GDist-RIA Crawler: A Greedy Distributed Crawler for Rich Internet Applications

Seyed M. Mirtaheri, Gregor V. Bochmann, Guy-Vincent Jourdan[1], and Iosif Viorel Onut[2]

[1] School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Ontario, Canada
staheri@uottawa.ca, gvj@eecs.uottawa.ca, bochmann@eecs.uottawa.ca
[2] Security AppScan® Enterprise, IBM
770 Palladium Dr, Ottawa, Ontario, Canada
vioonut@ca.ibm.com

Abstract. Crawling web applications is important for indexing, accessibility and security assessment. Crawling traditional web applications is an old problem, for which good and efficient solution are known. Crawling Rich Internet Applications (RIA) quickly and efficiently, however, is an open problem. Technologies such as AJAX and partial Document Object Model (DOM) updates only make the problem of crawling RIA more time consuming to the web crawler. One way to reduce the time to crawl a RIA is to crawl a RIA in parallel with multiple computers. Previously published *Dist-RIA Crawler* presents a distributed breath-first search algorithm to crawl RIAs. This paper expands Dist-RIA Crawler in two ways. First, it introduces an adaptive load-balancing algorithm that enables the crawler to learn about the speed of the nodes and adapt to changes, thus better utilize the resources. Second, it present a distributed greedy algorithm to crawl a RIA in parallel, called *GDist-RIA Crawler*. The GDist-RIA Crawler uses a server-client architecture where the server dispatched crawling jobs to the crawling clients. This paper illustrates a prototype implementation of the GDist-RIA Crawler, explains some of the techniques used to implement the prototype and inspects empirical performance measurements.

Keywords: Web Crawling, Rich Internet Application, Greedy Algorithm, Load-Balancing

1 Introduction

Crawling is the process of exploring and discovering states of a web application automatically. This problem has a long and interesting history. Throughout the history of web-crawling, the chief focus of web-crawlers has been on crawling traditional web applications. In these applications there is a one to one correspondance between the state of the web application and its URL. The new generation of web applications, called *Rich Internet Applications* (RIAs), take advantage of availability of powerful client-side web-browsers and shift some part of application logic to the client. This shift often breaks the assumption of one-to-one correspondance between the URL and the state of the application. Thus, unlike a traditional web application, in crawling a RIA it is not sufficient to discover all application URLs, and it involves discovering all application states.

In a RIA, a client-side page, associated with a single URL, often contains executable code that may change the state of the page as seen by the user. This state is stored within the browser, and is called the *Document Object Model* (DOM). Its structure is encoded in HTML and includes the program fragments executed in response to user input. Code execution is normally triggered by events invoked by the user, such as *mouse over* or *clicking* events. To ensure that a crawler finds all application content it must execute every events from every reachable application states. Thus, under the assumption that a RIA is deterministic, the problem of crawling is reduced to the problem of executing all events in the application across all reachable DOMs.

One can reduce the time it takes to crawl a RIA by executing the crawl in parallel on multiple computational units. By considering each state of the application on the client side (henceforth simply referred to as *state*) as a vertex and each JavaScript event as an edge, the problem of the parallel crawling a RIA is mapped to the problem of parallel exploration of a directed graph.

Dist-RIA Crawler [28] introduced a distributed crawler for RIAs that achieves parallelism by having all the crawlers go to each application state, however, each crawler only explores a specific subset of the events in that vertex. The union of all these events covers all of the events in the state. In Dist-RIA Crawler, each crawler node implements a breath-first search algorithm in its own scope.

Dist-RIA Crawler assigns equal number of events to each node. The underlying assumption is that all nodes have equal processing power, and thus equal workload is to be assigned to the nodes. To enhance Dist-RIA Crawler to take advantage of heterogeneous set of nodes available, this paper introduces a mechanism to adapt to the perceived speed and processing power of the nodes. This algorithm is explained in Section 3.

In the context of RIA crawling, crawling strategy refers to the strategy the crawler follows to decide the next event to execute. Dincturk et al. [5, 13, 15] studied several crawling strategies to optimize the crawl in two dimensions: reducing the total time of the crawl, and finding new application states as soon as possible in the crawl. Among the strategies studied, the greedy algorithm [29] scores well in the majority of cases, and it is much better than breath-first and depth-first search strategies. This algorithm always chooses the closest application state with an un-executed event, goes to the state and execute the event. This paper studies distribution of the greedy algorithm.

