Dist-RIA Crawler: A Distributed Crawler for Rich Internet Applications

Seyed M. Mirtaheri, Di Zou, Gregor V. Bochmann, Guy-Vincent Jourdan
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Ontario, Canada
{smirt016,dzou085}@uottawa.ca, {bochmann,gvj}@eecs.uottawa.ca

Iosif Viorel Onut
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, as old as the web itself. 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 makes the problem of crawling RIA more time consuming to the web crawler. To reduce the time to crawl a RIA, this paper presents a new distributed algorithm to crawl a RIA in parallel with multiple computers, called *Dist-RIA Crawler*. Dist-RIA Crawler uses the JavaScript® events in the DOM structure to partition the search space. This paper illustrates a prototype implementation of Dist-RIA Crawler and inspect empirical performance measurements.**

I. Introduction

The security of applications is an important, ongoing, and growing concern. Among the applications needing security, Web-based applications are at the forefront: being on the Internet, they are intrinsically exposed to attacks. Easily produced and rapidly changing, web-based applications are often found at the low end of software engineering standards. So-called *Rich Internet Applications* (RIAs), which execute important parts of the application logic in the browser on the user's side, just make the matter worse by providing new attack vectors and creating much more complex architectures.

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 should execute all events in all 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 DOM state (henceforth simply referred to as STATE) as a vertex and each JavaScript event as an edge, the problem of the parallel crawling of a RIA is mapped to the problem of parallel exploration of a directed graph. The distributed crawler introduced in this paper, called *Dist-RIA Crawler*, achieves parallelism by having all the crawlers go to each vertex, however, each crawler only explores a specific subset of the edges. Together all of the crawlers cover all of the edges in the application graph. The underlying assumption is that, on average, the number of events per STATE is larger than the number of crawlers, thus every crawler will have some events to execute in each STATE. While this is a reasonable assumption in large RIAs, it is not a necessary condition to be met for Dist-RIA Crawler to finish the crawl successfully.

In this paper, we introduce Dist-RIA Crawler and then explain some of the practical aspects and challenges faced in designing it. The contributions of this paper include: a new local partitioning algorithm based on JavaScript events, a distributed architecture for RIA crawling, and a prototype implementation and experimental evaluation of the proposed system.

The rest of this paper is organized as follows: We give an overview of the proposed partitioning algorithm and introduce the system components in Section II. In Section III, we explain the assumptions made in designing Dist-RIA Crawler. In Section IV, we explain the variables and the objects used to represent the state of the crawl internally. In Section V, we elaborate on the algorithm that runs on the nodes. In Section VI, we delve into some implementation issues and evaluate the performance of our prototype. In Section VII, we discuss the related works. Finally, in Section VIII we conclude this paper and propose some future improvements.

II. Overview

Dist-RIA Crawler consists of multiple running processes. Each process only has access to its own memory and processes communicate with each other through message passing. Processes are thus independent of each other and multiple processes can run on the same computer to take advantage of multi-core CPUs, or a single process can run on a single computer.

There are two types of processes in Dist-RIA Crawler. A special process called the *coordinator* is responsible for coordinating the crawl. All other processes are crawler processes, with the ability to visit a URL, emulate and control a virtual web browser. These crawler nodes are henceforth referred to as the *node*s. The processes communicate with each other in a star topology, where the coordinator is at the center of the star. Through communication with the coordinator, the nodes indirectly inform each other about the discovered STATEs.

Initially each node contacts the coordinator and gets two values back from the coordinator. The first value is the total number of nodes that join the crawl, called NUMBEROFNODES. The second value is a unique identifier number, assigned to the node by the coordinator, called NODEID where NODEID $\in [0, \text{NUMBEROFNODES})$.

Using NODEID and NUMBEROFNODES, the set of events that each node is responsible for is locally and deterministically determined, in such a way that every event is executed by one, and only one, node. This task is accomplished by an assignment function called ASSIGN. The ASSIGN function takes a STATE as a parameter, and having NODEID and NUMBEROFNODES as environmental variables, it determines the set of events in the STATE that the node is responsible for. For any STATE, the union of all events assigned by the ASSIGN function to different nodes is equal to the set of all events in the STATE (Thus every event in every STATE will be executed). Also, for any STATE, the intersection of events assigned by the ASSIGN function to two distinct nodes is empty (Thus there will not be any duplication of work).

