Distributed Crawling of Rich Internet Applications

Seyed M. Mir Taheri

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfilment of the requirements
for the Doctorate in Philosophy degree in Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Web crawlers visit internet applications, collect data, and learn about new web pages from visited pages. Web crawlers have a long and interesting history. Quick expansion of the web, and the complexity added to web applications have made the process of crawling a very challenging one. Different solutions have been proposed to reduce the time and cost of crawling.

New generation of web applications, known as Rich Internet Applications (RIAs), pose major challenges to the web crawlers. RIAs shift a portion of the computation to the client side. Shifting a portion of the application to the client browser influences the web crawler in two ways: First, the one-to-one correlation between the URL and the state of the application, that exists in traditional web applications, is broken. Second, reaching a state of the application is no longer a simple operation of navigating to the target URL, but often means navigating to a seed URL and executing a chain of events from it. Due to these challenges, crawling a RIA can take a prohibitively long time.

This thesis studies applying distributed computing and parallel processing principles to the field of RIA crawling to reduce the time. We propose different algorithms to concurrently crawl a RIA over several nodes. The proposed algorithms are used as a building block to construct a distributed crawler of RIAs. The different algorithms proposed represent different trade-offs between communication and computation.

This thesis explores the effect of making different trade-offs and their effect on the time it takes to crawl RIAs. We study the cost of running a distributed RIA crawl with client-server architecture and compare it with a peer-to-peer architecture. We further study distribution of different crawling strategies, namely: Breath-First search, Depth-First search, Greedy algorithm, and Probabilistic algorithm.

To measure the effect of different design decisions in practice, a prototype of each algorithm is implemented. The implemented prototypes are used to obtain empirical performance measurements and to refine the algorithms. The ultimate refined algorithm is used for experimentation with a wide range of applications under different circumstances.

This thesis finally includes two theoretical studies of load balancing algorithms and distributed component-based crawling and sets the stage for future studies.

Acknowledgements

I am grateful to many individuals for lending me their expertise and valuable time during my PhD program.

First and foremost, I wish to express my sincere gratitude to my supervisors Professor Gregor V. Bochmann and Dr. Viorel Iosif Onut. Their deep understanding of the subject, constant support from the initial draft of the thesis proposal to the final version of the thesis, and valuable technical and non-technical advices enabled me to develop a deep understanding of the subject.

I wish to thank Professor Guy-Vincent Jourdan for his industrial and academic expertise, and countless discussion sessions guided me and enabled this project to reach fruition.

I wish to thank the love of my life Ava Ahadipour for her endless patience, help and for believing in me.

Many regards to Dr. Kirsten C. Uszkalo, Anseok Jo, Jeff Sember, Mustafa Emre Dincturk, Di Zou, Ali Moosavi, Salman Hooshmand, Sara Baghbanzade, Alireza Farid Amin, Suryakant Choudhary, Khaled Ben Hafaiedh and Bo Wan for being part of this journey.

I would like to acknowledge that this work was in part supported financially by *IBM Center for Advanced Studies (CAS)* and *Natural Sciences and Engineering Research Council of Canada (NSERC).*

A very special thank to my family for their love and support.

Contents

List of Figures

Chapter 1

Introduction

A web application is a computer application that uses a web browser to interact with user. Crawling is the process of exploring the web applications automatically. A web crawler is a tool that performs crawling. The web crawler aims at discovering the web pages of a web application by navigating through the application. The web crawler usually simulates the possible user interactions and can only observe the client-side of the application. The web crawler can be used to automate testing and indexing of the application.

In the literature on web-crawling, a web crawler is basically a software that starts from a set of URLs and downloads all the web pages associated with these URLs. These initial URLs are called *Seed URLs*. After fetching a web page associated with a URL, the URL is removed from the working queue. The web crawler then parses the downloaded page, extracts the linked URLs that are included, and adds these new URLs to the list of seed URLs. This process continues iteratively until all the contents reachable from the seed URLs are reached, or at least one of the resources available exhausts.

The traditional definition of a web crawler assumes that all the content of a web application is reachable through URLs. Soon in the history of web crawling it became clear that such web crawlers cannot deal with the complexities added by interactive web applications that rely on the user input to generate web pages. This scenario often arises when the web application is an interface to a database and it relies on user input to retrieve contents from the database. The new field of *Deep Web-Crawling* was born to address this issue.

Availability of powerful client-side web-browsers, as well as the wide adaptation to technologies such as HTML5 and AJAX, gave birth to a new pattern in designing web applications called *Rich Internet Application* (RIA). RIAs move part of the computation from the server to the client. This new pattern of designing web applications led to complex client side applications that increased the speed and interactivity of the web application, while it reduced the network traffic per request.

1.1 Motivations for Crawling

There are several important motivations for crawling. The chief application of a web crawler is to fetch contents of web application. These contents are then indexed and used by search engines. As the amount of information on the web has been increasing drastically, web users increasingly rely on search engines to find desired data. In order for search engines to learn about the new data as it becomes available on the web, the web crawler has to constantly crawl and update the search engine database. The main three motivations for crawling are:

- Content indexing for search engines. Every search engine requires a web crawler to fetch the data from the web.

- Automated testing and model checking of the web application

- Automated security testing and vulnerability assessment. Many web applications use sensitive data and provide critical services. To address the security concerns for web applications, many commercial and open-source automated web application security scanners have been developed. These tools aim at detecting possible issues, such as security vulnerabilities and usability issues, in an automated and efficient manner[10, 40]. They require a web crawler to discover the states of the application scanned.

1.2 Challenges Faced

Despite many added values, RIAs introduced some unique challenges to web crawlers. In a RIA, user interaction often results in execution of client side *events*. Execution of an event in a RIA often changes the state of the web application on the client side, which is represented in the form of a tree with an application programming interface called *Document Object Model* (DOM)[77]. DOM both represents the structure of the application in the web browser and define methods to manipulate this structure. Henceforth, in this thesis we use DOM exclusively to refer to the structure and not the methods. Events that change the state of DOM does not necessarily change the URL. Traditional web crawlers rely heavily on the URL and changes to the DOM that do not alter the URL are invisible to them. Although deep web crawling increased the ability of the web crawlers to retrieve data from web applications, it fails to address changes to DOM that do not affect the URL. The new and recent field of *RIA web-crawling* attempts to address the problem of RIA crawling.

In a traditional web application a page is identified by a URL and has only a single state per URL. A transition between two states of the application is defined by a link that contains a URL which specifies the target state [96]. Therefore there is a one-to-one correspondence between the states of the application and its URLs.

The basic issues of crawling RIAs are well explained in [85]. In these applications a client-side page, usually associated with a single URL, often contains executable code that may change the state of the application that is seen by the user. This state is stored within the browser, and is represented and accessed through 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 of application contents it should execute all the events in all of the reachable application states. To do so, the crawler starts from the initial DOM executing events one at the time, and monitoring the state of the DOM. Upon discovery of a new DOM, it adds the newly discovered DOM to its working queue and marks the DOM events for future execution. 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. The basic problem with this approach is that the execution of all events can take a very long time and it is the intention of this thesis to reduce it.

1.3 Solution Proposed

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 as a vertex and each client-side event as an edge, the problem of the parallel crawling of a RIA is mapped to the problem of parallel directed graph exploration. This thesis proposes several algorithms to accomplish this task and evaluates their merit experimentally. We then explain some of the practical aspects and challenges faced in designing a distributed RIA crawler.

In the context of RIA web crawling, a *task* is a unit of work to be performed by a crawler node. In the context of RIA crawling, a task is the responsibility to execute one client side event in a certain application state. In order for the crawler to perform the task, it first has to reach the target application state. Thus a task comprises a path from a seed URL to the target application state as well as the event to be executed in that state.

This thesis provides the following contributions:

- New partitioning algorithms based on client-side events: To crawl a RIA in parallel, it is essential to execute all client-side events, without work duplication. This thesis introduces a concept called *partitioning algorithm*. The partitioning algorithm enables the nodes to crawl the RIA autonomously and independent of each other, while all client-side events are executed and no work is duplicated.

- Distributed client-server architectures for RIA crawling: A distributed client-server architecture was introduced to crawl RIAs in parallel, called *Dist-RIA*. In the Dist-RIA architecture nodes act autonomously deciding tasks to do. This architecture works with breath-first search strategy and requires a low network bandwidth.

- Distributed job-dispatching architectures for RIA crawling: A distributed job-dispatching architecture was introduced to crawl RIAs in parallel, called *Client-Server*. In the client-server architecture a single node calculates the tasks to do and dispatches them to the crawling nodes. The architecture can accommodate any crawling strategy including the probabilistic and greedy strategies.

- Distributed peer-to-peer architectures for RIA crawling: A distributed peer-to-peer architecture to crawl RIAs in parallel is introduced. Similar to the client-server architecture, the peer-to-peer architecture can accommodate any crawling strategy including the probabilistic and greedy strategies.

- Implementation and experimental evaluation of the proposed systems. The client-server and the peer-to-peer architectures are compared against each other using breath-first and greedy search strategies. Four strategies of breath-first, depth-first, greedy and probabilistic search strategies, are incorporated into the superior architecture. The four distributed strategies are studied across a set of RIAs to measure the performance and scalability of the proposed architecture. Among the crawling strategies, the greedy and probabilistic strategies are the superior strategies.

- Load balancing mechanisms to alleviate bottlenecks: Experiments with the proposed architectures prove the need for balancing the workload among the nodes. This thesis discusses two load-balancing algorithms introduced in the literature of web-crawling and explains how to utilize them in the context of RIA crawling. It also introduced two new load-balancing algorithms.

- Parallelization of component based crawling strategy: Component-based crawling is a recent and promising strategy to crawl RIAs. This strategy detects independent sections of the DOM (called *widgets*) and crawl each widget independent of other widgets. The final contribution of this thesis is how to distribute component-based strategy and run it in parallel.

1.4 Publications and Patents

This thesis has contributed to the following publications:

- SeyedM. Mirtaheri, GregorV. Bochmann, Guy-Vincent Jourdan, and IosifViorel Onut. *PDist-RIA Crawler: A Peer-to-Peer Distributed Crawler for Rich Internet Applications*, volume 8787 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014

- S. Choudhary, E. Dincturk, Seyed Mirtaheri, G. v. Bochmann, G.-V. Jourdan, and V. Onut. Model-based rich internet applications crawling: “menu” and “probability” models. In *Journal of Web Engineering*, volume 13, pages 243 – 262, 2014

- SeyedM. Mirtaheri, Gregor von Bochmann, Guy-Vincent Jourdan, and IosifViorel Onut. Gdist-ria crawler: A greedy distributed crawler for rich internet applications. In Guevara Noubir and Michel Raynal, editors, *Networked Systems*, Lecture Notes in Computer Science, pages 200–214. Springer International Publishing, 2014

- Seyed M Mirtaheri, Mustafa Emre Dinçtürk, Salman Hooshmand, Gregor V Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. A brief history of web crawlers. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 40–54. IBM Corp., 2013

- Seyed M Mirtaheri, Di Zou, Gregor V Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. Dist-ria crawler: A distributed crawler for rich internet applications. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pages 105–112. IEEE, 2013
- Suryakant Choudhary, Emre Dincturk, Seyed. Mirtaheri, Guy-Vincent Jourdan, Gregor. Bochmann, and Iosif Onut. Building rich internet applications models: Example of a better strategy. In Florian Daniel, Peter Dolog, and Qing Li, editors, *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 291–305. Springer Berlin Heidelberg, 2013
- Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M. Mirtaheri Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel 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

This thesis, also, contributed to the following patent applications:

- I.V. Onut, K.A. Ayoub, P. Ionescu, G.v. Bochmann, G.V. Jourdan, M.E Dincturk, and S.M. Mirtaheri. Representation of an element in a page via an identifier. [Patent]
- Jourdan-G.-V. Bochmann G.v. Mirtaheri S.M. Onut, I.V. A method of partitioning the crawling space of a rich internet application for distributed crawling, 2012. [Patent]
- Brake-N. Ionescu P. Smith D. Dincturk M.E. Mirtaheri S.M. Jourdan G.-V. Bochmann G.v. Onut, I.V. A method of identifying equivalent javascript events on a page, 2012. [Patent]

1.5 Outline of the thesis

The rest of this thesis is organized as follows:

- In Chapter 2, we state some of the challenges faced to design a crawler for RIAs. In this Chapter we also specify the scope of the project and the subset of problems that we are trying to solve.

- In Chapter 3, we summarize the history of parallel web-crawling, the current state of art, and the related works. Using the surveyed literature we show the taxonomy of web crawlers. In this chapter, we also give a brief introduction to distributed systems.

- In Chapter 4, a client-server communication protocol is introduced. This protocol is used to construct Dist-RIA architecture. Dist-RIA architecture parallelizes the breath-first search crawling strategy. It uses a primitive partitioning algorithm to partition the tasks among the nodes locally and autonomously, and only broadcasts the knowledge of states among the working nodes.

- In Chapter 5, some implementation improvements are proposed. The proposed changes are due to our experiments with Dist-RIA architecture.

- In Chapter 6, two more crawling architectures, client-server and peer-to-peer architectures, are introduced:

  - In the client-server architecture a centralized unit calculates the tasks and assign them to working nodes. In this architecture only the centralized unit has the knowledge of transitions and the states and the architecture results in a minimum network traffic.

  - In the peer-to-peer architecture a Peer-to-Peer communication protocol is used. Nodes in the peer-to-peer architecture first constructs an efficient Spanning Tree and then uses it to broadcast the knowledge of states and transitions.

- In Chapter 7, the peer-to-peer and the client-server architectures are used to crawl a set of six RIAs. Performance benchmarks of the architectures are shown in this chapter, and the efficiency of the architectures are explained.

- Finally, in Chapter 8, we conclude this thesis and talk about some of the areas of future work in the topic of distributed RIA Crawling.

This thesis also contains the following appendices:

- In Appendix A, load-balancing algorithms in the context of RIA crawling are introduced. In this appendix, two load-balancing algorithms for distributed crawling of traditional web applications in the literature are shown. These two models are adopted to RIA crawling. Additionally, two new models of task distribution are introduced.

- In Appendix B, a distributed component-based crawling algorithm to parallelized component-based crawling strategy is introduced. Implementation of the introduced concepts in this appendix, and experimenting with them are left for the future studies.

- In Appendix C, experimental results for cost of discovering application states using Breath-First and Greedy search strategies are presented.

- Finally, in Appendix D, experimental results for cost of discovering application states using Peer-to-Peer architecture are presented.

Chapter 2

Problem Statement and Scope of the Project

If an application state is considered to represent a vertex, and a client-side event considered to represent a transition between two vertices or a self-loop, a web application can be modelled as a directed graph. Using this technique the *World Wide Web* can also be modelled as a forest of such graphs. Thus, the problem of Web crawling is reduced to the problem of discovering all the vertices in this forest.

Web crawlers have always been challenged by the large size of the applications they crawl and tried to reduce the time it takes to crawl them. Crawling literature is rich in addressing efficiency and scalability in the context of traditional internet applications (i.e. the web applications where there is a one to one relation between the state of the application and its URL). The efficiency and scalability of crawling RIAs however is largely missing from the web crawling literature. This thesis targets reducing the time it takes to crawl RIAs and thus increasing its scalability through parallel processing and distributed computing.

2.1 Problem Definition

A web application can be modelled as a directed graph, and the *World Wide Web* can be modelled as a forest of such graphs. The problem of Web crawling is the problem of discovering all the nodes in this forest. In the application graph, each node represents a state of the application and each edge a transition from one state to another.

Web crawling problem is studied extensively in the literature. This thesis concentrates on two aspects of the problem:

- Crawling Rich Internet Applications (RIAs): Crawling traditional web applications, where there is a one-to-one correspondance between the state of the application and its URL is well studied. Crawling RIAs, on the other hand, is a recent and nascent problem. RIAs take advantage of client side events. These events introduce partial state update on the client side of the application; potentially without changing the URL. Therefore, in a RIA an application state may not be reachable directly. This complication factor makes the cost of reaching an application state in a RIA potentially higher.

- Distributed Web Crawling: This thesis explores utilizing parallel processing techniques to reduce the time it takes to crawl a RIA. Recently, several studies considered crawling RIAs sequentially. This thesis studies different architectures and algorithms to parallelize some of these sequential algorithms.

2.2 Constraints and Scope

Any solution to the problem of distributed RIA crawling, can be broken down into smaller subproblems. These subproblems represent the design decisions that have to be made before designing the crawler. Some of the design decisions we considered are:

- *Crawling Strategy* : There are two ways to reduce the time it takes to crawl a RIA: Using smart algorithms through crawling strategy and using multiple nodes to achieve parallelism. In recent years, *Model Based Crawling* (MBC) strategy has been studied extensively[13, 14, 31, 37]. Prior to these recent works, only depth-first-search, breath-first-search, and greedy crawling strategies have been described in the literature[17, 102]. Our team introduced a few new models including:

  - The Hypercube model[13, 36]: This model takes advantage of event ordering and when the ordering does not matter it creates an efficient model of the application. Based on the created model, it crawls the application efficiently.
  - The greedy strategy[102]: This strategy always find the closest un-executed event and executes it.
  - The probabilistic strategy[36, 38]: This strategy keeps statistics about the events, and gives priority to the events that leads to the discovery of new states.

  - The menu model[29]: This model assumes that an event always lead to the same target state. In the other words, if the same event is seen in two different states, irrelevant of the source states, both events lead to the same target state. This assumption is used to discover new states as early as possible during the crawl.

  Empirical measurements indicates that among these models, the greedy and the probabilistic models are the superior ones[31, 36]. These two models often surpass other models both in discovering states as early as possible, and in reducing the overall time of crawl. Thus, this thesis only focuses on these two models. We also study breath-first and depth-first strategies as the base case.

  In addition to the above-mentioned strategies, recently Moosavi[93] introduced component-based MBC. This model can detect client widgets and the independent parts in the client heuristically. By detecting this independence, the crawler requires to execute substantially less events in order to crawl the website in comparison to any other strategy. To set the stage for future works in this area, an abstract algorithm to parallelize this strategy is introduced too.

- *The extent and frequency of sharing information*: An important aspect of the parallel algorithm to crawl RIAs is the extent of sharing information among the crawling nodes. At one end of the spectrum, crawling nodes could share minimal amount of information among one another, such as the tasks to be done, or the discovered states. At the other end of spectrum, nodes could share all of the information they discover, including the transitions among the nodes. Another important aspect is the frequency of sharing information among the nodes. Nodes can share newly discovered information as soon as they become available, or they can do it in certain intervals. The extent and the frequency of sharing information effects the performance of the algorithm. This thesis inspects the effect of these two parameters in the performance of the web crawler.

- *Mainframe* vs *Commodity Hardware*: In designing a parallel algorithm to crawl RIAs, one can take advantage of mainframes and other expensive equipments. The other less costly alternative is to break down the computation over a set of cheap commodity computers. This thesis takes advantage of the latter approach and is designed to run on commodity computers.

- *System Characteristics*: Any communication protocol makes certain assumptions about the underlying communication infrastructure and the working nodes. This thesis assumes that:
  - Underlying communication medium is reliable and there are no messages lost.
  - Computational units are reliable and they do not die unexpectedly.
- *Scale*: There are assumptions about the size of the application concerning the number of states in the application and the number of transitions among those states, as well as the number of available computational resources. This thesis assumes that on average there are more transitions per state than the number of available computational units. In other words, this thesis addresses the problem of crawling a dense application over a small number of computers.

2.2.1 Assumptions about target RIA

We make two assumptions about the target RIA:

Deterministic RIAs

From the point of view of the crawler, a RIA can be deterministic or non-deterministic. The web crawler often does not have access to the state of the server, and can only capture the client side state of the application. Because the crawler cannot capture the state of the server, performing a specific event may lead to different application states depending on the server state. Thus even though a web application is deterministic as a 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 application states of an unknown non-deterministic RIA is not feasible. Duda et. al.[41] 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. The purpose of this thesis is to achieve full coverage of a given RIA in finite time, thus we only target deterministic finite RIAs.

We define a deterministic application as follows:*A RIA is deterministic if and only if given a seed URL and a chain of JavaScript events, by loading the seed URL and executing the chain of events sequentially the crawler always lands on the same target state.*

Given that the target RIA is deterministic and finite, we assume that there exists a finite set of events, and each application state is reachable by executing one of these events from some application state. In this model, it is assumed that all application states are reachable from the seed URL. 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*[8, 70, 73, 94, 105]. Our crawler uses a finite dictionary to assign values to open fields. Taking advantage of existing algorithms in assigning values to open fields is the topic of study in deep-web crawling and is beyond the scope of this thesis.

Client-side Events

We assume absence of external events. Newly introduced in HTML5 are *Web Sockets*[46]. These sockets allow the server to contact the web client. These special events are triggered by the environment and interfere with the assumption of deterministic RIA made by the crawler.

The functionality of the Web Sockets is achievable in HTML4 through *Comet*[110]. Comet withholds the HTTP request for a period of time. If during this time a message becomes available for the client, the server responds with the message to the client. If the period expires, the server returns the HTTP request and asks the client to send another request. In this model a never ending dialog continues between client and the server that allows to pass messages to the client in real time.

Existence of Web Sockets or Comet does not necessarily make an application indeterministic. Assuming that the crawler can capture these events, and that given these events deterministically change the state of the application, these events can be simply modelled as external events. These two assumptions, however, are hard to assert and verify. This thesis, thus, assumes that the target RIA does not use these functionalities. In other words, this thesis assumes that all events are triggered directly by the user, and no external events interfere with the state of the application.

In addition to the absence of external events, we only focus on JavaScript events and leave other client side events such as Flash events to the future studies.

2.3 Issues to be addressed

In addition to the design decisions states earlier, two more issues remain to be addressed:

- **Study relative cost and trade-offs to be made to adapt different search strategies**: As explained in Section 2.2, this thesis intends to address distributed crawling of RIAs with breath-first, depth-first, the greedy and the probabilistic crawling strategies. While the first two algorithms are easy to implement, the second two algorithms requires the knowledge of all transitions among different states, in order to calculate the next step in the algorithm. This thesis measures the cost of transferring such data among the nodes and shows the results of the trade-offs made.

- **Design a series of partitioning and load-balancing algorithms**: There are different approaches to distribute workload among the nodes. On one side of the spectrum, a static distribution allows each node to determine its share of the work autonomously and locally. On the other side of the spectrum, nodes can completely rely on a centralized unit to assign them tasks and do not participate in the process of assignment at all. This thesis studies these two approaches as well as another hybrid approach in between the two, and an adaptive distribution approach.

Chapter 3

Literature Review

This chapter reviews the relevant literature to designing a distributed RIA web crawler. It then, briefly reviews relevant sectors of distributed computing. Terminology and evaluation criteria defined in this chapter are used in the rest of the thesis.

3.1 Web Crawlers

In the literature on web-crawling, a web crawler is basically software that starts from a set of seed URLs, and downloads all the web pages associated with these URLs. After fetching a web page associated with a URL, the URL is removed from the working queue. The web crawler then parses the downloaded page, extracts the linked URLs from it, and adds to the list of seed URLs. This process continues iteratively until all of the contents reachable from seed URLs are reached.

The traditional definition of a web crawler assumes that all the content of a web application is reachable through URLs. Soon in the history of web crawling it became clear that such web crawlers cannot deal with the complexities added by interactive web applications that rely on user input to generate web pages. This scenario often arises when the web application is an interface to a database and it relies on user input to retrieve contents from the database. The new field of *Deep Web-Crawling* was born to address this issue.

Availability of powerful client-side web-browsers, as well as the wide adaptation to technologies such as HTML5 and AJAX, gave birth to a new pattern in designing web applications called *Rich Internet Application* (RIA). RIAs move part of the computation from the server to the client. This new pattern led to complex client side applications that increased the speed and interactivity of the application, while reducing the network traffic per request.

Despite the added values, RIAs introduced some unique challenges to web crawlers. In a RIA, user interaction often results in execution of client side *events*. Execution of an event in a RIA often changes the state of the web application on the client side, which is represented in the form of a *Document Object Model* (DOM)[77]. This change in the state of DOM does not necessarily mean changing the URL. Traditional web crawlers rely heavily on the URL and changes to the DOM that do not alter the URL are invisible to them. Although deep web crawling increased the ability of the web crawlers to retrieve data from web applications, it fails to address changes to DOM that do not affect the URL. The new and recent field of *RIA web-crawling* attempts to address the problem of RIA crawling.