In Dist-RIA Crawler, the nodes only broadcast the knowledge of application states, and no single node had the entire knowledge of the transitions between the states. This restriction does not allow a Dist-RIA Crawler to run the greedy algorithm: knowledge of application transitions is a prerequisite for the greedy algorithm. At the same time, broadcasting all transitions to the entire group of workers can make the network a bottleneck.

This paper introduces GDist-RIA Crawler, a client-server architecture to integrate the greedy algorithm into the architecture of the Dist-RIA Crawler. The GDist-RIA Crawler runs the greedy algorithm on the server and runs the crawling jobs to the client nodes. The server node is henceforth referred to as the *coordinator* and the client nodes responsible to crawl the website are henceforth referred to as the *nodes*. Nodes ask the coordinator for tasks to do, the coordinator runs the greedy algorithm on the application graph and responds them with a set of events to execute. Nodes execute the assigned tasks and inform the coordinator about the transition they discovered. The coordinator is the only computer that keeps the knowledge of application graph.

The greedy nature of the algorithm makes the GDist-RIA Crawler superior to the Dist-RIA Crawler (which runs breath-first search) by reducing the total number of events executed to crawl an application. The GDist-RIA Crawler is also superior to the centralized greedy algorithm in that it harnesses the power of multiple nodes to reduce the time it takes to crawl the target application. Further, it does not require the load-balancing algorithm introduced in Section 3 that is required by the breath-first search strategy, since only idle nodes ask for work from the coordinator, and thus no node becomes a bottleneck.

This paper contributes to the body of crawling literature by enhancing the previously presented Dist-RIA Crawler in two ways. First by introducing an adaptive load-balancing strategy to harness availability of heterogenous nodes. Second by introducing a client-server architecture to concurrently crawl RIAs. We share our empirical experience with the introduced model and some of the challenges we faced in capturing client-side events.

The rest of this paper is organized as follows: In Section 3 we introduce a new adaptive load-balancing algorithm. In Section 4 we give an overview of the GDist-RIA Crawler. In Section 5 we describe some of the technical aspects of implementing the GDist-RIA crawler. In Section 6 we evaluate various performance aspects of the GDist-RIA Crawler. In Section 2 we give an overview of the related works. Finally, in Section 7 we conclude this paper.

2 Related Works

This work is not the first of its kind in addressing the issue of RIA model construction and model checking. Duda et al. [16, 19, 24] uses Breadth-First search crawling strategy to crawl RIAs. Crawljax [25, 26] leans toward Depth-First search strategy. Other works aim at constructing the FSM model of the application [1–3, 23].

Model-based crawling is another area of research that gained momentum in recent years. Benjamin et al. [6, 14] present they hypercube model that assumes the target application is a hypercube. Choudhary et al. [10, 11] introduce Menu model that assumes events reach the same target state, irrelevant of the source state. Greedy strategy was explored by Peng et al. [29]; and Milani Fard and Mesbah [27] in a tool called FeedEx. An empirical comparison of different crawling strategies is done by Dincturk et al [13, 14].

Parallel crawling of traditional web applications has been explored extensively in the literature [7–9,17,18,20,21,30,31]. Parallel crawling of RIAs however is a new field and the only work we know of is Dist-RIA Crawler [28]. Dist-RIA Crawler performs a breath-first search over multiple independent nodes. This paper adds a load-balancing algorithm to the breath-first search. It also works on the superior and more efficient greedy algorithm.

A close topic to Model-based crawling is DOM equivalency. Duda et al. [16, 19, 24] used equality of DOMs to measure their equivalency. Crawljax [25, 26] uses edit distance to do so. Amalfitano et al. [2] compares the two DOMs based on the elements in them. *Imagen* [22] takes into account JavaScript functions closure, event listeners and HTML5 elements as well in identifying the state of the application. In this paper an DOM equality, the most strict form of DOM equivalency, is used.

3 Load-Balancing

The following notations are used in this section and the rest of the paper:

- s: Refers to an application state.
- e: Refers to an event.
- S: The total number of application states.
- E_s: The number of events in the application state s.
- E: Sum of the number of events in all application states.
- N: Number of crawler nodes.
- i: A unique identification number of a node, where $1 \leq i \leq N$.

As described earlier, in each state, Dist-RIA Crawler assigns equal shares of work to the nodes. The load-balancing algorithm presented in this section, refered to as *adaptive* approach, adjusts the portion of events assigned to each node as the crawling proceeds. The manipulation of the portion assigned to the nodes is used as a tool to reduce the workload of the overloaded nodes, and increase the workload of the idle nodes. One of the nodes, called *coordinator*, calculates the portion of the events to be assigned to each node at the time of state discovery. Tasks are not assigned equally, but assigned based on the perceived computational speed of the node and its current workload.

