 1 and 3 from Table 3, which DIXIE generates from the application query in Figure 7.

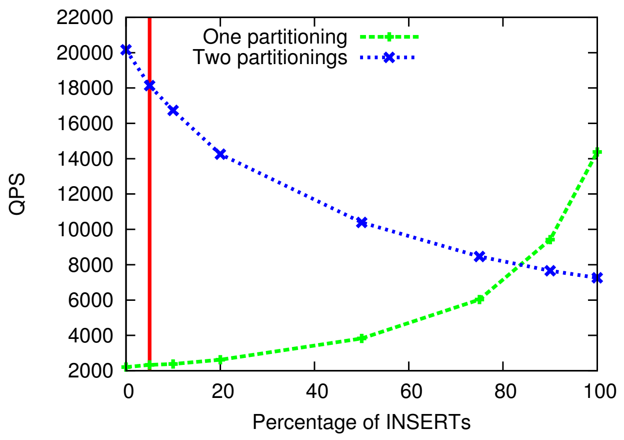


Figure 6: Throughput (queries per second) of a workload executing a combination of queries B and D in Table 4 on ten database servers. One line corresponds to a setup which partitions the page table only on page.id, and the second line corresponds to a setup with an extra copy of the page table, partitioned by page.title. The first setup sends reads to all ten servers and writes to one, the second setup reads from one server and writes to two.

```
SELECT   *
FROM     blogs, comments
WHERE    blogs.author = 'Bob'
AND      comments.user = 'Alice'
AND      blogs.id = comments.object
```

Figure 7: Alice's comments on Bob's blog posts.

If an optimizer were to mainly consider row retrieval cost, it would select Plan 3, the pushdown join, since it retrieves the fewest rows. The plan describes an execution which contacts all servers, but sends at most one or two rows back to the client.

Plan 1 contacts two servers in two steps: first to get Bob's blog posts, then to get Alice's comments on Bob's posts. Plan 1 requires sending back unnecessary rows in step one because it sends back all of Bob's blog posts, even though Alice only commented on one or two.

Figure 8 shows the throughputs for these plans with ten servers, varying the amount of data returned in step one of Plan 1 by increasing the number of blog posts per user. There are 1000 users and ten comments per user. A row in the blog posts table is approximately 900 bytes, and a row in the comments table is approximately 700 bytes. Queries use different values for "Alice" and "Bob" in Figure 7. The graph also shows DIXIE's cost predictions, based on the formula described in Section 5, inverted and scaled up to be in the same range as the measured QPS for comparison.

Figure 8 shows that if the query retrieves few enough rows in step one, the system can achieve a higher through-

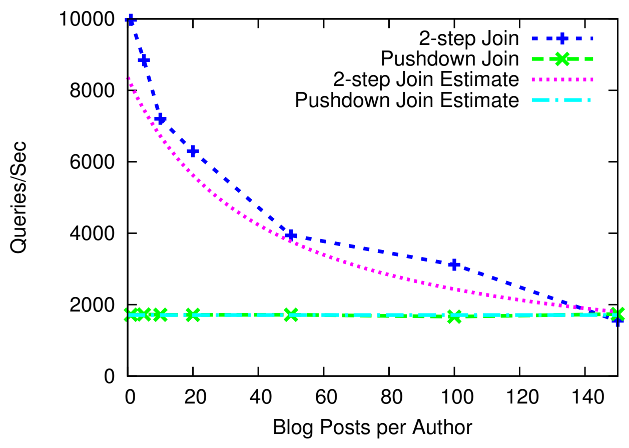


Figure 8: Throughput (queries per second) retrieved from ten database servers and DIXIE's predicted cost for each query plan.

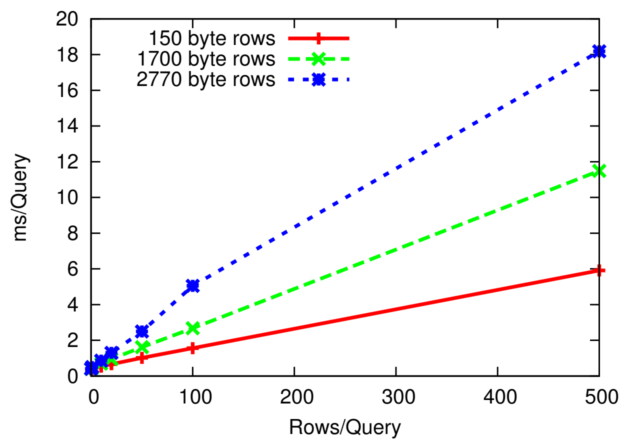


Figure 9: Measurement of milliseconds spent executing a query on a single EC2 MySQL server, varying the number of rows per query and bytes per row using multiple clients.

put with Plan 1 which sends dqueries to only two servers and retrieves more rows, than with Plan 3, which sends dqueries to all servers while retrieving fewer results. Using Plan 1 instead of Plan 3 when there are only ten blog posts per user gives a 4.2x improvement in throughput. This number would increase with more servers. When there are 145 rows returned per author Plan 1 is equivalent in throughput to Plan 3.

Since DIXIE's predicted query costs for the two plans cross shortly after 145 rows, it will usually choose the best plan. Its ability to do so depends on its having reasonable estimates for query overhead and row retrieval time. The next section investigates these two factors.

8.4 Cost Model

To test the accuracy of DIXIE's query cost models (described in Section 5) and measure the cost of per-query overhead, we set up a controlled set of experiments using one database based on the synthetic workloads. We varied the number of rows retrieved, the row size, and the percentage of queries sent to all shards. By varying the number of rows retrieved we can derive a ratio of query overhead to row retrieval time for a single server. We show that query overhead is independent of the size of the rows retrieved.

Setup. The database is on one Amazon EC2 as described in Section 8.1. In each experiment the database has one table of 100K rows, with nine either 15, 150, or 255 character columns. Each column has a different number of distinct keys, and as such a different number of rows returned when querying on that column. Every column has an index and each table fits in memory.

Workload. Throughput is measured by running as many client threads as necessary to saturate the database server (in these experiments 16), each generating and issuing queries of the form:

```
SELECT * FROM table1 WHERE c5 = ?
```

We vary the number of rows retrieved by the queries in a run by changing the restricted column in the query. Within a run each client thread issues a sequence of queries requesting a random existing value from one column (except for the run that measures retrieving zero rows), with a uniform distribution. The overall throughput of a run as measured in queries per millisecond is a sum of each client thread's throughput, measured as a sum of queries issued divided by the number of milliseconds in the run.

Figure 9 shows the time per query measured as $1/qps$ where qps is the throughput in queries per second, as a function of the number of rows returned by the query. This graph shows how total per-query processing time increases as the number of rows retrieved increases for different row sizes. This graph is fit by lines of the form $t_q = t_o + n_r * t_r$, where t_q is the total time of the query, t_o is query overhead, n_r is the number of rows retrieved per query, and t_r is the time to retrieve one row. On our experimental setup, for 150 byte rows, we measure query overhead as 0.45ms and the time to retrieve one row as 0.011ms. For 1700 byte rows, query overhead was .43ms and the time to retrieve a row was .022, and for 2700 byte rows the numbers were 0.5ms and .033ms. DIXIE only uses one value for $cost_r$, but this experiment shows that $cost_r$ varies for row size. Including metrics in the configuration files on the average size of rows in a table and using this in the cost formula would help DIXIE produce better query plans.

Using the formula in Section 5, DIXIE would estimate the cost of retrieving 20 rows from one server as .88, and retrieving 20 rows from two servers as 1.32, a 50% increase. Retrieving 100 rows from two servers instead of

Rows Returned	Servers	DIXIE's Cost	Time
20	1	.88	.95ms
20	2	1.32	1.37ms
100	1	2.64	2.67ms
100	2	3.08	3.24ms

Table 5: DIXIE plan cost estimation vs. actual time.

one server would be 16.7% higher. Table 5 shows the difference between DIXIE's plan costs and experimentally validated plan costs, in milliseconds. The actual time to request 20 rows from one server is .95ms and from two servers is 1.37ms, a 44% increase, and for 100 rows it is a 21% higher.

Query overhead is the MySQL server allocating resources to parse the query, obtain read table locks, and check indices or table metadata to determine if it has rows which match the query.

Since the per-row cost is roughly one twentieth the per-query overhead, DIXIE uses a $cost_r$ of 0.05 in the formula described in Section 5. So in the `blogs` and `comments` query in Figure 7, DIXIE would choose to execute the two-step join plan as long as selectivity statistics indicated that there were less than 162 blog posts per author. Note that for the purposes of showing how costs change according to column selectivity we are ignoring Plan 2, which DIXIE would also consider. This is 12% off from the measured optimal switchover point, shown in Figure 8, which is 145 blog posts per author.

9 Related Work

DIXIE relies on a large body of research describing how to build parallel databases and query optimizers. This section includes the most closely related systems.

DIXIE operates on horizontally partitioned databases on a shared-nothing architecture [22]. The benefits of this design were demonstrated in systems like Gamma [8], Teradata [26], and Tandem [12]. A number of more recent databases exploit horizontal partitioning. H-Store [15], Microsoft's Cloud SQL Server [3], and Google Megastore [10] are main-memory partitioned databases. These systems either prefer or require applications to execute queries which only touch a single partition, and do not describe how to efficiently execute queries that might require spanning partitions.

C-Store [23] and its successor Vertica [29] store copies of table columns partitioned on different columns. C-Store's query planner and optimizer consider which copies of a column to use in answering a query; its focus is parallelizing single queries that examine large amounts of data. Vertica has a sophisticated query optimizer which works over partitioned data on multiple servers, but its optimizer is also designed for large analytic queries, and chooses to favor colocated joins (what we call pushdown joins) where possible [28]. DIXIE relies on the fact that web application queries are often simple, selective, and use a small number of tables in joins. As shown in this work, there are queries where colocated joins are less efficient for these queries than other plans that DIXIE would choose.

The fractured mirrors work [20] presents the idea of storing tables in different formats on different disks to minimize disk seeks, but the authors do not consider partitions, or speak to the costs involved in distributed query execution and how this affects the choice of query plans.

Other work has made the point that social networks do not partition well, and suggested replication solutions [19, 18]. This work relies on network-style clustering in the data, and aims to put users on the same servers as their friends. DIXIE makes no such assumptions.

Schism [7] chooses good partitioning and replication arrangements with the goal of ensuring that transactions need never involve more than one server. Schism doesn't quantify the cost of query overhead or describe how to do query planning and optimization. [2] also investigates partition choice, focusing on warehouse datasets.

In the field of query optimization many systems have addressed how to execute distributed queries. The query optimizer in Orchestra [25], a peer-to-peer database built on a distributed hash table, estimates a plan's cost by considering the cost at the slowest node or link used in each plan stage, which will ultimately optimize for latency but not throughput.

Distributed INGRES [9] has a distributed query optimizer, but like the other systems mentioned above, it optimizes for reduced latency and parallelism. R* [17] is a distributed query optimizer which seeks to minimize total resources consumed, like DIXIE, but does not support the idea of table partitioning, so table access methods are limited. Most of these systems use replication for fault tolerance; none take advantage of table copies on different range partitions to execute plans that minimize machine accesses.

Kossman noted that when estimating costs of a query, communication costs including fixed per-message costs must be considered [16], and discusses choosing which replica of a table to use when executing a query. This paper extends upon that work by noting that in certain web workloads, this cost is the dominant cost of execution, and also proposing a new way of minimizing it.

Evaluation of Bubba [5] showed that when the system is CPU-bottlenecked, declustering degrades performance due to startup and communication costs, which are part of query overhead. DIXIE applies a similar idea to web application workloads, but goes beyond this to motivate keeping many copies of the data, and to use query overhead in the query optimizer to determine cost.

10 Conclusion

Due to the high cost of issuing unnecessary queries in a clustered database, and since web applications often have workloads which do not cleanly partition, developers should use multiple copies of tables partitioned on different columns. DIXIE is a query planner, optimizer, and executor for such a database. DIXIE can execute application SQL queries written for a single database against a partitioned database with multiple partitionings of tables without any additional code by the application developer. DIXIE chooses plans which have high throughput by using per-query server overhead as a dominant factor in calculating query costs.

Acknowledgements

We thank Ramesh Chandra, Sam Madden, and the anonymous reviewers for providing feedback that helped improve this paper. This work was supported by Quanta Computer and by the National Science Foundation.

References

[1] Wikipedia database dump. `http://dumps.wikimedia.org/`, 2008.

[2] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004.

[3] P. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. Lomet, R. Manne, L. Novik, and T. Talius. Adapting microsoft sql server for cloud computing. In *ICDE*, pages 1255–1263. IEEE, 2011.

[4] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *SIGMOD*, pages 128–136. ACM, 1982.

[5] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *SIGMOD*. ACM, 1988.

[6] C. Curino, E. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[7] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *VLDB*, 2010.

[8] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[9] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, page 180. ACM, 1978.

[10] J. Furman, J. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A scalable data system for user facing applications. *SIGMOD*, 2008.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[12] T. D. Group. NonStop SQL, a distributed, high-performance, high-reliability implementation of SQL. *Workshop on High Performance Transaction Systems*, 1987.

[13] HSQLDB. HSQLDB. `http://hsqldb.org/`, October 2011.

[14] JSQLParser. JSQLParser. `http://jsqlparser.sourceforge.net/`, October 2011.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB*, volume 1.

[16] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4), 2000.

[17] L. Mackert and G. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, 1986.

[18] J. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine (s) that could: Scaling online social networks. *ACM SIGCOMM Computer Communication Review*, 40(4):375–386, 2010.

[19] J. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. Scaling online social networks without pains. In *NETDB*, 2009.

[20] R. Ramamurthy, D. DeWitt, and Q. Su. A case for fractured mirrors. *VLDB*, 12(2), 2003.

[21] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[22] M. Stonebraker. The case for shared nothing. *Database Engineering Bulletin*.

[23] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *VLDB*.

[24] J. Tauber. Pinax. `http://www.pinaxproject.com/`, October 2011.

[25] N. Taylor and Z. Ives. Reliable storage and querying for collaborative data sharing systems. In *ICDE*, march 2010.

[26] Teradata. 1012 Data Base Computer, Concepts and Facilities. *Teradata Document C02-0001-05, Teradata Corporation, Los Angles, CA*, 1988.

[27] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11), 2009.

[28] Vertica. `http://www.vertica.com/2010/04/28/vertica-under-the-hood-the-query-optimizer`, April 2010.

[29] Vertica. Vertica. `http://vertica.com`, October 2011.

Gibraltar: Exposing Hardware Devices to Web Pages Using AJAX

Kaisen Lin
UC San Diego

David Chu James Mickens
Li Zhuang Feng Zhao
Microsoft Research

Jian Qiu
National University of Singapore

Abstract

Gibraltar is a new framework for exposing hardware devices to web pages. Gibraltar's fundamental insight is that JavaScript's AJAX facility can be used as a hardware access protocol. Instead of relying on the browser to mediate device interactions, Gibraltar sandboxes the browser and uses a small device server to handle hardware requests. The server uses native code to interact with devices, and it exports a standard web server interface on the localhost. To access hardware, web pages send device commands to the server using HTTP requests; the server returns hardware data via HTTP responses.

Using a client-side JavaScript library, we build a simple yet powerful device API atop this HTTP transfer protocol. The API is particularly useful to developers of mobile web pages, since mobile platforms like cell phones have an increasingly wide array of sensors that, prior to Gibraltar, were only accessible via native code plugins or the limited, inconsistent APIs provided by HTML5. Our implementation of Gibraltar on Android shows that Gibraltar provides stronger security guarantees than HTML5; furthermore, it shows that HTTP is responsive enough to support interactive web pages that perform frequent hardware accesses. Gibraltar also supports an HTML5 compatibility layer that implements the HTML5 interface but provides Gibraltar's stronger security.

1. Introduction

Web browsers provide an increasingly rich execution platform. Unfortunately, browsers have been slow to expose low-level hardware devices to JavaScript [8], the most popular client-side scripting language. This limitation has become particularly acute as sensor-rich devices like phones and tablets have exploded in popularity. A huge marketplace has arisen for mobile applications that leverage data from accelerometers, microphones, GPS units, and other sensors. Phones also have increasingly powerful computational and storage devices. For example, graphics processors (GPUs) are already prevalent on phones, and using removable storage devices like SD cards, modern phones can access up to 64GB of persistent data.

Because JavaScript has traditionally lacked access to such hardware, web developers who wanted to write device-aware applications were faced with two unpleasant choices: learn a new plugin technology like Flash which is not supported by all browsers, or learn a platform's native application language (e.g, the Win32 API for Windows machines, or Java for Android). Both choices limit the portability of the resulting applications. Furthermore, moving to native code eliminates a key benefit of the web delivery model—applications need not be installed, but merely navigated to.

1.1 A Partial Solution

To remedy these problems, the new HTML5 specification [10] introduces several ways for JavaScript to access hardware. At a high-level, the interfaces expose devices as special objects embedded in the JavaScript runtime. For example, the `<input>` tag [24] can reflect a web cam object into a page's JavaScript namespace; the page reads or writes hardware data by manipulating the properties of the object. Similarly, HTML5 exposes geolocation data through the `navigator.geolocation` object [27]. Browsers implement the object by accessing GPS devices, or network cards that triangulate signals from wireless access points.

Given all of this, there are two distinct models for creating device-aware web pages:

- Applications can be written using native code or plugins, and gain the performance that results from running close to the bare metal. However, users must explicitly install the applications, and the applications can only run on platforms that support their native execution environment.
- Alternatively, applications can be written using cross-platform HTML5 and JavaScript. Such applications do not require explicit installation, since users just navigate to the application's URL using their browser. However, as shown in the example above, HTML5 uses an inconsistent set of APIs to name and query each device, making it difficult to write generic code. Furthermore, by exposing devices through extensions of the JavaScript interpreter, the entire JavaScript runtime becomes a threat surface for a malicious web page trying to access unauthorized hardware—once a web page has compromised the browser, nothing stands between it and the user's devices. Unfortunately, modern browsers are large, complex, and have many exploitable vulnerabilities [4, 30, 34]. On mobile devices, browsers represent a key infection vector for malicious pages that steal SMS information [21], SD card data [23], and other private user information.