3.1.1 Evolution of Web Crawling Problem Definition

As web applications evolved, the definition of the state of the application evolved as well. In the context of traditional web applications, states in the application graph are pages with distinct URLs and edges are hyperlinks between pages i.e. there exists an edge between two nodes in the graph if there exist a link between the two pages. In the context of deep web crawling, transitions are constructed based on users input. This is in contrast with hyperlink transitions which always redirect the application to the same target page. In a deep web application, any action that causes submission of a form is a possible edge in the graph.

In RIAs, the assumption that pages are nodes in the graph is not valid, since the client side code can change the application state without changing the page URL. Therefore nodes here are application states denoted by their DOM, and edges are not restricted to forms that submit elements, since each element can communicate with the server and partially update the current state. Edges, in this context, are client side actions (e.g. in JavaScript) assigned to DOM elements and can be detected by web crawler. Unlike the traditional web applications where jumps to arbitrary states are possible, in a RIA, the execution of sequence of events from the current state or from a seed URL is required to reach a particular state.

The three models can be unified by defining the state of the application based on the state of the DOM as well as other parameters such as the page URL, rather than the URL or the DOM alone. A hyperlink in a traditional web application does not

Table 3.1: Different categories of web crawlers

Category	**Input**	**Application graph components**
Traditional	Set of seed URLs	Vertices are pages with distinct URL and a directed edge exist from page p_1 to page p_2 if there is a hyperlink in page p_1 that points to page p_2
Deep	Set of Seed URLs, user context specific data, domain taxonomy	Vertices are pages and a directed edge exists between page p_1 to page p_2 if submitting a form in page p_1 gets the user to page p_2.
RIA	A starting page	Vertices are DOM states of the application and a directed edge exists from DOM d_1 to DOM d_2 if there is a client-side JavaScript event, detectable by the web crawler, that if triggered on d_1 changes the DOM state to d_2
Unified Model	A seed URL	Vertices are calculated based on DOM and the URL. An edge is a transition between two states triggered through client side events. Redirecting the browser is a special client side event.

only change the page URL, but it also changes the state of the DOM. In this model changing the page URL can be viewed as a special client side event that updates the entire DOM. Similarly, submission of a HTML form in a deep web application leads to a particular state of DOM once the response comes back from the server. In both cases the final DOM states can be used to enumerate the states of the application. Table 3.1 summarizes different categories of web crawlers.

3.1.2 Requirements

Several design goals have been considered for web crawlers. *Coverage* and *freshness* are among the first [96]. Coverage measures the relative number of pages discovered by the web crawler. Ideally given enough time the web crawler has to find all pages and build the complete model of the application. This property is referred to as *Completeness*. Coverage captures the static behaviour of traditional web applications well. It may fail, however, to capture the performance of the web crawler in crawling dynamically created web pages. The search engine index has to be updated constantly to reflect changes in web pages created dynamically. The ability of the web crawler to retrieve latest updates is measured through *freshness*.

An important and old issue in designing web crawlers is called *politeness*[59]. Early web crawlers had no mechanism to stop them from flooding a server with many requests. As a result while crawling a website they could have lunched an inadvertent *Denial of Service*(DoS) attack and exhaust the target server resources to the point that it would interrupt normal operation of the server. Politeness was the concept introduced to put a cap on the number of requests sent to a web-server per unit of time. A polite web crawler avoids launching an inadvertent DoS attack on the target server. Another old problem that web crawlers faced are *traps*. Traps are seemingly large set of websites with arbitrary data that are meant to waste the web crawler resources. Integration of *black-lists* allowed web crawlers to avoid traps. Among the challenges web crawlers faced in the mid 90s was *scalability*[23]. Throughout the history of web-crawling, the exponential growth of the web and its constantly evolving nature has been hard to match by web crawlers. In addition to these requirements, the web crawler's model of application should be *correct* and reflect true content and structure of the application.

In the context of deep-web crawling Raghavan and Garcia-Molina[105] suggest two more requirements. In this context, *Submission efficiency* is defined as the ratio of submitted forms leading to result pages with new data; and *Lenient submission efficiency* measures if a form submission is semantically correct (e.g., submitting a company name as input to a form element that was intended to be an author name)

In the context of RIA crawling a non-functional requirement considered by Kamara et al. [14] is called *efficiency*. Efficiency means discovering valuable information as soon as possible. For example states are more important than transitions and should be found first instead of finding transitions leading to already known states. This is particularly important if the web crawler will perform a partial crawl rather than a full crawl.

This chapter defines web crawling and its requirements, and based on the defined model classifies web crawlers.

A brief history of traditional web crawlers[1], deep web crawlers[2], and RIA crawlers[3] is presented in Sections 3.2, 3.3, and 3.4, respectively. Based on this brief history and the model defined, taxonomy of web crawling is then presented in section 3.5. Section 3.6 presents some open questions and future works in web crawling.

3.2 Crawling Traditional Web Applications

Web crawlers were written as early as 1993. This year gave birth to four web crawlers: *World Wide Web Wanderer*, *Jump Station*, *World Wide Web Worm*[79], and *RBSE spider*. These four spiders mainly collected information and statistic about the web using a set of seed URLs. Early web crawlers iteratively downloaded URLs and updated their repository of URLs through the downloaded web pages.

The next year, 1994, two new web crawlers appeared: *WebCrawler* and *MOMspider*. In addition to collecting stats and data about the state of the web, these two web crawlers introduced concepts of *politeness* and *black-lists* to traditional web crawlers. *WebCrawler* is considered to be the first parallel web crawler by downloading 15 links simultaneously. From *World Wide Web Worm* to *WebCrawler*, the number of indexed pages increased from 110,000 to 2 million. Shortly after, in the coming years a few commercial web crawlers became available: *Lycos*, *Infoseek*, *Excite*, *AltaVista* and *HotBot*.

In 1998, Brin and Page[21] tried to address the issue of scalability by introducing a large scale web crawler called *Google*. Google addressed the problem of scalability in several ways: Firstly it leveraged many low level optimizations to reduce disk access time through techniques such as compression and indexing. Secondly, and on a higher level,

[1] See Olston and Najork[96] for a survey of traditional web crawlers.

[2] See He et al. [56] for a survey of deep web crawlers.

[3] See Choudhary et al. [31] for a survey of RIA crawlers.

Google calculated the probability of a user visiting a page through an algorithm called *PageRank*. PageRange calculates the probability of a user visiting a page by taking into account the number of links that point to the page as well as the style of those links. Having this probability, Google simulated an arbitrary user and visited a page as often as the user did. Such approach optimizes the resources available to the web crawler by reducing the rate at which the web crawler visits unattractive pages. Through this technique, Google achieved high *freshness*. Architecturally, Google used a master-slave architecture with a master server (called *URLServer*) dispatching URLs to a set of slave nodes. The slave nodes retrieve the assigned pages by downloading them from the web. At its peak, the first implementation of Google reached 100 page downloads per second.

The issue of scalability was further addressed by Allan Heydon and Marc Najork in a tool called *Mercator*[59] in 1999. Additionally Mercator attempted to address the problem of extendability of web crawlers. To address extensibility it took advantage of a modular Java-based framework. This architecture allowed third-party components to be integrated into Mercator. To address the problem of scalability, Mercator tried to solve the problem of *URL-Seen*. The URL-Seen problem answers the question of whether or not a URL was seen before. This seemingly trivial problem gets very time-consuming as the size of the URL list grows. Mercator increased the scalability of URL-Seen by batch disk checks. In this mode hashes of discovered URLs got stored in RAM. When the size of these hashes grows beyond a certain limit, the list was compared against the URLs stored on the disk, and the list itself on the disk was updated. Using this technique, the second version of Mercator crawled 891 million pages. Mercator got integrated into *AltaVista* in 2001.

IBM introduced *WebFountain*[43] in 2001. WebFountain was a fully distributed web crawler and its objective was not only to index the web, but also to create a local copy of it. This local copy was *incremental* meaning that a copy of the page was kept indefinitely on the local space, and this copy got updated as often as WebFountain visited the page. In WebFountain, major components such as the scheduler were distributed and the crawling was an ongoing process where the local copy of the web only grew. These features, as well as deployment of efficient technologies such as the *Message Passing Interface* (MPI), made WebFountain a scalable web crawler with high freshness rate. In a simulation, WebFountain managed to scale with a growing web. This simulated web originally had 500 million pages and it grew to twice its size every 400 days.

In 2002, *Polybot*[111] addressed the problem of URL-Seen scalability by enhancing the batch disk check technique. Polybot used a Red-Black tree to keep the URLs and when the tree grows beyond a certain limit, it was merged with a sorted list in main memory. Using this data structure to handle the URL-Seen test, Polybot managed to scan 120 million pages. In the same year, *UbiCrawler*[20] dealt with the problem of URL-Seen with a different, more peer-to-peer (P2P), approach. UbiCrawler used consistent hashing to distribute URLs among web crawler nodes. In this model no centralized unit calculates whether or not a URL was seen before, but when a URL is discovered it is passed to the node responsible to answer the test. The node responsible to do this calculation is detected by taking the hash of the URL and map it to the list of nodes. With five 1GHz PCs and fifty threads, UbiCrawler reached a download rate of 10 million pages per day.

In addition to Polybot and UbiCrawler, in 2002 Tang et al. [115] introduced *pSearch*. pSearch uses two algorithms called *P2P Vector Space Model (pVSM)* and *P2P Latent Semantic Indexing (pLSI)* to crawl the web on a P2P network. VSM and LSI in turn use vector representation to calculate the relation between queries and the documents. Additionally pSearch took advantage of *Distributed Hash Tables* (DHT) routing algorithms to address scalability.

Two other studies used DHTs over P2P networks. In 2003, Li et al. [68] used this technique to scale up certain tasks such as clustering of contents and bloom filters. In 2004, Loo et al. [72] addressed the question of scalability of web crawlers and used the technique to partition URLs among the crawlers. One of the underlying assumptions in this work is the availability of high speed communication medium. The implemented prototype requested 800,000 pages from more than 70,000 web crawlers in 15 minutes.

In 2005, Exposto et al. [45] augmented partitioning of URLs among a set of crawling nodes in a P2P architecture by taking into account servers' geographical location. Such an augmentation reduced the overall time of the crawl by allocating target servers to a node physically closest to them.

In 2008, an extremely scalable web crawler called *IRLbot* ran for 41.27 days on a quad-CPU AMD Opteron 2.6 GHz server and it crawled over 6.38 billion web pages[118]. IRLbot primarily addressed the *URL-Seen* problem by breaking it down into three sub-problems: CHECK, UPDATE and CHECK+UPDATE. To address these sub-problems, IRLbot introduced a framework called *Disk Repository with Update Management* (DRUM). DRUM optimizes disk access by segmenting the disk into several *disk buckets*. For each disk bucket, DRUM also allocates a corresponding bucket on the RAM. Each URL is mapped to a bucket. At first a URL was stored in its RAM bucket. Once a bucket on the RAM is filled, the corresponding disk bucket is accessed in batch mode. This batch mode access, as well as the two-stage bucketing system used, allowed DRUM to store a large number of URLs on the disk such that its performance would not degrade as the number of URLs increases.

3.3 Crawling Deep Web

As server-side programming and scripting languages, such as PHP and ASP, got momentum, more and more databases became accessible online through interacting with a web application. The applications often delegated creation and generation of contents to the executable files using *Common Gateway Interface* (CGI). In this model, programmers often hosted their data on databases and used HTML forms to query them. Thus a web crawler cannot access all of the contents of a web application merely by following hyperlinks and downloading their corresponding web page. These contents are *hidden* from the web crawler point of view and thus are referred to as *deep web*[56].

In 1998, Lawrence and Giles[66] estimated that 80 percent of web contents were hidden in 1998. Later in 2000, BrightPlanet suggested that the deep web contents is 500 times larger than what surfaces through following hyperlinks (referred to as *shallow web*)[16]. The size of the deep web is rapidly growing as more companies are moving their data to databases and set up interfaces for the users to access them[16].

Only a small fraction of the deep web is indexed by search engines. In 2007, He et al. [56] randomly sampled one million IPs and crawled these IPs looking for deep webs through HTML form elements. The study also defined a depth factor from the original seed IP address and constrained itself to depth of three. Among the sampled IPs, 126 deep web sites were found. These deep websites had 406 query gateways to 190 databases. Based on these results with 99 percent confidence interval, the study estimates that at the time of that writing, there existed $1,097,000$ to $1,419,000$ database query gateways on the web. The study further estimated that Google and Yahoo search engines each have visited only 32 percent of the deep web. To make the matters worse the study also estimated that 84 percent of the covered objects overlap between the two search engines, so combining the discovered objects by the two search engines does not increase the percentage of the visited deep web by much.

The second generation of web crawlers took the deep web into account. Information retrieval from the deep web meant interacting with HTML forms. To retrieve information hidden in the deep web, the web crawler would submit the HTML form many times, each time filled with a different dataset. Thus the problem of crawling the deep web got reduced to the problem of assigning proper values to the HTML form fields.

The open and difficult question to answer in designing a deep web crawler is how to meaningfully assign values to the fields in a query form[8]. As Barbosa and Freire[8] explain, it is easy to assign values to fields of certain types such as radio buttons. The difficult field to deal with, however, is text box inputs. Many different proposals tried to answer this question:

- In 2001, Raghavan and Garcia-Molina[105] proposed a method to fill up text box inputs that mostly depend on human output.

- In 2002, Liddle et al. [70] described a method to detect form elements and fabricate a HTTP GET and POST request using default values specified for each field. The proposed algorithm is not fully automated and asks for user input when required.

- In 2004, Barbosa and Freire[8] proposed a two phase algorithm to generate textual queries. The first stage collected a set of data from the website and used that to associate weights to keywords. The second phase used a greedy algorithm to retrieve as much contents as possible with minimum number of queries.

- In 2005, Ntoulas et al. [94] further advanced the process by defining three policies for sending queries to the interface: a random policy, a policy based on the frequency of keywords in a reference document, and an adaptive policy that learns from the downloaded pages. Given four entry points, this study retrieved 90 percent of the deep web with only 100 requests.

- In 2008, Lu et al. [73] map the problem of maximizing the coverage per number of requests to the problem of *set-covering*[32] and uses a classical approach to solve this problem.

3.4 Crawling Rich Internet Applications

Powerful client side browsers and availability of client-side technologies lead to a shift in computation from the server-side to the client-side. This shift of computation, also creates contents that are often hidden from traditional web-crawlers and are referred to as “Client-side hidden-web”[11]. In 2013, Behfarshad and Mesbah studied 500 websites and found that 95 percent of the subject websites contained client-side hidden-web, and among the 95 percent web-sites, 62 percent of the application states are considered client-side hidden-web. Extrapolating these numbers puts almost 59 percent of the web contents at the time of this writing as client-side hidden-web.

RIA crawling differs from traditional web application crawling in several frontiers. Although limited, there has been some research focusing on crawling of RIAs. One of the earliest attempts to crawl RIAs is by Duda et al. [42, 50, 78] in 2007. This work presents a working prototype of a RIA crawler that indexed RIAs using a Breath-First-Search algorithm. In 2008, Mesbah et al.[82, 86] introduced *Crawljax* a RIA crawler that took the user-interface into account and used the changes made to the user interface to direct the crawling strategy. Crawljax aimed at crawling and taking a static snapshot of each AJAX state for indexing and testing. In the same year, Amalfitano et al. [2, 3, 4, 5] addressed automatic testing of RIAs using execution traces obtained from AJAX applications.

This section surveys different aspects of RIA crawling. Different strategies can be used to choose an unexecuted event to execute. Different strategies effect how early the web crawler finds new states and the overall time of crawling. Section 3.4.1 surveys some of the strategies studied in recent years. Section 3.4.2 explains different approaches to determine if two DOMs are equivalent. Section 3.4.3 surveys parallelism and concurrency for RIA crawling. Automated testing and ranking algorithms are explored in Sections 3.4.4 and 3.4.5, respectively.

3.4.1 Crawling Strategy

Until recent years, there has not been much attention on the efficiency requirement, and existing approaches often use either Breadth-First or a Depth-First crawling strategy. Breath-First search was first discussed by Duda et al. [42, 50, 78] in 2007 in a tool called Crawljax. Later improvements on Crawljax enabled it to support different modes of operation[82, 86]. In one variation, Crawljax performs a depth-search algorithm, it stores the history of event execution and only executes an event if the event has not been executed before, regardless of the application state. Another more aggressive mode to operate forces Crawljax to execute all events and in-effect makes Crawljax to perform standard depth-first crawling strategy.

In 2008, Amalfitano et al. [2, 3, 4] studied testing and modelling of RIAs through execution traces. The proposed method acquires user traces manually and constructs a FSM model of the application[2]. *CrawlRIA* improved this work and removed the manual operation to generate the traces[4]. CrawlRIA runs a depth-first search strategy to emulate a user session and generate the traces.

Model-based crawling[14] was first introduced by Benjamin et al. in 2011. In this work, Benjamin et al. introduce a crawling strategy called *Hypercube strategy*. This model assumes that the application model is a hypercube, and based on this assumption chooses the next task to execute. If the application model is not a hypercube, the model deals with the violation of the assumption and adopts to it[36].

In 2012, Dincturk et al. [38] described *Probabilistic Strategy*. The probabilistic strategy takes into account the history of event execution, and based on that chooses an event that maximizes the probability of finding a new application state. In the same year, another strategy called *Greedy Strategy* was described by Peng et al. [102]. This strategy always executes the closest un-executed event. More formally, the algorithm starts a breath-first search on the unknown application graph from the current state. This algorithm differs from standard breath-first search that starts the search from the seed URL. The algorithm continues the search until it finds an un-executed event and then it executes it.

In 2013, Choudhary et al. [28, 29] described the *Menu Strategy*. This model-based crawling strategy assumes that the execution of an event type always leads to the same target state, regardless of the source state[29]. In the same year, Milani Fard and Mesbah[87] introduce *FeedEx*: a greedy algorithm to partially crawl a RIAs. FeedEx differs from Peng et al. [102] in that: Peng et al. [102] use a greedy algorithm in finding the closest unexecuted event, whereas FeedEx defines a matrix to measure the impact of an event and its corresponding state on the crawl. The choices are then sorted and the most impactful choice will be executed first. Given enough time, FeedEx will discover the entire graph of the application.

FeedEx defines the impact matrix as a weighted sum of the following four factors:

- Code coverage impact: how much of the application code is being executed.
- Navigational diversity: how diversely the crawler explores the application graph.
- Page structural diversity: how newly discovered DOMs differ from those already discovered.
- Test model size: the size of the created test model.

In the test cases studied, Milani Fard and Mesbah[87] show that FeedEx beats three other strategies of Breadth-First search, Depth-First search, and random strategy, in the above-mentioned four factors.

3.4.2 DOM Equivalence and Comparison

In the context of traditional web applications it is trivial to determine whether two states are equal: compare their URLs. This problem is not as trivial in the context of RIAs. Different chains of events may lead to the same states where the respective DOMs have only minor differences that do not effect the functionality of the state. Different researchers address this issue differently. Duda et al. [42, 50, 78] present one of the most aggressive approaches based on equality of the DOMS. In the presented papers, Duda et al. consider two states to be equal if the hashes of full DOMs of the two states are equal[50].

Crawljax [82] offers a less strict algorithm to determine the equality of DOMs. Crawljax calculates the edit distance (so-called Levenstein distance) to measure if two DOMs are equivalent. The distance is defined as the number of operations required to convert a DOM to another DOM. If this distance is less than a predefined threshold, two DOMs are considered equivalent.

Levenstein distance is not transitive. Therefore, Crawljax is susceptible to consider DOMs that have edit distance of more than the predefined threshold equivalent. Crawljax was later improved[86] to address this issue. In the new algorithm, a new DOM is considered to be new if its edit distance is more than the predefined threshold from all known states. This method solves the transitivity issue, however, it is expensive computationally, and requires the crawler to store all DOM trees.

Amalfitano et al. [3] describe an approach that ignores the ordering of DOM elements. In this approach, elements of different DOMs are compared against each other individually. If the same elements are found in two DOMs, two DOMs are considered equivalent. Amalfitano et al. also suggest variations of the method by including factors such as element path before considering the two elements equivalent.

In 2013, Lo et al. [71] in a tool called *Imagen*, consider the problem of transferring a JavaScript session between two clients. Imagen improves the definition of client-side state by adding the following items:

- JavaScript functions closure: JavaScript functions can be created dynamically, and their scope is determined at the time of creation.

- JavaScript event listeners: JavaScript allows the programmer to register event-handlers.

- HTML5 elements: Certain elements such as *Opaque Objects* and *Stream Resources*.

These items are not ordinarily stored in the DOM. Imagen uses code instrumentation and other techniques to add the effect of these features to the state of the application. To the best of our knowledge, Imagen offers the most powerful definition of a RIA state at the time of this writing. This definition has not been used by any web crawler yet, and its effect on the web crawler performance is an open research topic.

3.4.3 Parallel Crawling

To the best of our knowledge, prior to this thesis, the only two algorithms are proposed to achieve a degree of concurrency:

- Matter [78], proposed to use multiple web crawlers on RIAs that use hyperlinks together with events for navigation. The suggested method first applies traditional crawling to find the URLs in the application. After traditional crawling terminates, the set of discovered URLs are partitioned and assigned to event-based crawling processes that run independent of each other using their breadth-first strategy. Since each URL is crawled independently, there is no communication between the web crawlers.

  This approach achieves concurrency at the URL level and assumes that the application graph associated with each URL is small and can be crawled with one node. On the contrary, this thesis focuses on a giant application graph associated with a single URL and achieves concurrency at the application state level.

- Crawljax[86] used multiple threads for speeding up event-based crawling of a single URL application. The crawling process starts with a single thread (that uses depth-first strategy). When a thread discovers a state with more than one event, new threads are initiated that will start the exploration from the discovered state and follow one of the unexplored events from there.

  This approach relies on the shared memory among the threads and thus it not scalable. In this thesis we focus on achieving parallelism through independently running processes that communicate through message passing.

3.4.4 Automated Testing[4]

Automated testing of RIAs is an important aspect of RIA crawling. In 2008, Marchetto et al. [76] used a state-based testing approach based on a FSM model of the application. The introduced model construction method used static analysis of the JavaScript code and dynamic analysis of user session traces. Abstraction of the DOM states was used rather than the DOM states directly in order to reduce the size of the model. This optimization may require a certain level of manual interaction to ensure correctness of the algorithm. The introduced model produced test sequences that contained *semantically interacting events*[5]. In 2009, Marchetto and Tonella[75] proposed search-based test sequence generation using hill-climbing rather than exhaustively generating all the sequences up to some maximum length.

In 2009 and 2010, Crawljax introduced three mechanisms to automate testing of RIAs: Using *invariant-based* testing[83], security testing of interactions among web widgets [108], and regression testing of AJAX applications[108].

In 2010, Amalfitono et al. [4] compared the effectiveness of methods based on execution traces (user generated, web crawler generated and combination of the two) and existing test case reduction techniques based on measures such as state coverage, transition coverage and detecting JavaScript faults. In another study[5], authors used invariant-based testing approach to detect faults visible on the user-interface (invalid HTML, broken links, unsatisfied accessibility requirements) in addition to JavaScript faults (crashes) which may not be visible on the user-interface, but cause faulty behaviour.

3.4.5 Ranking

In the context of traditional web crawling, there is an extensive research in ranking different pages[96]. The only effort we are aware of to rank resources in the context of RIA crawling, is done by Frey[50] in 2007. Frey proposes an algorithm called *AjaxRank* to rank different states of RIA. AjaxRank adopts PageRank to RIA crawling by considering DOM states as application states, and transitions as hyperlinks. A connectivity-based graph is constructed and PageRank algorithm is implemented on top of the graph.

[4] For a detailed study of web application testing trends from 2000 to 2011 see Garousic et al. [53]

[5] Two events are semantically interacting if their execution order changes the outcome.