Ranges and *Strides* can be used to create a simple ASSIGN function. Let us deterministically order the set of JavaScript events in the page and assign each event a number called *offset*. In the case of strides, for instance, the ASSIGN function allocates all events whose offset is of the form: $\forall i \in \mathbb{N} : \text{NUMBEROFNODES} \times i + \text{NODEID}$ to a node whose identifier number is NODEID. A good ASSIGN function allocates JavaScript events uniformly among the nodes and thus prevents a subset of nodes from becoming a bottleneck. We use strides to construct the ASSIGN function and we leave more sophisticated ASSIGN functions, such as those based on *hashes*, for future studies.

A node is responsible for executing all the events assigned to it by the ASSIGN function for all STATEs. Every time a node learns about a new STATE, it runs the ASSIGN function to detect the set of JavaScript events it is responsible for. This set of events is added to a list structure called UNEXECUTEDEVENTS in the STATE under investigation. The STATE is then added to the WORKINGSTATES list and it will be explored eventually. If the execution of an event leads to the discovery of a new STATE, the node informs the coordinator, and the coordinator eventually propagates the discovery to the other nodes. Once a node executes all its events and has no more events to execute, it probes the coordinator for the list of STATEs newly discovered by other nodes. Eventually every discovered STATE will become common knowledge.

III. DESIGN ASSUMPTIONS

To facilitate deployment over the internet of the proposed distributed crawling system, and to ensure its effectiveness for security testing, Dist-RIA Crawler makes the following assumptions about the environment and the target RIA it crawls:

A. Environment

Reliability of nodes and communication channels is assumed. To facilitate the deployment of Dist-RIA Crawler over different firewall settings, it is also assumed that only the coordinator has a reachable IP address. In other words, the coordinator is an HTTP server and all other crawler nodes are HTTP clients. As a consequence, the coordinator has no means to contact the nodes and should it have a message for a node, it must wait until it is contacted by that node and respond with the message.

B. Target RIA

To automate security assessment of a RIA, the crawler has to find all STATEs and examine each STATE for security vulnerabilities.

The web crawler often do not have access to the state of the server, and can only capture the client side state of the application. Because the crawler can not capture the state of the server, performing a specific event may lead to two different STATEs. Thus even though a web application is deterministic at whole, because the crawler only has access to the client side state of the application, and has no access to the state of the server, the RIA may seem to be non-deterministic to the crawler.

Finding all STATEs of an unknown non-deterministic RIA is not feasible[1]. Dist-RIA Crawler only targets deterministic finite RIAs. More formally, Dist-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.

Given that the target RIA is deterministic and finite, Dist-RIA Crawler assumes that there exists a finite set of events, and each STATE is reachable by executing one of these events from another STATE. Open fields such as text boxes present a challenge to this assumption. Assigning meaningful data to open fields has been the topic of extensive research in the field of *deep-web crawling* [2]–[6]. Dist-RIA Crawler uses a finite dictionary to assign values for open fields. Taking advantage from existing algorithms in deep-web crawling is left for future studies.

IV. OBJECTS AND STATES

This section describes objects that are used to store the state of the crawl.

A. Application State

Application states are represented by STATE objects. STATE object in a node has the following attributes:

- STATEID: Hash of the DOM that uniquely identifies the STATE.
- PARENTSTATE: A STATE through which this STATE is reachable.
- PARENTEVENT: The event to be executed in order to reach this STATE from PARENTSTATE.
- EVENTS: An ordered list of events in the STATE. Although the node is not responsible to execute all of events in this list, having all events in STATE is

[1]Duda et. al. [1] suggest limiting the number of JavaScript events executed to avoid explosion of STATEs. The suggested technique achieves a partial crawl of non-deterministic RIA in a finite time.

required because in future the node may learn about a new STATE from the coordinator, and to reach that STATE it has to execute one of the events in the EVENTS list.

- UNEXECUTEDEVENTS: A list of the events that are to be executed by the node on the STATE. ASSIGN function is used to populate this list by taking a subset of EVENTS. Thus this list is disjoint from the lists on other states.