Ideally, we want the best of both worlds—device-aware, cross-platform web pages that require no installation, but whose security does not depend on a huge trusted computing base like a browser.

1.2 Our Solution: Gibraltar

Our new system, called Gibraltar, uses HTTP as a hardware access protocol. Web pages access devices by issuing AJAX requests to a *device server*, a simple native code application which runs in a separate process on the local machine and exports a web server interface on the localhost domain. If a hardware request is authorized, the device server performs the specified operation and returns any data using a standard HTTP response. Users authorize individual web domains to access each hardware device, and the device server authenticates each AJAX request by ensuring that the referrer field [7] represents an authorized domain.

Unlike HTML5, Gibraltar does not require the browser to be fully trusted. Indeed, in Gibraltar, the browser is sandboxed and incapable of accessing most devices. However, a corrupted or malicious browser can send AJAX requests to the device server which contain snooped referrer fields from authorized user requests. To limit these attacks, Gibraltar uses *capability tokens* and *sensor widgets* [12]. Before a web page can access hardware, it must fetch a token from the device server. The page must tag subsequent hardware requests with the fresh capability.

To prevent a malicious browser from surreptitiously requesting capabilities from the device server, Gibraltar employs sensor widgets. Sensor widgets are ambient GUI elements like system tray icons that indicate which hardware devices are currently in use, and which web pages are using them. Sensor widgets help a user to detect discrepancies between the set of devices that she expects to be in use, and the set of devices that are actually in use. Thus, sensor widgets allow a user to detect when a compromised browser is issuing hardware requests that the user did not initiate.

Using these mechanisms, a compromised browser in Gibraltar has limited abilities to independently access hardware (§5). However, a malicious browser is still the conduit for HTTP traffic, so it can snoop on data that the user has legitimately fetched and send that data to remote hosts. Gibraltar does not stop these kinds of attacks. However, Gibraltar is complementary to information flow systems like TightLip [39] that can prevent such leaks.

1.3 Advantages of Gibraltar

Gibraltar's device protocol has four primary advantages:

- **Ease of Deployment:** Gibraltar allows device-aware programs to be shipped as web applications that do not need to be installed. The device server does need to be installed, but it can ship alongside the browser and be installed at the same time that the browser itself is installed.
- **Security:** Compared to HTML5-style approaches which expose hardware by extending the JavaScript interpreter, Gibraltar has a much smaller attack surface. Gibraltar's HTTP protocol is a narrow waist for hardware accesses, and the device server is much simpler than a full-blown web browser; for example, our device server for Android phones is only 7613 lines of strongly typed Java code, instead of the million-plus lines of C++ code found in popular web browsers. Using capability tokens and sensor widgets, Gibraltar can also prevent (or at least detect) many attacks from malicious web pages and browsers. HTML5 cannot stop or detect any of these attacks.
- **Usability:** An HTTP device protocol provides a uniform naming scheme for disparate devices, and makes it easy for pages to access non-local devices. For example, a page running on a user's desktop machine may want to interact with sensors on the user's mobile phone. If a Gibraltar device server runs on the phone, the page can access the remote hardware using the same interface that it uses for local hardware—the only difference is that the device server is no longer in the localhost domain.
- **Backwards Compatibility:** It is straightforward to map HTML5 device commands to Gibraltar calls. Thus, to run a preexisting HTML5 application atop Gibraltar, a developer can simply include a translation library that converts HTML5 calls to Gibraltar calls but preserves Gibraltar's security advantages. The library can use Mugshot-style interpositioning [19] to intercept the HTML5 calls.

Since Gibraltar uses HTTP to transport hardware data, a key question is whether this channel has sufficient bandwidth and responsiveness to support real device-driven applications. To answer this question, we wrote a device server for Android mobile phones, and modified four non-trivial applications to use the Gibraltar API. Our evaluation shows that Gibraltar is fast enough to support real-time programs like games that require efficient access to hardware data.

2. Design

Gibraltar uses privilege separation [29] to provide a web page with hardware access. The web page, and the enclosing browser which executes the page's code, are both untrusted. Gibraltar places the browser in a sandbox which prevents direct access to Gibraltar-mediated devices. The small, native code device server resides in a separate process from the browser, and executes hardware requests on behalf of the page, exchanging data with the page via HTTP.

As shown in Figure 1, a Gibraltar-enabled page includes a JavaScript file called `hardware.js`. This library implements the public Gibraltar API. `hardware.js` fetches authentication tokens as described in Section 2.1, and translates page-initiated hardware requests into AJAX fetches as described in Section 3. `hardware.js` also receives and deserializes the responses. Note that `hardware.js` is merely a convenience library that makes it easier to program against Gibraltar's raw AJAX protocol; Gibraltar does not trust `hardware.js`, and it does not rely on `hardware.js` to enforce the security properties described in Section 5.

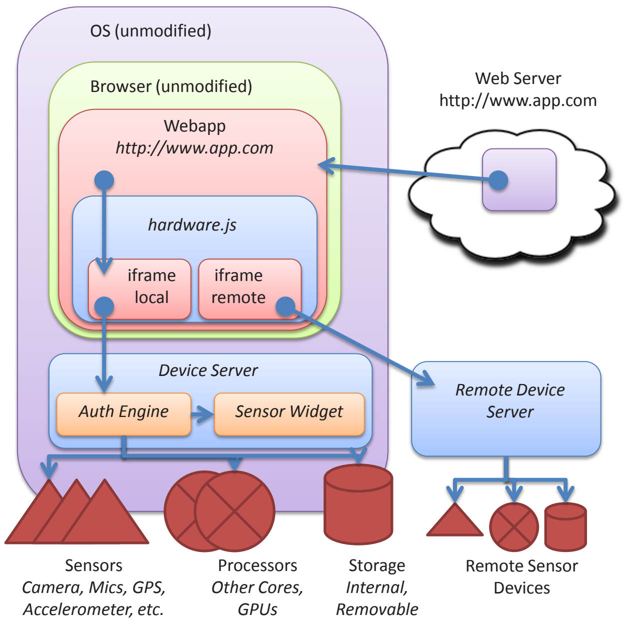


Figure 1. Gibraltar Architecture.

```
void handle_request(req){
   resp = new AJAXResponse();
   switch(req.type){
     case OPEN_SESSION:
        if(!active_tokens.contains(req.referrer)){
           resp.result = "TOKEN:" + makeNewToken();
           active_tokens[req.referrer] = resp.result;
        }
        break;

     case DEVICE_CMD:
        if(!authorized_domains[req.device].contains(
                                       req.referrer) ||
           (active_tokens[req.referrer] == null) ||
           (active_tokens[req.referrer] != req.token)){
           resp.result = "ACCESS DENIED";
        }else{
           resp.result = access_hardware(req.device,
                                         req.cmd);
           sensor_widgets.alert(req.referrer,
                                req.device);
        }
        break;

     case CLOSE_SESSION:
        if(active_tokens[req.referrer] == req.token)
           active_tokens.delete(req.referrer);
        break;
   }
   sendResponse(resp);
}
```

Figure 2. Pseudocode for device server.

Note that the device server resides in the localhost domain, whereas the Gibraltar-enabled page emanates from a different, external origin. By default, the same-origin policy would prevent the `hardware.js` in the web page from fetching cross-origin data from the localhost server. However, using the `Access-Control-Allow-Origin` HTTP header [37], the device server can instruct the browser to allow the cross-origin Gibraltar fetches. This header is supported by modern browsers like IE9 and Firefox 4+. In older browsers, `hardware.js` communicates with the device server using an invisible frame with a localhost origin; this frame exchanges Gibraltar data with the regular application frame using `postMessage()`. Similarly, Gibraltar can use a remote-origin frame to deal with off-platform devices.

2.1 Authenticating Hardware Requests

In Gibraltar, device management consists of three tasks: manifest authorization, session establishment, and session teardown. Figure 2 provides the relevant pseudocode in the device server. We discuss this code in more detail below.

Manifest authorization: On mobile devices like Android, users authorize individual applications to access specific hardware devices. Similarly, in Gibraltar, users authorize individual web domains like `cnn.com` to access individual hardware devices. When a page contacts the device server for the first time, the page includes a *device manifest* in its HTTP request. The manifest is simply a list of devices that the page wishes to access. The device server presents this manifest to the user and asks whether she wishes to grant the specified access permissions to the page's domain. If so, the device server stores these permissions in a database. Subsequent page requests for devices in the manifest will not require explicit user action, but if the page requests access to a new device, the user must approve the new permission.

Session management: Since Gibraltar hardware requests are expressed as HTTP fetches, a natural way for the device server to authenticate a request is to inspect its referrer field [7]. This is a standard HTTP field which indicates the URL (and thus the domain) of the page which generated the request. Unfortunately, a misbehaving browser can subvert this authentication scheme by examining which domains successfully receive hardware data, and then generating fake requests containing these snooped referrer fields. This is essentially a replay attack on a weak authenticator.

To prevent these replay attacks, the device server grants a *capability token* to each authorized web domain. Before a page in domain `trusted.com` can access hardware, it must send a session establishment message to the device server. The device server examines the referrer of the HTTP message and checks whether the domain has already been granted a token. If not,[1] the server generates a unique token, stores the mapping between the domain and that token, and sends the token to the page. Later, when the page sends an actual hardware request, it includes the capability token in its AJAX message. If the token does not match the mapping found in the device server's table, the server ignores the hardware request.

[1] We restrict each domain to a single token for security reasons that we describe in Section 5.1. However, this restriction does not prevent a domain from opening multiple device-aware web pages on a client—the pages can inform each other of the domain's token using the JavaScript `postMessage()` API.

A page sends a session teardown message to the device server when it no longer needs to access hardware, e.g., because the user wants to navigate to a different page. Upon receipt of the teardown message, the server deletes the relevant domain/token mapping. `hardware.js` can detect when a page is about to unload by registering a handler for the JavaScript `unload` event.

Sensor widgets: Given this capability scheme, a misbehaving browser that can only spoof referrers cannot fraudulently access hardware—the browser must also steal another domain's token, or retrieve a new one from the device server. As we discuss in Section 5, cross-domain token stealing is difficult if the browser uses memory isolation to partition domains. However, nothing prevents a browser from autonomously downloading a new security token in the background under the guise of an authorized domain, and then using this token in its AJAX requests. To prevent this attack, we use sensor widgets [12], which are ambient GUI elements like system tray icons that glow, make a noise, or otherwise indicate when a particular hardware device is in use. Sensor widgets also indicate the domains which are currently accessing hardware. Thus, if the browser tries to autonomously access hardware using a valid token, the activity will trigger the sensor widgets, alerting the user to a hardware request that she did not initiate.

The sensor widgets are implemented within the device server, not the browser. However, the browser can try to elude the widgets in several ways. In Section 5, we provide a fuller analysis of Gibraltar's security properties.

2.2 The Gibraltar API

Figure 3 lists the client-side Gibraltar API. Before a web page can issue hardware commands, it must get a new capability token via `createSession()`. Then, it must send its device manifest to the device server via `requestAccess()`. The device server presents the manifest to the user and asks her to validate the requested hardware permissions.

2.2.1 Sensor API

To provide access to sensors like cameras, accelerometers, and GPS units, Gibraltar provides a one-shot query interface and a continuous query interface. In keeping with JavaScript's event-driven programming model, `singleQuery()` and `continuousQuery()` accept an application-defined callback which Gibraltar invokes when the hardware data has arrived. The functions also accept the name of the device to query, and a device-specific `params` value which controls sensor-specific properties like the audio sampling bitrate. `continuousQuery()` takes an additional parameter representing the query frequency.

Different devices will define different formats for the `params` object, and different formats for the returned device data. However, much like USB devices, Gibraltar devices fall into a small set of well-defined classes such as storage devices, audio devices, and video devices. Thus, web pages can program against generic Gibraltar interfaces to each class; the device server and `hardware.js` can encapsulate any device-specific eccentricities.

Figure 3 also describes a sensor management interface. The power controls allow a page to shut off devices that it does not need; the device server ensures that a device is left on if at least one application still needs it. `sensorAdded()` and `sensorRemoved()` let applications register callbacks which Gibraltar fires when devices arrive or leave. These events are useful for off-platform devices like Bluetooth headsets and Nike+ shoe sensors [22].

2.2.2 Processor API

Multi-core processors and programmable GPUs are already available on desktops, and they are starting to ship on mobile devices. To let web pages access these extra cores, Gibraltar exports a simple multi-processor computing model inspired by OpenCL [13], a new specification for programming heterogeneous processors.

A Gibraltar *kernel* represents a computational task to run on a core. Kernels are restricted to executing two types of predefined functions. *Primitive functions* are geometric, trigonometric, or comparator operations. Gibraltar's primitive functions are similar to those of OpenCL. *Built-in functions* are higher-level functions that we have identified as particularly useful for processing hardware data. Examples of such functions are FFT transforms and matrix operations.

A web page passes a kernel to Gibraltar by calling `enqueueKernel()`. To execute a parallel vector computation with that kernel, the page calls `setKernelData()` with a vector of arguments; Gibraltar will instantiate a new copy of the kernel for each argument and run the kernels in parallel. A web page can also create a computation pipeline by calling `enqueueKernel()` multiple times with the same or different kernel. Gibraltar will chain the kernels' inputs and outputs in the order that the kernels were passed to `enqueueKernel()`. The page sets the input data for the pipeline by passing a scalar value to `setKernelData()`.

Once an application has configured its kernels, it calls `executeKernels()` to start the computation. Gibraltar distributes the kernels to the various cores in the system, coordinates cross-kernel communication, and fires an application-provided callback when the computation finishes.

2.2.3 Storage API

The final set of calls in Figure 3 provide a key/value storage interface. The namespace is partitioned by web domain and by storage device; a web domain can only access data that resides in its partitions. To support removable storage devices, Gibraltar fires connection and disconnection events like it does for off-platform sensors like Bluetooth headsets.

HTML5 DOM storage [11] also provides a key-value store. However, DOM storage is limited to non-removable

Call	Description
`createSession()`	Get a capability token from the device server.
`destroySession()`	Relinquish a capability token.
`requestAccess(manifest)`	Ask for permission to access certain devices.
`singleQuery(name, params)`	Get a single sensor sample.
`continuousQuery(name, params, period)`	Start periodic fetch of sensor samples.
`startSensor(name)`	Turn on a sensor.
`stopSensor(name)`	Turn off a sensor.
`sensorAdded(name)`	Upcall fired when a sensor is added.
`sensorRemoved(name)`	Upcall fired when a sensor is removed.
`getSensorList()`	Get available sensors.
`enqueueKernel(kernel)`	Queue a computation kernel for execution.
`setKernelData(parameters)`	Set the input data for the computation pipeline.
`executeKernels()`	Run the queued kernels on the input data.
`put(storename,key,value)`	Put value by key.
`get(storename,key)`	Get value by key.

Figure 3. Summary of `hardware.js` API. All calls implicitly require a security token and callback function.

media, and it does not explicitly expose the individual devices which are used for the underlying stable storage.

2.3 Remote device access

As we mentioned earlier, some devices may reside off-platform. If those devices run a Gibraltar server which accepts external connections, a web page can seamlessly access those devices using the same interface that it uses for local ones. This capability enables many interesting applications. For example, in Section 6, we evaluate a game that runs on a desktop machine but uses a mobile phone with an accelerometer as a motion-sensitive controller. In this example, the web page runs on the desktop machine, but the device server runs on the phone.

A device server accepts connections from localhost clients by default (subject to the authentication rules described in Section 2.1). For security reasons, a device server should reject connections from arbitrary remote clients. Thus, users must explicitly whitelist each external IP address or dynamic DNS name [35] that wishes to communicate with a device server. This is accomplished in a fashion similar to how the user authorizes device manifests (§2.1).

2.4 Sandboxing the Browser

Gibraltar is agnostic about the mechanism that prevents the browser from accessing Gibraltar devices. For example, mobile platforms like Android, iOS, and the Windows Phone provide device ACLs that makes it easy to prohibit applications from accessing forbidden hardware. Gibraltar is also compatible with other isolation techniques like hardware virtualization or binary rewriting.

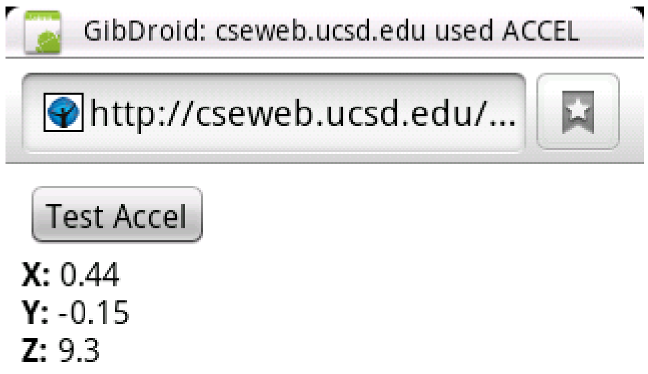
3. Implementation

Client-side Library: `hardware.js` encodes device requests using a simple XML string. Each request contains a security token, an action to perform, the target device, and optional device-specific parameters. For example, a request to record microphone data includes a parameter that represents the recording duration. Device responses are also encoded using XML. The response specifies whether the request succeeded, and any data associated with the operation. The device server encodes binary data in `Base64` format so that `hardware.js` can represent data as JavaScript strings.

Android Device Server: On Android 2.2, we implemented the device manager as a servlet for the i-jetty web server [2]. A servlet is a Java software module that a web server invokes to handle certain URL requests. The Gibraltar servlet handles all requests for Gibraltar device URLs. The servlet performs the authentication checks described in Section 2.1, accesses hardware using native code, and returns the serialized results. We refer to our Android implementation of Gibraltar as GibDroid.

The GibDroid device server has different probing policies for low throughput sensors and high throughput sensors. For low throughput devices like cameras, GibDroid accesses the sensor on demand. For devices like accelerometers that have a high data rate, the GibDroid server continuously pulls data into a circular buffer. When a page queries the sensor, the device server returns the entire buffer, allowing multiple data points to be fetched in a single HTTP round trip. Currently, GibDroid provides access to accelerometers, GPS units, cameras (both single pictures and streaming video), microphones, local storage, and native computation kernels.