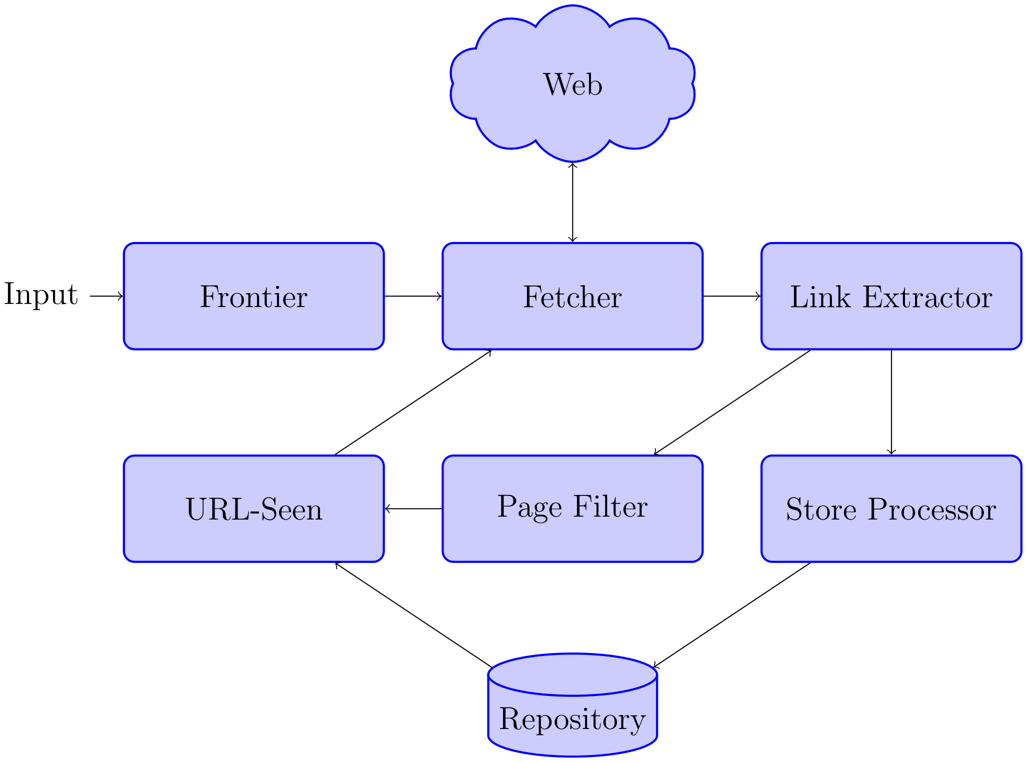


Figure 3.1: Architecture of a traditional web crawler.

3.5 Taxonomy and Evolution of Web Crawlers

The wide variety of web crawlers available are designed with different goals in mind. This section classifies and cross-measures the functionalities of different web crawlers based on the design criteria introduced in Section 3.1.2. It also sketches out a rough architecture of web crawlers as they evolve. Sections 3.5.1, 3.5.2 and 3.5.3 explain the taxonomy of traditional, deep, and RIA web crawlers, respectively.

3.5.1 Traditional Web Crawlers

Figure 3.1 shows the architecture of a typical traditional web crawler. In this model *Frontier* gets a set of seed URLs. The seed URLs are passed to a module called *Fetcher* that retrieves the contents of the pages associated with the URLs from the web. These contents are passed to the *Link Extractor*. The latter parses the HTML pages and extracts new links from them. Newly discovered links are passed to *Page Filter* and *Store Processor*. Store Processor interacts with the database and stores the discovered

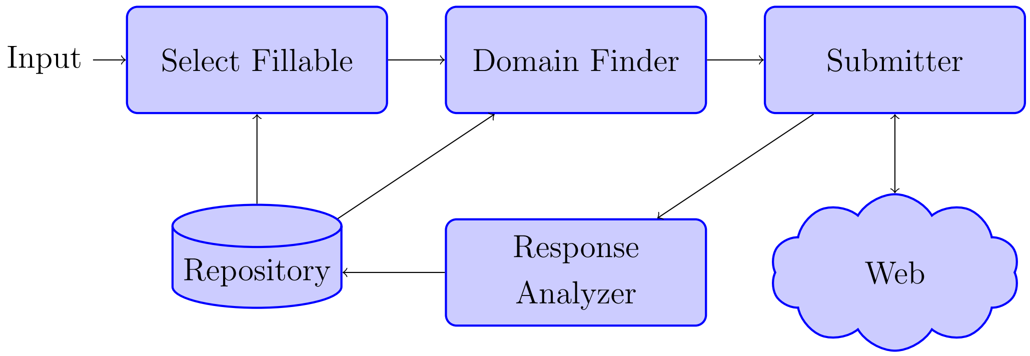


Figure 3.2: Architecture of a deep web crawler.

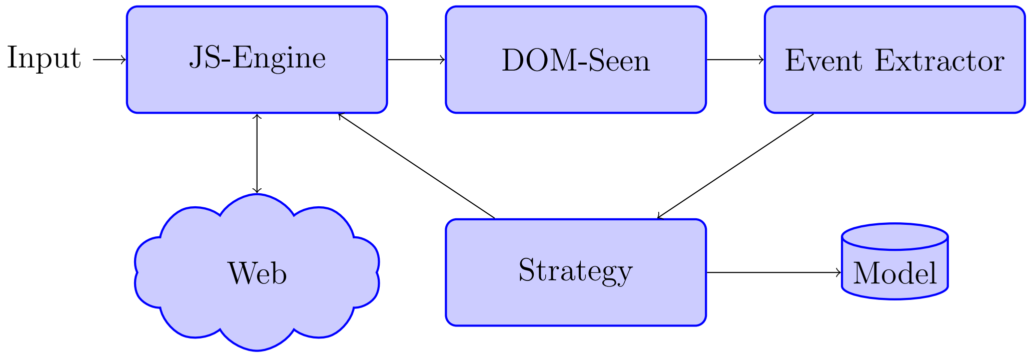


Figure 3.3: Architecture of a RIA web crawler.

Table 3.2: Taxonomy of traditional web crawlers

Study	**Component**	**Method**	**Goal**
WebCrawler, MOMspider[96]	Fetcher, Frontier, Page filter	Parallel downloading of 15 links, robots.txt, Black-list	Scalability, Politeness
Google[21]	Store processor, Frontier	Reduce disk access time by compression, PageRank	Scalability, Coverage, Freshness
Mercator[59]	URL-Seen	Batch disk checks, cache	Scalability
WebFountain[43]	Storage processor, Frontier, Fetch	Local copy of the fetched pages, Adaptive download rate, Homogenous cluster as hardware	Completeness, Freshness, Scalability
Polybot[111]	URL-Seen	Red-Black tree to keep the URLs	Scalability
UbiCrawler[20]	URL-Seen	P2P architecture	Scalability
pSearch[115]	Store processor	Distributed Hashing Tables (DHT)	Scalability
Exposto et al .[45]	Frontier	Distributed Hashing Tables (DHT)	Scalability
IRLbotpages[118]	URL-Seen	Access time reduction by disk segmentation	Scalability

Table 3.3: Taxonomy of deep web crawlers

Study	**Component**	**Method**	**Goal**
HiWe[105]	Select fillable, Domain Finder, Submitter, Response Analyst	Partial page layout and visual adjacency, Normalization by stemming etc, Approximation matching, Manual domain, Ignore submitting small or incomplete forms, Hash of visually important parts of the page to detect errors	Lenient submission efficiency, Submission efficiency
Liddle et al. [70]	Select fillable, Domain Finder	Fields with finite set of values, ignores automatic filling of text field, Stratified Sampling Method (avoid queries biased toward certain fields), Detection of new forms inside result page, Removal of repeated form Concatenation of pages connected through navigational elements, Stop queries by observing pages with repetitive partial results, Detect record boundaries and computes hash values for each sentence	Lenient submission efficiency, Submission efficiency
Barbosa and Freire[8]	Select fillable, Domain Finder, Response Analysis	Single keyword-based queries, Based on collection data associate weights to keywords and uses greedy algorithms to retrieve as much contents with minimum number of queries, Considers adding stop-words to maximize coverage, Issue queries using dummy words to detect error pages	Lenient submission efficiency, Submission efficiency
Ntoulas et al. [94]	Select fillable, Domain Finder	Single-term keyword-based queries, Three policies: random, based on the frequency of keyword in a corpus, and an Adaptive policy that learn from the downloaded pages. maximizing the unique returns of each query	Lenient submission efficiency, Submission efficiency
Lu et al. [73]	Select fillable, Domain Finder	querying textual data sources, Works on sample that represents the original data source, Maximizing the coverage	Lenient submission efficiency, Scalability, Submission efficiency

Table 3.4: Taxonomy of RIA web crawlers

Study	**Component**	**Method**	**Goal**
Duda et al. [42, 50, 78]	Strategy, JS-Engine, DOM-Seen	Breadth-First-Search, Caching the JavaScript function calls and results, Comparing Hash value of the full serialized DOM	Completeness, Efficiency
Mesbah et al. [82, 86]	Strategy, DOM-Seen	Depth-First-Search, Explores an event only once, New threads are initiated for unexplored events, Comparing Edit distance with all previous states	Completeness, State Coverage Efficiency, Scalability
CrawlRIA[2, 3, 4, 5]	Strategy, DOM-Seen	Depth-First strategy (Automatically generated using execution traces), Comparing the set of elements, event types, event handlers in two DOMs	Completeness
Kamara et al. [12, 14]	Strategy	Assuming hypercube model for the application, Using Minimum Chain Decomposition and Minimum Transition Coverage	State Coverage Efficiency
M-Crawler[30]	Strategy	Menu strategy which categorizes events after first two runs, Events which always lead to the same/current state has less priority, Using Rural-Postman solver to explore unexecuted events efficiently	State Coverage Efficiency, Completeness
Peng et al. [102]	Strategy	Choose an event from current state then from the closest state	State Coverage Efficiency
AjaxRank[50]	Strategy, DOM-Seen	The initial state of the URL is given more importance, Similar to PageRank, connectivity-based but instead of hyperlinks the transitions are considered hash value of the content and structure of the DOM	State Coverage Efficiency
Dincturk et al. [38]	Strategy	Considers probability of discovering new state by an event and cost of following the path to event's state	State Coverage Efficiency
Dist-RIA Crawler[89]	Strategy	Uses JavaScript events to partition the search space and run the crawl in parallel on multiple nodes	Scalability
Feedex[87]	Strategy	Prioritize events based on their possible impact of the DOM, Considers factors like code coverage, navigational and page structural diversity	State Coverage Efficiency

links. Page Filter filters URLs that are not interesting to the web crawler. The URLs are then passed to *URL-Seen* module. This module finds the new URLs that are not retrieved yet and passes them to Fetcher for retrieval. This loop continues until all the reachable links are visited.

Table 3.2 summarizes the design components, design goals and different techniques used by traditional web crawlers.

3.5.2 Deep Web Crawlers

Figure 3.2 shows the architecture of a typical deep web crawler. In this model *Select Fillable* gets as input a set of seed URLs, domain data, and user specifics. *Select Fillable* then chooses the HTML elements to interact with. *Domain Finder* uses these data to fill up the HTML forms and passes the results to *Submitter*. Submitter submits the form to the server and retrieves the newly formed page. *Response Analyser* parses the page and, based on the result, updates the repository; and the process continues.

Table 3.3 summarizes the design components, design goals and different techniques used by deep web crawlers.

3.5.3 RIA Web Crawlers

Figure 3.3 shows the architecture of a typical RIA web crawler. *JS-engine* starts a virtual browser and runs a JavaScript engine. It then retrieves the page associated with a seed URL and loads it through the virtual browser. The constructed DOM is passed to the *DOM-Seen* module to determine if this is the first time the DOM is seen. If so, the DOM is passed to *Event Extractor* to extract the JavaScript events from it. The events are then passed to the *Strategy* module. This module decides which event to execute. The chosen event is passed to JS-Engine for further execution. This process continues until all reachable states are seen.

Table 3.4 summarizes the design components, design goals and different techniques used by RIA web crawlers.

3.6 Some Open Questions in Web-Crawling

Traditional web crawling and its scalability has been the topic of extensive research. Similarly, deep-web crawling was addressed in great detail. RIA crawling, however, is a new and open area for research. Some of the open questions in the field of RIA crawling are the following:

- *Model based crawling*: The problem of designing an efficient strategy for crawling a RIA can be mapped to a graph exploration problem. The objective of the algorithm is to visit every node at least once in an unknown directed graph by minimizing the total sum of the weights of the edges traversed. This objective discovers all application states. The offline version of this problem, where the graph is known beforehand, is called the Asymmetric Traveling Salesman Problem (ATSP) which is NP-Hard. Although there are some approximation algorithms for different variations of the unknown graph exploration problem [39, 48, 51, 80], not knowing the graph ahead of the time is a major obstacle to deploy these algorithms to crawl RIAs.

  Model based crawling makes assumptions about the behaviour of the application and the structure of the application graph, and based on the assumptions made it crawl the application. These algorithms have to detect and deal with violations of the assumptions made. At the time of this writing, three major MBC strategies are: the hypercube model, the menu model, and the component-based model. While these three models cover a wide set of web applications, they are by no means sufficient and more work is required to cover a larger portion of the web applications that exist today.

- *Scalability of State-Seen*: Throughout the history of traditional web application crawling, the URL-Seen problem has been one of the biggest challenges. This problem does not exists in the context of RIA crawling since a typical RIA has a small set of URLs that are associated with a large set of application states. Inevitably, however, as RIAs scale a similar problem will present itself in the context of RIA crawling. This related future problem referred to as *State-Seen* problem answers: If an application state was seen before.

  Solutions to this open problem can be inspired from the work done on URL-Seen with one caveat: Comparing two URLs is an easy task, comparing two application states can be very difficult and is the topic of DOM equivalency. Thus, solutions to the state-seen problem requires combining some of the algorithms seen in URL-Seen problem with some of the algorithms seen in DOM equivalency.

- *Widget detection*: In order to avoid state explosion, it is crucial to detect independent parts of the interface in a RIA. This can effect ranking of different states, too.

- *Definition of state in RIA*: some HTML5 elements such as *web sockets* introduce new challenges to the web crawlers. Some of the challenges are addressed by Imagen[71], however many of these challenges remain open.

3.7 Distributed Computing

During early 70s, Organick[100] realized that it is cheaper to harness the power of multiple computers by interconnecting them together, rather than building an expensive super fast computer. Next few decades witnessed an explosion in engineering and deployment of a field in computer science called *distributed systems*, that is made of small inexpensive communicating nodes and achieves large computational power.

Over years, distributed systems has produced a large and fascinating literature, with may different models of distributed computing. Giving a thorough introduction of distributed systems is beyond the scope of this thesis. However some of the distributed systems taxonomy is essential to formally define the target distributed crawler. This section introduces basic models of distributed computing and briefly explain *Cloud Computing* and *Load Balancing*.

3.7.1 Basic Models

Parallel computing has been the direction of hardware and software design in the last two decades. Moore's law is still alive and the number of transistors are increasing exponentially. To accommodate large number of transistors hardware manufacturers adopted *multi-processor* and *multi-core* environments[58].

Distributed algorithms generally fall into three different models:

- **Shared memory**: In this model all processors have access to the same memory[34]. Shared memory systems exist in different scales: they can be as small as a multi-core CPU, or as large as a massive supercomputer with *Remote Direct Memory Access* (RDMA)[24]. It is often challenging to create a large scale Shared-memory system because in distributed systems CPUs are often not synchronized[58].

- **Parallel algorithm**: In this model the programmer can modify the network and computational resources available to set up a parallel environment. Computational nodes communicate with each other not by writing to the same memory space, but by sending messages to each other. Sorting networks are example of this model[32].

- **Distributed algorithms**: In this model, the programmer cannot choose the network structure and the program should work irrelevant of it. This is the most flexible model in harnessing different hardware settings available. In this model, often all nodes are running the same application. An application designed in this model can be modelled and verified by modelling the whole system as a graph, and modelling each node as a finite state machine. Similar to Parallel algorithms, in this model too nodes are communicating with messages.

3.7.2 Cloud Computing

Advancements made in network layer, storage technologies, multi-core CPUs, as well as the rapid growth of data size, prevalence of service computing led to the emergence of a new computing model called *Cloud Computing*[49]. In essence Cloud Computing addresses the question of the location of the infrastructure, and offers business owners cheaper infrastructures by economy of scale[112], through moving their computation to the network[119]. Cloud Computing differs from Grid Computing in its massive scalability and its abstraction of hardware and software by virtualization[49].

In 1961, John McCarthy predicted that computation may eventually be a public utility. Although the idea behind Cloud Computing is rather old, only recently it became widely deployed by industry[101]. The term "Cloud Computing" was first used by Google's CEO to describe Google's new business model[126]. Since then the term is defined in many different ways[119]. The National Institute of Standards and Technology (NIST) defines cloud computing as:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."[81]

By addressing difficult management problems, Cloud Computing quickly became popular: By deploying an application on the cloud a company requires no initial investment and enjoy a pay-as-you-go model. Further, Cloud Computing is an easy and cheap way of out-sourcing hardware and software maintenance. Last but not least, thanks to the wide availability of the Internet, applications deployed on Cloud are accessible all around the world at all time[126].

Cloud Computing heavily relies on vitalization to abstract the computing resources. The virtualization that happens in the Cloud environment is the chief factor that differentiate Cloud Computing from Grid Computing[119]. This factor also rewards Cloud environment with security by isolation and scalability. Virtualization happens on *infrastructure layer*, the layer on top of *hardware layer*, using software such as Xen[9]. The infrastructure layer creates a pool of resources and allows the *platform layer* to run the operating system and application framework on top of it. The last and top most layer of a cloud environment is *application layer*, in which the user applications are running.

3.7.3 Load Balancing in Distributed Systems

Load Balancing is a technique used to increase the performance by distributing the work among the nodes as evenly as possible[69]. A typical load balancing algorithm collects information on items such as the load level at different working units, and based on the collected information may migrate some data. Li and Lan[69] characterize different load balancing algorithms based on three dimensions: Static versus Dynamic, Centralized versus Distributed, and Application-level versus System-Level.

Static versus Dynamic

Static models make decisions about the distribution of the load only once and are not adaptive. Dynamic models on the other hand are constantly adapting and evolving. Traditionally dynamic load balancing algorithms are used in heterogeneous computing environments, and static load balancing algorithms are used in homogeneous computing environments[54].

Altilar and Paker[1] introduce a static partitioning and scheduling algorithm to process video frames. The proposed algorithm is generic and can be applied to any application with expensive independent data processing. Barbosa et al.[7] use linear algebra to statically distribute work in a homogeneous as well as a heterogeneous environment. Proposed algorithm seem to scale well, and be able to take advantage of the availability of a wide range of heterogeneous processing units. Genaud et al.[54] uses a linear programming algorithm and describe a static load balancing algorithm to optimally distribute data among nodes. Described algorithm takes as input network bandwidth as well as processing power of different nodes.

Dynamic load balancing strategies are categorized in three sub-categories[35]: The first approach, called *Recursive Bisection*, is to recursively divide the tasks plane by two until the number of sub-regions is equal to the number of workers[15, 113, 116]. The second approach, called *Space-Filling Curve*, maps and orders task in a one dimensional space. The tasks are cut into weighted pieces and are assigned to the workers[47, 92, 95, 103, 120]. The final approach, called *graph partitioning*, takes advantage of geometric locality and maps the problem of load balancing to the problem of graph partitioning and graph decomposition[22, 33, 57, 60, 64, 104, 113].

Hybrid solutions based on static and dynamic model exist too. Lu and Zomaya[74] propose a hybrid solution by partitioning the grid into different regions and integrating static and dynamic model in each region.

Centralized versus Distributed

Centralized models make decision on a single location, whereas Distributed models enjoy a distributed scheme to make load balancing decisions. Centralized approaches can be very efficient, however in many cases centralized approaches are susceptible not to scale as the centralized controller may become a bottleneck[74].

On the other hand, in a distributed model the communication cost may become prohibitive, if all nodes to know the full state of the system. To deal with potentially high communication cost in distributed models each node may store a partial state of the system. Probabilistic and Diffusion-based approaches can be used to achieve this[6, 74, 124].

Application-Level versus System-Level

Application-Level models target minimizing the task completion time, whereas System-Level models target maximizing the rate of resource utilization. System-Level load balancing can be reduced to distributed scheduling problem which in turn is a distributed constraint satisfaction problem[125]. Application-Level load balancing is the topic of extensive research. Different assumption about the underlying jobs leads to different policies to reduce the application completion time or *minimize its makespan*[69]. If the total completion time of each task is known the problem of load balancing reduces to the problem of scheduling and classical *online scheduling algorithm* find the optimal solution[32].

If the completion of tasks is not known stochastic approaches may be used to balance load among the nodes. Weiss[122] studies distribution of stochastic tasks over parallel machines. Weber[121] studies task distribution in an online fashion. In this model the assignment of jobs to the nodes is not necessarily pre-computed and task can be assigned to the nodes as the old ones finish. Goel and Indyc study load balancing in systems that tasks completion time follow Poisson and exponential distribution[55]. Kleinberg et al. address load balancing in the systems that tasks completion time is arbitrary[65]. Lehtonen looks into scheduling jobs with exponential completion time distribution over identical machines[67].

3.8 Summary

This chapter reviews the area of web crawling for the three kinds of web applications: traditional web applications, deep web applications, and rich internet application. As the web expands, efficient crawling of large web applications remain a challenge. This thesis contributes to the field of web crawling by harnessing the power of parallel processing to reduce the time it takes to crawl large RIAs.

Chapter 4

Dist-RIA: A client-server system architecture to crawl RIAs

In this chapter we introduce *Dist-RIA*[1], a distributed architecture to crawl RIAs using a star topology. We explain some of the practical aspects and challenges we faced in designing the Dist-RIA architecture.

The contributions of this chapter include:

- A new static partitioning algorithm based on JavaScript events.
- A distributed architecture for RIA crawling.
- A prototype implementation and experimental evaluation of Dist-RIA architecture.

The rest of this chapter is organized as follows: A high level view of the architecture is presented in Section 4.1. In Section 4.2, we explain the variables and the objects used to represent the state of the crawl internally. In Section 4.3, we explain the crawling protocol that the nodes and the coordinator use to communicate with each other. In Section 4.4, we delve into some implementation issues and evaluate the performance of our prototype. Finally, in Section 4.5 we conclude this chapter.

4.1 Architecture

There are two types of working components in a Dist-RIA Crawler. A special process called the *coordinator* is responsible for coordinating the communication among the

[1] Dist-RIA was first published in 3PGCIC 2013[89]

nodes. All other processes are working processes, and are henceforth referred to as the *node*s.

Processes communicate with each other in a star topology, where the coordinator is at the centre of the star. Through communication with the coordinator, the nodes indirectly inform each other of the discovered states. Through this communication the coordinator also learns about the load level of each node and the status of the crawl. Initially every node is assigned a unique identifier number by the coordinator, denoted by i as was explained in Section 4.2. The coordinator always initializes the nodes with i, propagates the knowledge of discovered states, and handles termination.

4.2 Objects and States

Application State object is used to represent an application state. This object is used to represent a state internally by the nodes and the coordinator. It is also used by the coordinator to disseminate the knowledge of discovered application states to the nodes. In addition to the application state object, the nodes and the coordinator have other internal objects to store the state of the crawl. These objects are explained in this section.

4.2.1 Application State

- *State identifier*: A unique identifier for the state, that is calculated as the hash of the serialized DOM.

- *Parent state*: The state identifier of a parent state through which the current state is reachable.

- *Parent event index*: The index of the event to execute from the parent state to reach the current state.

- *Number of events*, denoted by E: The number of events in the state.

- *Events*: The ordered list of events in the state. Each event has the following variables:

  - *JavaScript Event*: The JavaScript event to be executed by the node.

  - *Target State*: The target state reached by executing the event from the state. Ahead of the time the node may not know the value of this item, and thus it is initialized to null.

4.2.2 Coordinator State

The coordinator stores the following variables and objects:

- *Number of nodes*, denoted by N: The number of crawler nodes. Crawling starts after the N nodes have contacted the coordinator for the first time.

- *List of states*: The list of all states discovered by all nodes.