The purpose of the assignment is to drive all nodes to finish together. The portion of events in state s that belong to node i is represented by $P_{s,i}$ where $P_{s,i} \in [0, 1]$. The coordinator uses the assignment of tasks to different nodes as a means to increase the chance of all nodes to finish together, and no node becomes a bottleneck. To achieve this goal, for every node i, the coordinator uses the number of events executed so far by the node (called ET_i) to calculate the execution speed of the node. This execution speed is used to forecast the execution rate of the node in the future. Based on the calculated speed for all nodes, and the given remaining workload of each node, the coordinator decides the portion of the tasks that are assigned to each node.

3.1 Adaptive Load-Balancing Algorithm

Assume that a new state s is discovered at time t. The coordinator calculates v_i, the speed of node i, as:

$$v_i = ET_i / t \tag{1}$$

where ET_i is the number of events executed by node i so far. The remaining workload of node i can be calculated as the difference between the number of assigned events (called AT_i) and the number of executed events ET_i. Based on the calculated speed v_i, the coordinator calculates the time it takes for node i to finish execution of remaining events assigned to it. This time to completion is represented by TC_i and is calculated as follow:

$$TC_i = \frac{AT_i - ET_i}{v_i} \tag{2}$$

After the coordinator distributes the new events of a newly discovered state s among the nodes, the time to complete all events will change. Assuming node i will continue executing events at rate v_i, the new estimation for the time to finish, called TC'_i, is:

$$TC'_i = TC_i + \frac{P_{s,i} \times E_s}{v_i} \tag{3}$$

To drive all nodes to finish together, the coordinator seeks to make TC' equal for all nodes. That is, it seeks to make the following equation valid:

$$TC'_1 = TC'_2 = \cdots = TC'_N \tag{4}$$

Equations 4 can be re-written using equation 3:

$$TC_1 + \frac{P_{s,1} \times E_s}{v_1} = TC_2 + \frac{P_{s,2} \times E_s}{v_2} = \cdots = TC_N + \frac{P_{s,N} \times E_s}{v_N} \tag{5}$$

Let us take the first two expressions and re-write then:

$$TC_1 + \frac{P_{s,1} \times E_s}{v_1} = TC_2 + \frac{P_{s,2} \times E_s}{v_2} \tag{6a}$$

$$\Rightarrow \frac{(TC_1 + \frac{P_{s,1} \times E_s}{v_1} - TC_2) \times v_2}{E_s} = P_{s,2} \tag{6b}$$

Similarly $P_{s,2}$, $P_{s,3}$, $\ldots$ and $P_{s,N}$ can all be expressed as follow:

$$\forall i : 2 \le i \le N : P_{s,i} = \frac{(TC_1 + \frac{(P_{s,1} \times E_s)}{v_1} - TC_i) \times v_i}{E_s} \tag{7}$$

The coordinator intends to assign all of the events in the newly discovered states to the nodes. Thus the sum of all Ps for state s is 1. Therefore:

$$1 = \sum_{i=1}^{N} P_{s,i} \tag{8}$$

By expanding $P_{s,2}$, $P_{s,3}$, ... and $P_{s,N}$ in equation 8, using equation 7, we get:

$$1 = P_{s,1} + \sum_{i=2}^{N} \frac{(TC_1 + \frac{(P_{s,1} \times E_s)}{v_1} - TC_i) \times v_i}{E_s} \tag{9a}$$

$$\Rightarrow P_{s,1} = \frac{1 - \sum_{i=2}^{N} \frac{(TC_1 - TC_i) \times v_i}{E_s}}{1 + \frac{E_s}{v_1 \times E_s} \times \sum_{i=2}^{N} v_i} \tag{9b}$$

Given the value of $P_{s,1}$ using equation 9, the value of $P_{s,2}$, $P_{s,3}$, ... and $P_{s,N}$ can easily be calculated using equation 7.

The adaptive approach does not guarantee that all nodes finish together. The assignment eliminates bottlenecks only if there are enough events in a newly discovered state s to rescue every bottlenecked node. In the other words, if there are not enough events in s, and the workload gap between the nodes is large, the adaptive approach fails to assign enough jobs to all idle nodes and make them busy so that all nodes finish together.