Before a web page can receive hardware data from the device server, it must engage in a TCP handshake with the server and send an HTTP header. For devices with high data rates like accelerometers and video cameras, creating an HTTP session for each data request can hurt performance, even with sample batching. Thus, GibDroid allows the de-


GibDroid: cseweb.ucsd.edu used ACCEL

http://cseweb.ucsd.edu/...

Test Accel

X: 0.44
Y: -0.15
Z: 9.3

Figure 4. GibDroid uses the Android notification bar to hold sensor widgets.

vice server to use Comet-style [3] data pushes. In this approach, `hardware.js` establishes a persistent HTTP connection with the device server using a "forever frame." Unlike a standard frame, whose HTML size is declared in the HTTP response by the server, a forever frame has an indefinite size, and the server loads it incrementally, immediately pushing a new HTML chunk whenever new device data arrives. Each chunk is a dynamically generated piece of JavaScript code; the code contains a string variable representing the new hardware data, and a function call which invokes an application-defined handler. Forever frames are widely supported by desktop browsers, but currently unsupported by many mobile browsers. Thus, the device server reverts to request-response for mobile browsers.

GibDroid can stream accelerometer data and video frames using Comet data pushes. To handle video, our current implementation uses data URIs [18] to write `Base64`-encoded data to an `<image>` tag[2]. Many current browsers limit data URIs to 32 KB data; thus, data URIs are only appropriate for sending previews of larger video images. Our current GibDroid implementation displays video frames with a pixel resolution of 530 by 380. The device server uses Android's `setPreviewCallback()` call to access preview-quality images from the underlying video camera.

GibDroid supports the execution of kernel functions. However, our evaluation Android phone does not have secondary processing cores. Therefore, GibDroid kernels run in separate Java threads that time-slice the single processor with other applications.

As shown in Figure 4, GibDroid places sensor widgets in the standard notification bar that exists in all Android phones. The notification bar is a convenient place to put the widgets because users are already accustomed to periodically scanning this area for program updates. We are still experimenting with the visual presentation for the widgets, so Figure 4 represents a work-in-progress.

[2] In the next version of Gibraltar, `hardware.js` will write video frames to the bitmap of an HTML5 `Canvas` object [10].

Windows PC Device Server: We also wrote a device server for Windows PCs. This device server, written in C#, currently only provides access to the hard disk, but it is the target of active development. In Section 6.1, we use this device server to compare Gibraltar's performance on a multi-core machine to that of HTML5.

4. Applications

In this section, we describe four new applications which use the Gibraltar API to access hardware. We evaluate the performance of these applications in Section 6.

Our first application is a mapping tool similar to MapQuest [17]. This web page uses GPS data to determine the user's current location. It also uses Gibraltar's storage APIs to load cached maps tiles. More specifically, we assume that the phone's operating system prefetches map tiles, similar to how the Songo framework prefetches mobile ads [14]. The operating system stores the map tiles in the file system; for each cached tile, the OS adds the key/value pair `(tileId,fileSystemLocation)` to the mapping application's Gibraltar storage area. When the map application loads, it determines the user's current location and calculates the set of tiles to fetch. For each tile, it consults the tile index in Gibraltar storage to determine if the tile resides locally. If it does, the page loads the tile using an `<img>` tag with a `file://` origin; otherwise, the page uses a `http://` origin to fetch the image from the remote tile server.

The popular native phone application *Shazam!* identifies songs that are playing in the user's current environment. *Shazam!* does this by capturing microphone data and applying audio fingerprint algorithms. Inspired by *Shazam!*, we built Gibraltar Sound, a web application that captures a short sound clip and classifies it as *music*, *conversation*, *typing*, or *other ambient sound*. To classify sounds, we used Mel-frequency cepstrums (MFCC) for feature extraction, and Gaussian Mixture Models (GMM) for inference [16]. We implemented MFCC and GMM as native built-in kernels.

Our final applications leverage Gibraltar's ability to access off-platform devices. These pages load on a desktop machine's browser, but use Gibraltar to turn a mobile phone into a game controller. The first application, Gibraltar Paint, is a simple painting program in which user gestures with the phone are converted into brush strokes on a virtual canvas. Gestures are detected using the phone's accelerometer.

We also modified a JavaScript version of Pacman [15] to use a Gibraltar-enabled phone as a controller for a game loaded on the desktop browser—tilting the phone in a direction will cause Pacman to move in that direction. HTML5 cannot support the latter two applications because it lacks an API for remote device access.

5. Security

Any mechanism for providing hardware data to web pages must grapple with two questions. First, can it ensure that each device request was initiated by the user instead of a misbehaving browser? Second, once the hardware data has been delivered to browser, can the system prevent the browser from modifying or leaking that data in unauthorized ways? Gibraltar only addresses the first question, but it is complementary to systems that address the second. In Section 5.1, we describe the situations in which Gibraltar can and cannot prevent fraudulent hardware access. In Section 5.2, we describe how Gibraltar can be integrated with a taint tracking system to minimize unintended data leakage.

5.1 Authenticating Hardware Requests

In Gibraltar, there are five kinds of security principals: the user, the Gibraltar device server, the underlying operating system, web pages, and the web browser. Gibraltar does not trust the last two principals. More specifically, Gibraltar's security goal is to prevent unauthorized web pages from accessing hardware data, and faulty web browsers from autonomously fetching such data. Gibraltar assumes that the OS properly sandboxes the browser, and that the OS prevents the browser from directly accessing Gibraltar-mediated hardware; Gibraltar is agnostic to the particular sandboxing mechanism that is used, e.g., binary rewriting, virtual machines, or OS-enforced device ACLs provided by platforms like Android and iOS. Gibraltar assumes that the device server is implemented correctly, that the user can inform the device server of authorized web sites without interference, and that the operating system prevents the web browser from directly tampering with the device server. Thus, the only way that a faulty web page or browser can access hardware is by subverting the AJAX device protocol.

As shown in Figure 2, the device server will only respond to a hardware request if the request has an authorized referrer field and a valid authentication token; furthermore, the authorized domain cannot have another open session involving a different token. Thus, Gibraltar's security with respect to device D can be evaluated in the context of three parameters: whether the attacker can fake referrer fields, whether the attacker can steal tokens from domains authorized to access D, and whether the user currently has a legitimate, active frame belonging to a legitimately authorized domain. Figure 5(a) provides concrete threat examples that correspond to whether an attacker can fake referrers or steal tokens.

Figure 5(b) shows Gibraltar's attack resilience when the user does not have an authorized frame open. Figure 5(c) shows the attacker's power when the user has opened an authorized frame. In both cases, we see that an attacker cannot fraudulently access hardware if he cannot fake referrer fields. If the attacker can fake referrer fields, then his ability to fraudulently access device D depends on whether the user has already opened a frame for a domain that is authorized to access D. If no such frame is open, the attacker can successfully fetch an authentication token from the device server, since the domain will not have an outstanding token in circulation. The attacker's first hardware request will pass the device server's authentication tests, since the referrer will be authorized and the token will be valid. However, the device server will trigger the appropriate sensor widget for D, indicating the (spoofed) trusted domain that is accessing that device. At this point, the user can realize that she has not legitimately opened a frame in that domain, and she can shut down her browser or take other remediating steps. Although the browser has gained limited access to hardware data, Gibraltar can work in concert with a taint tracking system to prevent the data from being externalized (§5.2).

Now suppose that the attacker can fake referrer fields, and the user does have an authorized frame open (this is the right column of Figure 5(c)). If the browser uses a Gazelle-style architecture [36] and strongly isolates the attacker page from the authorized page, the attacker cannot inspect the token in the authorized page. Thus, the attacker must request a new token from the device server. However, the server will refuse this request because the domain in the referrer field will already have a token.

If the attacker can steal tokens *and* fake referrers, and the user already has an authorized frame open, then nothing prevents the attacker from opportunistically hiding his hardware requests within the background traffic from the legitimately authorized frame. Although current browsers do provide a modicum of domain isolation (e.g., via IE's process-per-tab model, or Chrome's process-per-site-instance model [31]), commercial browsers do not implement Gazelle-strength isolation. However, browsers are continually moving towards stronger isolation models, so we believe that soon, cross-frame token stealing will be impossible.

Gibraltar assumes that the operating system correctly routes packets to the device server. Thus, the device server can reject arbitrary connections from off-platform entities by verifying that the source in each AJAX request has a localhost IP address. If a user wants to associate a device server with a web page that resides off-platform, she must whitelist the external IP address, or notify the device server and the web page of a shared secret which enables the device server to detect trusted external clients. For example, the client web page might generate a random number and include this number in the first AJAX request that it sends to the device server. When the server receives this request, it can present the nonce to the user for verification.

Malicious local applications that are not web pages can also try to access hardware by contacting the device server. Sensor widgets provide some defense, but using tools like Linux's `lsof` or Windows' `Process Explorer`, the device server can simply reject localhost connections from programs that are not hosted within a web browser.

5.2 Securing Returned Device Data

The browser acts as the conduit for all AJAX exchanges, and it can arbitrarily inspect the JavaScript runtimes inside each page. Thus, once the browser has received hardware data (either because a user legitimately fetched it, or because the

	Cannot fake referrer	Can fake referrer
Cannot steal token	An unauthorized web page seeks hardware access while running in an uncompromised browser.	A malicious page subverts its container in a browser with per-frame memory isolation, but the page cannot peek into the address spaces of other frames in different processes.
Can steal token	The browser is uncompromised, but an authorized page in one tab willingly shares its token with an attacker page in another tab.	A malicious page subverts a monolithic, single-process browser, allowing it to steal a token from another authorized tab that is running.

(a) Example attacker scenarios.

	Cannot fake referrer	Can fake referrer
Cannot steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack detected (attacker can spoof referrer field from trusted domain and get a new token, but sensor widgets alert user to the fact that trusted domain X is accessing hardware but the user hasn't opened a page from X).
Can steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack detected (no legitimately authorized page exists from which the attacker can steal a token; thus, attacker is forced to download a new token as above, and is detected by the sensor widgets).

(b) Gibraltar attack resilience (no legitimately authorized page is running).

	Cannot fake referrer	Can fake referrer
Cannot steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack prevented (attacker cannot steal a token, so he must get a new one from the device server, pretending to be from domain X; however, X already has an outstanding token, so device server rejects the new token request).
Can steal token	Attack prevented (attacker cannot create an authorized referrer for its hardware request).	Attack succeeds (attacker can steal token from X's page and put X in its referrer field, effectively masquerading as X).

(c) Gibraltar attack resilience (a legitimately authorized page from domain X is running).

Figure 5. Gibraltar security properties.

browser stole/fetched a token and acquired the data itself), neither Gibraltar nor HTML5 can prevent the browser from arbitrarily inspecting, modifying, or disseminating the data.

Suppose that, through clever engineering, the browser cannot be subverted by malicious web pages. Further suppose that the browser is trusted not to fake referrer fields, steal tokens from authorized domains, or otherwise subvert the Gibraltar access controls. Even in these situations, malicious web pages can still leak hardware data to remote servers. For example, suppose that the user has authorized domain `x.com` to access hardware, but not `y.com`. The same-origin policy ostensibly prevents JavaScript running on `http://x.com/index.html` from sending data to `y.com`'s domain. However, this security policy is easily circumvented in practice. For example, the JavaScript in `x.com`'s page can read the user's GPS data and create an iframe with a URL like `http://y.com/page.index?lat=LAT_DATA long=LON_DATA`. By loading the frame, the browser implicitly sends the GPS data to `y.com`'s web server.

If the browser is trusted, it can prevent such leakage by tracking the information flow between Gibraltar AJAX requests and externalized data objects like iframe URLs. This is similar to what TaintDroid [6] does, although TaintDroid tracks data flow through a Java VM instead of a browser.

If the browser is untrusted, we can place the taint tracking infrastructure outside of the browser, e.g., in the underlying operating system. However, regardless of where the taint tracker resides, it must allow the user to whitelist certain pairs of domains and hardware data. For example, suppose that the user has authorized only `x.com` to access the GPS unit. Whenever the data flow system detects that GPS data is about to hit the network, it must ensure that the endpoint resides in x.com's domain, e.g., by doing a reverse DNS lookup on the endpoint's IP address.

If the taint system performs that check, it can prevent data from directly leaking to unauthorized domains. However, a full security suite requires both a taint tracker *and* Gibraltar, since a taint tracker alone cannot prevent several damaging attacks. For example, if only a taint tracker is present, a misbehaving browser could send hardware data to an authorized domain even if the user is not currently viewing a page in that domain. Such persistent snooping is problematic because it lets the authorized domain build a huge database of contextual information about the user, even though the user only intended for that data to be collected when she was actually browsing a web page from that domain. As shown in Figure 5(b), Gibraltar detects this attack if the user has not opened a page for an authorized domain. This is because the browser cannot surreptitiously stream data without triggering a sensor widget. However, if the user does have an authorized page open, and is running a browser with weak memory isolation, Gibraltar cannot stop the stolen token attacks shown in the bottom-right corner of Figure 5(c). Note that HTML5 cannot stop any of these attacks, since it lacks sensor widgets or a method for assigning device ACLs to web pages.

Note that taint tracking and whitelists cannot prevent all kinds of information leakage. For example, a malicious browser can post sensitive data to a whitelisted site using a format that the site does not treat as sensitive. For example, user data could be encoded as a comment in a web forum. When combined with cross-site request forgery (CSRF), an attacker may be able to download the exfiltrated user data. Gibraltar is compatible with approaches for stopping CSRF attacks (e.g., [25, 33]).

6. Evaluation

In this section, we ask two fundamental questions about Gibraltar's performance. First, is an HTTP channel fast enough to support high frequency sensors and interactive applications? Second, is Gibraltar competitive with HTML5 in terms of performance?

As described in Section 3, we wrote device servers for two platforms. The first server runs on Android 2.2 phones, and we tested it on two handsets: a Nexus One with 512 MB of RAM and a 1 GHz Qualcomm Snapdragon processor, and a Droid X with 512 MB of RAM and a 1 GHz Texas Instruments OMAP processor. We also wrote a device server for Windows PCs. We tested that server on a Windows 7 machine with 4 GB of RAM and an Intel Core2 processor with two 2.66 GHz cores.

6.1 Access Latency

Multi-core machines: We define a device's *access latency* as the amount of time that a client perceives a synchronous device operation to take. Figure 6 shows access latencies

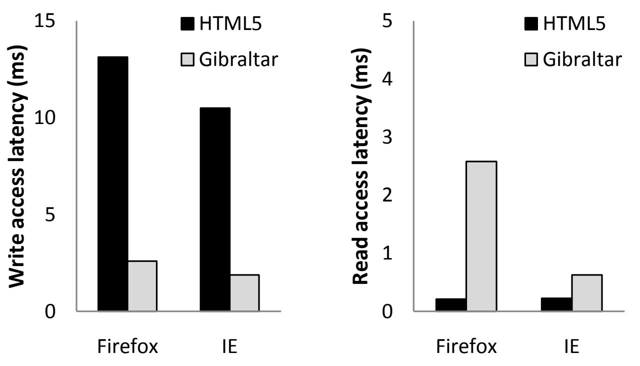


Figure 6. Read and write latencies to the hard disk on the dual-core desktop machine.

for the hard disk on the dual-core desktop machine. Each bar represents the average of 250 trials, with each read or write involving 1 KB of data. HTML5 disk accesses were implemented using the DOM storage API [11], whereas Gibraltar disk accesses were handled by the device server and accessed a partitioned region of the local file system owned by the device server. All reads targeted prior write addresses, meaning that the reads should hit in the block cache inside the device server or the HTML5 browser.

The absolute latencies for Gibraltar's disk accesses are small on both Firefox 3.6 and IE8. For example, a Gibraltar-enabled page on IE8 can read 1 KB of data with a latency of 0.62 ms; on Firefox, the page can perform a similar read with 2.58 ms of latency. While Gibraltar's read performance is worse than that of HTML5, it is more than sufficient to support common use cases for local storage, such as caching user data to avoid fetching it over a slow network.

For disk writes on both browsers, Gibraltar is more than five times faster than HTML5. This is because the Gibraltar device server asynchronously writes back data, whereas Firefox and IE have a write-through policy. Switching Gibraltar to a write-through policy would result in similar performance to HTML5, since the primary overhead would be mechanical disk latencies, not HTTP overhead.

Single-core machines: Our desktop machine had a dual-core processor, meaning that the device server and the web browser rarely had to contend for a core. In particular, once the device server had invoked a `send()` system call to transfer device data to the browser, the OS could usually swap the browser immediately onto one of the two cores. On a single core machine, the browser might have to wait for a non-trivial amount of time, since multiple processes besides the browser are competing for a single core.

Figure 7 depicts access latencies to the Null device on the Droid X phone (the Null device immediately returns an empty message). By using `setsocketopt()` to disable the TCP Nagle algorithm, we prod TCP into sending small packets immediately instead of trying to aggregate several

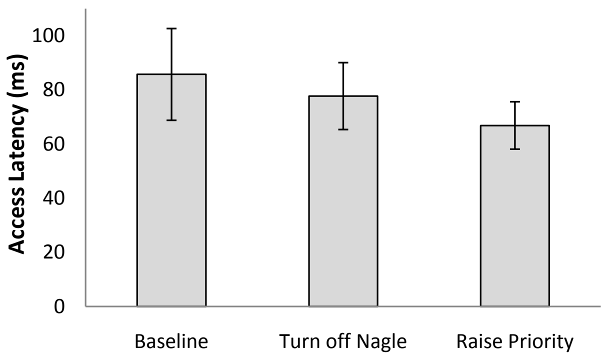


Figure 7. GibDroid Null-device access latencies (single-core phone). Error bars represent one standard deviation.

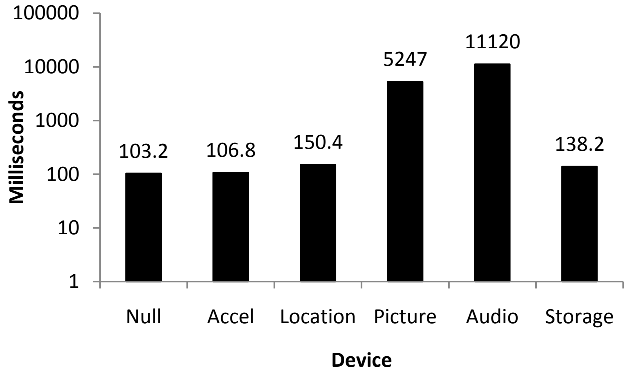


Figure 8. Nexus One access latencies (mobile browser accessing local hardware). Note that the y-axis is log-scale.