- *List of states per node*: The list of states' state identifier that each node knows. The coordinator keeps track of the states that a node knows about. If a node contacts the coordinator and probes it for new states, the coordinator will only reply back with the newly discovered states that the node does not yet know about.

- *List of nodes status*: This list stores the *node status* of all nodes. The node status is explained in Section 4.2.3.

4.2.3 Node State

Each node stores the following variables and objects:

- *Node identification number*, denoted by i: A unique identifier allocated to the node by the coordinator.

- *Coordinator address*: The coordinator's address through which the node contacts the coordinator.

- *Seed URL*: The URL for the RIA. We assume this URL does not change across different states of the RIA.

- *Pending tasks*: A list of tasks that the node is responsible to do.

- *States*: The list of states that the node knows about.

- *Status*: The status of the node. The node may be in one of the following states:

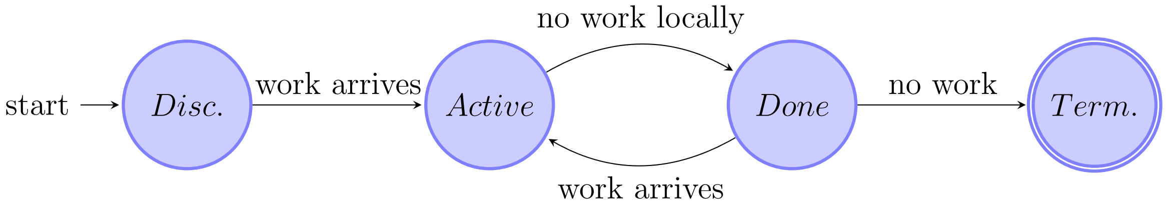


Figure 4.1: The node status state diagram: *Disconnected*, *Active*, *Done* and *Terminate* .

- *Disconnected*: The node is not yet initialized.
- *Active*: The node has been initialized and has work to do.
- *Done*: The node has finished executing all its tasks and does not have any work to do.
- *Terminate*: Crawling is finished and the node can leave the system.

Figure 4.1 shows the possible status transitions.

In Dist-RIA Crawler, initially the coordinator address is the only global constant, and it is assumed to be common knowledge before the crawling begins. Having the coordinator address, the node retrieves its identification number and the seed URL from the coordinator upon initialization. It then proceeds with the crawling algorithm described in Section 4.3.1.

4.3 Crawling Protocol

Dist-RIA Crawler is composed of a communication protocol between the nodes and the coordinator. This protocol deals with the crawling of the application, the exchange of the application states between the nodes and the coordinator, as well as the termination of the crawling process once all the states are discovered and all of the events have been executed.

Every time a node discovers a new state, it informs the coordinator. A node that has no more work to do also contacts the coordinator and probes the coordinator for more work to do. The coordinator may answer the contacting node with a new task assigned to it, or a notification that the crawl is over. Should a node receive new tasks from the coordinator it will add them to its pending tasks.

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.

Once a node executes all the events that it is responsible for, it asks the coordinator for new tasks. The coordinator responds as follows: If the status of all nodes are done, the coordinator initiates the termination process by sending a terminate message to all nodes. If not, the coordinator orders the node to ask again by sending a *stay* message. The node then probes the coordinator again and the loop continues until either more work becomes available or a termination order arrives.

Section 4.3.1 explains the crawling algorithm as it runs on the nodes.

4.3.1 Protocol Definition

Algorithm 1 describes the Crawling Algorithm as it is executed at each node. A node starts in the disconnected state. The node goes from the disconnected state to the active state upon getting the initial token which contains the seed URL, the node identification number i, and the load balancing approach from the coordinator. It then iteratively executes tasks and removes them from its pending tasks. When the list becomes empty, the node moves to the done state. At this point, it calls the coordinator, and depending on the answer received, it either goes back to the active state if more work becomes available, or it goes to the terminate state if no more work is available globally. The crawling algorithm invokes a set of procedures, and sends certain messages to the coordinator. These procedures and messages are explained below.

The following procedures communicate with the coordinator:

- *GetInitialTokenFromCoordinator* The node contacts the coordinator and gets a token that contains the following two items:
  - The unique node identification number i.
  - The seed URL.

  This method takes no input, and outputs the retrieved message.

Algorithm 1 Crawling algorithm (as executed at each node)

```
initialToken ← GetInitialTokenFromCoordinator()
i ← initialToken.identificationNumber
SeedURL ← initialToken.seedURL
nodeStatus ← Active
while (nodeStatus is not Terminate) do
    if pendingTasks is Empty then
        pendingTasks ← GetTasksFromCoordinator()
        if pendingTasks is Empty then
            nodeStatus ← Done
        else
            nodeStatus ← Active
        end if
        TerminationStatus ← SendStatusToCoordinator(nodeStatus)
        if TerminationStatus is Terminate then
            nodeStatus ←  Terminate
        end if
    else
        TaskToExecute  ← getTask(currentState, pendingTasks)
        currentState  ← ExecuteTask(TaskToExecute)
        if currentState is not in discoveredStates then
            push currentState to discoveredStates
            SendNewStateToCoordinator( currentState )
            pendingTasks ← GetTasksLocally(currentState)
        end if
        pendingTasks.Remove(taskToExecute)
    end if
end while
```

- *GetTasksFromCoordinator*: The node contacts the coordinator and asks for tasks. The coordinator responds with a set of tasks, or an empty result. This method takes no input, and outputs the answer received from the coordinator.

- *SendStatusToCoordinator*: Sends the status of the node to the coordinator. The coordinator will respond with the status of global termination, which can be either of:

  - **Active**: If the node is busy.
  - **Stay**: If there exist more work to do globally.
  - **Terminate**: If there exist no more work to do globally, and all nodes are in done state.

  This method takes the status of the node as input, and outputs the global termination status retrieved from the coordinator.

- *SendNewStateToCoordinator*: Sends a newly discovered state to the coordinator. This method takes the state to be sent to the coordinator as input, and has no output.

Other local procedures invoked during the crawling algorithm are:

- *getTask*: Uses a greedy strategy to search in the application graph for a pending task, and returns it. The greedy algorithm, which finds the closest task to the current state, is described by Dincturk et al[38]. A summary of the algorithm is as follows:

  1. Construct a temporary graph of states by:
     (a) Adding a node to the graph for every application state.
     (b) Adding a transition to the graph per event that has known source and target states.
  2. Add a transition from each node in the graph to the node that corresponds to the seed URL and uses the reset cost to assign weight to these transition. The reset cost is the average cost of retrieving the seed URL. This value is an environmental constant.

3. Iterates through all the tasks in the pending tasks. If the state for the task does not exist in the application graph (i.e. this is a new state discovered by another node), add a new node to the graph. Add a transition from parent state of the task state holder to the newly created nodes.
4. Run a Breadth-First-Search from the current state until a state among the pending tasks is found.

This method takes the current state, and the set of pending tasks as input, and outputs a task to be executed.

This greedy algorithm is not the only strategy to find a task. In the case of a full crawl, it is a good practice to choose tasks that minimize the overall crawling time. In the case of a partial crawl, it is a good practice to choose tasks that increase the chances of discovering new states earlier in the crawl. Menu and probabilistic model-based crawling algorithms[28, 31, 38] address this issue in depth and attempt to find as many states as early as possible in the crawl. Also if one categorizes pending tasks based on their holder states, this procedure can be used to implement Dept-First-Search and Breath-First-Search strategies:

   - LIFO: A *Last-In-First-Out* order of picking results in the implementation of a Dept-First-Search (DFS) strategy within the scope assigned to the node.
   - FIFO: A *First-In-First-Out* order of picking results in the implementation of a Breath-First-Search (BFS) strategy within the scope assigned to the node.

- *ExecuteTask*: Executes the task passed to it. This procedure takes the task to be executed as input and returns the state that is reached by executing the task as output.

- *GetTasksLocally*: This procedure gets a state as input, uses a partitioning algorithm and outputs a set of tasks. Dist-RIA Crawler uses a simple and primitive partitioning algorithm where it breaks the events into sets of almost equal sizes, one set per node. Each set is then deterministically and locally allocated to the node that it belongs to. Partitioning algorithm is formally defined and explained in Section 5.2.

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 4.1: AJAX file browser testbeds

4.4 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. All of the communications happen in JSON format in UTF8 encoding

4.4.1 Test-Application[3]

A jQuery-based AJAX file browser library[4] (Figure 4.2) 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 4.1).

[2] Similar to [84], 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/

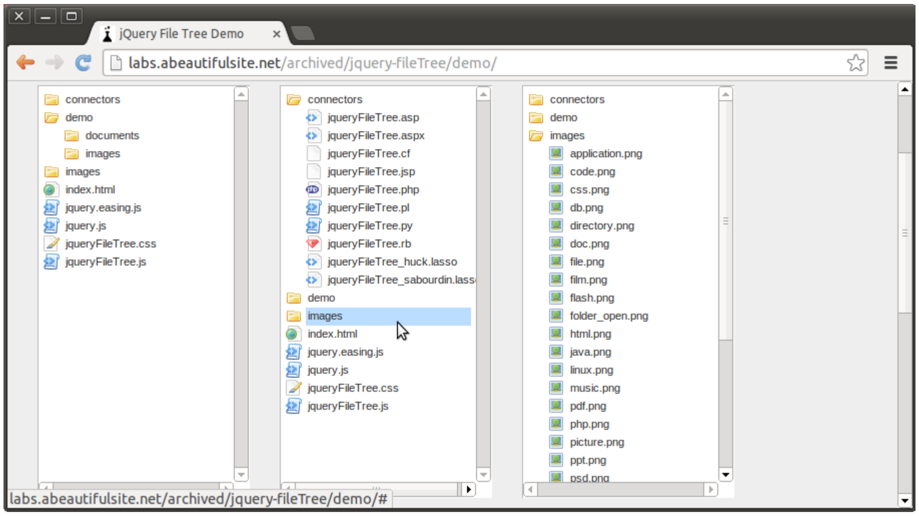


Figure 4.2: File tree browser RIA screen-shot

4.4.2 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 4.3 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

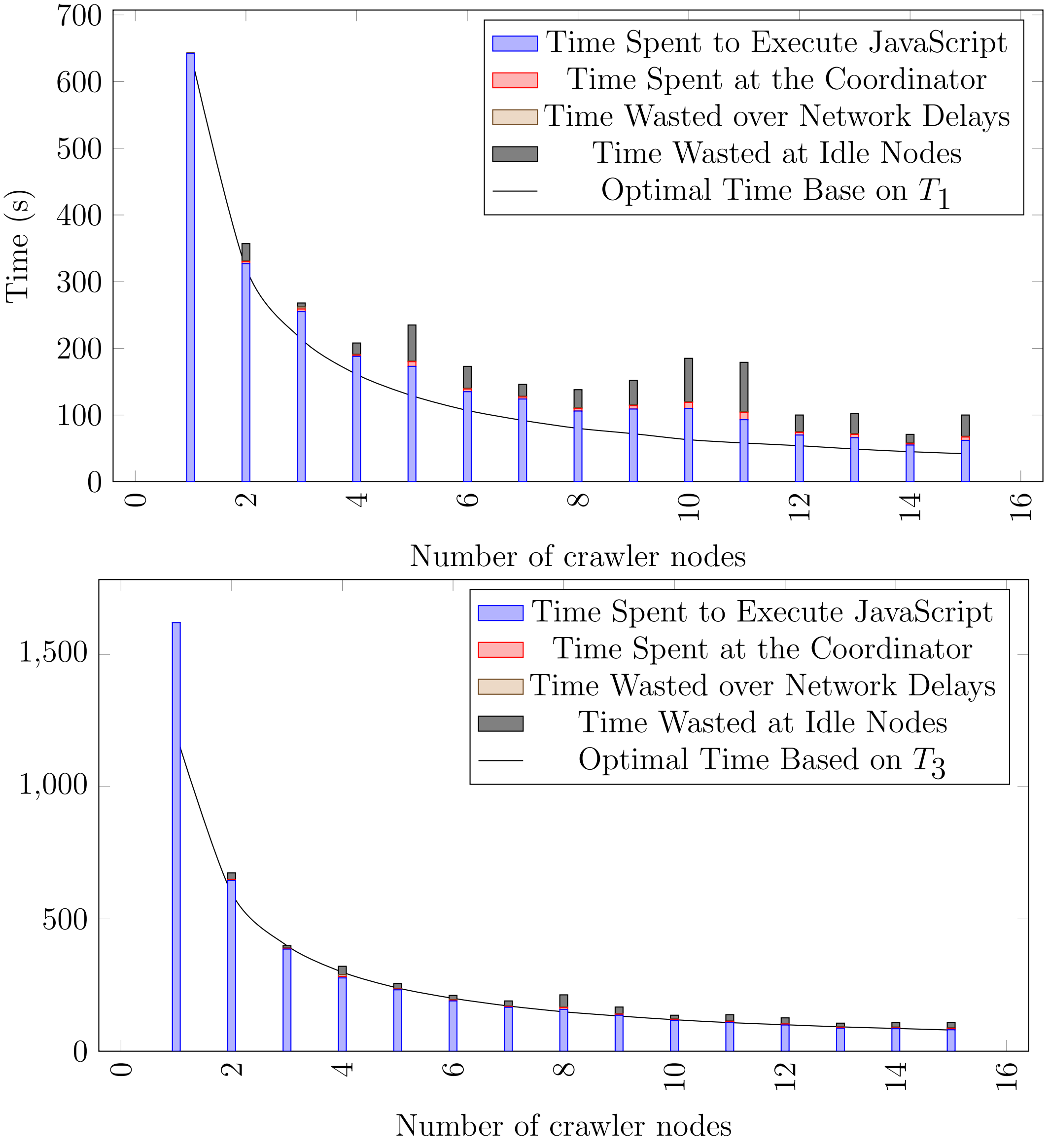


Figure 4.3: Time to crawl a RIA with multiple nodes: Apache HTTPD source code file browser (top), and Apache Cassandra source code file browser (down).

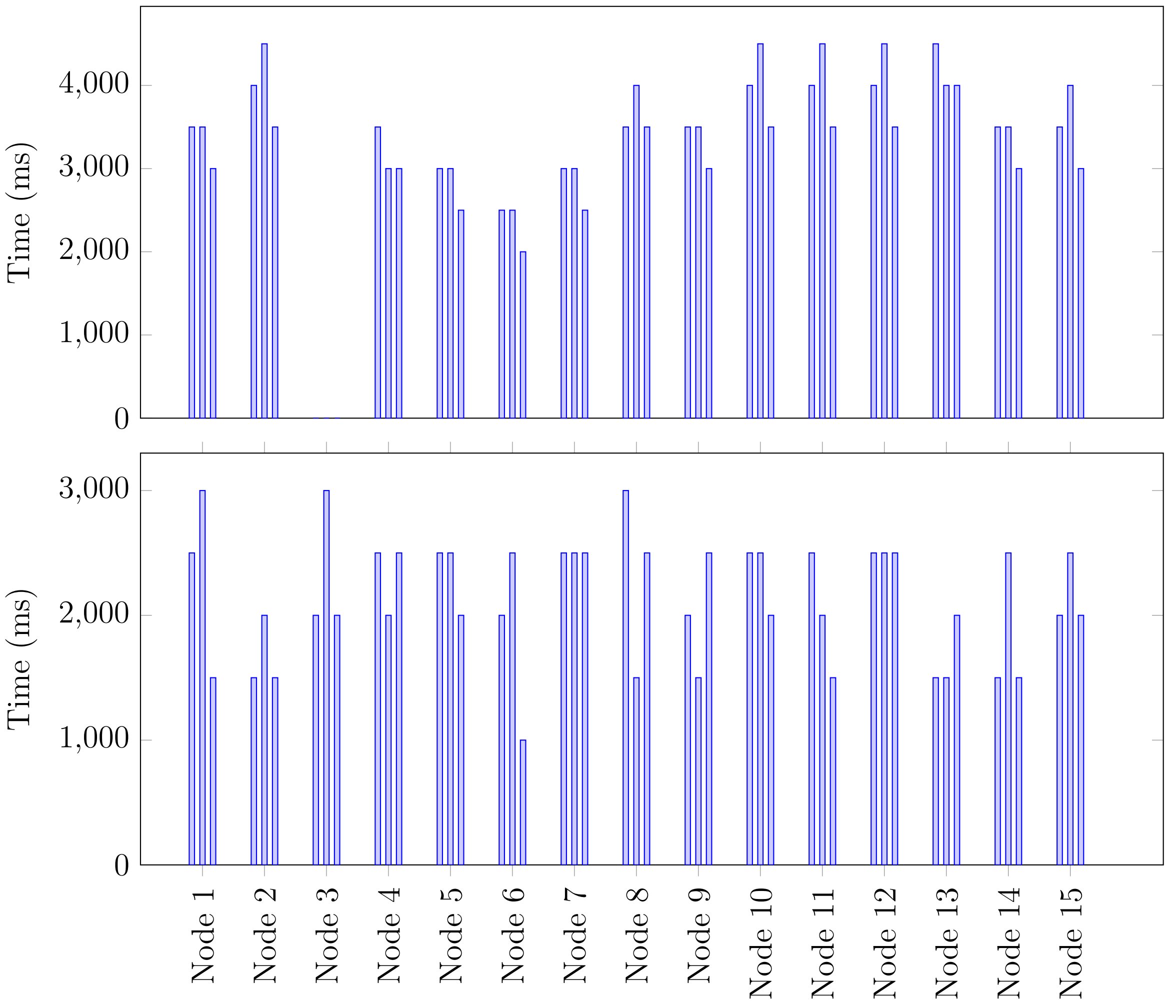


Figure 4.4: Idle time distribution during parallel crawl of AJAX file browser with 15 nodes: Apache HTTPD (upper figure), and Apache Cassandra (lower figure) source codes.

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 4.3: 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 4.4 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 4.3 and 4.4 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.

4.5 Conclusion

This chapter considers distributed crawling of RIAs using a Breadth-First strategy. A new load partitioning algorithm was proposed and a system for distributed crawling of RIAs, called Dist-RIA Crawler, was 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.

Although Dist-RIA Crawler scales well up to a number of nodes, its suffer from the high cost of executing events. This cost is primarily the result of using a Breath-First Strategy. As Dincturk[36] shows the overall cost of crawl can be reduced substantially by using Greedy or Probabilistic model. Additionally, a technical limitation that Dist-RIA Crawler faces is using a V8 JavaScript engine. At the time of this writing, V8 engine offers limited access to the JavaScript virtual machine.

To reduce the high cost of Breadth-First algorithm, and to take advantage of a more open JavaScript engine, Chapter 6 shows a distributed architecture to crawl a website using the greedy algorithm implemented over an open source JavaScript engine called $PhantomJS$.

Chapter 5

Some Implementation Issues

The Dist-RIA Crawler design described in the last chapter explained 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 the events in the state. In Dist-RIA Crawler, each crawler node implements a Breath-First algorithm in its own scope.

In the context of RIA crawling, the term *crawling strategy* refers to the strategy the crawler follows to decide the next event to execute. Dincturk et al. [13, 31, 38] studied several crawling strategies to optimize the crawl according to two criteria: reducing the total time of the crawl, and finding new application states as soon as possible during the crawl. Among the strategies studied, the greedy algorithm[102] scores well in the majority of cases. The greedy strategy is particularly much better than the breath-first search strategy. This algorithm always chooses the closest application state with an un-executed event, goes to the state and executes the event.

In the Dist-RIA Crawler, the nodes only broadcast the knowledge of application states, and no single node has the entire knowledge of the transitions between 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. In addition, through our experiments with the Dist-RIA Crawler, we learned about other short-comings as well. This Chapter addresses these short-comings as follow:

- **Client-side events**: Due to the use of the V8-Engine, Dist-RIA Crawler cannot capture client-side event listeners. This, merely technical, limitation of the V8-Engine stems from its API. This limitation did not pose a problem for the experiments presented in Chapter 4, since the target applications considered did not use any event that is un-detectable by the Dist-RIA Crawler. However, it restricts the crawler to crawl other RIAs.

To deal with this problem, we replaced the V8-Engine with a virtual web browser. Details on this replacement and its ramifications are described in Section 5.1.

- **Partitioning Algorithm**: The Dist-RIA Crawler used a simple range-based partitioning algorithm. This algorithm may lead to unequal partitioning and may make some nodes bottlenecks. Section 5.2 describes different partitioning algorithms and chooses the most efficient algorithm.

In addition to these improvements, we measure performance of different operations in this chapter. The improvements described and the performances measure, are used to construct the efficient crawlers described in the next chapters.

The rest of this chapter is as follows:

- In Section 5.1, we describe how to run a full-fledged browser using PhantomJS.
- In Section 5.2, we describe how to improve the partitioning algorithm.
- In Section 5.3, we measure performance overhead of running operations such as calculating the next task to execute.
- Finally, we conclude this chapter in Section 5.4.

5.1 Running a Full-Fledged Browser

The Dist-RIA Crawler abstracted the interaction with the web application through a V8 JavaScript Engine. As mentioned in Section 4.5, V8 limits our ability to interact with the web application. To by-pass this limitation, we switched from the V8 JavaScript engine to a full-fledged headless browser with the ability to render CSS and take screenshots from the web application. This browser is called *PhantomJS*[1], an open-source headless WebKit.

Switching from a JavaScript engine to a full-fledged browser comes with a caveat: the asynchronous nature of JavaScript effects the crawling strategy. The crawler can no longer 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

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

schedule an event to happen in the future. When these events happen the state of the application may change. In other words, there are two main types of events that may have dormant ramifications. These two event categories include: *Asynchronous calls* and *Clock events*. The JavaScript engine (henceforth referred to as *JS-Engine*) deals with these events as follows:

- **Asynchronous calls**: 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 asynchronous calls in progress and wait for their completion before continuing. Section 5.1.1 describes a code-instrumenting technique to identify the asynchronous calls in progress. This section also shows how the crawler application can get a notification upon initiation and termination of any such events in the target application.

- **Clock events**: Time events are another type of events that JS-Engine needs to be aware of. These events happen at some time in the future and, similarly to asynchronous calls, JavaScript does not offer a mechanism to keep track of them. Section 5.1.2 describes another code-instrumenting method to identify and handle such events. Having these notifications, the crawler application can decide the right time to retrieve the current state of the target application and trigger the next event.

In addition to asynchronous calls and clock events, another code-instrumentation is also required on the JavaScript part of the web application to fully detect client-side events. The JS-Engine needs to identify the user interface events. Client-side events that leave a footprint in the DOM are easy to detect: Traversing the DOM and inspecting each element can find these events. Unfortunately not all client side events are extractable from the application DOM. Another category of events are *Attached events*. These events are attached to an element using *addEventListener* and do not reflect themselves on the DOM. Section 5.1.2 describes a mechanism to identify such events.

The code snippets described in Sections 5.1.1, 5.1.2 and 5.1.2 are injected into the header of the target web application using a proxy server. They are injected in such a way as to ensure that they run prior to running any of the RIA's original JavaScript code.