4 Overview of the GDist-RIA Crawler

This section describes the crawling algorithm that the GDist-RIA crawler uses.

4.1 Design Assumptions

The GDist-RIA Crawler makes the following assumptions:

- **Reliability**: Reliability of nodes and communication channels is assumed. It is also assumed that each node has a reachable IP address.
- **Network Bandwidth**: It is assumed that the crawling nodes and the coordinator can communicate at a high speed. This makes the network delay intangible. Note that there is no assumption made about the network delay between the server or servers hosting the target application and the crawling nodes.
- **Target RIA**: The GDist-RIA Crawler only targets deterministic finite RIAs. More formally, the GDist-RIA Crawler assumes that visiting a URL always leads to the same STATE; and from a given STATE, execution of a specific JavaScript event always leads to the same target STATE.

4.2 Algorithm

The GDist-RIA Crawler consists of multiple nodes. The nodes do not share memory and work independently of each other. Nodes communicate with the coordinator using a client-server architecture. Nodes start by contacting the coordinator for the seed URL. After loading the seed URL (i.e. the URL to reach the starting state of the RIA), and after executing any path of events, a node sends the hash of the serialized DOM (henceforth referred to as the ID of s), as well as E_s to the coordinator.

In response, the coordinator who has the knowledge of the application graph calculates the closest application state to s with an unexecuted event and sends a chain of events that lead to that state back to the probing node. This path may start with a reset order by visiting the seed URL. In addition, the coordinator sends the index of the un-executed event in the target state to the probing node.

The probing node executes the assigned event and sends the transition to the coordinator. The coordinator again runs the greedy search algorithm and responds to the client with a new chain of events. This process continues until all the events in all the application states are executed. If at any point the coordinator realizes that there is no path from the state of the probing node to a state with unexecuted events, it orders the node to reset. In effect, by reseting the node jumps back to the seed URL. Since all application states are reachable from the seed URL, the node will find events to execute after the reset.

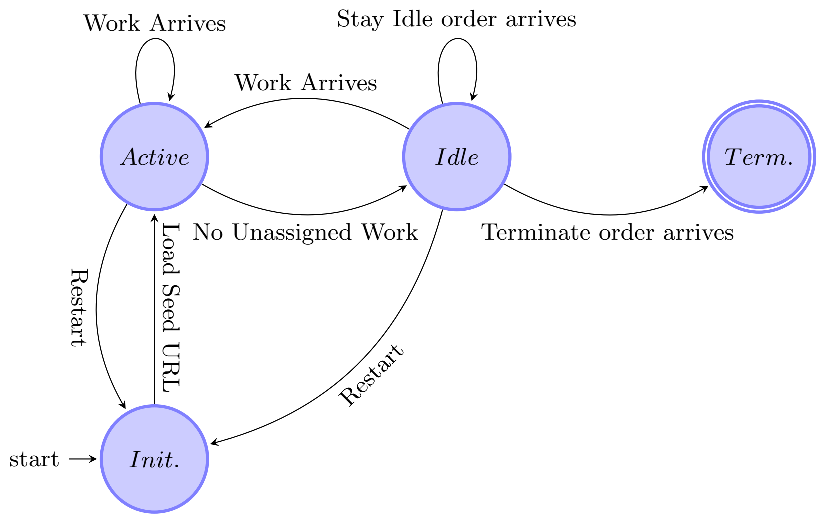

Fig. 1: The Node Status state diagram.

Figure 1 shows the node state diagram of a crawler node. The crawler starts in the *Initial* state. In this state, the crawler starts up a headless browser process. It then loads the seed URL in the headless browser and goes into the *Active* state. Crawling work happens in the Active state. After finishing the assigned task, the node goes to the *Idle* state. The node stays in the Idle state until either more work becomes available or a termination order from the coordinator marks the end of the crawl. During Active and Idle states, the coordinator may order the node to restart so it can reach states that are unreachable from the current state of the node.

4.3 Termination

When the following two conditions are met the coordinator initiates the termination protocol by sending all nodes a *Terminate* order:

– All nodes are all in Idle state.
– There is no Unassigned work in the coordinator i.e. all events in the discovered states are assigned to the nodes.

5 Implementation

To ensure that the proposed algorithm is practical a prototype of the system was implemented. This section explains some of the technical challenges in implementing the prototype of the GDist-RIA crawler.