B. Node State

Each node stores the following variables and objects:

- COORDINATORADDRESS: The coordinator's address.
- SEEDURL: The initial RIA's URL that is immutable across different STATEs of the RIA. A new URL is treated as a separate RIA.
- NUMBEROFNODES: The total number of nodes.
- NODEID: A unique identifier allocated to the node by the coordinator.
- DISCOVEREDSTATES: The list of STATEs that the node knows about.
- WORKINGSTATES: The list of STATEs that have events in their UNEXECUTEDEVENTS list. This list is initialized with the SEEDURL STATE.
- CURRENTSTATE: The current working STATE.
- NODESTATUS: Represent the status of the node and has one of the following values (Figure 1):
  - DISCONNECTED: The node is not initialized.
  - ACTIVE: The node has been initialized and has work to do.
  - DONE: The node has finished executing all its events and does not have any more work to do. This state marks temporary local termination.
  - TERMINATED: Crawling is finished and the node can leave the system. This state marks the global termination.

Initially the COORDINATORADDRESS is the only global constant and common knowledge. Having the COORDINATORADDRESS, the node initializes itself by retrieving the NODEID, NUMBEROFNODES and SEEDURL from the coordinator. It then proceeds with the crawling algorithm described in Section V.

C. Coordinator State

The coordinator stores the following variables and objects:

- NUMBEROFNODES: The number of nodes. The coordinator has advance knowledge of this number and based on that synchronizes all node at the beginning of the crawling.
- STATELIST$_{nodeID}$: The coordinator keeps track of the STATEs that each node knows about through this list. In this list there is an element per node and each element is an array of STATEIDs of the STATEs that the corresponding node is aware of.
- NODESTATUSLIST: Stores the NODESTATUS of each node.

V. DISTRIBUTED CRAWLING ALGORITHM

A. Coordination Protocol

Nodes start at DISCONNECTED state. Initially each node sends a message, called GETCREDENTIALSFROMCOORDINATOR, to the coordinator asking for its NODEID, the NUMBEROFNODES, and the SEEDURL. The coordinator synchronizes the nodes by waiting to get NUMBEROFNODES requests before responding. Upon arrival of NUMBEROFNODES messages, the coordinator responds to all nodes with the requested information. After arrival of the the coordinator respond, nodes go to the ACTIVE state and crawling begins.

During the crawling, each node contacts the coordinator when it discovers a new STATE, or when it has no more work to do. More formally, the messages sent to the coordinator during the crawling algorithm are:

- SENDNEWSTATETOCOORDINATOR: The node sends a newly discovered STATE to the coordinator.
- GETNEWSTATESFROMCOORDINATOR: The node asks the coordinator for newly discovered STATEs by other nodes.
- SENDNODESTATUSTOCOORDINATOR: The node sends its NODESTATUS to the coordinator.

Through SENDNEWSTATETOCOORDINATOR and GETNEWSTATESFROMCOORDINATOR nodes indirectly propagate discovered STATEs to each other. Upon getting a request from a node, the coordinator responds with one of the following messages:

- STATES: A set of newly discovered STATEs by other nodes, if any. Should a node receive newly discovered STATEs from the coordinator, it will add them to its WORKINGSTATES and DISCOVEREDSTATES. Once a node executes all the events that it is responsible for, it goes into the DONE state, and it informs the coordinator.
- TERMINATED order: Marks the global termination, which means that all nodes are in the DONE state, thus the crawl is over.
- STAY order: The crawl is not over, however, there are no new STATEs for the node. Thus the node stays in the DONE status.

We next describe in more details the crawling algorithm as it runs on the nodes.

B. Local Crawling Algorithm

Algorithm 1 describes the crawling algorithm run locally on each node. The node iteratively executes events and removes STATEs with no event to execute from its WORKINGSTATES list. When the list becomes empty, the node moves to the DONE state. At this point, it either goes back to the ACTIVE state if more work becomes available, or it goes to the TERMINATED state if no more work is available globally.

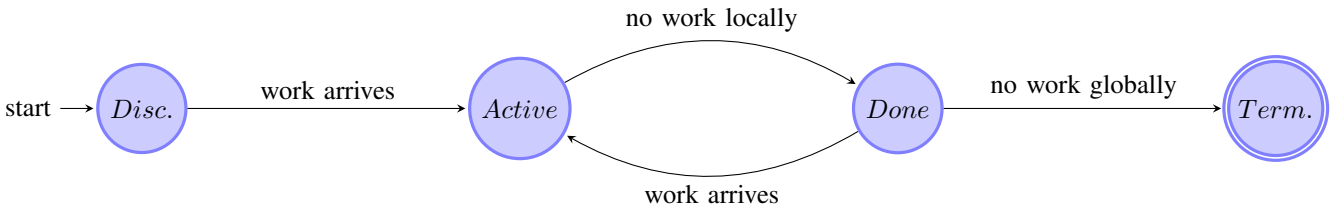

Fig. 1: The Node Status state diagram.