Listing 5.1: 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);
};
```

5.1.1 Handling asynchronous calls

Executing an event on the RIA may start asynchronous HTTP calls to the server. It is the responsibility of the JS-Engine to wait for all asynchronous calls to finish before it continues its interaction with the application. There are unfortunately no ways for the JS-Engine to know if there are asynchronous calls in progress without modifying the target web application. Fortunately, however, one can inject a JavaScript code into the RIA to re-define the browser's native code that performs asynchronous calls and add necessary measures so that the target application keeps track of the asynchronous calls.

Let us define two JavaScript functions called *ajaxStartNotification* and *ajaxStopNotification*. These two functions are to be called every time an asynchronous call starts or finishes, respectively. Listing 5.1 shows how to redefine asynchronous *send* and *onreadystatechange* operations, such that the target web application notifies the crawler application automatically upon start and finish of asynchronous call[2].

As the listing shows, JS-Engine redefines both the *send* method and the *onreadystatechange* variable. A new method, called *sendOriginal*, is defined in the context of *XMLHttpRequest* so that it has access to all the methods and attributes of the *XMLHttpRequest* object, and populates it with the contents of the *send* method, which is web-browser native code.

[2] In this thesis we only show the technique for *XMLHttpRequest* the module responsible for asynchronous calls in many popular browsers such as Firefox and Chrome. Microsoft Internet Explorer, however, does not use this module, and instead it uses *ActiveXObject*. Similar measures can be taken for *ActiveXObject*.

Listing 5.2: Hijacking setTimeOut

```
var setTimeoutOriginal = setTimeout;
setTimeout = function(x,y){
       var self = this;
       var newX = function(x){
              x();
              parent.setTimeoutEndNotification(self.setTimeoutHandle);
       };
       this.setTimeoutHandle = setTimeoutOriginal(newX,y);
       parent.setTimeoutStartNotification(this.setTimeoutHandle);
       return this.setTimeoutHandle;
};
```

In effect, JS-Engine renames *XMLHttpRequest.send* to *XMLHttpRequest.sendOriginal*. It then redefines the *send* method of *XMLHttpRequest*. The new *send* method first notifies the parent about the start of a new asynchronous call, then proceeds with the original *send* method that is pointed at by *sendOriginal*.

A similar measure is taken for the *XMLHttpRequest . onreadystatechange* variable. This call back function is first stored in a variable called *onreadystatechangeOriginal*. The *onreadystatechange* is then redefined as a function. This function first performs the original *onreadystatechange* function, then notifies the JS-Engine that the asynchronous call is over. Note that *this* pointer has to be passed to the *onreadystatechangeOriginal* function so that the original *onreadystatechange* function is called in the right context.

5.1.2 Handling clock events

The second source of asynchronous behaviour of a RIA with respect to the time comes from executing clock functions, such as *setTimeout* and *setInterval*. These methods are used to trigger events in the future. In many cases, such events can help animating the website, and adding fade-in or fade-out effects. Knowledge of the existence of such dormant functions may be necessary for the JS-Engine. This sections elaborates on receiving notifications from the *setTimeout* and *clearInterval* functions. Similar measures can be used to receive notifications from *setInterval* events.

setTimeout is used to call a function, passed as an argument, after a specified number

Listing 5.3: Hijacking clearInterval

```
var newClearInterval = clearInterval;
clearInterval = function(x){
        parent.clearIntervalNotification(x);
        return newClearInterval(x);
};
```

of milliseconds. Listing 5.2 shows how to inform the JS-Engine of the creation of a dormant events with *setTimeout*, and also after the function is fired up. The variable *setTimeoutHandle* is used to store the handle to the event, and is passed to the JS-Engine. This variable is passed to the JS-Engine in case the crawler intends to ignore certain clock events.

clearInterval is used to terminate prematurely a clock event. Listing 5.3 shows how to get a notification in the JS-Engine upon firing a *clearInterval* function. Similar to *setTimeout*, in this case too, the handle to the clock event (i.e. x) is passed to the JS-Engine, in case the engine needs to disregard a certain clock event.

Handling Attached Events

The final challenge faced by the JS-Engine is to detect 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*. Listing 5.4 shows a technique to keep track of events added and removed through *addEventListener* and *removeEventListener*.

This listing first defines a global object called *_eventListeners*. When a call is made to addEventListener an entry is added to this 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.

5.2 Partitioning Algorithms

A primitive partitioning algorithm based on ranges was introduced in the Dist-RIA Crawler. In a RIA, it is often the case that the relative location of events does not

Listing 5.4: Hijacking event listeners

```
var _eventListeners;
var addEventListenerOrig = Element.prototype.addEventListener;
var removeEventListenerOrig = Element.prototype.removeEventListener;
var addEventListener=function(type,listener) {
       _eventListeners[type][listener] = this;
       addEventListenerOrig(type,listener);
};
var removeEventListener=function(type,listener) {
       delete _eventListeners[type][listener];
       removeEventListenerOrig(type,listener);
};
Element.prototype.addEventListener=addEventListener;
Element.prototype.removeEventListener=removeEventListener;
if (HTMLDocument) {
       HTMLDocument.prototype.addEventListener=addEventListener;
       HTMLDocument.prototype.removeEventListener=removeEventListener;
}
if (Window) {
       Window.prototype.addEventListener=addEventListener;
       Window.prototype.removeEventListener=removeEventListener;
}
```

change much between different states. For example, the location of navigation menus may always remain on the top of the page, or a log-in link tends to remain on the top-right corner of the page. By allocating events based on ranges, some nodes tend to get similar events. If the density of time consuming JavaScript events happened to be high in some ranges in a state, chances are that the same scenario happens on other states. To solve this issue, this section describes better partitioning algorithms.

By using a partitioning algorithm, nodes decide the next event to execute locally based on the events allocated by the partitioning algorithm. A partitioning algorithm takes as input a list of events in the page and the node identifier and returns the list of events that belong to the identified node. Any partitioning algorithm has to fulfill the following two conditions:

- Guarantee that all events are executed in all states.
- Lack of work duplication: the partitioning algorithm should assign each event to only one node.

The partitioning algorithm is to divide the events in the page into almost equal subsets and allocate each subset to a node. As described in Chapter 2, the assumption is that the number of events in the page is much higher than the number of nodes. These subsets are not necessarily ranges. Some partitioning algorithms that perform this function are described below.

- **Range-Based Partitioning algorithm**: The algorithm divides the number of events in the page E_S by total number of nodes N, and gets N ranges, one for each node. The algorithm then allocates each range to one node, starting from the first node. In the other words, an event in state s belongs to node i if, given its event identifier e_{id}, the following condition holds:

  $$\lfloor \frac{e_{id}}{N} \rfloor = i$$

  This algorithm almost equally distributes the work among the nodes. In the worst case scenario, if E_S is not a multiple of N, some of the nodes will get one less event to execute than the others. This algorithm was used in Chapter 4 to construct the Dist-RIA Crawler.

- **Modulo-Based Partitioning algorithm**: Another similar approach is to use a modulo function. In this case, the partitioning algorithm divides the events in the page into a set of strides, one stride per node. More formally, an event in state s belongs to node i if its event identifier e_{id} satisfies the following condition:

$$e_{id}\ mod\ N = i$$

- **Hash-Based Partitioning algorithms**: In cryptography, a hash function is defined as a function $H : K \times M \rightarrow (0,1)^n$ where K is the key, M is a plain-text, and $(0,1)^n$ is a fixed size string representing the hash of the plain-text M with key K[109]. In effect hash functions achieve a deterministic random mapping of input to a finite output set. This valuable attribute can be used to map events in the page to a finite set of nodes. To achieve this, the event identifier is to be hashed, and the hash is to be mapped to $\{0, 1, \ldots, N-1\}$. More formally, given a hash function H, a random number R as the key, an event in state s belongs to node i if its event identifier e_{id} satisfies the following condition:

$$\frac{(H(R, e_{id})) \times N}{2^n} = i$$

It is often the case that many events are repeated in different states of the application, and further they have the tendency of keeping a relative location in the DOM. Range-based and modulo-based partitioning algorithms are susceptible to make some of the nodes bottlenecks by constantly assigning them time consuming events. Due to properties of hash functions, they seem to randomly distribute events among the nodes. Thus, hash-based partitioning functions are less susceptible to make some nodes bottlenecks.

The performance of range-based and modulo-based partitioning algorithms can be improved by random and deterministic permutation of events in the page. pseudorandom number generator algorithms are useful to achieve this. Formally, a pseudorandom generator $G : D^{in} \rightarrow D^{out}$ maps the inputs from input domain D^{in} into the output domain D^{out}. In addition to this requirement, pseudorandom number generators are often good in creating an output that is equally distanced in D^{out} irrelevant of the input[19]. A thorough survey of pseudorandom number generators is available by Ritter[106]. If before applying range-based and modulo-based algorithm a pseudorandom permutation is applied to the events in the page, the two partition algorithms are no longer vulnerable to assigning time consuming events to a set of nodes.

In case of Breath-First, Depth-First, and greedy strategies, it is trivial to create a partitioning algorithms since events to not have types. In these cases, any of the partitioning algorithms described above can be used as they are. In case of the probabilistic model, if no unexecuted event is found in the current state, a search is initiated from the current state to find a state with an un-executed events which has the highest probability to lead to a new state. In this strategy events have types, and the history of execution of an event type is required before the node can calculate the probability that an event from a given type can lead to a new state. To address this issue, the partitioning algorithms assign events based on their types to the nodes. In this model, events in the page are first categorized into their types, then event types are allocated to the identified node.

5.2.1 Implementation of Partitioning Algorithms

This section explains some implementation details about hashing functions and pseudorandom number generators. Due to their trivial nature, details about range-based and modulo-based partitioning algorithms are omitted.

Hash functions

Two widely used families of hash algorithms are: *message-digest family*[107] (e.g. MD4 and MD5) and *secure hash family*[123] (e.g. SHA-2 and SHA-3). For the best choice of hash function we look into the CPU consumption of each algorithm. Relative performance of these functions are evaluated in Section 5.2.2.

Pseudorandom Number Generator

Linear Congruential Generators[117] (LCGs) are a class of algorithms to generate pseudorandom numbers. This is one of the oldest and most widely used classes to generate pseudorandom numbers. This class is defined as a generator series X where:

$$X_n = aX_{n-1} + c \ (mod\ M)$$

where a, c, and M are integer factors[117]. While LCGs are fast and memory efficient algorithms, they suffer from poor dimensional distribution[44]. In our application, lack of a good dimensional distribution defeats the purpose of using a pseudorandom number generator.

Lagged Fibonacci Generators (LFGs) are an improvement over linear congruential generators with a better dimensional distribution[62]. An LFG generates series X where:

$$X_n = (X_{n-p} \odot X_{n-q}) \ (mod\ M)$$

Listing 5.5: Fisher-Yates Shuffle