5.1 Running a Virtual Headless Browser

The GDist-RIA Crawler uses an engine, called *JS-Engine*, to handle web client events[3]. The primary task of JS-Engine is to execute JavaScript events and it uses *PhantomJS*[4], an open source headless WebKit, to emulate a browser with the capability to do so.

Due to the asynchronous nature of the JavaScript, the crawler can not simply trigger an event and consider the execution finished when the call returns. Executing an event in JavaScript may trigger an asynchronous call to the server, or schedule an event to happen in the future. When these events happen the state of the application may change. More formally, two main types of the events that may have dormant ramifications include: *Asynchronous calls* and *Clock events*.

Upon triggering an event on the target application, the JS-Engine waits until the event and all its ramifications are over. For this to happen successfully, the JS-Engine requires a mechanism to keep track of all asynchronous calls in progress and wait for their completion before continuing. Unfortunately, JavaScript does not offer a method to keep track of AJAX calls in progress. Thus the JS-Engine redefines *send* and *onreadystatechange* methods of *XMLHttpRequest* object, the native JavaScript object responsible for performing asynchronous requests, such that the target web application notifies the crawler application automatically upon start and finish of every asynchronous call (Listing 1.1)[5][6].

[3]This paper only focuses on JavaScript events and leaves other client side events such as Flash events to the future studies.

[4]`http://phantomjs.org/`

[5]*XMLHttpRequest* is the module responsible for asynchronous calls in many popular browsers such as Firefox and Chrome. Microsoft Internet Explorer however does not use module, and instead it uses *ActiveXObject*.

[6]Due to space limitation rest of code snippets in this section are omitted.

Listing 1.1: Hijacking Asynchronous Calls

```
XMLHttpRequest.prototype.sendOriginal = XMLHttpRequest.prototype.send;
XMLHttpRequest.prototype.send = function (x){
       var onreadystatechangeOriginal = this.onreadystatechange;
       this.onreadystatechange = function(){
              onreadystatechangeOriginal(this);
              parent.ajaxFinishNotification();
       }
       parent.ajaxStartNotification();
       this.sendOrig(x);
};
```

The second source of asynchronous behaviour of a RIA with respect to the time comes from executing clock functions, such as *setTimeout*. This methods is used to trigger an event in the future. In many cases, such events can help animating the website, and adding fade-in fade-out effects. Knowledge of the existence of such dormant functions may be necessary to the JS-Engine. Similar to the asynchronous events, JavaScript does not offer a method to keep track of the time events. Thus JS-Engine re-defines *setTimeout* to hijack time events.

JS-Engine needs to identify the user interface events (i.e. the events that can be triggered by the user interacting with the interface) in the page. Events that leave a footprint in the DOM are easy to detect: Traversing the DOM and inspecting each element can find these events. Attached events using *addEventListener*, however, do not reflect themselves on the DOM. The final challenge faced by JS-Engine is to detect these client-side events attached through event listeners.

These events are added through a call made to *addEventListener* and are removed through a call made to *removeEventListener*. To handle event listeners, JS-Engine redefines *addEventListener* and *removeEventListener* methods such that whenever a call is made to *addEventListener* an entry is added to a global object, and when a call is made to *removeEventListener* the corresponding element is removed. Hence at any given point, JS-Engine can simply check the contents of this object to get elements with attached events.

6 Evaluation

The coordinator prototype is implemented in PHP 5.3.10 and MySQL 14.14. The coordinator contacts the node using SSH channel. The nodes are implemented using PhantomJS 1.9.2, and they contact the coordinator through HTTP. The coordinator as well as the nodes are hosted on a Linux® Kernel 3.8.0 operating system with an Intel® Intel® Core(TM)2 Duo CPU E8400 @ 3.00GHz and 3GB of RAM. The communication happens over a 10 Gbps network.

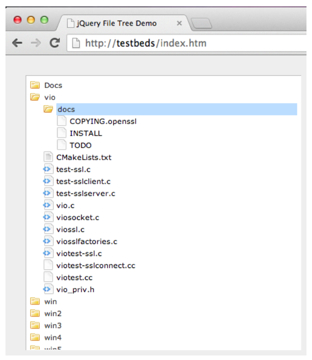

Fig. 2: File tree browser RIA screen-shot

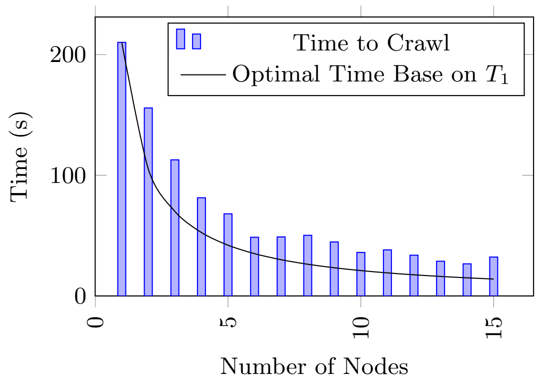

Fig. 3: The total time to Crawl the target RIA with multiple nodes.