Algorithm 1 Crawling Algorithm (As Executed at Each Node)

```
GETCREDENTIALSFROMCOORDINATOR()
NODESTATUS ← ACTIVE
while (NODESTATUS is not TERMINATED) do
    if WORKINGSTATES is Empty then
        GETNEWSTATESFROMCOORDINATOR()
        if WORKINGSTATES is Empty then
            NODESTATUS ← DONE
            SENDNODESTATUSTOCOORDINATOR()
        else
            NODESTATUS ← ACTIVE
        end if
    else
        stateToVisit ← PICKSTATE(WORKINGSTATES)
        eventToExecute ← PICKUNEXECUTEDEVENT(stateToVisit)
        EXECUTEEVENT(stateToVisit, eventToExecute)
        if CURRENTSTATE is not in DISCOVEREDSTATES then
            push CURRENTSTATE to DISCOVEREDSTATES
            push ASSIGN(CURRENTSTATE) to CURRENTSTATE.UNEXECUTEDEVENTS
            push CURRENTSTATE to WORKINGSTATES
            SENDNEWSTATETOCOORDINATOR( CURRENTSTATE )
        end if
        stateToVisit.REMOVEUNEXECUTEDEVENT(eventToExecute)
        if stateToVisit.UNEXECUTEDEVENTS is empty then
            WORKINGSTATES.REMOVESTATE(stateToVisit)
        end if
    end if
end while
```

The crawling algorithm invokes a set of procedures. These procedures are explained below.

- PICKSTATE: Picks a STATE from the list. This procedure can be used to implement different crawling strategies. Some simple examples are:
  - LIFO: A *Last-In-First-Out* order of picking results in a Dept First Search (DFS) strategy within the scope available to the node.
  - FIFO: A *First-In-First-Out* order of picking results in a Breath First Search (BFS) strategy within the scope available to the node.
- PICKUNEXECUTEDEVENT: Chooses an event to be executed. In the case of a full crawl, it is a good practice to choose events that minimize the overall crawling time. In the case of a partial crawl, it is a good practice to choose events that increase the chances of discovering new STATEs earlier in the crawl [7]. The so-called *Menu* and *Statistical* model-based crawling algorithms [8], [9] address this issue and attempt to find as many STATEs early in the crawl. In this paper, we choose the first unexecuted event and leave more elaborate approaches to future studies.
- EXECUTEEVENT: Executes the event specified. Should the execution of an event lead to a new STATE, this method would add the newly discovered STATE to WORKINGSTATES and DISCOVEREDSTATES lists.
  Should the event to execute does not reside on a different STATE than CURRENTSTATE, the crawler first have to get to the STATE.
  This can be done by visiting the SEEDURL and executing a chain of events that leads to the target STATE. As a future improvement, this method can be optimized by searching the current known graph of the application to find a shorter path from the CURRENTSTATE to the target STATE [8], [9].
- ASSIGN(STATE): This is the ASSIGN function, which finds the events the node is responsible for and adds them to STATE's UNEXECUTEDEVENTS.

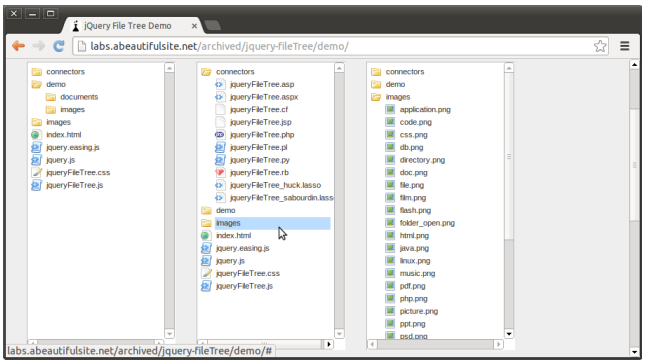

Fig. 2: File tree browser RIA screen-shot

- REMOVEUNEXECUTEDEVENT: Removes an event from the STATE's UNEXECUTEDEVENTS list.

VI. IMPLEMENTATION AND EVALUATION

To evaluate the Dist-RIA Crawler architecture, a prototype of the system was implemented. [2] The Coordinator is implemented in PHP 5.2.10 and MySQL 5.0.77, and runs on an Apache web-server hosted on a Linux® Kernel 2.6 operating system. The coordinator process runs on a machine with an Intel® Xeon® CPU E7330 @ 2.40GHz and 4GB of RAM.