```
function fisherYateShuffle ( unshuffledArray) {
  for (var index = unshuffledArray.length - 1; index > 1; index--) {
    var randomIndex = getPseudoRandom(index);
    var temp = unshuffledArray[index];
    unshuffledArray[index] = unshuffledArray[randomIndex];
    unshuffledArray[randomIndex] = temp;
  }
  return unshuffledArray;
}
```

where p and q are predefined lag factors and $\odot$ represents a generic binary operation such as addition, subtraction, or bitwise exclusive-or[62].

Event List Permutation

Given a deterministic pseudorandom number generator function, the set of events in the page are to be shuffled deterministically. A simple yet powerful shuffling algorithm is called *Fisher-Yates shuffle*, also known is *Knuth shuffle*[18]. This time optimal algorithm has the complexity of $O(n)$ and achieves a uniform distribution given the proper random number generator.

The shuffling algorithm works by taking an array of events as input. It goes through the elements of the array one by one from the last element to the first. At each step it chooses a random element from the beginning of the list to the current position in the list, and swaps the content of the current element with the randomly chosen element. Code snippet 5.5 shows the implementation of the algorithm in JavaScript. In the code snippet, $getPseudoRandom$ function represent the pseudorandom number generator that, given an upper bound, returns a random number from 0 to the given upper bound.

5.2.2 Evaluation of Partitioning Algorithms

To choose the most effective partitioning algorithm, in this section we compare the relative performance of hash functions and pseudorandom number generators. Figure 5.1 compares the performance of different hash algorithms as well as the random shuffle. Two algorithms from the message digest family (i.e. RIPEMD160 and MD5), three algorithms

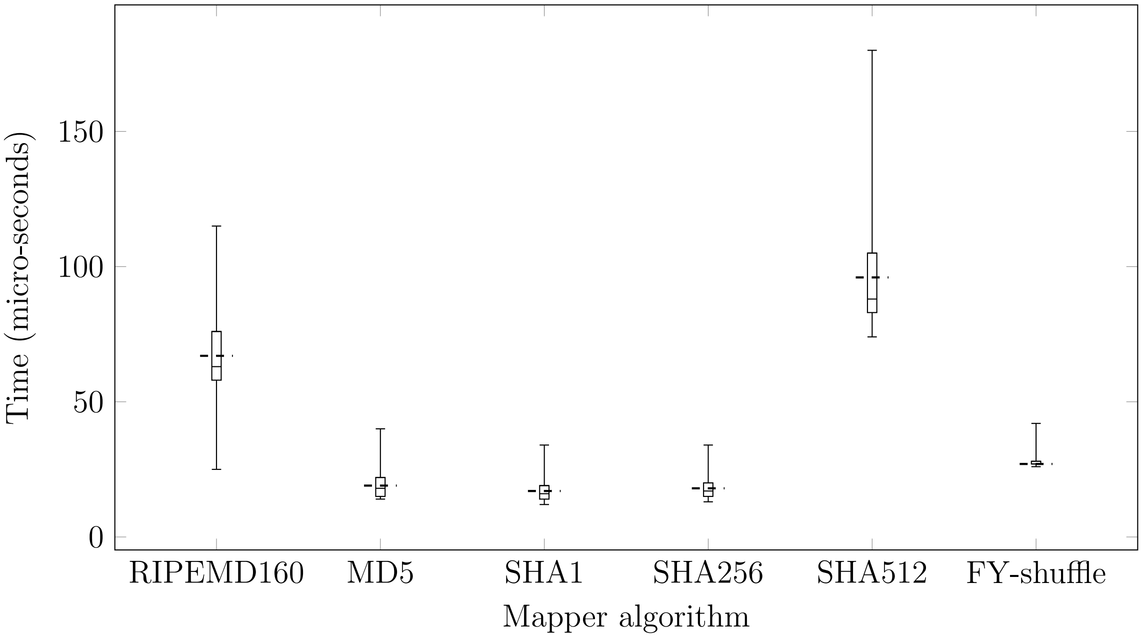


Figure 5.1: Time performance comparison between the hashing algorithms

from secure hash family (i.e. SHA1, SHA256 and SHA512), as well as the Fisher-Yates shuffling algorithm with Lagged Fibonacci Generator (LFG) are compared. The hasher algorithms are implemented in JavaScript language by an open source library called *crypto-js*[3], and the LFG pseudorandom number generator is implemented by another open source library called *Fibgen.js*[4].

There is a box plot per hash function, and one for the shuffling function. Each box plot shows the quartiles of the time it takes for the algorithm to map an input to its output. The input represent the position of an event in the page and is an integer. The output is the new position of the event in the page and is another integer. In addition to the quartiles, a dashed line shows the average time for the algorithm to perform its operation. As for input, integers between 0 and 999 are chosen. Each experiment was repeated 1000 times and the times presented are the average amount. Numbers are reported in logarithmic scale, and the time unit is micro-second.

As the figure shows, the two fastest algorithm to take a hash of an input are MD5 and SHA1, with SHA1 out-performing MD5 slightly. Based on the results of this experiment, SHA1 is chosen as the mapping function for the partitioning algorithm used by the peer-

[3] `https://code.google.com/p/crypto-js/`

[4] `https://gist.github.com/wivlaro/6375828`

to-peer architecture.

5.3 Performance Measurements

In this section we measure the performance of certain operations used by the crawling algorithm. These measurements are used in later chapters to justify some of the design decisions made.

5.3.1 Time to transmit messages

As described before, communication happens through message passing. In this section we measure the efficiency of message passing. Figure 5.2 shows the time it takes to send a message from a node to another, in logarithmic scale. Each message was sent 100 times and the distribution of measured time is indicated by the corresponding stack-bar. The measured times include the time it takes to encode the message by the sender and the time it takes to decode it by the receiver.

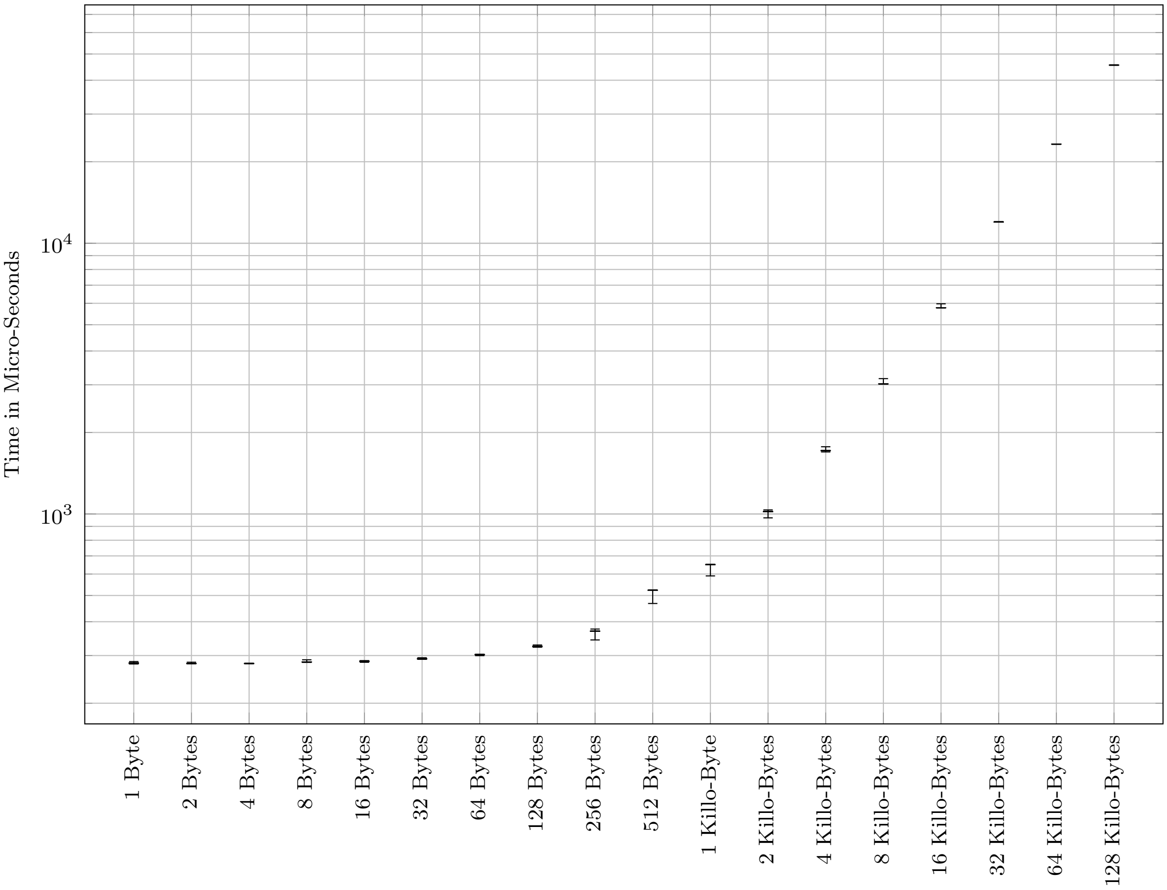


Figure 5.2: Cost of sending messages between nodes

Given the measured times, we can calculate overhead of sending different messages during the crawl process:

- **State**: A message to inform another node about a new state discovery contains state identifiers, number of events, a parent state identifier, and an event identifier in the parent state that leads to the discovered state. These items, along with the message header take between 128 to 256 bytes. Thus an average delay to inform another node about a state is from 324 to 369 Micro-Seconds.

- **Transition**: A message to inform another node about a transition contains source identifiers, target state identifiers, and event identifier. Similar to the *State* message, a transition message takes between 128 to 256 bytes. Thus an average delay to inform another node about a transition is expected to be from 324 to 369 Micro-Seconds Micro-Seconds.

- **Termination Token**: Size of termination token varies depend on the number of crawler nodes, number of discovered states, and number of visited states by crawler nodes. In worst case, the token contains a state identifier for all application states, and all nodes have visited all states. Among the test applications used in this thesis, *Dyna-Table* with 448 states has the largest number of application states. Assuming we are crawling this application with 20 nodes, termination token can get as large as 8 Kilo-Bytes. Thus in the worst case scenario where the token is at its largest size, the average time it takes to send the token from a node to another is less than 3 milli-seconds.

5.3.2 Time to Calculate the Task to Execute

After execution of an event, the crawler has to calculate the next task to execute. Different crawling strategies run different algorithms to calculate the next task to execute. In this section we measure the time it takes to calculate the next task to execute.

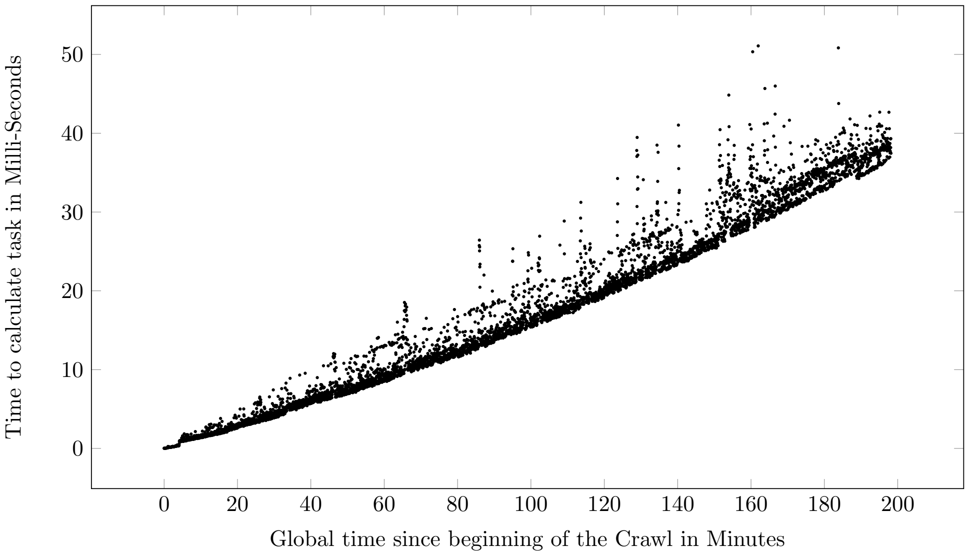


Figure 5.3: Time to calculate next to execute as crawling Dyna-Table web application proceeds, using one node with Breath-First Search Strategy

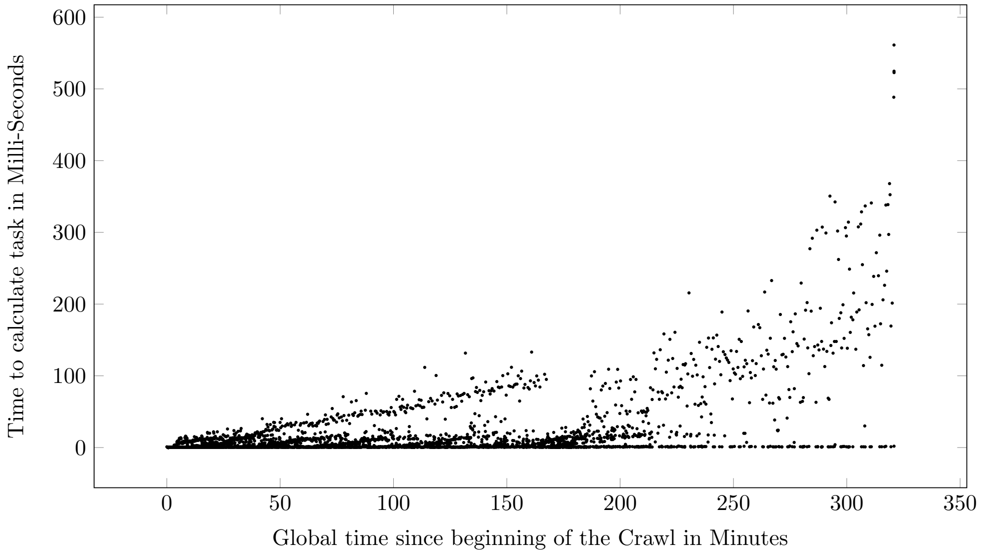


Figure 5.4: Time to calculate next to execute as crawling Dyna-Table web application proceeds, using one node with Depth-First Search Strategy

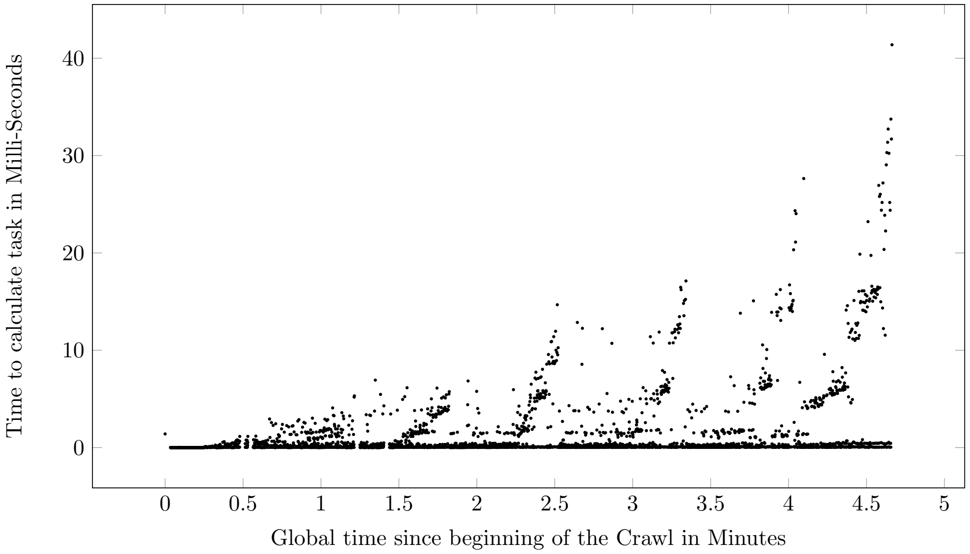


Figure 5.5: Time to calculate next to execute as crawling Dyna-Table web application proceeds, using one node with Greedy Strategy

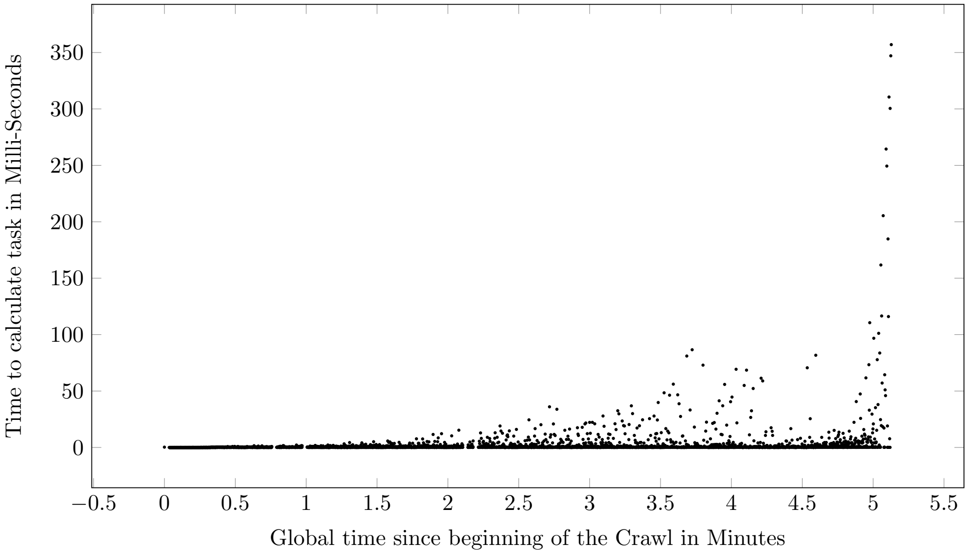


Figure 5.6: Time to calculate next to execute as crawling Dyna-Table web application proceeds, using one node with Probabilistic Strategy

Figures 5.3, 5.4, 5.5 and 5.6 show the time it takes to calculate the next task using breath-first search, depth-first search, greedy and probabilistic strategies, respectively. Table 5.1, summarizes these figures and show the average time to calculate the next task. In these figures y-axis shows the time to calculate the next event, and x-axis represent the clock since the beginning of the crawl. As the figures show:

- The time to calculate the next task to execute, tends to rise steadily using breath-first search strategy. In this algorithm, after finishing each task, the crawler looks for the next task to execute by looking into the seed URL and then the most immediate children for a task to perform. As the crawl proceeds the algorithm should go deeper in the graph before it can find a new task. Thus the cost of running the application tends to increase as the crawl proceeds.

- Unlike breath-first search strategy, in other crawling strategies, the time it takes to calculate the next task to execute does not always go up as the crawl proceeds. In the case of depth-first search strategy it is not unusual to find a task to do in the current state of the application. This results often in very small calculation times. If, however, the crawler does not find a task close to the current state of the application, the depth-first search strategy continues its search from the seed URL which causes large calculation times.

Table 5.1: Average time to calculate the next event for different crawling strategies, while crawling Dyna-Table web application with one node

Strategy	Depth-First Search	Breath-First Search	Probabilistic	Greedy
Average Time in Milli-Seconds	11.867	17.459	1.693	0.946

Table 5.2: Average number of events in a task for different crawling strategies, while crawling Dyna-Table web application with one node

Strategy	Depth-First Search	Breath-First Search	Probabilistic	Greedy
Average Number of Events	9.85	8.00	1.85	1.67

- The time it takes to calculate the next task to execute is generally lower in the greedy algorithm than in both breath-first and depth-first search strategies. In this algorithm, the crawler does not have to study all options in the current branch, as it is the case for depth-first search strategy; nor does it have to start from the seed URL and check all immediate children before going to further children for a task. Thus the greedy algorithm has a higher chance of finding tasks faster than the other two algorithms.

- In most cases, it takes the probabilistic strategy similar time it takes the greedy strategy to calculate the next task. In some cases, however, the algorithm requires to search further in the application graph before it can find an event that has a high priority. In these cases, it takes the probabilistic strategy a very long time to find the next task to execute.

As Figures 5.3, 5.4, 5.5 and 5.6 show, in majority of the cases, it takes a few milliseconds to calculate the next task to execute. Executing the task which often involves executing several client side events, possibly interacting with the server, and possibly performing a reset, often takes much longer than calculating the next task to execute.

5.3.3 Number of events in Tasks

A task is often composed of several events to be executed. The number of events in each task is the primary factor that determines the time it takes to execute the task. Different crawling strategies create tasks with different number of events. In this section we measure the number of events in the tasks calculated by different crawling strategy.

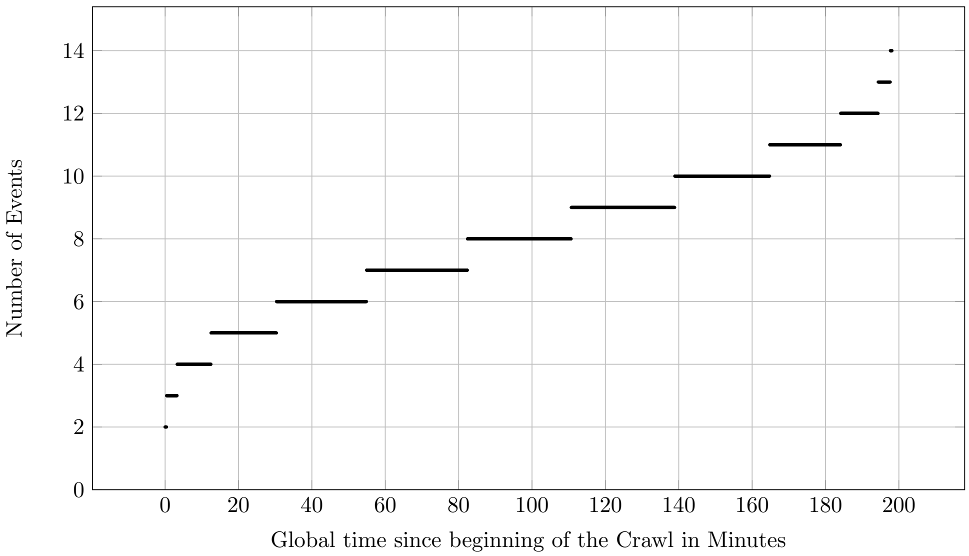


Figure 5.7: Number of events to execute as crawling Dyna-Table web application proceeds, using one node running Breath-First Search Strategy

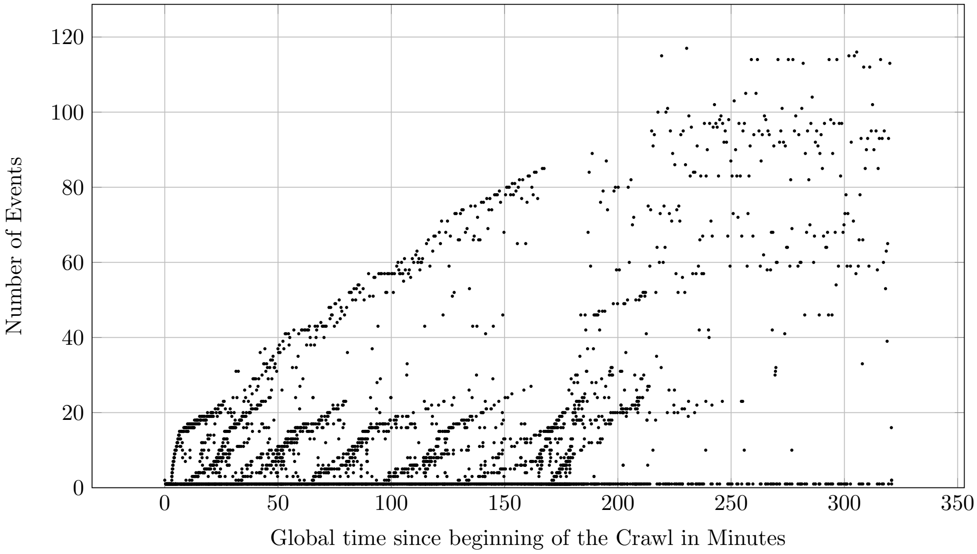


Figure 5.8: Number of events to execute as crawling Dyna-Table web application proceeds, using one node running Depth-First Search Strategy

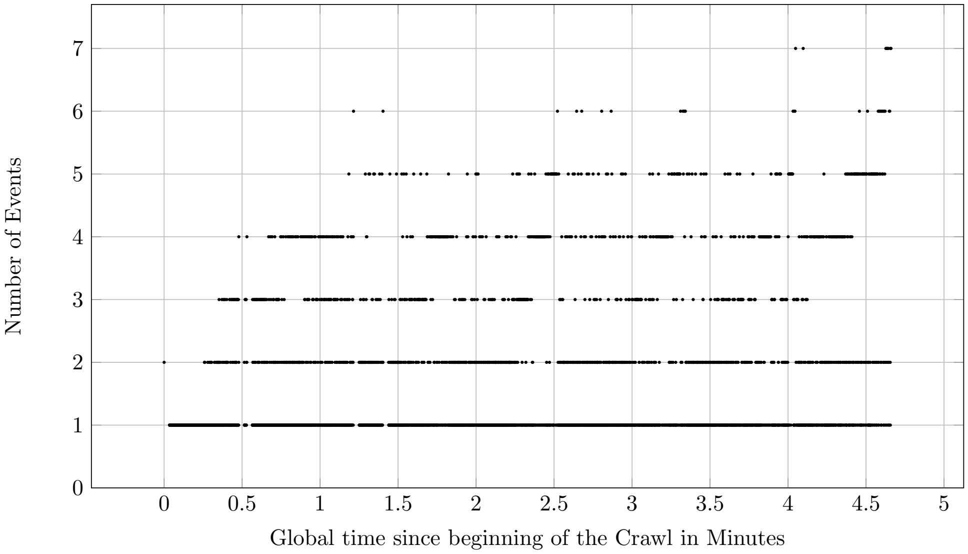


Figure 5.9: Number of events to execute as crawling Dyna-Table web application proceeds, using one node running Greedy Strategy

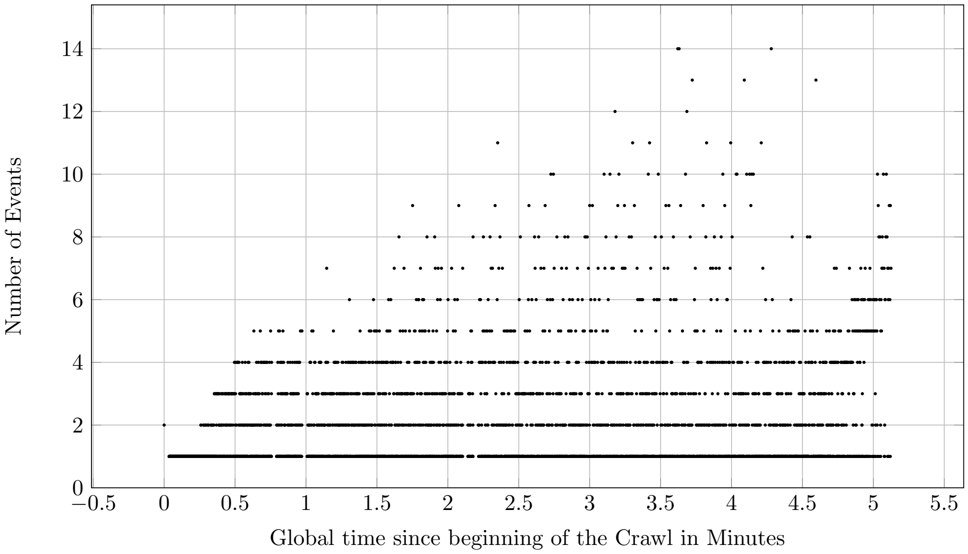


Figure 5.10: Number of events to execute as crawling Dyna-Table web application proceeds, using one node running Probabilistic Strategy

Table 5.2, shows the average number of events in each task for different crawling strategies. Figures 5.7, 5.8, 5.9 and 5.10 show the length of events in tasks created to crawl Dyna-Table target application, using breath-first search, depth-first search, greedy and probabilistic strategies, respectively. In these figures y-axis shows the number of events, and x-axis represent the clock since the beginning of the crawl. As the figures show:

- As the crawling proceeds, the number of events in tasks increases using the breath-first search strategy. This length, however, does not increase modestly and in Dyna-Table application this length can be as high as 14 events.

- The depth-first search strategy often generate tasks with small number of events in them. In many cases, however, the algorithm is very in-efficient and generates tasks with a large number of events in them. The unusual large number of events caused when the first time the crawler discovers a state it is after a long chain of events. To click on the next event in the state, the crawler has to go through the same long chain of events.

- The greedy strategy represent a very efficient strategy. As Figure 5.9 shows often very few events exist in each task. The rare worst case scenario happens with 7 events in a task.

- The probabilistic strategy has similarities to the greedy strategy. This strategy often generates tasks similar to the greedy strategy. In some cases however, the strategy generates tasks with a larger number of events to find new states faster. This strategy is often as efficient as the greedy strategy, but in the worst case scenario can generates tasks as long as the one generated using the breath-first search strategy.

The time it takes to execute individual events depends highly on the target application and the server hosting it. Execution of JavaScript events, that do not trigger an asynchronous call to the server, is substantially faster than the events that interact with server. For example, in Dyna-Table web application, execution of events that do not interact with the server always take less than 20 milliseconds. In the same application, events that do interact with the server often take more than 85 milliseconds to execute, and can take up to 1.3 seconds.

5.4 Conclusion

This chapter addressed several short-comings of the Dist-RIA Crawler, and measure performance of several operations:

- A new virtual web browser was described that has the ability to capture client-side events and handle asynchronous events.

- Several partitioning algorithms were described and performances were compared against each other.

- The time to send messages of different sizes over the network was measured and was used to calculate the cost of communication between the nodes.

- The time to calculate the next task to execute and the cost of executing the task in terms of number of events that has to be run using different strategies is measured. Additionally, the cost of communication through network was measured.

Next chapter describes two new distributed architectures for crawling RIA with the ability to run any crawling strategy. Techniques described in this chapter, as well as the lesson learned by measuring performance overhead of different operations are used in designing the next RIA crawler.

Chapter 6

Crawling Architectures: Client-Server and Peer-to-Peer

Chapter 4 introduced our first effort in creating a distributed crawling architecture for RIAs. This chapter expands our initial effort by introducing new crawling architectures and relaxing some of the assumptions made in Chapter 4. Two new crawling architectures introduced in this chapter are called *client-server* and *peer-to-peer*.

The rest of this chapter is organized as follows:

- Distribution of crawling control and the architectural choices are discussed in Section 6.1.
- Performance properties of different crawling strategies are discussed in Section 6.2.
- The client-server architecture is described in Section 6.3.
- The peer-to-peer architecture is described in Section 6.4.
- Finally, in Section 6.5 we conclude this chapter.

6.1 Distribution of Crawling Control

From a high-level point of view, we can adopt two distributed models to crawl RIAs:

- Client-Server architecture: In this model, nodes rely on a centralized unit to dispatch jobs to them.

- Peer-to-Peer architecture: In this model, nodes calculate tasks to do locally and communicate with each other in a peer-to-peer fashion.

6.1.1 Client-Server Architecture

In this architecture a centralized unit is responsible for allocating tasks to the nodes. Nodes do not communicate with each other directly and they only communicate with the centralized unit, henceforth referred to as the coordinator. In this architecture, nodes contact the coordinator, asking for a task. Based on the crawling strategy, the known application graph, and the state of the node, the coordinator calculates a task to be executed and returns the task to the node. The node executes the task and sends the new transition back to the coordinator while requesting another task to execute.

In this architecture the coordinator keeps the entire application graph. Since the coordinator calculates the next task to do, nodes do not require to have the knowledge of the transitions or states in the application graph. Therefore, broadcast is not required in this architecture.

6.1.2 Peer-to-Peer Architecture