6.1 Testbed

To measure the performance of the crawler in practice a jQuery based RIA called *jQuery file tree*[7] was chosen. This open source library creates a web interface that allows the user to brows a set of files and directories through a browser. Similar to most file browsers, directories can be expanded and collapsed, leading to the new client side states. Expanding a directory triggers an asynchronous call to the server to retrieve the contents of that directory. Picture 2 shows a picture of a jQuery file tree application.

6.2 Results

To capture the performance of the algorithm as the number of nodes increases, we crawled the target RIA with different number of nodes, from 1 node to 15 nodes.

Figure 3 shows the total time it takes to crawl the RIA as the number of nodes increase (the bar chart) and compares it with the theoretical optimal time to crawl the RIA with multiple nodes (the line chart). The theoretical optimal time it calculated by taking the time it takes to crawl the RIA with one node (T_1), and divide the number by the number of nodes used by the crawler. This theoretical number serves as a base line to measure the efficiency of the crawler. A the figure shows, a good speedup is achieved as the number of nodes increases. The best performance is achieved with 14 nodes.

The performance of the crawler in Figure 3 is better described by breaking down the time into most time consuming operations. Box plots in Figures 4, 5, 6 and 7 show this break down:

– Figure 4: This plot shows the time it takes to load the seed URL into JS-Engine. This plot is interesting in that, this operation is the only operation

[7] `http://www.abeautifulsite.net/blog/2008/03/jquery-file-tree/`

that gets more expensive as the number of crawlers increase. Compared to normal asynchronous calls, the seed URL contains large files and libraries. As the number of crawling nodes increase, the host server disk operation becomes a bottleneck and a jump is observed around node 6.

- Figure 5: This plot shows the time it takes for the coordinator to maintain and update the application graph. This includes adding new states and transitions to the application graph stored in the MySQL database. As expected, this operation is impacted by the number of crawlers.
- Figure 6: This plot shows the time it takes for the coordinator to calculate the closest state from the state of the probing node with un-executed events in it. The time to do this calculation does not vary much and it is often close to 50 milliseconds. The calculation itself is rather fast, and the majority of the 50 milliseconds is spent on retrieving the application graph from the database and constructing the auxiliary structures in the memory. As expected, the figure shows that the measured values are independent of the number of crawlers and are not impacted by it.
- Figure 7: Finally this plot shows the time it takes to execute a single JavaScript event. Based on our calculations, executing JavaScript events is fairly fast when there is no asynchronous call to the server. Asynchronous calls make event execution time substantially longer, and often increase the execution time by two orders of magnitude. At the scale we ran the experiments, the application server is not bottlenecked by executing JavaScript events. Eventually as the number of nodes increases, the application server will become a bottleneck and the time it takes to execute asynchronous requests rises.

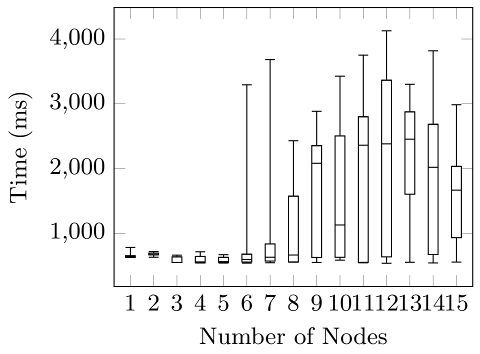

Fig. 4: Time to load the seed URL into JS-Engine.

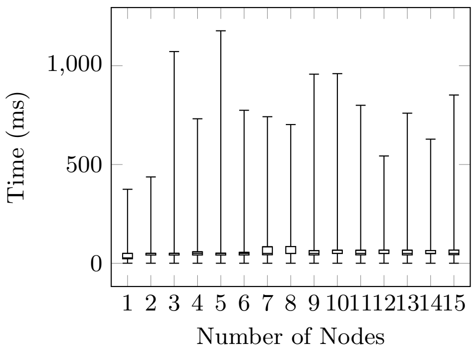

Fig. 5: Time to update application graph.