The crawler nodes are implemented in C#.NET using the .NET 4 framework. V8-engine is used to emulate a browser with the capability to run JavaScript events. Each crawler process runs on the Windows® 7 Enterprise operating system hosted on a separate machine with an Intel Core 2 Due and 1GB of RAM.

The nodes and the coordinator communicate using the HTTP protocol over TCP channels using a 10G-bps local area network.

A. Test-Application[3]

A jQuery-based AJAX file browser library[4] (Figure 2) is used to construct the test-applications by applying the browser to the file folders. To avoid explosion of STATEs, we disabled the caching mechanism included in the library, and configured the application so that it only shows one open sub-folder at any given time. In this test-application there is one STATE for each sub-folder in the given file folder. The number of transitions from a STATE depends on its location and dept of the open sub-folder in the file hierarchy. The test-applications are two file folders that contain the source code of two open source projects (Table I).

B. Results and Discussion

This section presents the experimental results of crawling the test-applications using our prototype. The experiments measure the efficiency of the Dist-RIA Crawler in harnessing the computational power available to it. We measure the effect of increasing the number of crawler nodes on the time it takes to crawl a given RIA. We further capture the time spent to execute JavaScript events, the network delay, the time spent in the coordinator, and time wasted while being idle. Each test-application is crawled in 15 settings with 1 to 15 nodes. We ran each experiment three times and the presented results are the average of these three runs.

Figure 3 shows the time it takes to crawl the test-applications in parallel using different number of crawling nodes, and shows the break-down of the time in each case. As the figure shows Dist-RIA Crawler is more effective in the larger test-application compared to the smaller one. In the case of the smaller test-application, on average 76.38 percent of the total to crawl was spent executing JavaScript events, and 19.77 percent of it was spent being idle. Whereas, in the case of the larger test-application, on average 85.31 percent of the total time was spent executing JavaScript events, and only 11.34 percent of it spent being idle. In both cases, network delay and the time spent at the coordinator are minimal, and as the number of crawler nodes increases a satisfactory speedup is observed.

Along with the measured time, we also depict the time it takes to crawl each test-bed with one node, divided by the number of nodes used to run the experiment. This number is used to present the optimal hypothetical case: If it takes T_1 seconds to crawl a RIA with one node, one expects that n nodes will take take at least T_1/n seconds to crawl the same RIA. We call this expected number the theoretical *optimal time* which is shown on the chart as a line.

Optimal time is only meaningful if the CPU is the bottleneck. In case of the larger test-application, a better than optimal speedup is observed in Figure 3: by increasing the number of nodes from one to three we achieve speed-up of more than three. This speedup is achieved since the processing power is not the bottleneck when crawling the larger application with one node and two nodes, but memory swapping is. This effect disappears with a higher number of nodes as the given RAM suffices. To eliminate the effect of memory swapping, in the case of the larger test-application the optimal time is extrapolated based on the time it takes to crawl the application with three nodes.

The main challenge for scalability is the idle time. This is the time that a subset of nodes have nothing to do and are waiting for other nodes to do their work. Figure 4 shows the idle times of crawling test-applications with 15 nodes. As explained above, each experiment was repeated three times and each of the three bars for each node in this chart represent the idle time for that node in one of the runs. As both Figures 3 and 4 show, idle times are relatively for the smaller test-application. More specifically, in the smaller test-application node number 3 is the bottleneck in all runs with 15 nodes, whereas the larger test-application enjoy a more equal distribution of idle times. The use of a strict-stride based ASSIGN function partially explains this discrepancy. This function may make a node bottleneck by assigning a larger number of time-consuming events to it. This problem is application-specific and smaller applications are more susceptible to it. The use of more randomized ASSIGN functions (e.g. hash-based), and the deployment of load balancing algorithm can alleviate this problem.

[2] Similar to [10], only the JavaScript events that are triggered directly as a result user interaction with the RIA are executed.