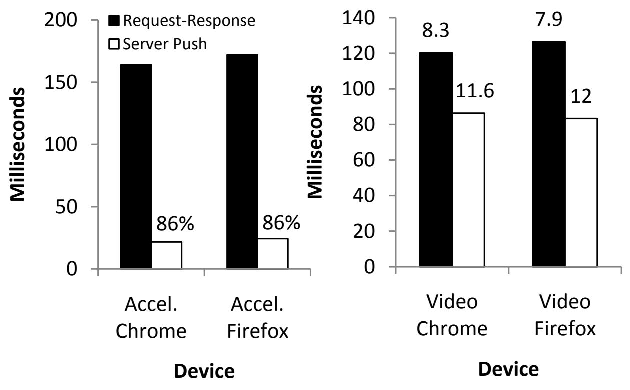


Figure 9. Nexus One access latencies (desktop browser accessing phone hardware). Top-bar numbers for accelerometer represent improvements in sample frequency; top-bar numbers for video represent frame rates.

small packets into one large one. This decreases the average access latency from 87 ms to 78 ms; it also decreases the standard deviation from 34 ms to 25 ms. By raising the priority of the device server thread and the receiving browser thread, we can further decrease the latency to 67 $\pm$ 18 ms. However, the raw performance is still worse than in the dual-core case due to scheduling jitter. For example, looking at single-core results for individual trials, we saw access latencies as low as 29 ms, and as high as 144 ms.

Multi-core processors are already pervasive on desktop systems, and new mobile phones and tablets like the LG Optimus 2X have dual-core processors. Thus, we expect that scheduling jitter will soon become a non-issue for Gibraltar. In the rest of this section, we provide additional evaluation results using the single-core Nexus One phone. We show that even on a single-core machine, Gibraltar is fast enough to support interactive applications.

Accessing Sensors on the Nexus One: Figure 8 depicts the access latency for various devices on the Nexus One phone. The accelerometer and the GPS unit are the sensors that applications query at the fastest rate. Figure 8 shows that the accelerometer can be queried 9.4 times a second, and the GPS unit can be queried 6.6 times a second. As we discuss in Section 6.4, these sampling rates are sufficient to support games and interactive mapping applications.

Accessing the camera or the microphone through Gibraltar is much more expensive than accessing the accelerometer. However, most of the latency arises from the inherently expensive initialization costs for those devices. For example, GibDroid adds 160 ms to the inherent cost of sampling 10 seconds of audio data, and 560 ms to the inherent cost of taking a picture. In both cases, the bulk of Gibraltar's overhead came from the `Base64` encoding that the device server must perform before it can send binary data to the application.

The results in Figure 8 used the request-response version of the Gibraltar protocol. On browsers that support forever frames (§3), Gibraltar can use server-push techniques to decrease client-perceived access latencies to devices. Figure 9 quantifies this improvement for desktop browsers accessing phone hardware over a wireless connection. For example, for video on Firefox, frame access latencies decreased from 126 ms to 83 ms; this improved the streaming rate for live video from 8 frames per second to 12. For the accelerometer, access latencies decreased from 173 ms to 22 ms, allowing the client to fetch accelerometer readings at a rate of 45 Hz. This was close to the native hardware limit of 50 Hz. Note, however, that the performance gains in both cases arose not just from the server-push technique, but from the fact that the device server and the web browser ran on different machines (and thus different processors). This ameliorated some of the scheduling jitter that arises when the device server and the browser run on the same core.

6.2 Sampling Throughput

Low access latencies improve the freshness of the data that the client receives. However, the client may still be unable to receive data at the native sampling frequency. Thus, the device server continuously gathers information from high data

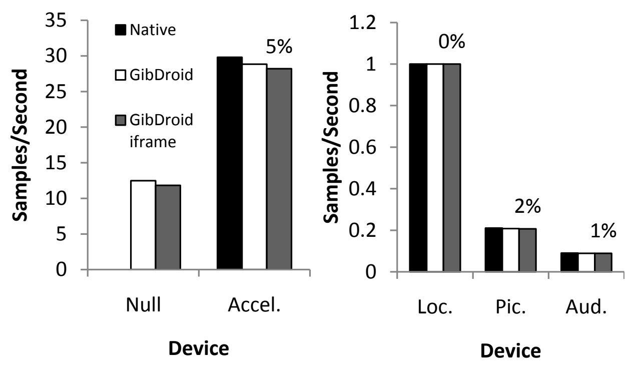


Figure 10. GibDroid sampling throughputs. Top-bar fractions show the percentage slowdown of Gibraltar with an iframe compared to native execution.

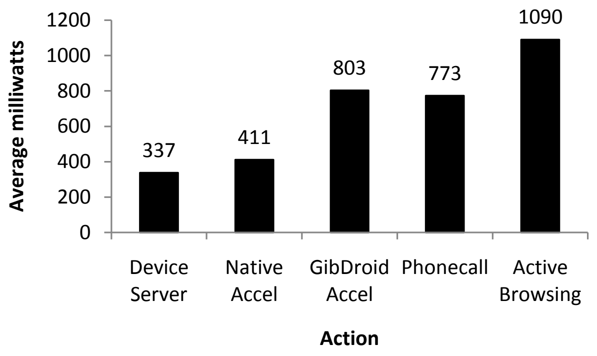


Figure 11. GibDroid power consumption.

rate devices like the accelerometer and the GPS unit. When the server gets a read request for such a device, it returns all of the data that has accumulated since the last query. Thus, an application can analyze the entire data stream even if it cannot access every sample at the native data rate.

Figure 10 depicts GibDroid's sampling throughput using the built-in Android browser to access phone hardware. Each bar represents the maximum number of data samples accessible per second to a native application, a Gibraltar page using an inner iframe (§2), and a Gibraltar page in which the outer iframe directly issues AJAX requests. Throughput degradation was less than 5% for all devices. Figure 10 also shows that cross-frame `postMessage()` overhead was minimal. Note that the accelerometer throughput was greater than the Null device throughput because GibDroid batched multiple accelerometer samples per HTTP response.

6.3 Power

On mobile devices, minimizing power consumption is extremely important. To measure Gibraltar's impact on battery life, we attached a Monsoon Power Monitor [20] to the Nexus One. The Monsoon acted as an energy source while simultaneously measuring how much power it transferred to the phone. We set the phone's screen brightness to the minimum setting and enabled the phone's "airplane mode" when running tests that did not involve radios.

Figure 11 shows power consumption in several different scenarios. The first bar depicts the power consumption for an idle device server. Running an idle server costs 337 mW, and this is essentially the base cost of having the phone turned on. Continuously querying the accelerometer in a native application requires 411 mW. In contrast, using GibDroid to continuously query the device costs 803 mW; however, this cost includes the power spent by the server *and* the browser. By comparison, making a phone call requires 773 mW, and actively browsing the Internet uses over 1W. Thus, we believe that Gibraltar's power usage is similar to that of other mobile applications.

6.4 Applications

For the final part of our evaluation, we examined the performance of the four Gibraltar-enabled applications that we described in Section 4. We evaluated all four applications on the GibDroid platform.

Our map application took an average of 64 ms to load a cached map tile, but 372 ms to fetch one from the Internet. This result is not surprising, since accessing local storage should be faster than pulling data across the wide area.

For our audio classification application, the key performance metric is how long the classification takes. For a 52 KB WAV file representing 10 seconds of data, feature extraction took approximately 6 seconds, and classification of the result took 1.5 seconds. These experiments used a JavaScript implementation of the classification algorithms. For larger audio files, the application could use Gibraltar's native computation kernels to boost performance.

We evaluated Paint and Pacman by running them on a Chrome desktop browser which communicated with GibDroid through a USB cable. Paint was able to sense 9.83 motions per second; this number is an application-level latency that includes the Gibraltar access latency and the overhead of updating the HTML `Canvas` object. Pacman had similar performance. In both cases, the phone was able to control the application with no user-perceived delay. We plan to run further tests over a wireless network which allows the phone to be untethered from the desktop.

7. Related Work

In Section 1, we described the disadvantages of using native code plugins like Flash to provide hardware access to web pages. We also described why HTML5 is a step in the right direction, but not a complete solution.

Like Gibraltar, Maverick [32] provides web pages with hardware access. Maverick lets web developers write USB drivers using JavaScript or NaCl. Maverick sandboxes each untrusted page and USB driver; the components exchange messages through the trusted Maverick kernel. Maverick differs from Gibraltar in three key ways. First, Maverick is limited to the USB interface, whereas Gibraltar's client-side JavaScript library can layer arbitrary hardware protocols atop HTTP. Second, unlike USB, HTTP provides straightforward support for off-platform devices. Third, Maverick does not have mechanisms like sensor widgets that detect misbehaving applications. Thus, Maverick cannot prevent buggy or malicious pages from using the driver infrastructure in ways that the user did not intend. Maverick does have better performance than the current implementation of Gibraltar since Maverick provides IPC via native code NaCl channels instead of via standard HTTP over TCP. However, with kernel support for fast-path localhost-to-localhost TCP connections, and/or NIC support for offloading TCP-related computations to hardware, we believe that Gibraltars performance can approach that of Maverick.

PhoneGap [26] is a framework for building cross-platform, device-aware mobile applications. A PhoneGap application consists of JavaScript, HTML, CSS, and a bundled chrome-less browser whose JavaScript runtime has been extended to export hardware interfaces. Like Gibraltar, PhoneGap allows developers to write device-aware applications using the traditional web stack. Compared to Gibraltar, PhoneGap has three limitations. First, PhoneGap's hardware interface is philosophically equivalent to the HTML5 interface, and thus has similar drawbacks with respect to interface and security. Second, a PhoneGap program is a native application and must be explicitly installed, unlike a Gibraltar web page. Third, PhoneGap applications run within the `file://` protocol, not the `http://` protocol. Thus, unlike Gibraltar web pages, PhoneGap programs are not restricted by the same domain policy. This allows a PhoneGap program to load multiple frames from multiple domains and manipulate their data in ways that would fail in the `http://` context and violate the security assumptions of the remote domains.

In Palm's webOS [1], applications are written in JavaScript, HTML, and CSS. However, these programs are not web applications in the standard sense—they rely on webOS' special runtime, and they will not execute inside actual web browsers. The webOS runtime is a customized version of the popular WebKit browser engine. It exposes HTML5-style device interfaces to applications, and thus suffers from the problems that we discussed in prior sections.

Microkernel browsers like OP [9] and Gazelle [36] restructure the browser into multiple untrusted modules that exchange messages through a small, trusted kernel. Gibraltar's device server is somewhat like a trusted microkernel which mediates hardware access. However, previous microkernel browsers do not change the hardware interface exposed to web pages, since these browsers use off-the-shelf JavaScript runtimes that export the HTML5 interface.

Several projects from the sensor network community expose hardware data using web protocols [5, 28, 38]. However, these systems do not address the security challenge of authenticating hardware requests that emanate from potentially untrustworthy browsers. Gibraltar also exports a richer interface for device querying and management.

8. Conclusions

Gibraltar's key insight is that web pages can access hardware devices by treating them like web servers. Gibraltar sandboxes the browser, shifts authority for device accesses to a small, native code device server, and forces the browser to access hardware via HTTP. Using this privilege separation and sensor widgets, Gibraltar provides better security than HTML5; the resulting API is also easier to program against. Experiments show that the HTTP device protocol is fast enough to support real, interactive applications that make frequent hardware accesses.

References

[1] M. Allen. *Palm webOS: The Insider's Guide to Developing Applications in JavaScript using the Palm Mojo Framework.* O'Reilly Media, Sebastopol, CA, 2009.

[2] J. Bartel and et. al. i-Jetty: Webserver for the Android mobile platform. `http://code.google.com/p/i-jetty`.

[3] D. Crane and P. McCarthy. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0.* Apress, New York, NY, 2008.

[4] M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *Proceedings of USENIX Workshop on Offensive Technologies*, 2008.

[5] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. pREST: A REST-based Protocol for Pervasive Systems. In *Proceedings of IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, October 2004.

[6] W. Enck, P. Gilbert, B. gon Chun, L. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI*, 2010.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[8] D. Flanagan. *JavaScript: The Definitive Guide, Fifth Edition.* O'Reilly Media, Inc., Sebastopol, CA, 2006.

[9] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 402–416, Washington, DC, USA, 2008. IEEE Computer Society.

[10] I. Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML. `http://dev.w3.org/html5/spec/`.

[11] I. Hickson. Web Storage: Editor's Draft, August 20, 2010. `http://dev.w3.org/html5/webstorage`.

[12] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, cameras, and other sensors. In *Proceedings of W2SP*, 2010.

[13] Khronos Group. OpenCL. `http://www.khronos.org/opencl`.

[14] E. Koukoumidis, D. Lymberopoulos, J. Liu, and D. Burger. Improving Mobile Search User Experience with SONGO. Microsoft Research Tech Report MSR-TR-2010-15, February 18, 2010.

[15] N. Landsteiner. How to Write a Pacman Game in JavaScript. `http://www.masswerk.at/JavaPac/`

pacman-howto.html.

[16] H. Lu, W. Pan, N. D. Lane, T. Choudhury, and A. T. Campbell. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09*, pages 165–178, New York, NY, USA, 2009. ACM.

[17] MapQuest. Mapquest maps. http://www.mapquest.com/.

[18] L. Masinter. The "data" URL Scheme. RFC 2397 (Draft Standard), August 1998.

[19] J. Mickens, J. Howell, and J. Elson. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI*, San Jose, CA, April 2010.

[20] Monsoon Soluitions Inc. Monsoon power monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[21] R. Naraine. Pwn2Own 2010: iPhone hacked, SMS database hijacked. ZDNet. http://www.zdnet.com/blog/security/pwn2own-2010-iphone-hacked-sms-database-hijacked/5836, March 2010.

[22] Nike. Nike+. http://nikerunning.nike.com/nikeos/p/nikeplus/en_US/.

[23] K. Noyes. Android Browser Flaw Exposes User Data. PC-World. https://www.pcworld.com/businesscenter/article/211623/android_browser_flaw_exposes_user_data.html, November 24 2010.

[24] I. Oksanen and D. Hazael-Massieux. HTML Media Capture. http://www.w3.org/TR/html-media-capture/.

[25] OWASP (The Open Web Application Security Project. CSRF-Guard. https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project.

[26] Phonegap. PhoneGap. http://www.phonegap.com/.

[27] A. Popescu. Geolocation API specification. http://dev.w3.org/geo/api/spec-source.html.

[28] B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services for sensor device interoperability. In *Proceedings of IPSN*, pages 567–568, Washington, DC, USA, 2008. IEEE Computer Society.

[29] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of USENIX Security*, 2003.

[30] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of USENIX Security*, 2009.

[31] C. Reis and S. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of EuroSys*, Nuremberg, Germany, April 2009.

[32] D. Richardson and S. Gribble. Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices. In *Proceedings of USENIX WebApps*, Boston, MA, June 2011.

[33] P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent Client-side Mitigation of Malicious Cross-domain Requests. In *International Symposium on Engineering Secure Software and Systems*, February 2010.

[34] Secunia. Secunia Advisory SA29787: Mozilla Firefox Javascript Garbage Collector Vulnerability. http://secunia.com/advisories/29787, April 21 2008.

[35] P. Vixie. Dynamic Updates in the Domain Name System (DNS Update). RFC 2136 (Draft Standard), April 1997.

[36] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-principal OS Construction of the Gazelle Web Browser. In *Proceedings of USENIX Security*, pages 417–432, 2009.

[37] World Wide Web Consortium (W3C). Access Control for Cross-Site Requests. W3C Working Draft, September 12 2008.

[38] D. Yazar and A. Dunkels. Efficient Application Integration in IP-Based Sensor Networks. In *Proceedings of BuildSys*, November 2009.

[39] A. R. Yumerefendi, B. Mickle, and O. P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of NSDI*, 2007.

LIBERATED: A fully in-browser client and server web application debug and test environment

Derrell Lipman
University of Massachusetts Lowell

Abstract

Traditional web-based client-server application development has been accomplished in two separate pieces: the frontend portion which runs on the client machine has been written in HTML and JavaScript; and the backend portion which runs on the server machine has been written in PHP, ASP.net, or some other "server-side" language which typically interfaces to a database. The skill sets required for these two pieces are different.

In this paper, I demonstrate a new methodology for web-based client-server application development, in which a simulated server is built into the browser environment to run the backend code. This allows the frontend to issue requests to the backend, and the developer to step, using a debugger, directly from frontend code into backend code, and to debug and test both the frontend and backend portions. Once working, that backend code is moved to a real server. Since the application-specific code has been tested in the simulated environment, it is unlikely that bugs will be encountered at the server that did not exist in the simulated environment.

I have implemented this methodology and used it for development of a live application. All of the code is open source.

1 Introduction

Web-based client-server applications can be difficult to test and debug. Disparate development environments on the client and server sides, distinct skill sets for each, and a network that precludes easy synchronous debugging all impede debugging at the client side. Sometimes, the server environment provides little debugging and testing infrastructure.

I will describe here an architecture and framework that allows writing both the frontend code that runs on the client machine (i.e., in the browser) and the backend code that typically runs on a server machine, in a single language. Furthermore, this architecture allows debugging and testing the entire application, both frontend *and* backend, within the browser environment. Once the application is tested, the backend portion of the code can be moved to the production server where it operates with little, if any, additional debugging.

1.1 Typical web application development

There are many skill sets required to implement a modern web application. On the client side, initially, the user interface must be defined. A visual designer, in conjunction with a human-factors engineer, may determine what features should appear in the interface, and how to best organize them for ease of use and an attractive design.

The language used to write the user interface code is most typically JavaScript [6]. There need be at least a small amount of HTML to load the JavaScript code. Many applications are written using a JavaScript framework such as jQuery, ExtJS, or qooxdoo. Developers must therefore be fluent with both the language and the framework.