6.3 Discussion

From the presented break down, it is obvious that the most time consuming operation is loading the seed URL into the JS-Engine. The second most time

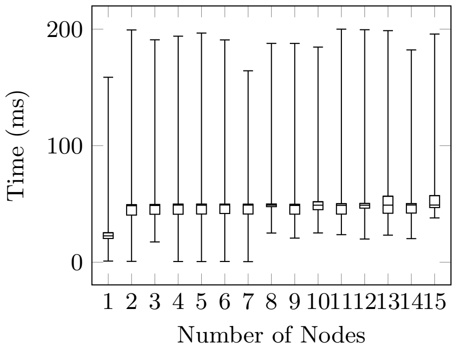

Fig. 6: Time to calculate the next task using the greedy algorithm.

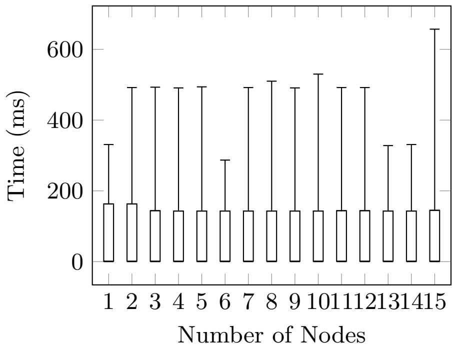

Fig. 7: Time to execute JavaScript events.

consuming operation that happens frequently is executing JavaScript events. Executing a JavaScript event can be particularly time consuming if it involves an asynchronous call to the server.

The design decision of performing the greedy algorithm in a centralized location is inspired by the large discrepancy in the time it takes to find the path greedily and the time it takes to execute the path. As the experiments presented suggests, executing a single asynchronous event can take an order of magnitude longer than calculating the entire shortest path.

At the scale presented in this paper, the coordinator is far from being a bottleneck. As the number of crawling nodes increases, however, the coordinator is bound to become one. In Dist-RIA Crawler [28] nodes uses a deterministic algorithm to autonomously partition the search space and execute JavaScript events in the application. As a future improvement, similar techniques can be used to improve the GDist-RIA crawler by allowing the crawling nodes to autonomously decide (at least partly) the events to execute.

7 Conclusion and Future Improvements

This paper studies distributed crawling of RIAs using a greedy algorithm. A new client-server architecture to dispatch crawling jobs among the crawling nodes, called GDist-RIA Crawler, is introduced. Upon finishing a task, nodes ask the coordinator for the next tasks to do. The coordinator runs the greedy algorithm to assign new task to the probing node, and responds the node with the task. A prototype of the algorithm is implemented and experimental results are provided.

The GDist-RIA Crawler achieves a satisfactory speed up while running the system with up to 15 crawling nodes. This speedup is a result of the low cost of running the greedy search in the application graph at the coordinator, compared to executing the found path by a crawler node. The GDist-RIA Crawler can be improved in many directions, including: Multiple Coordinators to scale better, a peer-to-peer architecture is to shift the greedy algorithm from the coordinator to the crawling nodes, parallelizing other Model-based Crawling strategies (such as probabilistic model or menu model) [4, 5, 12, 15], and Cloud Computing to be more elastic with respect to the resources available and disappearing resources.

Acknowledgements

This work is largely supported by the IBM® Center for Advanced Studies, the IBM Ottawa Lab and the Natural Sciences and Engineering Research Council of Canada (NSERC). A special thank to Sara Baghbanzadeh.

Trademarks

References

1. D. Amalfitano, A. R. Fasolino, and P. Tramontana. Reverse engineering finite state machines from rich internet applications. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 69–73, Washington, DC, USA, 2008. IEEE Computer Society.
2. D. Amalfitano, A. R. Fasolino, and P. Tramontana. Experimenting a reverse engineering technique for modelling the behaviour of rich internet applications. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 571 –574, sept. 2009.
3. D. Amalfitano, A. R. Fasolino, and P. Tramontana. Rich internet application testing using execution trace data. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 274–283, Washington, DC, USA, 2010. IEEE Computer Society.
4. K. Benjamin, G. v. Bochmann, G.-V. Jourdan, and I.-V. Onut. Some modeling challenges when testing rich internet applications for security. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 403–409, Washington, DC, USA, 2010. IEEE Computer Society.
5. K. Benjamin, G. von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I.-V. Onut. A strategy for efficient crawling of rich internet applications. In *ICWE*, pages 74–89, 2011.