[3] http://ssrg.eecs.uottawa.ca/papers/DistRIA-3PGCIC-2013/testbeds.tar

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

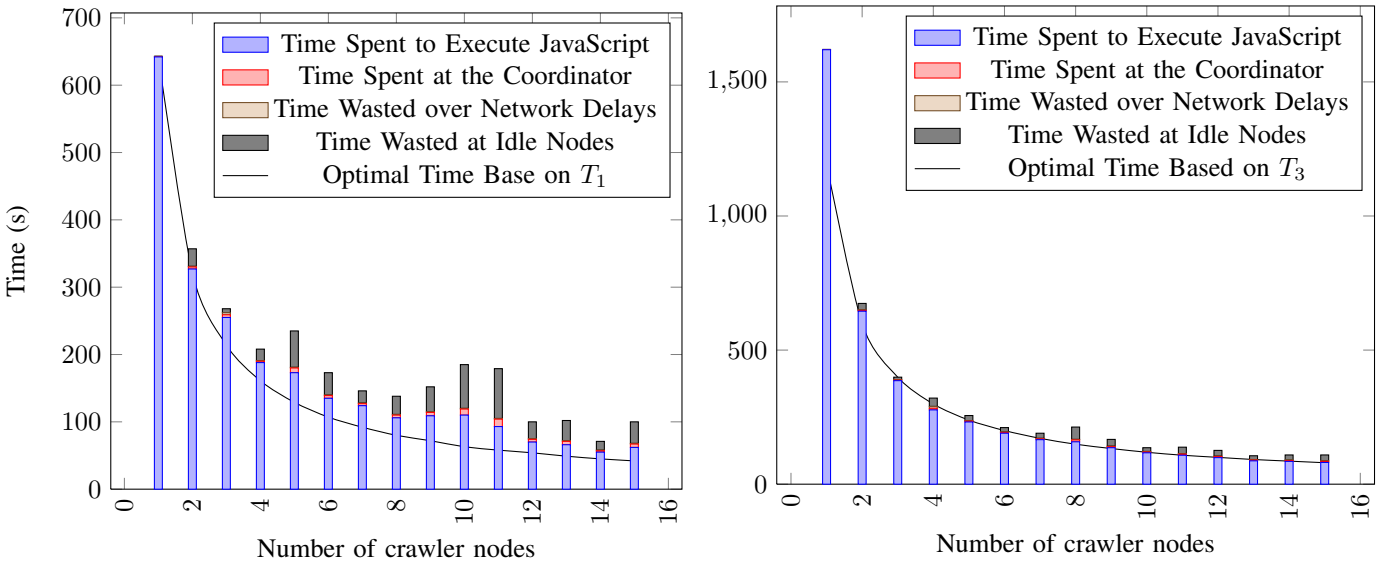

Fig. 3: Time to crawl a RIA with multiple nodes: Apache HTTPD source code file browser (left), and Apache Cassandra source code file browser (right).