In this architecture, the nodes use partitioning algorithms described in Chapter 5 to locally determine the next task to execute. Depending on the crawling strategy used, nodes require different amounts of information:

- To run the breath-first and the depth-first search strategies in a peer-to-peer environment, it is sufficient for the nodes to have the knowledge of application states. To run these strategies, the nodes do not require the transitions to calculate the next task to execute.

- To run the greedy and the probabilistic strategies, however, the knowledge of transitions is crucial. Both algorithms require this knowledge to find the closest task to do. In addition, the probabilistic strategy requires the probabilities that each type of transition leads, to calculate the probability that a task can lead to a new state.

- Similar to the greedy and the probabilistic strategies, to run the component-based strategy, it is required to broadcast states and transitions. In this model, however, states are defined differently. A distributed algorithm to run component-based strategy is described in more details in Chapter B.

Therefore, to run the breath-first and depth-first search strategies, only new states need to be broadcasted; and to run the greedy and probabilistic strategies both states and transitions are to be broadcasted to all nodes.

6.1.3 Notations

The following notations are used in the rest of this chapter:

- s: An application state.
- e: An event.
- S: The total number of application states.
- E_S: The number of events in the application state s.

6.2 Performance Properties and Architectural Choices

In Chapter 5 we measured the performance of several operations: communication between nodes, calculating the next task to execute, and the time it takes to execute each task. As the experimental results in the chapter show:

- The time required for communicating a state or transition between two nodes is less than a millisecond.
- The time required for calculating the next event to execute is often less than 100 milliseconds.
- The time required for executing a JavaScript event is often between 10 to 1000 milliseconds. A task is often composed of several JavaScript events, so the result of executing a task can take up to several seconds.

Given the large discrepancies between the time it takes to perform these operations, and the low network delay, two architectures can be devised:

- A client-server architecture: The time it takes to execute a task is often an order of magnitude longer than the time it takes to calculate the task. It is thus reasonable to expect a good performance if a single unit calculates the next job and dispatches them to nodes to run the jobs. It is also expected that, as the number of nodes increases, eventually this unit becomes a bottleneck and the performance of the crawler remains the same or deteriorate. For example, based on the experimental measurements presented in Chapter 5, we expect that when crawling Dyna-Table web application using this architecture, the unit becomes a bottleneck when there are 12 or more crawler nodes.

  Based on this observation, we devise the client-server-based architecture. This architecture takes advantage of the large gap between the time it takes to calculate the next task compared to executing it, and as explained does not require broadcasting of transitions for any strategy. Henceforth, we call this architecture *client-server*.

- A peer-to-peer architecture: The time it takes to broadcast a message is orders of magnitude lower than the time it takes to calculate a task or execute it. It is thus reasonable to expect a good performance from a peer-to-peer architecture where nodes broadcast states and if necessary transitions. As the number of nodes increases, the number of messages per second increases too, and the network is bound to become a bottleneck. For example, based on the experimental measurements presented in Chapter 5, we expect that when both states and transitions are broadcasted, by crawling Dyna-Table application with 434 nodes or more the network becomes a bottleneck. Before reaching to this point, however, a good performance speedup is expected with this architecture.

  Based on this observation, we devise a peer-to-peer architecture. This architecture takes advantage of low network delay and relies on broadcasting the information needed. Through elimination of any centralized unit, this architecture tries to achieve a better scalability. Henceforth, this architecture is referred to as *peer-to-peer*. We incorporate a batching mechanism to deter network from becoming a bottleneck.

6.3 Client-Server Architecture

In this architecture, all nodes transfer the knowledge of new transitions to a special node, called coordinator. The coordinator maintains the knowledge of the application graph, calculates tasks to accomplish and dispatches jobs to the nodes. In this client-server architecture, nodes contact the coordinator and ask for tasks to do. The coordinator responds with a task and the node executes it. This architecture can be used to run any algorithm on the server and dispatch 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*.

Minimally, a task can be a single JavaScript event to execute. This is the case when a node contacts the coordinator asking for work and there is an un-executed JavaScript event in the current state of the contacting node. If there is no un-executed JavaScript event in the current state, the coordinator returns a chain of JavaScript events that first lead the node to a state with an un-executed events, then have it execute an un-executed event in that state. A special event can be a reset where the node loads the seed URL and goes back to the initial state.

6.3.1 Initialization

This architecture contains several crawling nodes. The nodes do not share memory and work independently of each other. Nodes communicate with the coordinator using a client-server scheme. The crawling nodes are initialized as follow:

- The coordinator has the seed URL, and the IP address and port number for each crawler node. Using a combination of IP address and port number allows us to run multiple crawler nodes on a single computer.

- The crawler nodes have the IP address and port number of the coordinator.

When initialized, the crawler nodes contact the coordinator using its address and ask for work, and the crawling phase begins.

6.3.2 Algorithm

Initially, the coordinator sends the reset order to the crawler nodes. After loading the seed URL (i.e. the URL to reach the starting state of the RIA), a node asks for a task by sending 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 next task to execute based on the crawling strategy. For instance, in the case of the greedy strategy, the coordinator finds the closest application state to s with an unexecuted event to the current state of the node. It then constructs a path of events that the node has to take to reach s and concatenate the path with the unexecuted event in s. This path is then sent back to the probing node.

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 resetting 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. A node stays active while there is work available. If no work is available, the node becomes idle. The node stays in the Idle state until either more work becomes available or a termination order from the coordinator.

6.3.3 Termination

Detecting termination is trivial, and it has a similarity to Dist-RIA Crawler as described in Chapter 4. In this architecture the coordinator keeps track of the executed events. When the following two conditions are met the coordinator initiates the termination protocol by sending all nodes a *Terminate* order in response to a new task request:

- There is no Unassigned work in the coordinator i.e. all events in the discovered states are assigned to the nodes.

- Nodes are all in Idle state.

When the two conditions are satisfied, the coordinator concludes that there is no work available now (the first condition), and there will not be any work available in future (the second condition). Therefore it is safe to terminate the crawl.

6.4 Peer-to-Peer Architecture

Unlike the client-server architecture, nodes in the peer-to-peer architecture cannot rely on a centralized unit to collect and disseminate the information among the nodes. To run the greedy and the probabilistic strategies on the peer-to-peer architecture, it is required to broadcast the knowledge of new transitions to all nodes.

6.4.1 Initialization

In this architecture, the following information are global knowledge:

- The IP address and the port number of all nodes.
- The seed URL.
- The overlay network for broadcasting.
- The node identifier of each node. This identifier is a unique number from 0 to the number of nodes minus one.

Nodes are initialized with these information before crawling begins.

6.4.2 Algorithm

Nodes in the peer-to-peer architecture can be in four states of *Awake*, *Working*, *Idle*, and *Terminated*. Initially all nodes are in the *Awake* state. The crawl starts when the node with node identifier 0 broadcasts a message that moves all nodes to the *Working* state. In this state nodes run the crawling algorithm. Nodes find the next event to execute locally and deterministically. When a node is done with all of the events to execute, it will go to the *Idle* state. Every time the node discovers and executes an event it will broadcast the information as described earlier.

If the node has nothing to do it goes to the *Idle* state. In this state the node waits for the termination state token or a new state. If a new state becomes available, it goes back to the *Working* state. If the termination token arrives, the node runs termination algorithm to determine termination status. This algorithm is described in details in the next section.

6.4.3 Termination

Since there is no centralized unit in the peer-to-peer architecture, a peer-to-peer protocol is used to determine termination. The termination protocol runs along with the crawling algorithm throughout the crawling phase. This protocol works by passing a token called termination state token around a ring overlay network that goes through every node. The termination token contains the following objects:

- List of state IDs for discovered application states: This list has an element per discovered state. As the token goes around the ring more state IDs are added to this list.
- Number of known application states for each node: When the token visits a node, the node counts the number of application states it knows about and reflect them in this list.

Using the information stored in the token, the termination algorithm described below decides weather to pass the token to the next node, or initiate termination.

Termination Algorithm

When the token arrives at a node, the node is either in the Working or Idle state. If it is in the Working state it will continue executing tasks and hold on to the token until it goes into the Idle state. In the Idle state the node performs the following steps:

1. The node updates the token with the new application states it knows about that are not yet included in the token.

2. The node updates the number of states it knows about in the token.

3. If the status of the node is not indicated to be Idle in the token, the node updates its status to Idle in the token. This situation happens if this is the very first time the node takes the token. Initially status of all nodes is set to Active in the token. As the node goes around the ring, it can only pass a node if the node is in idle state. Thus after one round of going around the ring, all nodes status will be Idle.

4. The node loops through the list of node states in the token and if it finds at least one node that is in Working state, the node passes the token to the next node in the ring.

5. If all nodes are in Idle state in the token, the node loops through the list of number of known states to each node in the token and compares the number of known states for each node against the number of application state IDs in the token. If there is at least one node that does not know all states discovered, the node passes the token to the next node.

6. If the last two steps do not pass the token to the next node, the node concludes that crawling is over and it initiates a termination by broadcasting a termination order to all the nodes.

Proof of correctness

Let us assume that the algorithm is not correct and the termination is initiated while there are still events to execute. Without loss of generality let us assume node A initiated the termination order. Since there is at least one event to execute there is at least one node that is not idle. Let us refer to this node as node B. The termination order cannot be initiated if the token indicates that node B is not in idle state. Thus, node B was in idle state when the token visited it after node B passed the token to the next node, a message arrived to it with a new state and the node became busy. Let us call the sender of the message node C. Node C can either be one of the nodes that the token visited on its way from node B to node A, or one of the nodes outside this path.

Node C cannot be one of the nodes that the termination token visited on its way from node B to node A. If that was the case, on its visit to node C the new state would be added to the list of application states in the token, the termination order would not be initiated by node A since the number of states known by node B was lower than the number of application states known in the token. So node C is not visited by the token on its way from node B to node A.

For the same reason that was stated for node B, node C was idle at the time the token visited it and another node send it a message with a new state. The sender, henceforth referred to as node D cannot be on the way from node C to node A, for the same reason node C cannot be on the way from node B to node A.

This reasoning does not stop to node D and it continues indefinitely. Since number of nodes that are not on the way of token from the sender node to node A is finite, eventually we run out of nodes to be potential senders, and thus the initial message telling node B about a new state could have never been initiated. Thus the termination algorithm is proven to be correct by contradiction.

6.5 Conclusion

This chapter used experiments and techniques discussed in Chapter 5 to make high level decisions about possible distributed crawling architectures. Based on these decisions two new distributed architectures to crawl RIAs, a client-server and a peer-to-peer one, are described. Both architectures can incorporate any crawling strategy, including the greedy and the probabilistic model.

Chapter 7 experimentally evaluates the performance of the three crawler against each other on four different web applications.

Chapter 7

Experimental Results

In this chapter we study the relative performance of peer-to-peer and client-server architectures. Additionally, we study performance of the peer-to-peer architecture in more depth. This chapter contains the following sections:

- The testing environment and the target web applications are described in Section 7.1.
- The performance criteria for the comparison are explained in Section 7.2.
- Experimental results to compare performance of client-server architecture against peer-to-peer architecture are explained in Section 7.3:
  - Experimental results on the time it takes to finish the crawl in Section 7.3.1.
  - Experimental results on the time it takes to discover new application states in Section 7.3.2.
  - In Section 7.3.3, we discuss about the relative performance of the two architectures.
- Experimental results on running different crawling strategies on the peer-to-peer architecture are explained in Section 7.4:
  - Experimental results on the time it takes to finish the crawl in Section 7.4.1.
  - Experimental results on the time it takes to discover new application states in Section 7.4.2.

  - In Section 7.4.3, we discuss about the experimental results presented in Sections 7.4.1 and 7.4.2.

- Finally this chapter is concluded in Section 7.5.

7.1 Test-Bed

For experimental results discussed in this chapter, the nodes and the coordinator are implemented as follow:

- The JavaScript engine in the nodes are implemented using PhantomJS 1.9.2.
- Strategies are implemented in the C programming language and GCC version 4.4.7 is used to compile them.