6. K. Benjamin, G. Von Bochmann, M. E. Dincturk, G.-V. Jourdan, and I. V. Onut. A strategy for efficient crawling of rich internet applications. In *Proceedings of the 11th international conference on Web engineering*, ICWE'11, pages 74–89, Berlin, Heidelberg, 2011. Springer-Verlag.
7. P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Proc Australian World Wide Web Conference*, 34(8):711–726, 2002.
8. P. Boldi, A. Marino, M. Santini, and S. Vigna. Bubing: Massive crawling for the masses.
9. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.
10. S. Choudhary. M-crawler: Crawling rich internet applications using menu metamodel. Master's thesis, EECS - University of Ottawa, 2012. `http://ssrg.site.uottawa.ca/docs/Surya-Thesis.pdf`.
11. S. Choudhary, E. Dincturk, S. Mirtaheri, G.-V. Jourdan, G. Bochmann, and I. Onut. Building rich internet applications models: Example of a better strategy. In F. Daniel, P. Dolog, and Q. Li, editors, *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 291–305. Springer Berlin Heidelberg, 2013.
12. S. Choudhary, M. E. Dincturk, G. von Bochmann, G.-V. Jourdan, I.-V. Onut, and P. Ionescu. Solving some modeling challenges when testing rich internet applications for security. In *ICST*, pages 850–857, 2012.
13. S. Choudhary, M. E. Dincturk, S. M. M. G. von Bochmann, G.-V. Jourdan, and I.-V. Onut. Crawling rich internet applications: The state of the art. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, Riverton, NJ, USA, 2012. IBM Corp.
14. M. E. Dincturk. Model-based crawling - an approach to design efficient crawling strategies for rich internet applications. Master's thesis, EECS - University of Ottawa, 2013. `http://ssrg.eecs.uottawa.ca/docs/Dincturk_MustafaEmre_2013_thesis.pdf`.
15. M. E. Dincturk, S. Choudhary, G. von Bochmann, G.-V. Jourdan, and I.-V. Onut. A statistical approach for efficient crawling of rich internet applications. In *ICWE*, pages 362–369, 2012.
16. C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. Ajax crawl: Making ajax applications searchable. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 78–89, Washington, DC, USA, 2009. IEEE Computer Society.
17. J. Edwards, K. McCurley, and J. Tomlin. An adaptive model for optimizing performance of an incremental web crawler, 2001.
18. J. Exposto, J. Macedo, A. Pina, A. Alves, and J. Rufino. Geographical partition for distributed web crawling. In *Proceedings of the 2005 workshop on Geographic information retrieval*, GIR '05, pages 55–60, New York, NY, USA, 2005. ACM.
19. G. Frey. Indexing ajax web applications. Master's thesis, ETH Zurich, 2007. `http://e-collection.library.ethz.ch/eserv/eth:30111/eth-30111-01.pdf`.
20. A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2:219–229, 1999.
21. J. Li, B. Loo, J. Hellerstein, M. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. *Peer-to-Peer Systems II*, pages 207–215, 2003.

22. J. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 815–825. ACM, 2013.
23. A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 121–130, Washington, DC, USA, 2008. IEEE Computer Society.
24. R. Matter. Ajax crawl: Making ajax applications searchable. Master's thesis, ETH Zurich, 2008. `http://e-collection.library.ethz.ch/eserv/eth:30709/eth-30709-01.pdf`.
25. A. Mesbah, E. Bozdag, and A. v. Deursen. Crawling ajax by inferring user interface state changes. In *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ICWE '08, pages 122–134, Washington, DC, USA, 2008. IEEE Computer Society.
26. A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *TWEB*, 6(1):3, 2012.
27. A. Milani Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, page 10 pages. IEEE Computer Society, 2013.
28. S. M. Mirtaheri, D. Zou, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut. Dist-ria crawler: A distributed crawler for rich internet applications. In *In Proc. 8TH INTERNATIONAL CONFERENCE ON P2P, PARALLEL, GRID, CLOUD AND INTERNET COMPUTING*, 2013.
29. Z. Peng, N. He, C. Jiang, Z. Li, L. Xu, Y. Li, and Y. Ren. Graph-based ajax crawl: Mining data from rich internet applications. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on*, volume 3, pages 590 –594, march 2012.
30. V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *In Proc. of the Int. Conf. on Data Engineering*, pages 357–368, 2002.
31. H. tsang Lee, D. Leonard, X. Wang, and D. Loguinov. Irlbot: Scaling to 6 billion pages and beyond, 2008.