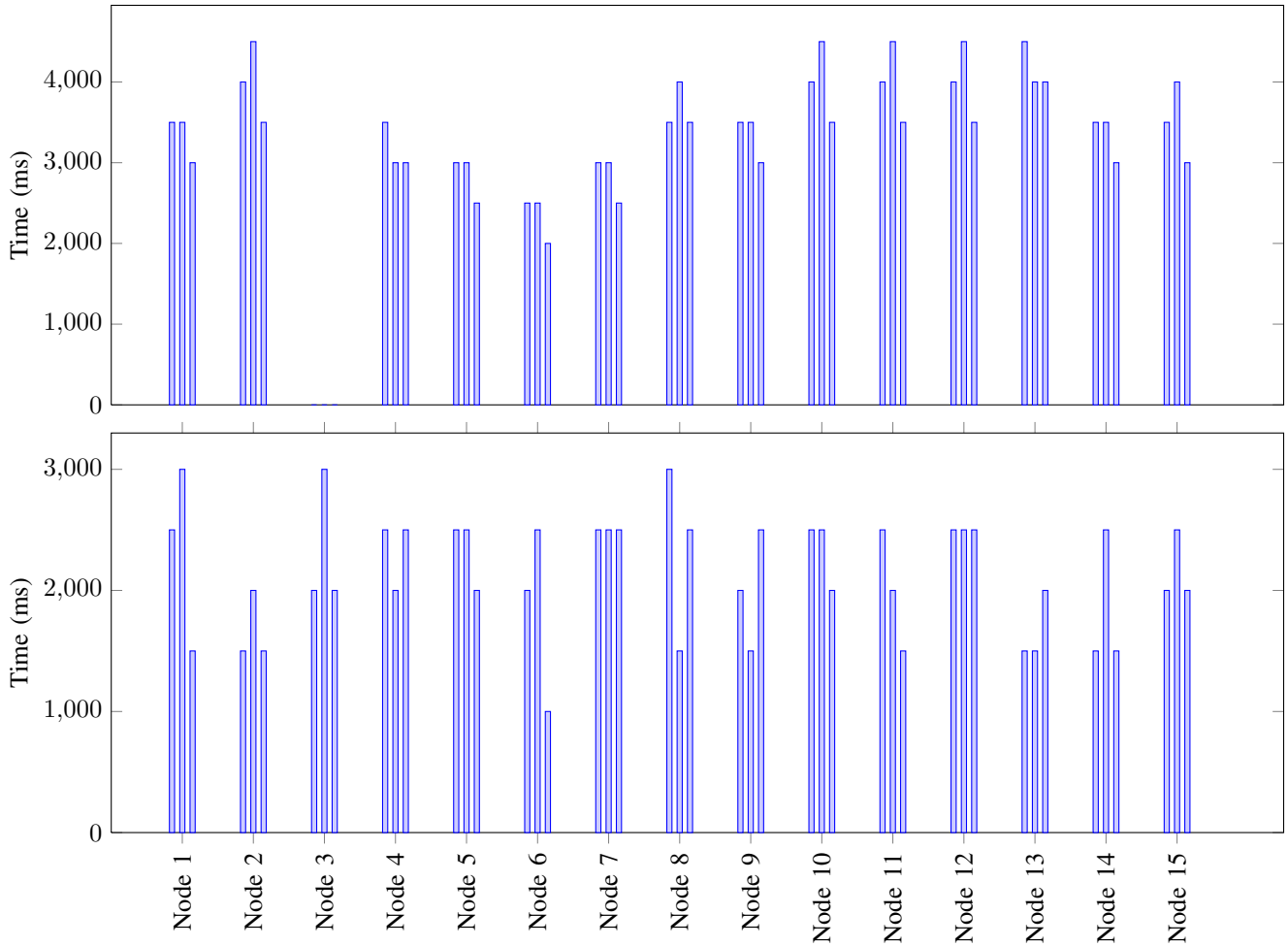

Fig. 4: Idle time distribution during parallel crawl of AJAX file browser with 15 nodes: Apache HTTPD (upper figure), and Apache Cassandra (lower figure) source codes.

Browsed Source Code	States (Nodes)	Transitions (Edges)	Average Number of Edges per Node	Total JavaScript Events Executed
Apache HTTPD 2.4.3	91	2,293	25.197	7,461
Apache Cassandra 1.2.1	163	4,816	29.546	27,149

TABLE I: AJAX file browser testbeds

VII. Related Works

The problems of crawling a RIA have been studied in recent years. Amalfitano et al. [11] describe techniques and tools for testing a RIA based on passively analyzing log-files and reconstituting the user sessions. Mesbah et al. describe a method to crawl RIA applications through simulating user interactions by identifying JavaScript events from the user interface and firing them to reach different states of the application [10], [12]–[14]. Duda et al. [1] describe a method for testing a RIA that contains AJAX calls to the server. The described algorithm is based on a Breadth-First-Search (BFS), and is optimized to avoid repeated execution of the same AJAX calls, similar to [7]. Different assumptions about the structure of the web applications lead to different models for crawling. A survey paper on model-based crawling was written by Choudhary et al. [9]. Benjamin et al. studied the *hypercube* model of crawling in which the assumption is that the order of executing events does not matter [7], [15]. Dincturk et al. [16] studied a statistical approach in choosing the order of events. Other works in this area include [17] and [18]. To the best of our knowledge, however none of these studies target distributed crawling of RIAs.

Distributed crawling of traditional web applications has been described in the literature extensively [19]. Loo et al. [20] describe distributed web crawling where a large number of crawling engines share the task of crawling the Internet. The distribution of the tasks of crawling different URLs is performed by hashing the URL (either only the host-name part, or the entire URL) and distributing the resulting hash values to the different crawlers, for instance, using a distributed hash table (DHT). Exposto et al. [21] also include geographic information about the crawlers and the searched servers into the task distribution algorithm in order to allocate a crawler that is close to the server to be crawled. Boldi et al. [22], in the paper on the UbiCrawler, shows how the so-called consistent hashing approach can be used to allocate the tasks to different crawlers in such a way that there are only minimal changes when some crawler disappears or new crawlers come in. This approach can be used to obtain better fault tolerance. These works mostly deal with the crawling of the traditional web applications based on a page URL. To the best of our knowledge none of the studies above handles distributed crawling of all of the application STATES associated with a single URL.