Debugging is generally accomplished using a debugger provided by the browser, or a plug-in to the browser.

The backend software includes the web server and database engine. Recent statistics [7] show that PHP and ASP.NET are the most popular languages for writing the backend code. Each provides a mechanism for receiving requests in the agreed upon application communication protocol (encoding) from the frontend. These languages also provide a means of communicating with a separate database server, or to an embedded database.

The application-specific backend code, or "business logic," is usually initiated by a web server which may or may not provide mechanisms for easy debugging of the application code. When a debugger is not available, the developer must rely on `print` or `log` statements to ascertain the code location of problems.

With the differing coding language and operating environment comes unique debugging methodologies. The skill sets required for debugging at the client and server are different, so any debugging session may require the availability of multiple people. Making debugging even more difficult is the asynchronous nature of the client-server interaction. Request messages are sent via the transport, and at some future time, response messages are returned. This separation of client and server means that it is not possible to use a debugger at the browser to step into code which is running on the server, nor even set a breakpoint that would allow stopping at the server-side handler for a key or button press at the user interface.

1.2 Research question

With the afore-mentioned problems in mind, I ask:

Is it feasible to design an architecture and framework for client-server application implementation that allows:

1. *all application development to be accomplished primarily in a single language;*
2. *application frontend and backend code to be tested and debugged within the browser environment; and*
3. *debugged and tested application-specific backend code to be moved, unchanged, from the browser environment to the real server environment, and to run there?*

In order to accomplish this, we first need a language that can be used both in the browser and on the server. For cross-browser use, the only viable choice is JavaScript. We therefore need a JavaScript implementation of the backend code that could run both in the browser and on the server, which can talk to whatever server-side database is to be used. The desired architecture is depicted in Figure 1.

Additionally, we need some form of abstraction that encompasses the set of database operations that are performed. The mechanism must map to a particular database on the server, and to a simulation of the database in the browser.

Two new questions arise out of such an architecture:

1. *How much of a compromise does this architecture impose, i.e., what common facilities become unavailable or more difficult to use?*
2. *Does this new architecture create new problems of its own?*

2 Introducing LIBERATED

LIBERATED is an architecture and JavaScript library upon which full web applications can be built. **LIBERATED** allows a web application to be debugged and tested, fully within the browser environment. Once all code is working, that same code can be moved to a real server, and run there. **LIBERATED** truly lives up to its name, liberating the developer from many of the hassles of traditional web application debugging.

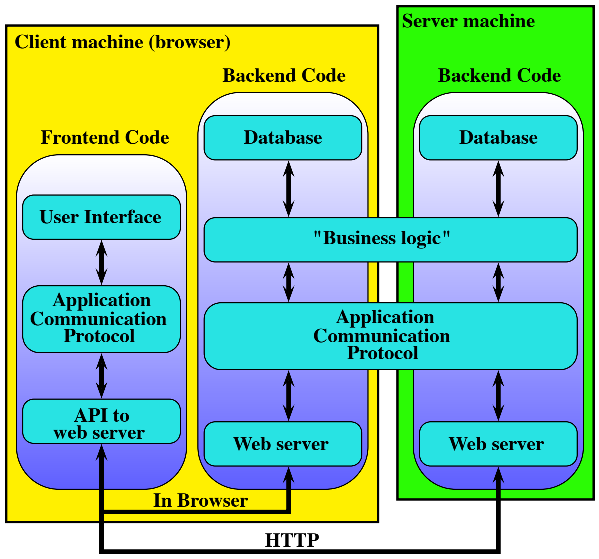


Figure 1: Desired architecture

LIBERATED is extensible. At present, it provides the following components:

- Database abstraction, used by an application
- Mapping from the database abstraction to the App Engine datastore
- Mapping from the database abstraction to SQLite[1]
- Mapping from the database abstraction to a simulated database which runs in the browser
- JSON-RPC Version 2.0 server
- Web server interface for App Engine
- Web server interface for the Jetty web server[2]
- Transport simulator to talk to an in-browser web server
- Hooks into the qooxdoo JavaScript framework, to allow use of the transport simulator in addition to its standard transports[3]

The following sections will discuss the overall architecture of **LIBERATED** and provide additional details of important components.

2.1 Architecture

In the backend, when using **LIBERATED**, an application's "business logic" code interacts with the database

[1] `http://sqlite.org`
[2] `http://jetty.codehaus.org/jetty/`
[3] `http://qooxdoo.org` (pronounced ['kuksdu:])

using a database abstraction provided by **LIBERATED**. Using the database abstraction allows the actual database to be *real* or *simulated*. A real database is the App Engine datastore, SQLite, MySql, etc, whereas a simulated database runs in the browser. Similarly, the backend receives requests from the frontend via a **transport** that can be either *real*, communicating across a network, or *simulated*, communicating solely within the browser.

LIBERATED handles requests which arrive via the selected **transport**. With a real web server such as provided by App Engine or Jetty, requests arrive via the HTTP protocol. When requests arrive via the simulated transport, they are placed on a queue, and handled in sequence from there, by a simulated web server.

The web server, whether *real* or *simulated*, determines which handler should process a request. A handler for the JSON-RPC server is currently implemented. Others, such as for REST could be added.

2.2 Development environment

The JavaScript framework upon which **LIBERATED** is implemented is qooxdoo. The qooxdoo framework provides a traditional class-based object programming model, and a wealth of additional functionality including classes to assist communicating over a network. There is nothing qooxdoo-specific, however, to this technology, and **LIBERATED** can be used in a non-qooxdoo-based application.

2.3 Database abstraction

In a common SQL-accessed relational database, data is organized into **tables** with names that identify the type of data that is stored in the **table**. A **table** contains **rows** of data, each with a common set of **columns** or **fields**. Each **row** is uniquely identified by a **key** value contained in one or more of its columns.

The database abstraction in **LIBERATED** is built upon a class called `liberated.dbif.Entity`. Each "table" can be thought of as being defined as a separate subclass of `liberated.dbif.Entity`. An instance of one of those subclasses, referred to as an **entity**, represents a row from that table. Each subclass of `liberated.dbif.Entity` defines a unique **entity type**.

`liberated.dbif.Entity` contains a method for registering the **properties** (like column names and types) which are members of each entity of that **entity type**.

To add a new object to the database, an entity of the proper subclass of `liberated.dbif.Entity` is instantiated, its property values set, and its `put()` member method called. When instantiating the subclass, the **key field(s)** of the **entity type** can be provided, to retrieve a specific existing object from the database. The `liberated.dbif.Entity.query()` function is used to retrieve specified sets of objects of an entity type from the database.

At present, relationships among entity types must be maintained by the application. Future plans include improvements in this area.

2.4 JSON-RPC server

The JSON-RPC server accepts incoming requests and returns responses in the format specified by the JSON-RPC Version 2.0 standard. [3] Remote procedure call methods are registered as a tuple consisting of the name of the method, a function that implements the remotely-accessible method, and an array that lists the names of the parameters. The latter allows requests to use either positional parameters or named parameters.

3 Example use of LIBERATED

To demonstrate, in part, how **LIBERATED** is used, consider a database entity which implements a counter. This simple entity type is shown in Listing 1.

Listing 1: Entity type definition for a simple counter

```
qx.Class.define("example.ObjCounter",
{
  extend : liberated.dbif.Entity,

  construct : function(id)
  {
    // Pre-initialize field data
    this.setData({ "count" : 0 });

    // Call the superclass constructor
    this.base(arguments, "counter", id);
  },

  defer : function()
  {
    var Entity = liberated.dbif.Entity;

    // Register the entity type
    Entity.registerEntityType(
      example.ObjCounter,
      "counter");

    // Register the properties
    Entity.registerPropertyTypes(
      "counter",
      {
        "id"    : "String",
        "count" : "Integer"
      },
      "id");
  }
});
```

The key field for this entity type is a string, referred to as *id*. As soon as this class has been loaded, the `defer()` function is called, which registers the entity type, so it is immediately available for use once the entire application has been loaded. The name of this class (`example.ObjCounter`) and the entity type name ("counter") are provided in the entity type registration, as shown on lines 19–21. This entity type has two properties: the counter's *id* and its *count*, which are registered on lines 24–30.

When a new object of this class is instantiated, default data is provided for the *count* field: it is initialized to zero, by line 8.

Listing 2 shows how remote procedure calls are implemented. Line 6 begins the registration of the remote procedure named "countPlusOne". Line 7 maps that name to the `countPlusOne` method which begins at line 13. Line 8 shows the list of parameters that are expected or allowed to be passed to the "countPlusOne" RPC. In this case, a single parameter, a counter ID, is expected.

Listing 2: RPC to increment a counter

```
1  qx.Mixin.define("example.MCounter",
2  {
3    construct : function()
4    {
5      // Register the 'countPlusOne' RPC
6      this.registerService("countPlusOne",
7                           this.countPlusOne,
8                           [ "counterId" ]);
9    },
10
11   members :
12   {
13     countPlusOne : function(counter)
14     {
15       var              counterObj;
16       var              counterDataObj;
17
18       liberated.dbif.Entity.asTransaction(
19         function()
20         {
21           // Get the counter object
22           counterObj =
23             new example.ObjCounter(counter);
24
25           // Get the application data
26           counterDataObj =
27             counterObj.getData();
28
29           // Increment the count
30           counterDataObj.count++;
31
32           // Write it back to the database
33           counterObj.put();
34
35         }, [], this);
36
37       // Return new counter value
38       return counterDataObj.count;
39     }
40   }
41 });
```

The implementation of `countPlusOne()` begins a database transaction to ensure that all manipulation of the database is accomplished based on a consistent database state. The function passed as the first parameter to `asTransaction()` will be called once a transaction has been established. When that function completes, the transaction will be ended.

The function to be run as a transaction begins at line 19. It first obtains the current counter object based on the specified counter ID, at line 22, and then retrieves that object's data map, at line 26. The data map contains the values of the two fields in this entity type (*id* and *count*).

The *count* field is incremented, and then the counter object is written back to the database with line 33.

The return value of this function, the counter's new value, is returned by `asTransaction()` after ending the transaction.

4 Discussion

One of the clear benefits of the **LIBERATED** architecture is that key portions of debugging and testing can be easier to handle than with traditional client-server applications. In this section, I will discuss some techniques that are now available, and our experience using them.

4.1 Debugging

The frontend and backend are traditionally initially debugged in isolation. They are often written in different languages, may be developed by different teams, and may not be able to run on the same machine. The interface between them may be implemented solely to a service API specification, with little ability for the frontend and backend to interact until both are nearly completed. There is often no easy way to use a single debugger to step through the code. It may be possible to have separate frontend and backend debuggers, but some server environments do not provide any easy means of debugging, and developers resort to `print` or `log` statements in the code.

With an application developed with **LIBERATED**, debugging of frontend and backend code need not be accomplished in isolation, both are written in the same language, and the service API can be exercised easily during development. This allows early and iterative debugging during the development process. The developer can use a debugger running in the browser to step from frontend code into backend code, or set breakpoints in backend code and then interact with the user interface to cause a request to be sent to the backend... and immediately have the debugger stop at that breakpoint.

4.2 Debugging Experience

During the course of developing the App Inventor Community Gallery, a complete application built upon **LIBERATED**, the **LIBERATED** architecture time and again proved itself to be a highly efficient and easy to use development and debugging environment. Instead of developing the frontend and backend code in isolation, we implemented and tested new user interface features and any corresponding backend changes concurrently. With **LIBERATED**, when new code doesn't work as intended, our typical debug cycle is:

1. Set a breakpoint in the remote procedure call implementation in the backend code. Run the program.
2. If the breakpoint in the RPC is hit, review the received parameters to ensure they are as expected. Step through the RPC implementation, noting variable changes, return values from functions, etc., until the problem is identified.
3. If the breakpoint in the RPC implementation is not hit, this indicates that there is likely a problem in the way the RPC is called. Set a breakpoint in the new frontend code, where the remote procedure call is initiated.
4. Run the program again, and at the breakpoint, ensure that the proper remote procedure call is being requested, and that the parameters have the expected values. If not, fix the problem, and repeat the process.
5. If, upon running the program in the previous step, the breakpoint is not hit, normal frontend debugging procedures are used to ascertain where the code is faulty.

5 Related work

I have been unable to find any literature or related projects which accomplish all of my goals set forth in Section 1.2. Although there is work in progress on the various sub-pieces described here, there appear to be none that would allow an application to be written in a single language, debugged and tested in the browser, and allow debugged, tested code to then be moved to the real server. Significant work which encompasses or relates to portions of my goals is described here.

5.1 Server-side JavaScript

The three JavaScript engines in common use are V8, used in the Chrome browser; SpiderMonkey, embedded in a number of Mozilla products; and Rhino, an implementation of JavaScript written in Java, also from the Mozilla Foundation. Each engine allows adding scripting to an application, so it is easy to build products around the engine. A plethora of such products have shown up in the last few years [8].

5.2 Web standard database interfaces

Work is progressing on a standard database interface for local storage of data at the browser. The proposal gaining acceptance for a browser database interface is *Indexed Database API* [5]. The Indexed Database API provides a programmatic database interface somewhat similar to the database abstraction in **LIBERATED**. Once it is widely available, the Indexed Database API could be used for an improved client-side simulated database in **LIBERATED**.

5.3 Reducing the distinction between client and server

The problem of different languages for client and server development is being tackled in different ways by various projects. The following sections describe some current work in progress.

5.3.1 Google Web Toolkit

Google's answer to unifying the client and server languages for web application development is called the Google Web Toolkit [1]. GWT allows the developer to write client-side code in Java, which is then translated into JavaScript to run in the browser. GWT is essentially backend-agnostic. GWT allows writing frontend applications in Java, and optionally also writing backend applications in Java, to accomplish the language unification.

5.3.2 Plain Old Webserver

Plain Old Webserver (POW) is a browser add-on that provides a web server that runs in the browser. The server "uses Server Side Javascript (SJS), PHP, Perl, Python or Ruby to deliver dynamic content." [4] Using Plain Old Webserver allows cross-platform, consistent access to a single server implementation. It runs on Firefox, on Linux, Mac, or Windows. It does not, however, provide the ability to step from frontend code into backend code.

5.3.3 Wakanda

Wakanda [2] provides a datastore and HTTP server, a Studio to visually design both the user interface and the data models which define how the datastore is organized, and a high-integrated code editor. It also provides the communications mechanism between frontend and backend, and data binding of user interface components to the datastore. The server-side language is JavaScript. Wakanda comes close to meeting the requirements of my research question, but it lacks **LIBERATED**'s critical ability to debug round trip operations, e.g., to trace into backend code upon initiation of a request via a frontend user action. It is also not fully cross-platform. The Wakanda Studio works only on Mac OS X and Windows, not on Linux. (The Wakanda Server, however, does run on Linux.)

6 Conclusions

The implementation of **LIBERATED** shows that an architecture that meets my research questions from Section 1.2 is feasible. **LIBERATED** allows both the frontend and backend of the application to be coded in JavaScript. With the simulated server running the backend code in the browser, all of the code can be debugged purely within the browser, with no need for an external server to run the backend code. Breakpoints can be set in backend code, within the browser, or the developer can step directly from frontend code into backend code. Finally, as has been shown with the Google App Engine and Jetty/SQLite interfaces of **LIBERATED**, the working application-specific backend code can be moved to a real server environment and run there.

The answers to my follow-up questions in Section 1.2 are not as clear cut, however.

6.1 Compromises of this approach

Although the architechture of **LIBERATED** is easy to work with and accomplishes the goals set out by my research question, a number of open issues remain, and it is yet to be determined how much impact these might have. These mostly pertain to the database abstraction. To wit:

- Testing a large web app often requires a substantial database. The current simulation database in **LIBERATED** is not adequate for complete testing of an application.
- **LIBERATED** does not yet provide for automated operations based on relations between entities.
- The complete set of property types which an application may use is defined by **LIBERATED**. The target database may allow other types.
- Some datastores, e.g., Google App Engine, do not require a pre-defined schema, but **LIBERATED** requires one.

6.2 New problems of this approach

There have been few new problems seen as a result of using this approach. The most obvious one is that server-side JavaScript is still young, and plentiful libraries of code are not yet available. Even now, though, **Node.js** is building a large library of code, easily `require()`'d (included) from custom code.[4] As server-side JavaScript matures, it appears likely that this problem may simply evaporate.

7 Recommendations

LIBERATED is a working implementation that is being used in a significant application. There is ample related and continuation work that can be done, however.

[4] `http://nodejs.org/`

The most urgent need is a rigorous evaluation of the benefits of **LIBERATED** vs. one or more traditional development paradigms. At present, my conclusions are based only on the development of App Inventor Gallery by one team of developers.

Additionally, there are some obvious improvements that can be made.

- Relationships between objects in **LIBERATED** are ad hoc, maintained exclusively by the application. Object relationships should be defined in the **LIBERATED** database abstraction, allowing for such things as automatic retrieval of related records or cascading deletes.
- The simulation database driver could use the HTML5 Indexed Database for a more capable simulated database.
- Query operators other than "and" should be supported.

Acknowledgments

The inspiration for **LIBERATED** was App Inventor Community Gallery, which was developed under a grant from Google to Professor Fred Martin at UMass Lowell.

Availability

The fully-open-source **LIBERATED** and App Inventor Community Gallery are available from their respective github repositories:

```
https://github.com/liberated/liberated
https://github.com/app-inventor-gallery/aig
```

References

[1] Google Web Toolkit Overview. `http://code.google.com/webtoolkit/overview.html`.

[2] 4D. Wakanda JS.everywhere(). `http://www.wakanda.org/features`.

[3] JSON-RPC Working Group. JSON-RPC 2.0 specification. `http://jsonrpc.org/spec.html`, Mar. 2010.

[4] Kellogg, D. Plain Old Webserver. `http://davidkellogg.com/wiki/Main_Page`.

[5] W3C. Indexed Database API. `http://dvcs.w3.org/hg/IndexedDB/raw-file/tip/Overview.html`.

[6] Web Technology Surveys. Usage of client-side programming languages for websites. `http://w3techs.com/technologies/overview/client_side_language/all`, January 2012.

[7] Web Technology Surveys. Usage of server-side programming languages for websites. `http://w3techs.com/technologies/overview/programming_language/all`, January 2012.

[8] Wikipedia. Comparison of server-side JavaScript solutions. `http://en.wikipedia.org/wiki/Comparison_of_server-side_JavaScript_solutions`.

JavaScript in JavaScript (js.js): Sandboxing third-party scripts

Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta
Princeton University

Abstract

Running on billions of today's computing devices, JavaScript has become a ubiquitous platform for deploying web applications. Unfortunately, an application developer who wishes to include a third-party script must enter into an implicit trust relationship with the third-party—granting it unmediated access to its entire application content.

In this paper, we present js.js, a JavaScript interpreter (which runs in JavaScript) that allows an application to execute a third-party script inside a completely isolated, sandboxed environment. An application can, at runtime, create and interact with the objects, properties, and methods available from within the sandboxed environment, giving it complete control over the third-party script. js.js supports the full range of the JavaScript language, is compatible with major browsers, and is resilient to attacks from malicious scripts.

We conduct a performance evaluation quantifying the overhead of using js.js and present an example of using js.js to execute Twitter's Tweet Button API.

1 Introduction

The web has undoubtedly become one of the most dominant application deployment platforms. Thanks to its wide support from today's consumer devices—from desktops and laptops to tablets and smartphones—the web's scripting language, JavaScript, is available on billions of devices.

One problem facing a web application developer is the implicit trust of including third-party scripts. A third-party script is a JavaScript file included from a party other than the application owner. Examples of commonly used third-party scripts are Google Analytics, Facebook's Like button, Twitter's Tweet buttons, and advertising platforms. When including one of these third-party scripts, the application is trusting the third-party script to execute only what is expected of it. The third-party, however, has full access to the application. It could redirect the page, modify the DOM, or insert malware.

An application owner could download the third-party script and serve it from his or her own servers. This at least ensures that the script being run by the application hasn't been modified without the application owner knowing. However, this is not always possible with dynamically-generated scripts (*e.g.*, advertisements), and it still doesn't ensure that the third-party script is not malicious. Third-party services often compress their code (*e.g.*, using the closure compiler), producing a large soup of JavaScript that can make it very difficult for a human or static analyzer to verify its behavior. Alternatively, the application could include the third-party scripts in an iframe, but iframes still have privileges (*e.g.*, alerts, redirection, etc.) that the application might want to disallow. It also requires cumbersome inter-iframe messaging for communication.

Static analyzers can be used to rewrite third-party JavaScript [16, 9, 2, 17] before it gets executed. This is often used on small, user-submitted widgets to guarantee their safety, but doesn't provide flexible, fine-grained control over the third-party script's privileges, and therefore, are not applicable to large, third-party libraries. Other approaches extend the browser itself to provide security mechanisms for third-party scripts [12]. While a nice approach, adopting a new standard in all major browsers is difficult and breaks backwards-compatibility.

In this paper, we present js.js, a JavaScript interpreter that runs on top of JavaScript. It allows site operators to mediate access to a page's internals by executing a third-party script inside a secure sandbox. We created a prototype js.js implementation by compiling the SpiderMonkey [3] JavaScript engine to LLVM [11] and then translating it to JavaScript using Emscripten [20]. The implementation is used to demonstrate the js.js API security features and benchmark performance on the SunSpider benchmark suite [4].

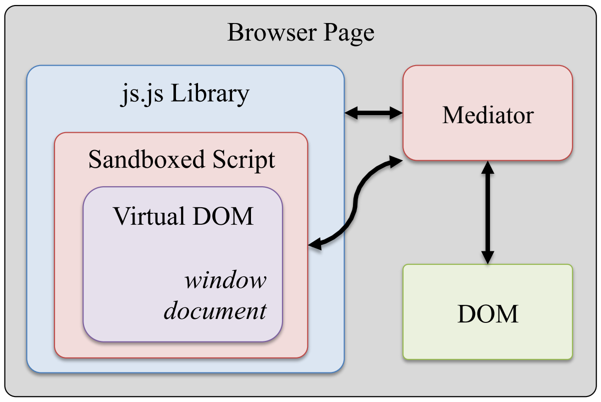


Figure 1: js.js architecture for example application

2 Design

The design goals for js.js are as follows:

- **Fine-grained control**: Rather than course-grained control, *e.g.*, disallowing all DOM access, an application should have fine-grained control over what actions a third-party script can perform.
- **Full JavaScript Support**: The full JavaScript language should be supported, including *with* and *eval*, which are impossible to support with static analysis.
- **Browser Compatibility**: All major browsers should be supported without plugins or modifications.
- **Resilient to attacks**: Resilient to possible attacks such as page redirection, spin loops, and memory exhaustion.

With these goals in mind, the js.js API has been designed to be very generic, similar in structure to the SpiderMonkey API. Rather than being specific to a web environment, the js.js API can be used to bind any kind of global object inside the sandbox space. Initially, a sandboxed script has no access to any global variables except for JavaScript built-in types (*e.g.*, Array, Date, and String), but the application can add additional names. In the web environment, these include global names like *window* and *document*. The js.js API allows an application, for example, to add a global name called *alert* that, when called inside the sandbox, calls a native JavaScript function. This way, the application using js.js has complete control over the sandboxed script since the only access the sandbox has to the outside is through these user defined methods. Thus these methods must give the script access only to the elements that the user allows.

Figure 1 shows an example application architecture using js.js. The *Mediator* is a JavaScript application that uses the js.js library to execute a third-party script in a sandbox. The *Virtual DOM* is comprised of the usual web-specific global variables that a script expects, but instead of referring directly to the browser, the mediator intercepts all access, such that it can allow or reject requests.

```
var src = "nativeAdd(17, 2.4);";
var jsObjs = JSJS.Init();

function nativeAdd(d1, d2) {
  return d1 + d2;
}

var dblType = JSJS.Types.double;
var wrappedNativeFunc = JSJS.wrapFunction({
  func: nativeAdd,
  args: [dblType, dblType],
  returns: dblType});

JSJS.DefineFunction(jsObjs.cx, jsObjs.glob,
    "nativeAdd", wrappedNativeFunc, 2, 0);

var rval = JSJS.EvaluateScript(jsObjs.cx,
    jsObjs.glob, src);

//Convert result to native value
var d = rval && JSJS.ValueToNumber(jsObjs.cx
    , rval);
```

Figure 2: Example of binding a native function to the global object space of a sandboxed script.

The js.js API aims to be easy to use and flexible. The example in Figure 2 demonstrates using the API to bind to the sandboxed environment, a global function called *nativeAdd* that accepts two numbers as arguments and returns the sum. `Init` initializes a sandboxed environment with standard JavaScript classes and an empty global object. `wrapFunction` is a helper function that allows an application to specify expected types of a function call. If the wrong types are passed to the function, an error is triggered in the sandbox space, which will result in an error handler being called in native space, allowing applications to detect sandboxed errors. `DefineFunction` binds the wrapped function to a name in the global object space of the sandbox, `EvaluateScript` executes the script, and `ValueToNumber` converts the result of evaluating the expression to a native number.

In addition to primitive types like bool, int, and double, the js.js API also includes helper functions for binding more complex types like objects, arrays, and functions to the sandboxed space. With the js.js API, an application can expose whatever functionality of the DOM it wants to a sandboxed script. It can also be used to run user-submitted scripts in a secure way, even providing a custom application-specific API to its users. Currently, creating this virtualized DOM is fairly complex. As future work, we wish to extend the js.js API to make it easier to use by allowing the user to easily setup white/black lists of browser elements, sites, etc.

	Fine-Grained DOM Control	Full JS Support	Page Redirection	Spin Loop / Terminate	Memory Exhaustion	Suspend / Resume
Direct Include	✗	✓	✗	✗	✗	✗
iframe	✗	✓	✗	✗	✗	✗
Web Worker	✗	✓	✓	✓	✗	✗
Static	✓	✗	✓	✗	✗	✗
Static + Runtime	✓	✗	✓	✗	✗	✗
js.js	✓	✓	✓	✓	✓	✓

Figure 3: Table of related work and the attack vectors which they can protect against or support (✓) and those that they cannot (✗). "Static" refers to purely static techniques such as ADsafe and Gatekeeper, while "Static + Runtime" refers to techniques such as Caja and WebSandbox.

The power of *eval* and *with* make them difficult to execute securely. For example, malicious code can use *eval* to disable or circumvent any protections that have been added through JavaScript code. Thus, most other techniques either completely prohibit using them or provide some limited version. However, the powerful sandboxing of scripts that js.js employs means that even *eval* and *with* can be executed securely as they can still only access the protected Virtual DOM.

Since js.js contains a full JavaScript interpreter (a compiled version of SpiderMonkey in our prototype implementation), it supports all variants of JavaScript that the interpreter supports. The js.js API allows an application to specify what version of the JavaScript language to use, anywhere from 1.0 to 1.8. Since an application can bind any name to the sandboxed space, full DOM support can be emulated. In addition, js.js is currently compatible with Google Chrome 7+, Firefox 4+, and Safari 5.1+. Because it requires Typed Array support, js.js will not support Internet Explorer until version 10 (currently in development) is released.

Another benefit of having the interpreter in JavaScript is that js.js has full execution and environmental control of sandboxed scripts. Thus it is relatively simple for js.js to prevent scripts staying in infinite loops or consuming large quantities of memory by placing optional checks inside the interpreter loop. This kind of protection is typically not possible in a normal protection system given the nature of JavaScript.

As seen in Figure 3, js.js is the only technique to meet all of our desired goals. A more detailed discussion of related work can be found in Section 6.

3 Implementation

Our initial prototype implementation of the js.js runtime has been created by compiling the SpiderMonkey [3] JavaScript interpreter to LLVM [11] bytecode using the Clang compiler and then using Emscripten [20] to translate the LLVM bytecode to JavaScript.

Emscripten works by translating each LLVM instruction into a line of JavaScript. Typed Arrays (a browser standard) allows Emscripten to emulate the native stack and heap, such that loads and stores can be translated to simple array accesses. When possible, Emscripten translates operations to native JavaScript operations. For example, an add operation is translated into a JavaScript add operation. An LLVM function call is translated into a JavaScript function call. It also has its own version of a libc implementation. By doing this, the translated output can achieve good performance.

SpiderMonkey comprises about 300,000 lines of C and C++. A lot of our implementation effort was spent patching SpiderMonkey so that it compiles into LLVM bytecode that is compatible with Emscripten's translator. Due to JavaScript's inability to execute inline assembly, the JIT capabilities of SpiderMonkey were disabled. We also contributed patches to Emscripten for corner cases that it had previously not encountered. We then wrote the js.js API wrapper around the resulting interpreter. Thus the js.js API greatly resembles SpiderMonkey's JSAPI.

The translated SpiderMonkey shared library (`libjs.js`) is 365,000 lines of JavaScript and 14MB in size. After closure compiling (`libjs.min.js`), it is 6900 lines and 3MB in size. After gzipping (which all browsers support), it is 594KB. Our wrapper API is about 1000 lines of code. The compiled SpiderMonkey library is available under the Mozilla Public License, while the rest (wrapper script and build scripts) are available under a BSD License. The library can be found at `https://github.com/jterrace/js.js`.

4 Demo Application

To give an example of running a third-party script using js.js, this section describes how to run Twitter's Tweet Button inside js.js. Twitter makes a script available for embedding a button on an application's website. Normally, an application loads Twitter's script directly from platform.twitter.com/widgets.js. Instead, we serve the *unmodified* widget code from our own server, as this version is confirmed to work with js.js, and use js.js to interpret it.

The Twitter widget script is 47KB of complicated, closure-compiled JavaScript. It expects to be running inside a web page with full access to the DOM. The basic flow of the script is as follows: a large amount of boilerplate code runs first that checks browser compatibility, the DOM is searched for `<a>` elements that match the twitter class selector, and each match is replaced with a new `<iframe>` element containing the Twitter button. Given that it spans a large portion of the DOM API, it is a good representative example of third-party scripts. Supporting this in js.js involved all of the following functionality:

- Binding many global objects to the sandbox space that the browser compatibility code checks for, such as `location`, `screen`, `navigator`, and `window` along with many of their properties and functions.
- Allowing the sandboxed code to bind to event handlers such as `DOMContentLoaded`, the `<iframe>` `onload` event, and the `message` event handler (used for inter-iframe communication).
- Many document utility functions such as `getElementsByTagName`, `getElementById`, and `createElement`.
- Wrapping real DOM elements with sandbox-space objects that provide functions like `getAttribute` and `setAttribute`, returning the real DOM element attributes when necessary.

When providing these objects, methods, functions, and handlers, we only provided the sandboxed code with just enough functionality that it can achieve its goal—creating a Twitter button—without allowing it access to any other unnecessary functionality. This demo script can be found at `http://jterrace.github.com/js.js/twitter/`.

5 Evaluation

We evaluate the js.js prototype implementation with both microbenchmarks of its API functions as well as with the SunSpider JavaScript Benchmark Suite [4]. The evaluation platform is a Macbook Pro with a 2.4 GHz P8600 and 4GB of RAM. The native tests were performed using SpiderMonkey (tag 20111220) with the JIT disabled. The js.js runtime was compiled from SpiderMonkey (tag 20110927) using clang and LLVM version 3.0, and Emscripten version 2.0.

Figure 4 shows the mean time (across ten executions) required to execute the startup and shutdown routines for the js.js runtime as well as the time required to evaluate a simple `1+1` expression. The overhead of creating the runtime environment to start executing a script is not an expensive cost.

Function	**Time (ms)**
libjs.min.js load	84.9
NewRuntime	25.2
NewContext	35.8
GlobalClassInit	15.5
StandardClassesInit	60.1
Execute 1+1	70.6
DestroyContext	33.3
DestroyRuntime	1.8

Figure 4: Mean (across 10 executions) runtime for various js.js initialization and execution procedures.

Figure 5 shows the SunSpider benchmark results for js.js in both Chrome and Firefox. For Chrome, most benchmarks fall in the 100x to 200x slowdown range, while Firefox lags behind in some benchmarks. This slowdown is due to a combination of inefficiency in the Emscripten compiler and the overhead of running JavaScript within JavaScript. Further efforts to improve the Emscripten compiler, along with manual optimization of the resulting JavaScript, could result in even better performance. However, this level of overhead could be acceptable to protect websites from untrusted third-party scripts. Trusted JavaScript code can still run natively alongside js.js protected code, especially effective if js.js is running in a Web Worker[1] The majority of the execution time is spent in js.js's main interpreter loop, a very large function that Emscriptenm, and JIT compilers in browsers, do not do very well at optimizing. We are currently working on ways to improve the performance of this function.

Also note that although the performance overhead of running js.js is high, other implementations of the js.js API could improve performance. For example, an implementation of the API could be built with Native Client [6] or incorporated directly into future versions of browsers, with the pure JavaScript implementation being used as a fallback if no faster implementations are available.

6 Related Work

iframes have been widely adopted because of the flexibility they provide for sandboxing third-party pages. An iframe allows each third-party script to have its own page

[1]Note that since the DOM API is synchronous and Web Workers are asynchronous, a blocking mechanism, such as the HTML5 file-system API, would have to be used.

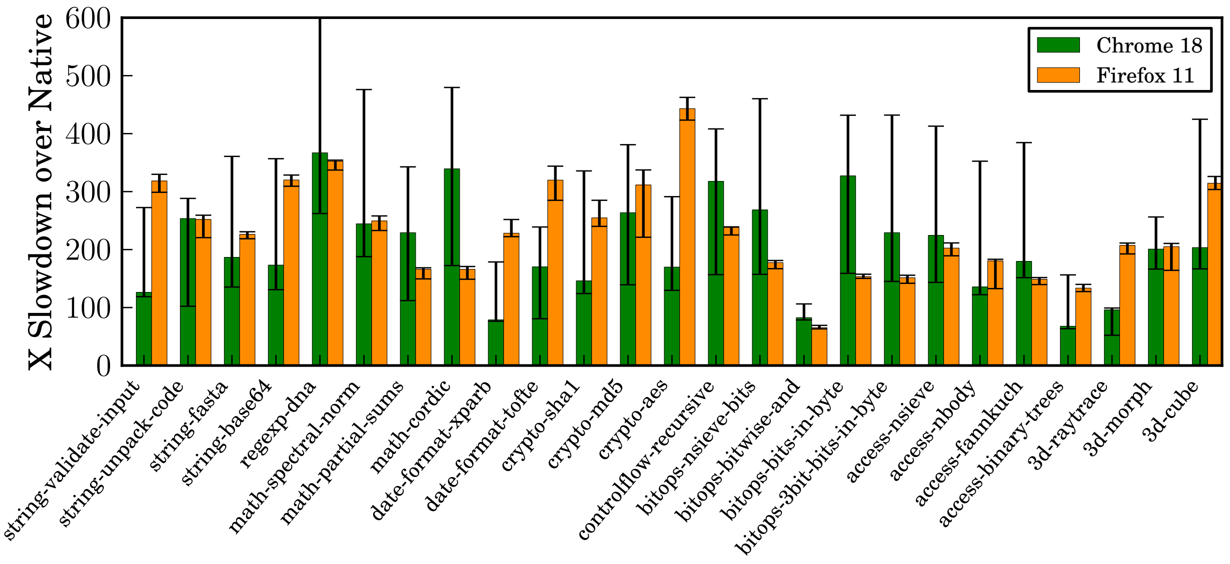


Figure 5: Relative Slowdown of js.js running the Sunspider benchmark in Google Chrome and Firefox 9. The median of ten runs of each benchmark is shown, with error bars corresponding to the minimum and maximum.

and the cross-origin policy prevents it from accessing the DOM outside of where it originates, however this approach has not fared well because today's web pages are complex and limiting a third-party's access to a single fixed-size iframe is not flexible enough. In addition, cross-iframe messaging is cumbersome—requiring established message-passing protocols between parties. iframes also don't prevent page redirection, window alerts, browser denial-of-service (via spin-loop), and memory exhaustion.