VIII. Conclusion and Future Improvements

This paper considers distributed crawling of RIAs. A new load partitioning algorithm is proposed and a system for distributed crawling of RIAs, called Dist-RIA Crawler, is introduced. Dist-RIA Crawler partitions the crawling task based on the JavaScript events of a given page by assigning different subsets of events to different crawling nodes. A special node called the coordinator is used to control the system and to distribute discovered application states among the crawling nodes, as well as to perform load balancing. A prototype of the system is implemented and evaluated with up to 15 nodes. For large RIAs, the implemented prototype demonstrate satisfactory speed up.

Dist-RIA Crawler can be improved in several directions:

- **Load Balancing**: A load-balancing technique, that dynamically changes its behaviour based on the infrastructure available to the system, can improve this work and reduce the idle times. This is particularly useful if on average the number of events in each page is lower than the number of crawler nodes.
- **Model-based Crawling**: Dist-RIA Crawler is based on BFS. BFS does not always choose the best event to execute next to discover STATEs early and minimize the time it takes to finish the crawl. Model-based crawling can be used to achieve these goals by choosing events in different orders [7], [8], [15], [16].
- **Cloud Computing**: The proposed architecture is not elastic with respect to the available resources. Support for dynamic working nodes allows the system to take advantage of the appearing nodes, and deals with leaving nodes.
- **Fault Tolerance**: The current implementation does not consider the possibility of failing nodes and thus every failing of a node leads to a failure of the system. Check points can be integrated to make the system more resilient to failures.

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 Mustafa Emre Dincturk.

REFERENCES

[1] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou, "Ajax crawl: Making ajax applications searchable," in *ICDE*, pp. 78–89, 2009.

[2] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, (San Francisco, CA, USA), pp. 129–138, Morgan Kaufmann Publishers Inc., 2001.

[3] S. W. Liddle, D. W. Embley, D. T. Scott, and S. H. Yau1, "Extracting Data behind Web Forms," *Lecture Notes in Computer Science*, vol. 2784, pp. 402–413, Jan. 2003.

[4] L. Barbosa and J. Freire, "Siphoning hidden-web data through keyword-based interfaces," in *In SBBD*, pp. 309–321, 2004.

[5] A. Ntoulas, "Downloading textual hidden web content through keyword queries," in *In JCDL*, pp. 100–109, 2005.

[6] J. Lu, Y. Wang, J. Liang, J. Chen, and J. Liu, "An Approach to Deep Web Crawling by Sampling," *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, vol. 1, pp. 718–724, 2008.

[7] K. Benjamin, G. von Bochmann, M. E. D. Dincturk, G.-V. Jourdan, and I.-V. Onut, "A strategy for efficient crawling of rich internet applications," in *ICWE*, pp. 74–89, 2011.

[8] 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*, pp. 362–369, 2012.

[9] 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), IBM Corp., 2012.

[10] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 210–220, IEEE Computer Society, 2009.

[11] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "Techniques and tools for rich internet applications testing," in *WSE*, pp. 63–72, 2010.

[12] D. Roest, A. Mesbah, and A. van Deursen, "Regression testing ajax applications: Coping with dynamism," in *ICST*, pp. 127–136, 2010.

[13] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 35–53, 2012.

[14] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, pp. 3:1–3:30, Mar. 2012.

[15] 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, (Washington, DC, USA), pp. 403–409, IEEE Computer Society, 2010.

[16] 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*, pp. 850–857, 2012.

[17] 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, (Washington, DC, USA), pp. 121–130, IEEE Computer Society, 2008.

[18] A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing web applications by modeling with fsms," *Software and Systems Modeling*, vol. 4, pp. 326–345, 2005.

[19] C. Olston and M. Najork, "Web crawling," *Foundations and Trends in Information Retrieval*, vol. 4, no. 3, pp. 175–246, 2010.

[20] B. T. Loo, S. Krishnamurthy, and O. Cooper, "Distributed web crawling over dhts," Tech. Rep. UCB/CSD-04-1305, EECS Department, University of California, Berkeley, 2004.

[21] T. Vazão, M. M. Freire, and I. Chong, eds., *Information Networking. Towards Ubiquitous Networking and Services, International Conference, ICOIN 2007, Estoril, Portugal, January 23-25, 2007. Revised Selected Papers*, vol. 5200 of *Lecture Notes in Computer Science*, Springer, 2008.

[22] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Proc Australian World Wide Web Conference*, vol. 34, no. 8, pp. 711–726, 2002.