There has been a lot of work [17, 15, 14, 9, 1, 7] in the area of static javascript analysis of third-party scripts to restrict content and enforce security policies. These implementations typically restrict the way the JavaScript language features are used. They enforce these restrictions by using static analysis techniques to check the parameters passed to the various functions used by the script. Since much of the policy enforcement is done statically, these solutions typically have good runtime performance. However, it is very hard to determine the security aspects of such parameters by plain parameter checking unless one does a very robust execution tracing at runtime. For example, GateKeeper [9] employs a parameter checking model, but they cannot check the safety of the complex but useful functions, such as eval, setTimeout etc., whose parameters need to be passed to the JavaScript parser. In the cases of FBJS [1] and ADsafe [17], untrusted scripts are allowed to make calls to an access-controlled DOM interface, which again supports a very restricted version of Javascript and many of these access control checks are not sound. The cost in employing a restricted JavaScript subset is that some scripts may not conform to the subset, requiring they be ported.

Many recent techniques [16, 2, 8, 18] have taken the approach of transforming untrusted JavaScript code dynamically to interpose runtime policy enforcement checks. These works try to cover the many diverse ways in which a malicious code may subvert static policy enforcement checks. But even these policies restrict features of JavaScript (*e.g.*, `eval` and `with`) or some functions that eschew redefinition as in BrowserShield [18]. WebSandbox [2], for example, adopts a parameter checking model to verify the parameters being passed, but additionally creates a virtualized environment for third-party scripts in which the variables have a different namespace than what is visible to the native engine. However, the arguments of functions like eval, when generated dynamically, would bypass such instrumentation. Since the execution is still done on the native JavaScript engine, eval cannot be safely executed in the WebSandbox approach.

The Google Caja project [16] enforces security policies using a mixture of static and runtime techniques. Caja provides a compiler that transforms (cajoles) the third-party script into a milder version with less capability, *i.e.*, it restricts the way a script might use the DOM API or various JavaScript constructs such as `with` and `eval`. This is done by verifying that the script adheres to the required security policy using static analysis. Where it cannot confirm that the script is well behaved, it will annotate the application with runtime checks. In contrast, access to the DOM with js.js is unavailable by default and has to be explicitly allowed for the third-party script to be able to access a DOM object. This negates the need for changing the script source or restricting the way the third-party applications use JavaScript APIs. Instead, js.js captures the accesses to DOM objects using callbacks and enforces a security policy at runtime. js.js also allows for pausing or terminating the execution of runaway scripts while Caja cannot handle this issue (without solving the halting problem). Caja also requires server-side execution, while js.js is client-side only.

The recent introduction of Web Workers [5] has enabled a way of sandboxing third-party scripts, but to an extreme extent. A Web Worker prevents not just a malicious third-party script but any third-party script from accessing the DOM at all. A script running in a Web Worker essentially runs in parallel to the application UI and hence can be killed at any time by the parent application that forks it, preventing loop attacks. But the forked worker has very limited functionality, having to communicate with the parent through message-passing. This requires rewriting third-party scripts, decreasing its usability.

AdJail [13], has less restriction on JavaScript functionality and adopts access control mechanisms to regulate the access to host objects. But the access control model applied in this case is not flexible enough to dictate how the object is used once a third-party script validates access to it. Our approach gives such flexibility by letting site operators build wrappers to functions that pose a security risk.

A different approach is for the website owner to ask the underlying browser to enforce the owner's policies on any third-party JavaScript content, leaving the enforcement entirely to the browser's discretion. Using this method, a wide variety of fine-grained security policies can be enforced with low overhead as illustrated in Content Security Policies [19], BEEP [10] and Conscript [12]. Such a collaborative approach seems sound in the long term but today's browsers do not agree on a standard for publisher-browser collaboration, resulting in a large gap in near-term protection from malicious third-party scripts.

7 Conclusion

We have created the js.js API and runtime which allows for controlled and secure execution of untrusted JavaScript code. Our initial prototype implementation has been created by compiling the SpiderMonkey JavaScript engine to JavaScript. We then implemented the js.js API in JavaScript as a wrapper around the SpiderMonkey API. Using this API, we show secure execution of Twitter's Tweet Button and we evaluate the performance overhead of running JavaScript in JavaScript by evaluating on the SunSpider benchmark suite. In the future, we hope to both increase the performance of js.js as well as show its security potential on increasingly interesting examples.

Acknowledgements

We would like to thank Michael Freedman for his advisement and Alon Zakai for his help and excellent work on Emscripten.

References

[1] Facebook javascript. `http://developers.facebook.com/docs/fbjs/`.

[2] Microsoft web sandbox. `http://www.websandbox.org/`.

[3] Spidermonkey. `https://developer.mozilla.org/en/SpiderMonkey`.

[4] Sunspider javascript benchmark. `http://www.webkit.org/perf/sunspider/sunspider.html`.

[5] Web workers. `http://dev.w3.org/html5/workers/`.

[6] Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., and Yee, B. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proc. PLDI 2011*.

[7] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for javascript. In *Proc. PLDI 2009*.

[8] Felt, A., Hooimeijer, P., Evans, D., and Weimer, W. Talking to strangers without taking their candy: isolating proxied content. In *Proc. SNS 2008*.

[9] Guarnieri, S., and Livshits, B. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proc. USENIX Security 2009*.

[10] Jim, T., Swamy, N., and Hicks, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proc. WWW 2007*.

[11] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO 2004*.

[12] Livshits, B., and Meyerovich, L. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proc. IEEE Security and Privacy 2010*.

[13] Louw, M. T., Ganesh, K. T., and Venkatakrishnan, V. N. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proc. USENIX Security 2010*.

[14] Maffeis, S., Mitchell, J., and Taly, A. Run-time enforcement of secure javascript subsets. In *Proc. W2SP 2009*.

[15] Maffeis, S., and Taly, A. Language-based isolation of untrusted javascript. In *Proc. CSF 2009*.

[16] Miller, M. S., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized javascript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[17] Politz, J. G., Eliopoulos, S. A., Guha, A., and Krishnamurthi, S. ADsafety: Type-based verification of javascript sandboxing. In *Proc. USENIX Security 2011*.

[18] Reis, C., Dunagan, J., Wang, H. J., Dubrovsky, O., and Esmeir, S. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM TWEB 2007*.

[19] Stamm, S., Sterne, B., and Markham, G. Reining in the web with content security policy. In *Proc. WWW 2010*.

[20] Zakai, A. Emscripten: An LLVM-to-javascript compiler. `https://github.com/kripken/emscripten/blob/master/docs/paper.pdf`.

Aperator: Making Tweets Enable Actionable Commands on Third Party Web Applications

Peter Zakin
Princeton University
pzakin@princeton.edu

Soumya Sen
Princeton University
soumyas@princeton.edu

Mung Chiang
Princeton University
chiangm@princeton.edu

Abstract

Twitter has become a persistent part of our digital lives, connecting us not only to our individual audiences but also to an entire landscape of applications built for the web. While much has been done to support the Twitter ecosystem outside of Twitter, little has been done within Twitter to power those same applications. This work introduces a service called *Aperator*, which supports application-specific actionable commands through tweets. This ability creates several interesting opportunities for both end-users and application developers building on the Twitter platform. For example, the actionable command capability allows a link that a Twitter user shares with his followers to be directly added to any of the user's connected link sharing networks, such as Delicious or Read it Later. The client side of this system has a console for end-users to sign up and provide their login credentials for various web services that our system supports: Delicious, Foursquare, Read it Later, Foursquare etc. The system's backend has two cron jobs that run every minute to: (a) retrieve and parse tweets from a specific twitter account and store them in a command form in a MySQL database, and (b) execute the unexecuted commands found in the users tweets. This paper describes the concept, implementation, and results from an experimental study of this new application.

1 Introduction

Since its launch in 2006, Twitter has become one of the most important social properties on the web, trailing only Facebook and YouTube in terms of traffic [1]. Beyond the popularity of its own service, however, Twitter has also promoted the growth and engagement of third party websites through its API. As of May 2011, there are over 660k developers building applications on the Twitter API and over 900k operating applications [5]. The Twitter API has four major offerings: Twitter for Websites enables visitors at third party sites to make use of basic Twitter functionalities like following and tweeting on third party sites; the Search API provides query-access to recent Tweets; the REST API provides a way for developers to access user data and execute most of the main functionality of the Twitter service; lastly, the Streaming API permits an uninterrupted connection to the Twitter Firehose for developers to make use of data-sets.

Generally speaking, consumer web applications predominantly use the REST API and do so in three primary ways: to publicize user activity on the Twitter network (e.g. Foursquare, Quora), to stream user activity (e.g. Summify), or to operate their own Twitter client (e.g. TweetDeck). But the service that this paper describes represents a new mode for relating consumer web applications to Twitter.

The motivation for this approach is that although today's social web is great for sharing [8, 9], with so many apps it can be hard to connect what we share in one network to our presence on others. Moreover, for those who spend most of their time on only the major networks, it can be difficult to keep up with their audience on others [7]. Some networks today do provide limited interactivity through automatic forwarding of all user updates to a few other networks, as shown in Figure 1.1. But instead of either forwarding all updates from one network to the others or needing to log in and post on multiple networks, what if a user had the ability to selectively post from one network to all the other networks that the user cares about? Aperator [2, 3] realizes this vision by enabling users to operate real commands on a set of different web apps just by tweeting.

1.1 Contributions

Aperator demonstrates a new means of posting, which creates numerous benefits including:

- A method for *granular* cross-network posting: Cross-network posting is currently possible from

Twitter to Facebook and from Foursquare to Twitter, among others. But this form of cross-network posting is automatic and applies to all posts or none at all. The selective nature of aperator commands makes cross-network posting granular.

- A solution to make *multi-network online presences* more convenient for users: Although users may maintain several presences across a variety of web properties, it can be difficult to actively contribute on all of them. Generally and not at all surprisingly, users spend most of their time on larger networks like Facebook and Twitter as opposed to smaller networks like Delicious or Read it Later. Since aperator enables users to post content on some of those smaller networks from Twitter, it makes the multi-network presence more plausible.

- A way to increase *engagement* for third party applications: By making posting on third party applications as simple as tweeting, users can broadcast to multiple networks through one single interface. Thus, aperator can boost engagement on third party applications as another avenue for posting.

- A new *platform* for application development: Since users can interact with third party applications through the Twitter interface, aperator demonstrates the possibility of purely back-end applications. As an example, we created a prototype for such an application built on top of the aperator platform. The application, called "sms", provided the basic functionality of a group text messaging application from Twitter by using Twilio's service. Users could sign up by logging into aperator and editing their sms settings, which entailed adding or editing groups of numbers. If users wanted to send a text message to the members of their sms-groups, they would tweet: "@aperator sms #GROUPNAME Message".

Although increasing connectivity among different web applications in itself is not a new idea, Aperator's key contribution is in demonstrating what this system can enable. Aperator can be seen as a first step towards a TwitterOS, and using Twitter to provide some interesting features and services as discussed below.

First, there is a familiar and easy to use interface and you can access it from a wide range of clients. As a result, we did not develop a separate stand-alone shell-client on *aperator.com* and instead let users interface directly through Twitter.

Second, when users start using aperator services extensively and more features are developed by third-party developers using aperator as a platform, then user's command history will be publicly available for innovating a

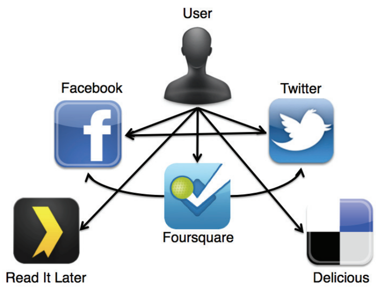


Figure 1: Partial interactivity among different social network applications

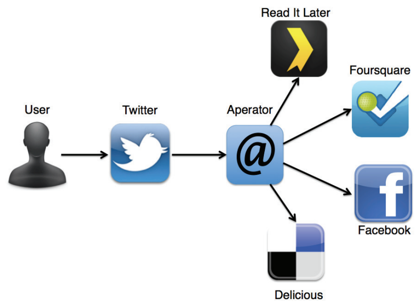


Figure 2: Aperator allows users to tweet selectively to audiences on different networks

variety of new services. For example, users can benefit from searching and finding out how their fellow users are using various sequence of commands to accomplish some particular task. While there are privacy concerns to be addressed, the presence of such a platform can create new possibilities for technical innovations.

Third, while Twitter already permits in-tweet commands for cross-posting to LinkedIn and Facebook, this capability stems from coordinated relationships between Twitter and other applications. Aperator democratizes this capability, making cross-posting simple even for organizations that are smaller than Facebook or LinkedIn. Additionally, Aperator commands have the potential to execute more sophisticated operations than posting, while Twitter's current model for cross-posting seems ill-poised to do so.

It should be noted that Aperator is not a 'tweeting from the command line' application like Twidge, Twitter CLI etc; it is effectively accessible from any Twitter client. Also, since Aperator is not a client interface, the recent moves by Twitter to discourage third party Twitter client developments in favor of consistent user experience [6] is not at odds with our approach.

1.2 Approach

Aperator [2] is a platform for making tweets actionable on third party web applications. By tweeting to the @aperator account, users are able to post content on web applications like Facebook, Foursquare, Delicious and

Read it Later, as illustrated in Figure 2. Just as a command line connects its users to applications on their operating system, Aperator connects its users (through Twitter) to applications on the social web. And like the command line, Aperator operates with a strict syntax, even as it enables powerful capabilities through a simple interface. The syntax currently supports four commands:

1. Link submission to Delicious: "@aperator delicious www.example.com [optional text]".
2. Link submission to Read it Later: "@aperator ril www.example.com [optional text]".
3. Post a status update on Facebook: "@aperator fb This is a status update".
4. Check-in on Foursquare: "@aperator 4sq Example Location".

In the absence of a revenue source to pay for users' texting, the sms app has not been featured in the public release of Aperator, but it does demonstrate a potentially powerful model. Building the sms application was greatly simplified since the user interface was almost entirely located on Twitter – the only exception was the group set-up page, on which users created groups and specified the names and numbers of its members. This shows that application developers can build mostly back-end applications that require little to no front-end interface since the end-users tweets have been shown to be sufficient as a means for user input.

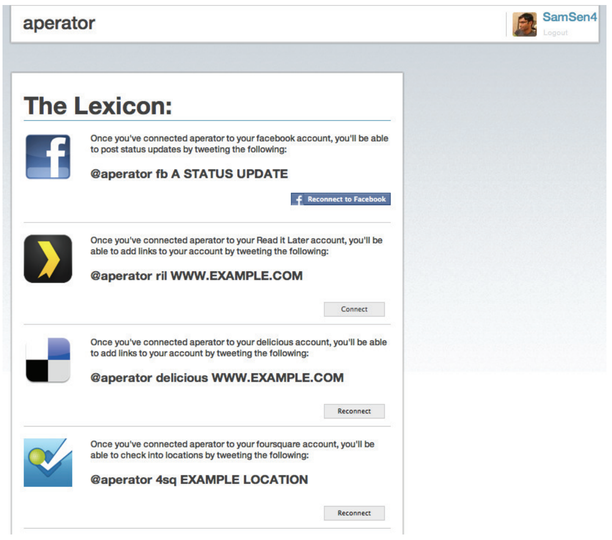
This paper is organized as follows: Section 2 describes the client-side and back-end architecture of Aperator. The implementation details, challenges, and limitations are elaborated in Section 3. Initial performance results from a small-scale functionality test of the system is reported in Section 4. Section 5 discusses the potential of app-specific command execution capability and future extensions, followed by conclusions drawn in Section 6.

2 Architecture

2.1 Client-side design

To get started on the service, users are required to sign up with their Twitter credentials. After Twitter authentication, users are redirected to aperator's signup page, where they also create their own Aperator login credentials. When users are logged in, they are presented with a visual display of aperators "Lexicon", which lists the available applications that can be connected alongside their associated command-syntax, like in Figure 3.

In order to start using any of the four apps that are currently supported by aperator – Foursquare, Delicious, Facebook and Read it Later – users first connect them to



The Lexicon:

Once you've connected aperator to your facebook account, you'll be able to post status updates by tweeting the following:

@aperator fb A STATUS UPDATE

Once you've connected aperator to your Read it Later account, you'll be able to add links to your account by tweeting the following:

@aperator ril WWW.EXAMPLE.COM

Once you've connected aperator to your delicious account, you'll be able to add links to your account by tweeting the following:

@aperator delicious WWW.EXAMPLE.COM

Once you've connected aperator to your foursquare account, you'll be able to check into locations by tweeting the following:

@aperator 4sq EXAMPLE LOCATION

Figure 3: Signup screen with a list of supported web apps

Aperator. For users wishing to connect aperator to their Foursquare or Facebook accounts, users follow a connect button to either apps' oAuth 2.0 authentication process, which upon verification, will redirect the user back to Aperator. But because Delicious and Read it Later utilize HTTP-Auth, Aperator stores users login credentials for these services. Therefore, when users choose to connect either of these two services to aperator, they are taken to a connect page. Upon successful connection, users are redirected back to the aperator home screen.

After users have connected aperator to any of the available apps, they can reconnect to an app if they changed their login credentials or if they happened to inadvertently revoke access to aperator. To reflect the ability to reconnect an app with different login credentials, what was previously labeled a "connect" button is now labeled for connected users as "reconnect."

2.2 Back-end design

While the end-user interface occurs almost entirely on Twitter's cross-platform properties after signup, the implementation and core of the system runs on the services server, which runs as an Amazon Web Services instance. The back-end processes involve two main parts: storing tweet-commands and executing commands that have not been executed.

Tweets are stored through two cron processes that run every minute, called stream1.php and stream2.php. Both processes are identical but for a sleep cycle in stream2.php which lasts 30 seconds. This way, tweets are captured from Twitter twice every minute – nearly every 30 seconds. The frequency of tweet-capturing can be increased as the service's adoption grows. Although Twitter does not release a formal rate-limit for the Search

API, the risks of being throttled seem to advise a conservative approach to tweet queries.

It should be noted that using the Streaming API is more conducive to the demands of this application since it allows developers to maintain long-lasting connections to the Twitter Firehose. In future releases, the Streaming API should be utilized since it would relinquish the need to run multiple cron processes and concerns about rate-limiting. However, given the prototype nature of the current version, we presently capture tweet-commands using the Search API.

The following snippet demonstrates the cURL response used to query the Search API:

```
curl http://search.twitter.com/search.json?q=\%
40aperator&include_entities=true
```

The search is specifically for all @mentions of @aperator, specifying tweet-entities as the return type. Specifying “include_entitites=true” in the request asks Twitter to include the expanded URL of links that Twitter has converted to the t.co link-shortened format. This request to Twitter returns a JSON response:

```
{"result_type":"recent"},
"profile_image_url":"...., ....,
"text":" @aperator ril http://t.co/gzPGiehI",
"to_user":"aperator",
"to_user_id":427607438,
"to_user_id_str":"427607438",
"to_user_name":"Aperator",
"created_at":"Wed, 11 Jan 2012 19:27:43 +0000",
"from_user":"pzakin", ....
```

The JSON specifies the constituent elements of a tweet object, e.g. “to_user_name” and “text”, and is easily parsed. From here, the “from_user” property of the tweet is checked against a MySQL table of Aperator users in order to make sure that tweets from non-users will not be processed. Assuming that the tweet comes from a registered user and that the app specified is valid, the command is stored in a MySQL table. The second half of the implementation consists of a process that executes commands that are yet to be executed. This process is managed by a cron job that runs each minute. To maximize for speed of execution–i.e., limiting the lag between catching the tweet from the Twitter Search response and its execution–the execution process operates inside a while loop, which iterates not less frequently than every five seconds. Upon execution, the command is designated as executed and ignored by the executing process in the future. Figure 4 shows the overall architecture and the processes that constitute the aperator system.

3 Implementation

The client console was developed using basic HTML, CSS and Javascript–the latter of which was mostly im-

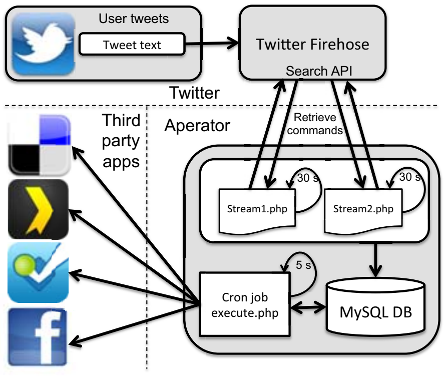


Figure 4: Aperator architecture

plemented using the javascript framework, jQuery. The back-end was built on a LAMP stack that was hosted on an EC2 instance from Amazon Web Services. As has already been described, the system itself was powered primarily by the Twitter API, which connected the aperator application to users’ tweets.

Although our current initial prototype uses the Search API, the use of Streaming API and even the REST API are also feasible. The Streaming API represents the best method for accessing @mentions of @aperator because it streams tweets almost instantaneously. We chose to use the Search API in the current prototype merely to avoid potential complications with maintaining a long-standing HTTP connection. Additionally, we opted against the REST API because of its requirement for authenticated use of the API.

3.1 Limitations

Although the Search and Streaming APIs can rapidly stream @mentions of @aperator, both cannot access @mentions coming from private users. Therefore, Twitter users who maintain a private account viewable only to their followers are presently unable to use the service.

Going forward, we have considered improving our current implementation by utilizing the Streaming API to catch @mentions of @aperator instantaneously. Additionally, to provide service access to Twitter users with private accounts, we will add a process utilizing the REST API that should run in parallel to our Streaming API connection. Since the REST API requires authentication, private @mentions will be retrievable as long as we require users to follow @aperator when they sign up, which can be easily added to our signup process.

Additionally, the current implementation for checking in on Foursquare merely “shouts” a venue name. We do not specify a ‘venue id’ that would enable a traditional Foursquare check-in. In order to retrieve a ‘venue id’, we would need to specify latitude and longitude coordinates. In fact, this should be possible since users can add their location to their tweets. Unfortunately, at this time, the Tweet entities returned by the Search API have returned "NULL" values for the "geo" field that otherwise should specify the needed coordinates. Until this issue has been resolved by Twitter, we are unable to check-in users to registered Foursquare locations.

3.2 Challenges

Besides the issues discussed regarding the Foursquare client, it is worth mentioning some other challenges and the workarounds we undertook to help coordinate access to Delicious and Read it Later in a user-friendly way.

First, Twitter shortens all links to the t.co wrapper format. When links are added to Read it Later or Delicious, it is important that they are translated back to an expanded URL. Otherwise, users viewing their saved links would fail to recognize the destinations of their links. Since Twitter provides an "expanded_url" field in their JSON response to search queries, we use it so that when the expanded_url is valid, the delicious request adds the expanded URL rather than the t.co shortened link.

Lastly, the Delicious API requires a "description" parameter for link-submission. In order to provide a meaningful description, we decided to scrape its "<title>" tag from its DOM structure. The code we used for that purpose is given below:

```
function scrapeTitle($url)
{   $file = file_get_contents($url);
    preg_match('/<title>(.*)<\/title>/i',
    $file, $title);
    $description = trim($title[1]);
    return urlencode($description);}
```

4 Evaluation

4.1 Deployment

The service website [2] was formally opened up to users on January 2, 2012 for experimentation and testing. Since then, 47 users have signed up for the service and 82 commands have been issued from Twitter, as shown in Figure 5. Although the adoption has been slow in the absence of promotional efforts on our part for this non-commercial service, our focus has been on testing and improving the system functionality as opposed to enlarging user base. Going forward, it will be important to reach out to Twitter users and adapt the service based on user feedback. Increasing the number of applications and services that users can connect to from Aperator will also be explored in the future.

4.2 Performance

We evaluate the performance of the system by measuring the execution time of commands issued to Aperator from

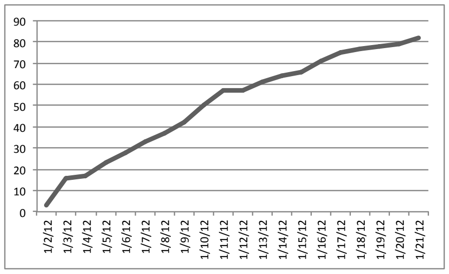


Figure 5: Cumulative growth in tweets sent to Aperator

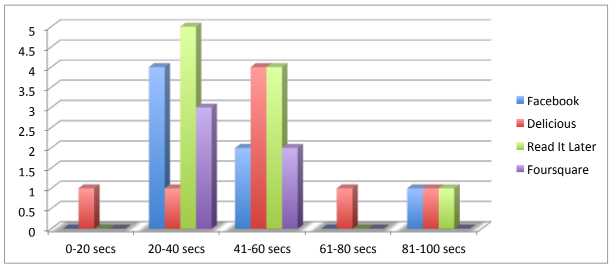


Figure 6: Histogram of the execution time of 30 commands for different web apps

user tweets. Each time a user submits a tweet, we record the time at which the tweet was created as well as the time at which the command was executed. Thus, execution time measures the time taken from a tweet post on Twitter to its execution for posting on another network. Based on a sample of 30 tweets, with execution times ranging from 20 seconds to 94 seconds, we found the mean execution time to be 44.23 seconds. The estimation of execution times for tweets of different third-party services is shown in Table 1.

Figure 6 shows a more granular picture of the distribution of execution times for tweet commands operated on by Aperator for different applications. The performance figures show that Delicious and Read it Later commands have relatively slower execution because both implementations entail an additional process of scraping titles for submitted links, which we've detailed before.

5 Discussions

Despite the presence of APIs across the vast majority of popular web services, the flow of data generally moves towards the larger networks as smaller networks publicize activity to a greater audience. In a sense, Aperator is a serious step towards moving more data from the larger networks to the smaller ones–which in effect, amounts merely to the ability to post content from a larger network to a smaller one. Because users spend more time on larger networks, Aperator effectively lowers the barriers to engagement that plague smaller networks.

While we currently support commands only coming from Twitter, it is possible and perhaps advantageous to replicate the Aperator service for Facebook. Sending

Table 1: Command Execution Time Statistics

Application	Mean (secs)	Std. Dev. (secs)
Delicious	50.25	20.58
Read It Later	43.4	21.72
Facebook	44.29	24.02
Foursquare	36.2	10.98

commands to third party web applications from Facebook would work quite similarly to the Twitter implementation. Users would merely update their status with specific commands for third party web applications. Admittedly, this would require a syntax adjustment because Facebook does not support user-to-application communication. Thus, to specify an Aperator command in Facebook, we plan on introducing a syntax such as: "## delicious www.example.com".

5.1 Related Developments

Increasing connectivity among different web applications is not a unique idea, even though Aperator is in many ways a unique implementation. IFTTT [4], for example, works with a host of different applications to automate actions between different services. For instance, if one uploads a file to her dropbox folder, IFTTT might send a tweet or a text message or post a status update on any number of services etc. The range of "recipes" permitted by IFTTT is extremely compelling when considering the potential benefits of inter-app synergy.

IFTTT has become a popular service in a new category of applications we might well call "API plumbing," but it certainly need not be the last. Aperator differs from IFTTT in the granularity of control it offers and the dynamism with which users can specify executable commands directly as tweets. There is tremendous potential to improve and rethink the relationships between different apps: synergy paves the way for new, unexplored experiences in the consumer web. Some of those experiences that deserve further exploration are presented next.

5.2 Future Use-cases

Facebook and Twitter have popularized the idea of signing up for web services using their respective login credentials. Supporting Facebook and Twitter authentication on third party applications expands the reach of their networks and for the benefit of third party properties, simplifies the authentication process and increases engagement. One of the ways in which Aperator increases the accessibility of third party web applications might very well be in authentication. Instead of signing up with Facebook or Twitter, users could simply tweet: @aperator install APPNAME. Once again, the analogy of the command line serves inspiration well. For just as "yum install emacs" provides a simple model for package installation on the command line, "@aperator install APPNAME" could provide an easier way for users to register on third party web applications as Aperator funnels authentication tokens in a pipeline from Facebook or Twitter to another application.

Right now, the Twitter presence of web applications amounts to little more than a stream of updates and announcements. But through Aperator, applications can begin to have actionable presences inside of a network. The suggestion, begged by the use-case, may be that Twitter could function in parallel to HTTP as an application medium. There is some truth, then, to the words of Paul Graham, the founder of Y Combinator, who wrote: *"Successful new protocols are rare... So any new protocol is a big deal. Each one of those protocols has spawned many successful companies. Twitter will too."*

Following such thoughts, a package installer for web applications is just the tip of the iceberg when it comes to new apps that can be developed on the Aperator platform.

6 Conclusions

This work introduces a new idea and an initial system prototype for a service, called Aperator, which supports application-specific actionable commands through tweets. Aperator facilitates granular cross-network posting and increases user convenience, thus opening up avenues for greater social utility and interactivity across web services. We review several benefits of this approach for both end-users and third-party applications, provide an architecture for enabling such services, report on initial performance results, and outline several potentially interesting extensions of this system.

References

[1] Alexa. http://www.alexa.com.

[2] Aperator. http://www.aperator.com.

[3] Aperator Demo. http://www.youtube.com/watch?v=tBzqShO29Xw.

[4] IFTTT. http://ifttt.com/.

[5] Twitter. http://www.twitter.com.

[6] Brown, M. Twitter Clamps Down On Third Party Clients. Wired [online], 14 March 2011. http://www.wired.com/epicenter/2011/03/twitter-third-party-clients/all/1.

[7] Bruno Goncalves, Nicola Perra, A. V. Validation of Dunbar's number in Twitter conversations, May 2011. arXiv:1105.5170v2.

[8] Krishnamurthy, B., Gill, P., and Arlitt, M. A few chirps about twitter. In *Proceedings of the first workshop on Online social networks* (2008), WOSN '08, pp. 19–24.

[9] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *Proc. of the 19th int'l conference on World wide web* (2010), WWW, pp. 591–600.

Don't Repeat Yourself: Automatically Synthesizing Client-side Validation Code for Web Applications

Nazari Skrupsky Maliheh Monshizadeh Prithvi Bisht Timothy Hinrichs
V.N. Venkatakrishnan Lenore Zuck

Department of Computer Science
University of Illinois at Chicago

Abstract

We outline the groundwork for a new software development approach where developers author the server-side application logic and rely on tools to automatically synthesize the corresponding client-side application logic. Our approach uses program analysis techniques to extract a logical specification from the server and synthesizes client code from that specification. Our implementation (WAVES) synthesizes interactive client interfaces that include asynchronous callbacks whose performance and coverage rival that of manually written clients, while ensuring that no new security vulnerabilities are introduced.

1 Introduction

Current practices in mainstream web development isolate the construction of the client component of an application from the server component. These practices are a byproduct of the fact that the client component is often written using a different programming language and platform (HTML and JavaScript in a web browser) than the server (e.g., PHP, Java, ASP), therefore necessitating developers with different skill sets. Independent development is problematic when the client and server share application logic. In this paper, we are concerned with a specific kind of application logic shared by the client and server: the input validation logic. Performing input validation on the client improves the user experience because of immediate feedback about errors, and if the validation is entirely self-contained on the client, it reduces network and server load. Performing input validation on the *server* is necessary for security, since a malicious user can otherwise bypass the client validation and supply invalid data to the server [2]. Necessarily then the client and the server must implement the same input validation logic if the application is to give users the interactive experience they expect, while ensuring the security of the application.

In this paper, we pursue a new methodology that aims to improve the development process and achieve a higher level of consistency. In our approach, web developers author the server side input validation of a web application, and WAVES automatically synthesizes the input validation for the client. If the input validation must change, the developer changes the server-side code and reruns WAVES. The benefits of our approach include:

- *Development efficiency*. Developers no longer repeat themselves— client validation code is automatically synthesized.
- *Greater compatibility and code efficiency*. The potential for validation mismatches between client and server is reduced, because developers can specify all validation checks in server code and use tools to generate equivalent validation code optimized for the client.
- *Improved security*. Our approach allows the development team to spend more time on the server side component, and encourages the specification of all validation checks in the server code itself.

Our implementation of WAVES uses program analysis techniques to automatically extract a logical representation of the input validation checks on the server and then synthesizes efficient client-side input validation routines. Of particular note is that WAVES generates code for validation checks that involve server-side state and utilize asynchronous requests (AJAX) to perform the required validation. The high-level challenges that WAVES addresses include:

- *Inference of server-side constraints*. The server-side validation may be performed in terms of server-side variables within deeply nested control flows of the application. The server-side constraints must be extracted and expressed in terms of the form fields.
- *Validation involving the server*. Some validation may involve server-side state for a variety of reasons.

2 Our Approach

WAVES (Web Application Validation Extraction and Synthesis), incorporates client side validation in applications in the following four conceptually distinct phases.

(1) Server analysis. WAVES first extracts the input validation constraints enforced by the server using dynamic program analysis. The key insight is that when the server is given an input it accepts, that input is processed along a success path. WAVES captures a sequence of if-statements along this path, which contains all the input validation constraints. With the execution trace, WAVES then rewrites the if-statements in terms of the original form field inputs and produces a list of potential input validation constraints. It then analyzes each one to determine if it is truly an input validation constraint—one that when falsified causes the server to reject the input. WAVES then identifies which constraints are dependent on the server's environment (the *dynamic* constraints) and which are not (the *static* constraints).

(2) Client-side code generation. Next, WAVES synthesizes client-side code to check the extracted constraints each time the user changes the value of a form field. Static constraints can be checked directly by JavaScript code, but dynamic constraints can only be checked by the server. So for each form field, WAVES generates client side code that first checks if any static constraints are violated and if not sends a message via AJAX to the server asking if any of the dynamic constraints are violated.

(3) Server-side code generation. The asynchronous messages sent by the client to check the dynamic constraints for a form field can only be responded to by special-purpose server-side code. (The original code assumes the user provided values for all form fields, but the clients asynchronous messages aim to check constraints even before the user completes the form.) Thus, to generate the proper server code that permits dynamic constraint checking on the client, WAVES performs code slicing on the server code to create an AJAX stub.

(4) Integration. After code generation, the client is augmented with event handlers that properly invoke the generated code and inform the user of errors. Server-side integration requires only uploading the generated AJAX stub code to the server's application directory.

3 Evaluation

We implemented WAVES for web applications written in PHP and clients written in HTML/JavaScript. Our implementation builds on Kaluza [4] (an SMT solver), and Pixy [3] (a tool for PHP dependency analysis).

To evaluate our approach we selected one form from each of the three medium to large and popular PHP applications. For each selected form, we first manually analyzed the server-side code and identified the constraints being checked — we call this the "ideal" synthesis and use it to assess the effectiveness of WAVES. For each application, Column 2 of Table 1 shows the ideal number of constraints (static + dynamic). As shown in the next column, WAVES was able to synthesize over 83% of the constraints identified by the ideal synthesis.

Application	**Ideal**	**WAVES**	**Existing**
B2Evolution	10+1	7+1	0
WeBid	17+8	16+6	0
WebSubRev	5+1	4+1	5+0

Table 1: WAVES synthesized 83% constraints successfully.

We also compared the code WAVES synthesized with code written manually by application developers. The third application in our test suite, `WebSubRev`, already had client-side validation. For this form, the server-side code checked 6 constraints (Column 1 Table 1), and the developer written client-side code checked 5 constraints (all of which were static). WAVES generated 4 static constraints and 1 dynamic constraint, therefore synthesizing 80% of the static constraints and 100% of the dynamic constraints. (The reason WAVES missed one constraint was due to a limitation of Kaluza.) We refer the interested reader to a more detailed technical report [1] that provides an in-depth treatment of issues involved in realizing WAVES as well as experimental data created for our tool.

4 Conclusion

The novel approach to developing web applications reported in this paper allows the developer to improve security (without sacrificing client interactivity) by focusing on hardening the server-side input validation. Our experimental results indicate that automated synthesis can result in highly interactive web applications, and the synthesized checks rival human-generated code in coverage and expressiveness.

References

[1] Automatically Synthesizing Client-side Validation Code for Web Applications. http://alcazar.sisl.rites.uic.edu/wavesTR.pdf, 2012.

[2] Bisht, P., Hinrichs, T., Skrupsky, N., Bobrowicz, R., and Venkatakrishnan, V. N. NoTamper: Automatic blackbox detection of parameter tampering attacks on web applications. In *the 18th ACM Conference on Computer and Communications Security* (Oct. 2010).

[3] Jovanovic, N., Kruegel, C., and Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *the 27th IEEE Symposium on Security & Privacy* (2006).

[4] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A Symbolic Execution Framework for JavaScript. In *SP'10: the 31st IEEE Symposium on Security and Privacy* (2010).