- All crawlers, and the coordinator, use the Message Passing Interface (MPI)[114] as the communication mechanism. MPI is an open standard communication middleware developed by a group of researchers with background both in academia and industry. MPI aims at creating a communication system that is interoperable and portable across a wide variety of hardware and software platforms. Efficient, scalable, and open source implementations of MPI are available such as OpenMPI[52] and MPICH[63]. MPICH version 3.0.4 is used to implement the communication channel in our experiments.
- All nodes, as well as the coordinator in case of client-server architecture, run on Linux kernel 2.6.

The nodes are hosted on Intel® Core(TM)2 Duo CPU E8400 @ 3.00GHz and 3GB of RAM. The coordinator is hosted on Intel® Xeon® CPU X5675 @ 3.07GHz and 24GB of RAM. The communication happens over a 10 Gbps local area network. We ran each experiment three times and the presented numbers are the average of those runs.

To compare the relative performance of the crawler, we implemented all crawlers in the same programming language and used MPI as communication channel. In all cases the C programming language is used to calculate the next task to do.

In the client-server architecture, nodes only communicate with the coordinator, and coordinator calculates the next task to perform. In the peer-to-peer architecture, nodes

Table 7.1: Implementation summary of different crawling architectures

Architecture	Communication Topology	Termination Protocol	Calculation of Next Task
Client-Server	Star	By coordinator	By coordinator
Peer-to-Peer	Fully connected	Ring-based	By node

communicate directly through MPI. In the peer-to-peer architecture calculation of the next task to execute happens on each node locally.

Table 7.1 summarizes certain implementation aspects of the three architectures.

In order to compare the performance of the two architectures we crawled six different target web applications using all web crawlers. In all cases we used breath-first and Greedy search strategies. Six target applications are chosen to measure the performance of the crawler. Except for *Test-RIA*, all the target web applications are real world web applications and none of them are developed for the purpose of this thesis. The target web applications are chosen based on their size, complexity and client side features they use. They are explained in the following.

7.1.1 Test-RIA

Test-RIA (Figure 7.1) is a toy example created by our team to measure efficiency and performance of a web crawler on a simple RIA. Albeit small, this example incorporates many features of JavaScript that pose a challenge to web-crawlers, such as attached events and asynchronous calls. This application has 39 states and a total of 305 events.

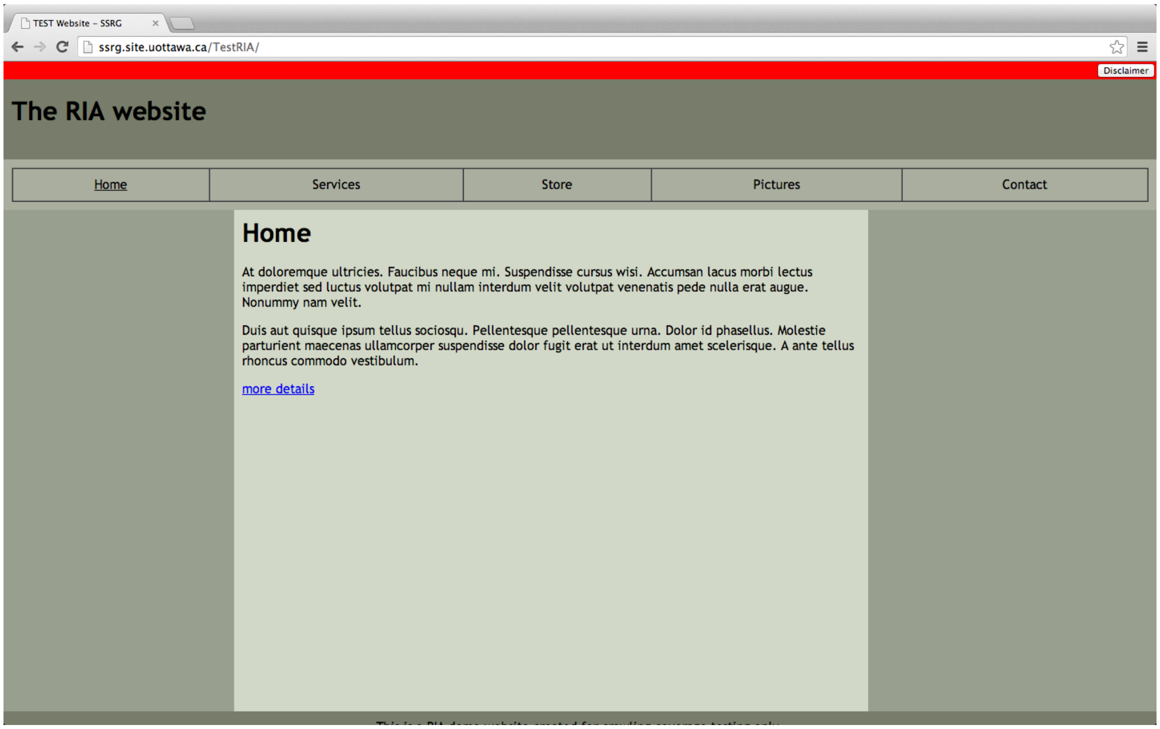
7.1.2 Altoro-Mutual

Altoro-Mutual (Figure 7.2) is a web application created by IBM to test web crawlers and security scanners. This website simulates a fake e-commerce system.

This application is developed using *ASP.NET*, runs on a Windows Operating System with IIS, and has 45 states and $1,210$ events.

7.1.3 Dyna-Table

Dyna-Table (Figure 7.3) is a real world example of a JavaScript widget, with asynchronous calls ability, that is incorporated into larger RIAs. This widget helps developer



The RIA website

Home

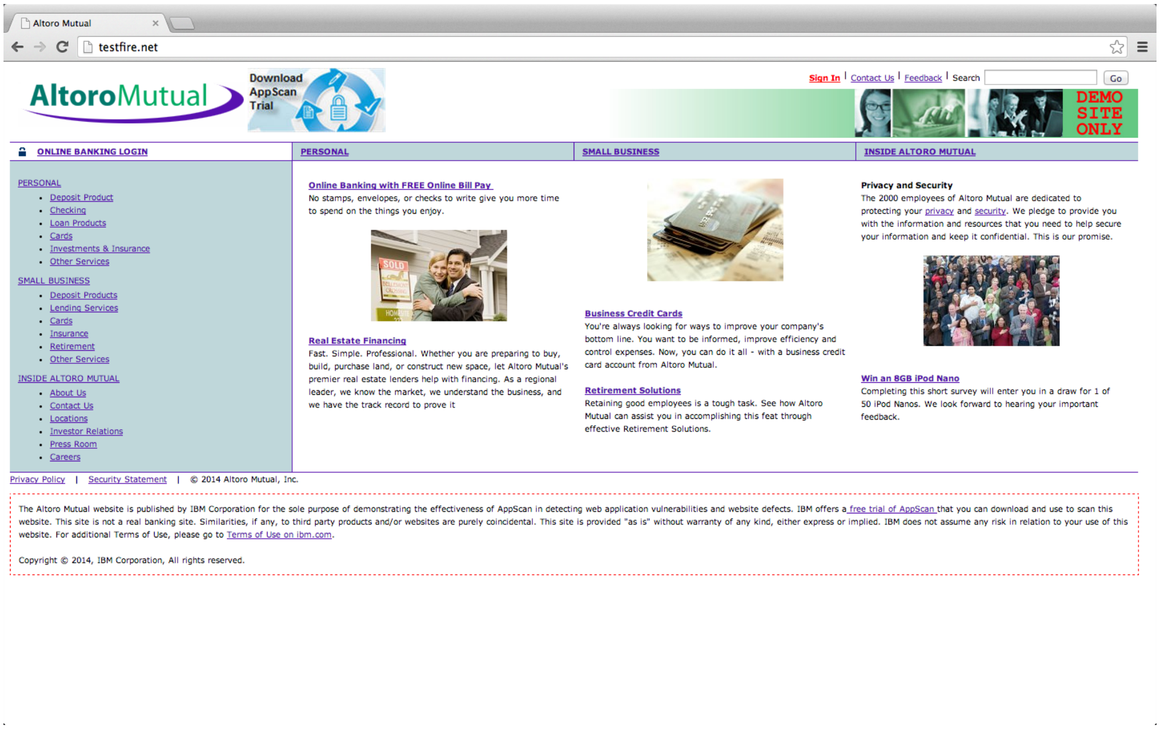
At doloremque ultricies. Faucibus neque mi. Suspendisse cursus wisi. Accumsan lacus morbi lectus imperdiet sed luctus volutpat mi nullam interdum velit volutpat venenatis pede nulla erat augue. Nonummy nam velit.

Duis aut quisque ipsum tellus sociosqu. Pellentesque pellentesque urna. Dolor id phasellus. Molestie parturient maecenas ullamcorper suspendisse dolor fugit erat ut interdum amet scelerisque. A ante tellus rhoncus commodo vestibulum.

more details

This is a RIA demo website created for crawling coverage testing only.

Figure 7.1: Test-RIA screen-shot



PERSONAL
- Deposit Product
- Checking
- Loan Products
- Cards
- Investments & Insurance
- Other Services

SMALL BUSINESS
- Deposit Products
- Lending Services
- Cards
- Insurance
- Retirement
- Other Services

INSIDE ALTORO MUTUAL
- About Us
- Contact Us
- Locations
- Investor Relations
- Press Room
- Careers

Online Banking with FREE Online Bill Pay
No stamps, envelopes, or checks to write give you more time to spend on the things you enjoy.

Real Estate Financing
Fast. Simple. Professional. Whether you are preparing to buy, build, purchase land, or construct new space, let Altoro Mutual's premier real estate lenders help with financing. As a regional leader, we know the market, we understand the business, and we have the track record to prove it

Business Credit Cards
You're always looking for ways to improve your company's bottom line. You want to be informed, improve efficiency and control expenses. Now, you can do it all - with a business credit card account from Altoro Mutual.

Retirement Solutions
Retaining good employees is a tough task. See how Altoro Mutual can assist you in accomplishing this feat through effective Retirement Solutions.

Privacy and Security
The 2000 employees of Altoro Mutual are dedicated to protecting your privacy and security. We pledge to provide you with the information and resources that you need to help secure your information and keep it confidential. This is our promise.

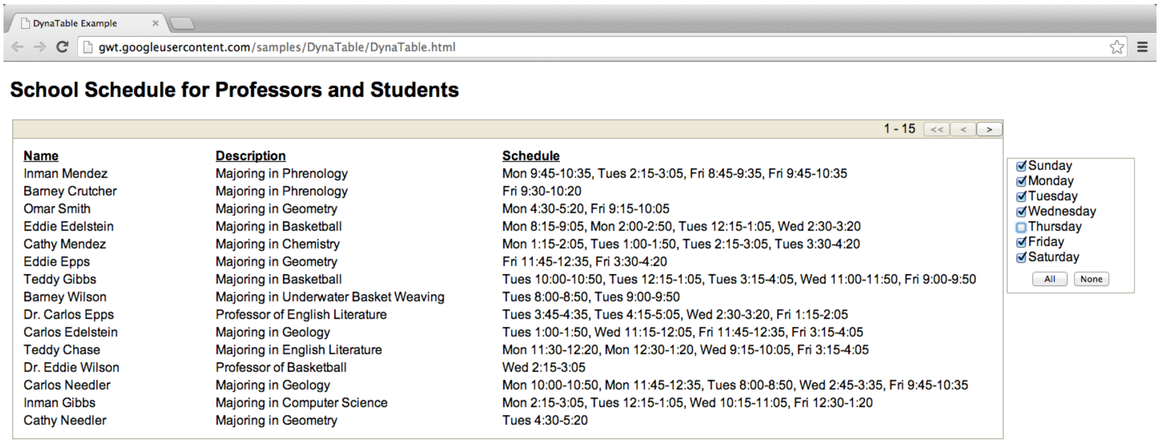
Win an 8GB iPod Nano
Completing this short survey will enter you in a draw for 1 of 50 iPod Nanos. We look forward to hearing your important feedback.

Privacy Policy | Security Statement | © 2014 Altoro Mutual, Inc.

The Altoro Mutual website is published by IBM Corporation for the sole purpose of demonstrating the effectiveness of AppScan in detecting web application vulnerabilities and website defects. IBM offers a free trial of AppScan that you can download and use to scan this website. This site is not a real banking site. Similarities, if any, to third party products and/or websites are purely coincidental. This site is provided "as is" without warranty of any kind, either express or implied. IBM does not assume any risk in relation to your use of this website. For additional Terms of Use, please go to Terms of Use on ibm.com.

Copyright © 2014, IBM Corporation, All rights reserved.

Figure 7.2: Test-RIA screen-shot



School Schedule for Professors and Students

Name	Description	Schedule
Inman Mendez	Majoring in Phrenology	Mon 9:45-10:35, Tues 2:15-3:05, Fri 8:45-9:35, Fri 9:45-10:35
Barney Crutcher	Majoring in Phrenology	Fri 9:30-10:20
Omar Smith	Majoring in Geometry	Mon 4:30-5:20, Fri 9:15-10:05
Eddie Edelstein	Majoring in Basketball	Mon 8:15-9:05, Mon 2:00-2:50, Tues 12:15-1:05, Wed 2:30-3:20
Cathy Mendez	Majoring in Chemistry	Mon 1:15-2:05, Tues 1:00-1:50, Tues 2:15-3:05, Tues 3:30-4:20
Eddie Epps	Majoring in Geometry	Fri 11:45-12:35, Fri 3:30-4:20
Teddy Gibbs	Majoring in Basketball	Tues 10:00-10:50, Tues 12:15-1:05, Tues 3:15-4:05, Wed 11:00-11:50, Fri 9:00-9:50
Barney Wilson	Majoring in Underwater Basket Weaving	Tues 8:00-8:50, Tues 9:00-9:50
Dr. Carlos Epps	Professor of English Literature	Tues 3:45-4:35, Tues 4:15-5:05, Wed 2:30-3:20, Fri 1:15-2:05
Carlos Edelstein	Majoring in Geology	Tues 1:00-1:50, Wed 11:15-12:05, Fri 11:45-12:35, Fri 3:15-4:05
Teddy Chase	Majoring in English Literature	Mon 11:30-12:20, Mon 12:30-1:20, Wed 9:15-10:05, Fri 3:15-4:05
Dr. Eddie Wilson	Professor of Basketball	Wed 2:15-3:05
Carlos Needler	Majoring in Geology	Mon 10:00-10:50, Mon 11:45-12:35, Tues 8:00-8:50, Wed 2:45-3:35, Fri 9:45-10:35
Inman Gibbs	Majoring in Computer Science	Mon 2:15-3:05, Tues 12:15-1:05, Wed 10:15-11:05, Fri 12:30-1:20
Cathy Needler	Majoring in Geometry	Tues 4:30-5:20

☑ Sunday
☑ Monday
☑ Tuesday
☑ Wednesday
☐ Thursday
☑ Friday
☑ Saturday

Figure 7.3: Test-RIA screen-shot

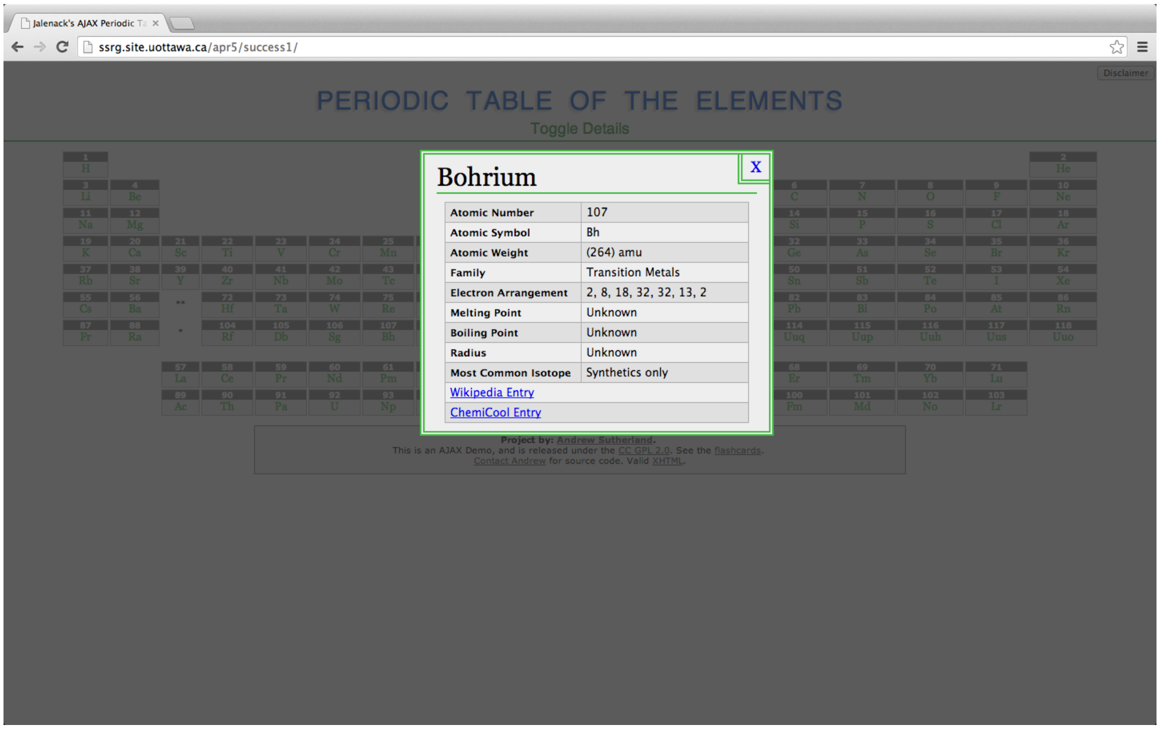
to handle large interactive tables. It allows to show a fixed number of rows per page, to navigate through different pages, to filter content of a table based on given criteria, and to sort the rows based on different fields.

This application was developed using the *Google Web Toolkit* and has 448 states and a total of $5,380$ events.

7.1.4 Periodic-Table

This educational open source application creates an interactive periodic table (Figure 7.4). This application allows the user to click on each element and show the user information about the element in a pop-up window. The application can display the periodic table in two modes: the small mode, and the large mode. The two modes are identical in terms of functionality, however they offer two very different interfaces. Once an element is clicked and the pop-up window shows up for the element, other elements can be clicked or the pop-up window can be closed.

This application is the largest target application we use. It is developed in PHP and JavaScript, and it has 240 states and a total of $29,040$ events.


Jalenack's AJAX Periodic Ta
ssrg.site.uottawa.ca/apr5/success1/
Disclaimer
PERIODIC TABLE OF THE ELEMENTS
Toggle Details
X

Bohrium

Atomic Number	107
Atomic Symbol	Bh
Atomic Weight	(264) amu
Family	Transition Metals
Electron Arrangement	2, 8, 18, 32, 32, 13, 2
Melting Point	Unknown
Boiling Point	Unknown
Radius	Unknown
Most Common Isotope	Synthetics only
Wikipedia Entry	
ChemiCool Entry	

Project by: Andrew Sutherland.
This is an AJAX Demo, and is released under the CC GPL 2.0. See the flashcards.
Contact Andrew for source code. Valid XHTML.

Figure 7.4: Test-RIA screen-shot

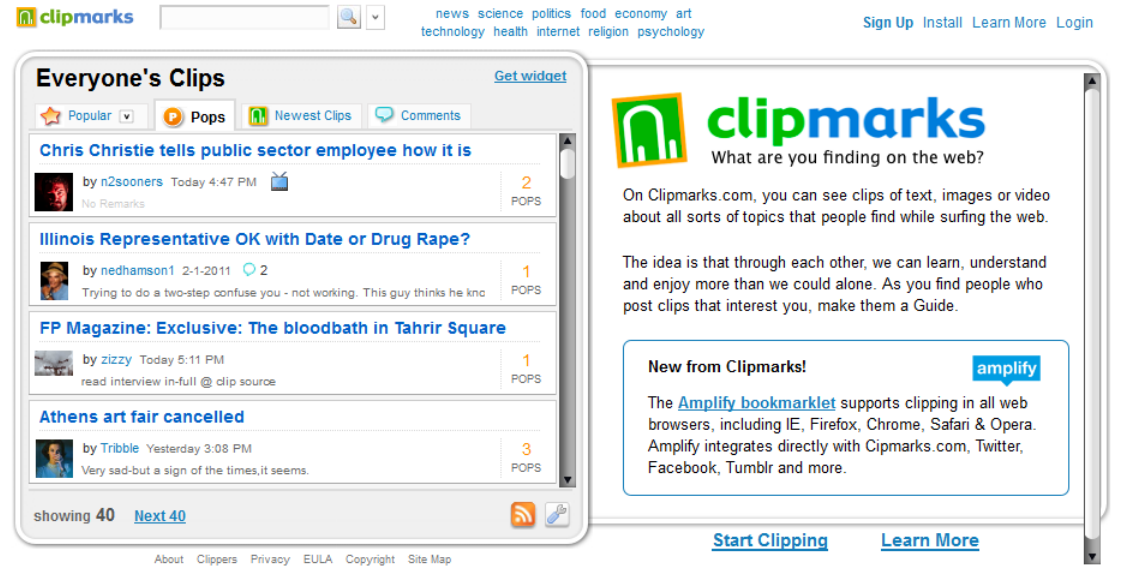
7.1.5 ClipMarks

ClipMarks is a RIA that allows the user to share contents from other websites, with other members of the ClipMarks (Figure 7.5). List of shared contents (called *Clips*) is available on the first page of the application. Users can vote on a clip, and the number of users who have voted on a clip shows up next to the clip. By clicking on the number, the list users who voted to the clip pops-up. Items within this pop-up window are linked to individual user profiles. By going to a user profile, it is possible to see the list of clips shared by the user.

This application has 129 states and a total of 10, 580 events.

7.1.6 Elfinder

Elfinder is an open source and interactive RIA file-browser (Figure 7.6). This application allows the user to interact with individual files and folders on the server. The user can create, modify, delete and view files and folders. To simplify the application, and avoid state explosion, we disabled the application from modifying and deleting contents on the server. This application has 1, 360 states and a total of 43, 816 events.



clipmarks

news science politics food economy art
technology health internet religion psychology

Sign Up Install Learn More Login

Everyone's Clips

Get widget

Popular Pops Newest Clips Comments

Chris Christie tells public sector employee how it is
by n2sooners Today 4:47 PM
No Remarks
2 POPS

Illinois Representative OK with Date or Drug Rape?
by nedhamson1 2-1-2011 2
Trying to do a two-step confuse you - not working. This guy thinks he knc
1 POPS

FP Magazine: Exclusive: The bloodbath in Tahrir Square
by zizzy Today 5:11 PM
read interview in-full @ clip source
1 POPS

Athens art fair cancelled
by Tribble Yesterday 3:08 PM
Very sad-but a sign of the times,it seems.
3 POPS

showing 40 Next 40

About Clippers Privacy EULA Copyright Site Map

clipmarks
What are you finding on the web?

On Clipmarks.com, you can see clips of text, images or video about all sorts of topics that people find while surfing the web.

The idea is that through each other, we can learn, understand and enjoy more than we could alone. As you find people who post clips that interest you, make them a Guide.

New from Clipmarks! amplify

The Amplify bookmarklet supports clipping in all web browsers, including IE, Firefox, Chrome, Safari & Opera. Amplify integrates directly with Cipmarks.com, Twitter, Facebook, Tumblr and more.

Start Clipping Learn More

Figure 7.5: ClipMarks screen-shot

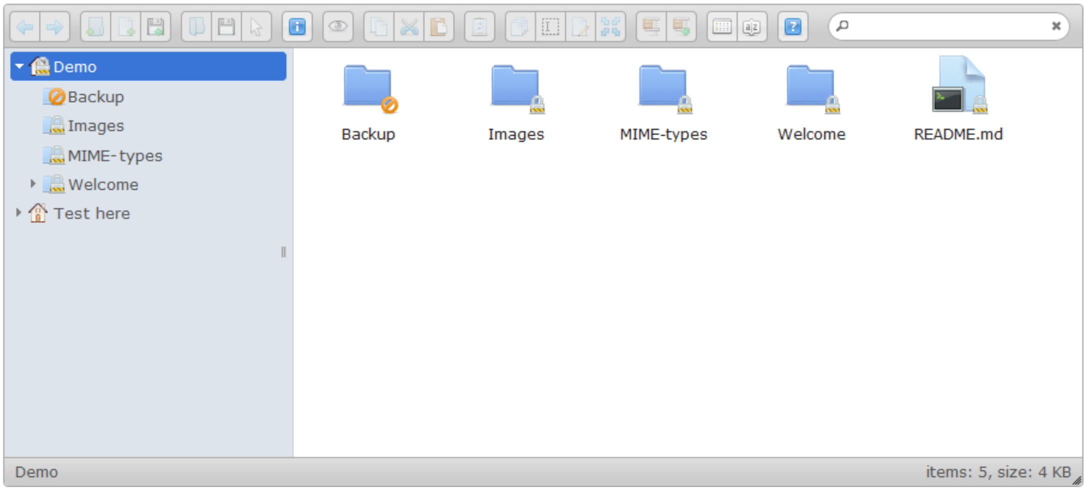


Figure 7.6: Elfinder screen-shot

Table 7.2: Target applications graph summary

Application Name	Number of States	Number of Transitions	Minimum Events per Page	Maximum Events per Page	Average Events per Page
Test-RIA	39	305	5	10	7.82
Altra-Mutual	45	1,210	3	34	26.89
Dyna-Table	448	5,380	12	13	12.01
Periodic-Table	240	29,040	119	122	121.00
Clipmarks	129	10,581	69	129	82.02
Elfinder	1360	43,817	15	1359	32.22

7.1.7 Summary of Test Applications

Table 7.2 shows some information about the graph of the target web applications. As the table shows, the four test beds represent four category of web applications:

1. Test-RIA represents small size RIAs. These applications have a small number of states, and a small number of events per page. Also the number of events per page do not change significantly from page to page.

2. Altoro-Mutual represents a medium size RIA. This application has a larger number of events per page than Test-RIA, however, there are pages where the number of events are lower than the number of web crawlers we have used.

3. Dyna-Table shows a large size RIA with a small number of events per page.

4. Periodic-Table, ClipMarks, and Elfinder show large size RIAs with a large number of events per page. In these application all states have more events per page than the number of web crawlers and the application graph is dense.

7.2 Comparison Criteria

We have compared the performance of the crawlers in two ways:

1. The total time of the crawl: This factor measures how fast the crawler can explore the entire application graph and execute all transitions. Ultimately the focus of

this thesis is to reduce the time it takes to crawl RIAs and this factor comes first, naturally.

2. Efficiency of the crawler in terms of state discovery: This factor measures how quickly the crawler discovers new states. Early discovery of states have many important implications. Firstly, in case of a partial crawl it is useful to have as many states as early as possible in the crawl. Secondly, discovering states early in the crawl increases the chance of finding faulty states earlier. Lastly, discovering states early in the crawl helps preventing nodes from being idle while other nodes are discovering new states. Efficiently of state discovery depends largely on the crawling strategy[36].

We measure the performance of the two architectures using two crawling strategies:

1. Breath-First search strategy: In this strategy, nodes perform a breath-first search from the root. If during event execution the application graph changes by other nodes (i.e. new states are discovered or new transitions are added due to other nodes), the node starts a new breath-first search from the root.

2. Greedy strategy[102]: In this strategy nodes start by performing the greedy algorithm from the root. After performing the first task, when a node requires the calculation of the next task to execute, the latest known application graph is used to perform a greedy algorithm from the current state of the application, and this step is repeated until the node terminates. Similar to the breath-first search strategy, JavaScript events on the page are partitioned based on their location on the DOM.

7.3 Experimental Results: Client-Server versus Peer-to-Peer

7.3.1 Time to finish crawl

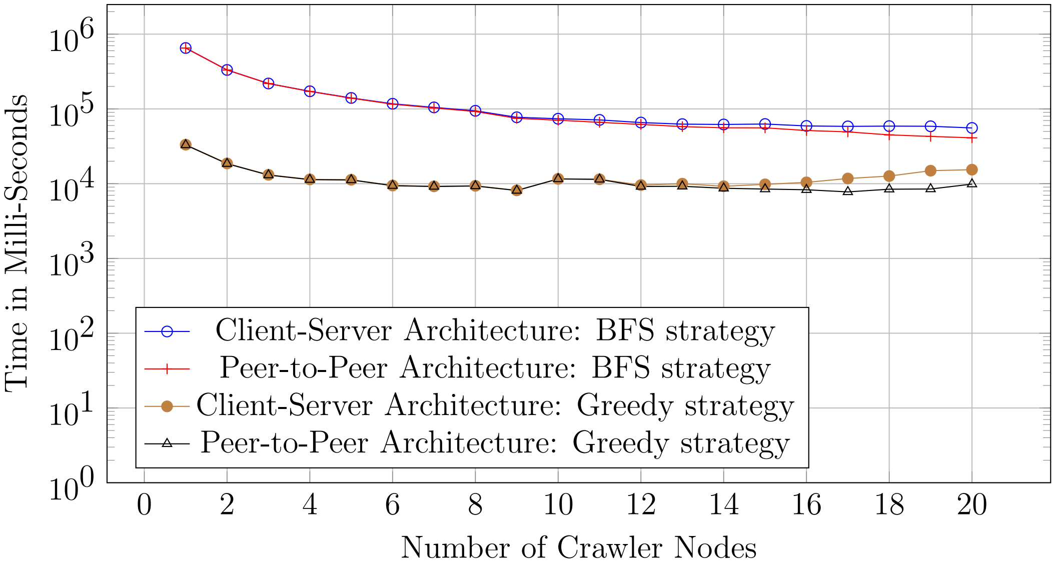


Figure 7.7: The total time to crawl Test-RIA in parallel using different architectures

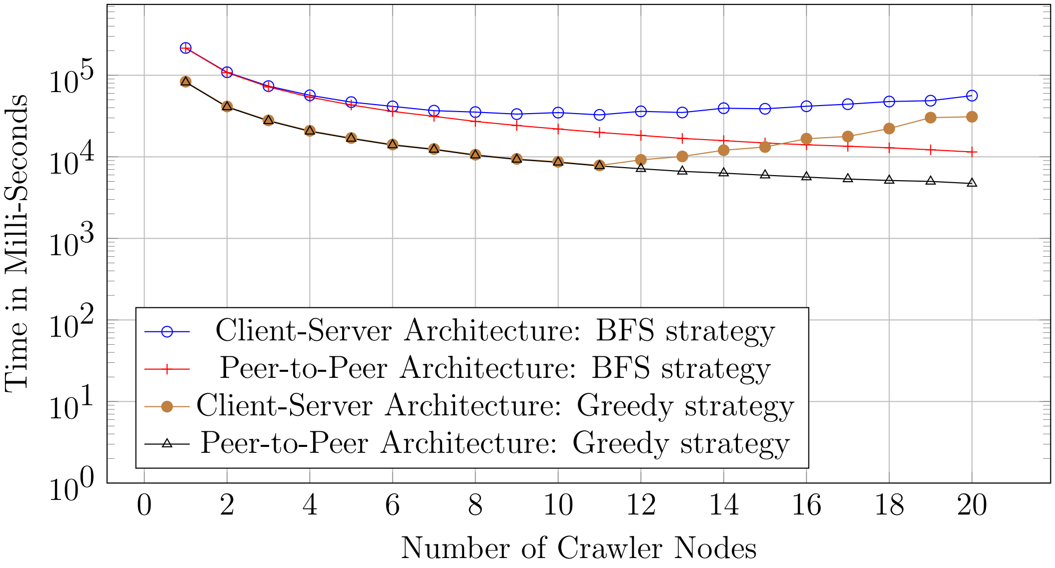


Figure 7.8: The total time to crawl Altra-Mutual in parallel using using different architectures

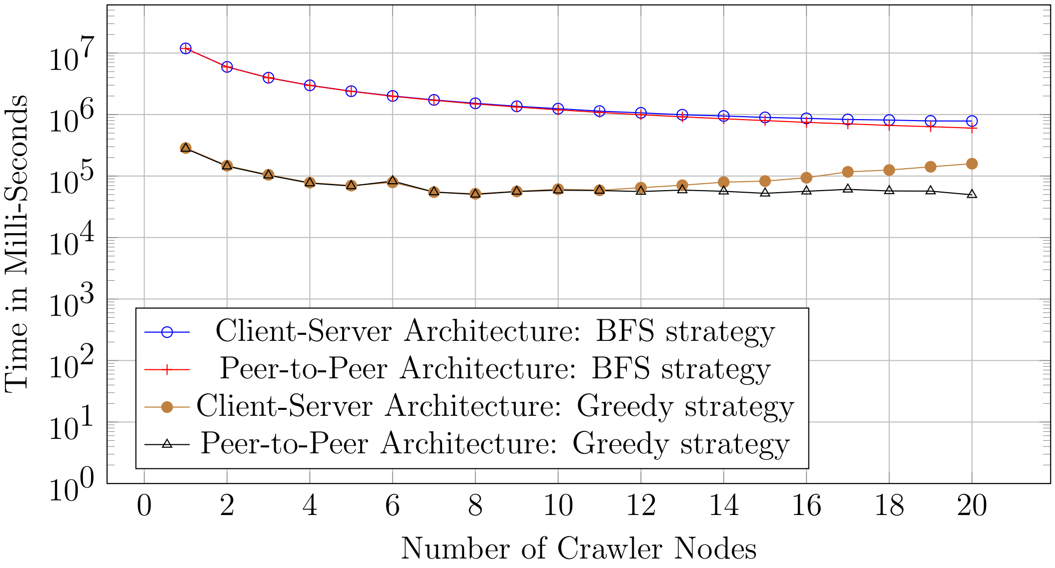


Figure 7.9: The total time to crawl Dyna-Table in parallel using using different architectures

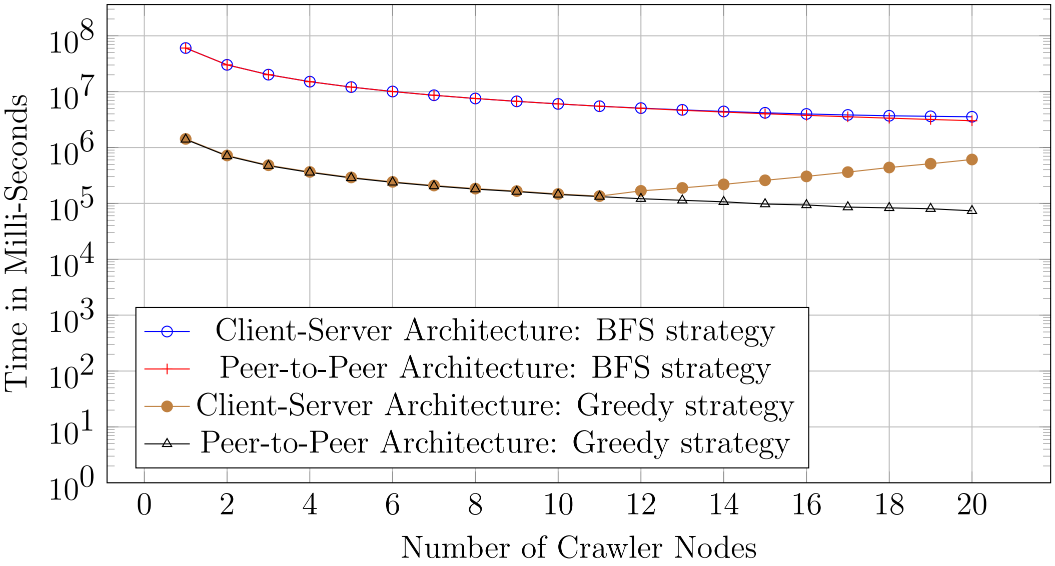


Figure 7.10: The total time to crawl Periodic-Table in parallel using using different architectures

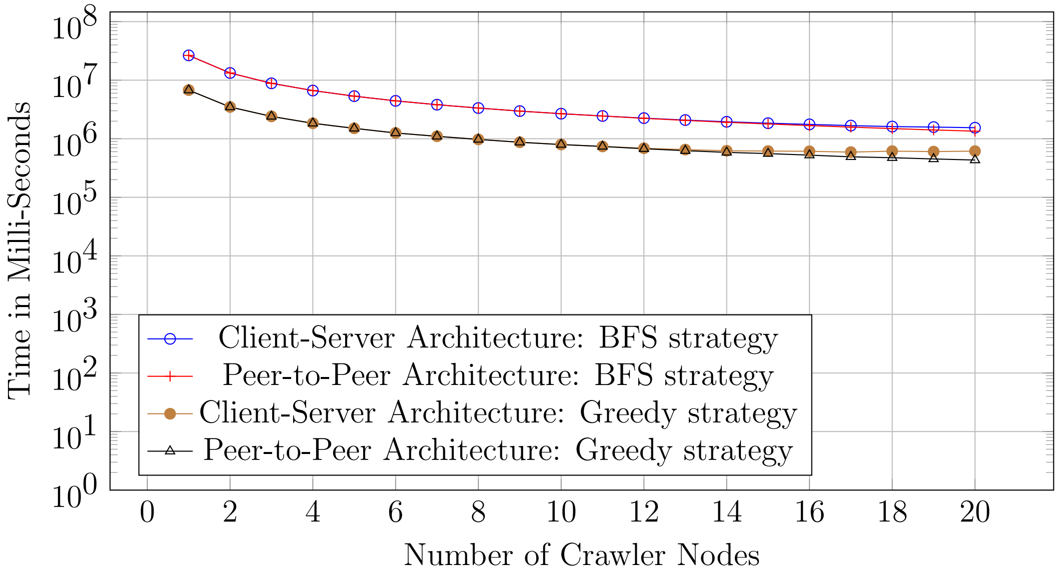


Figure 7.11: The total time to crawl Clipmarks in parallel using using different architectures

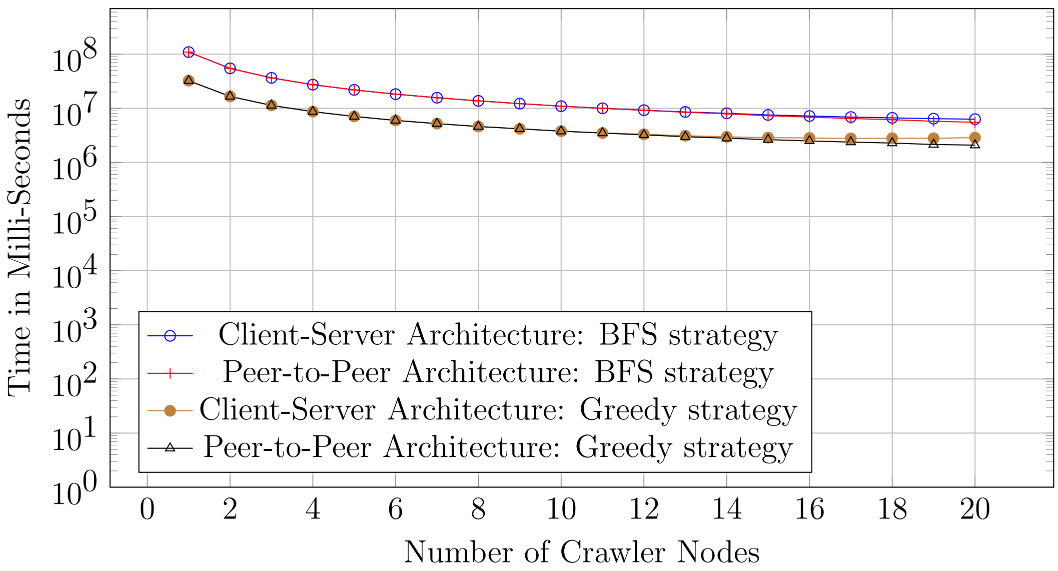


Figure 7.12: The total time to crawl Elfinder in parallel using using different architectures

7.3.2 Time to Discover New States

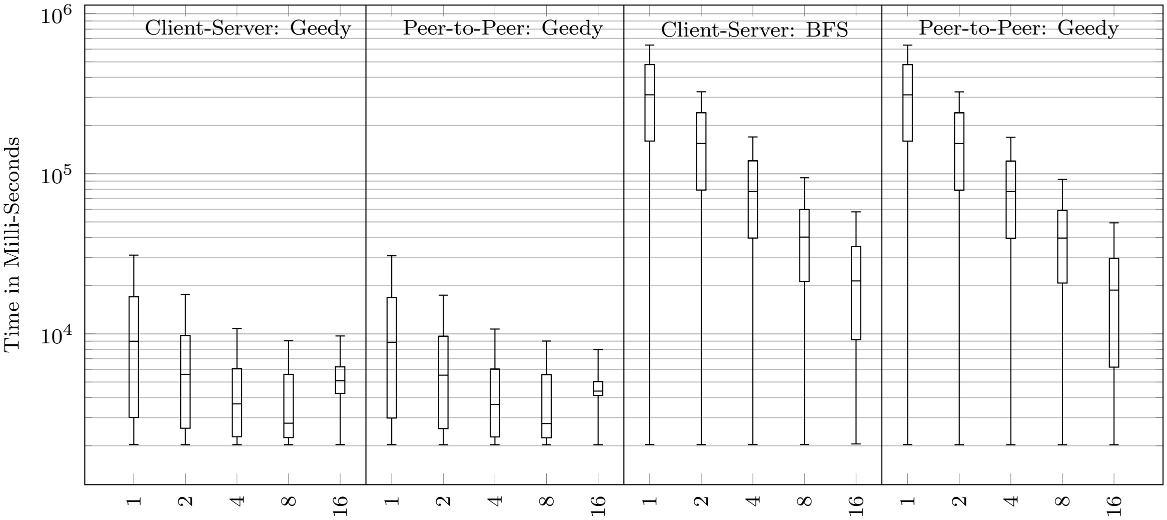


Figure 7.13: Cost of discovering Test-RIA application states using different architectures

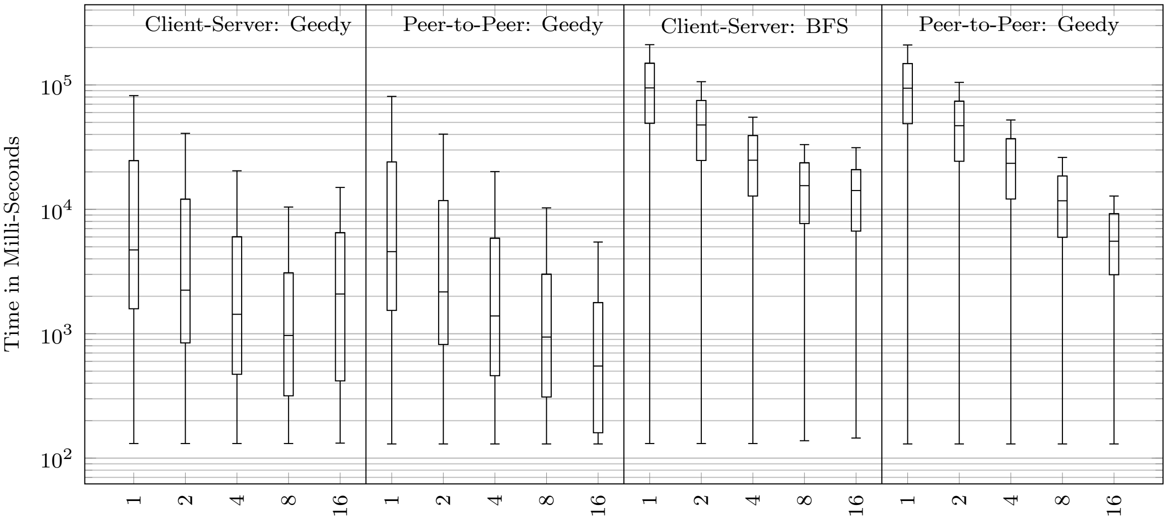


Figure 7.14: Cost of discovering Altra-Mutual application states using different architectures

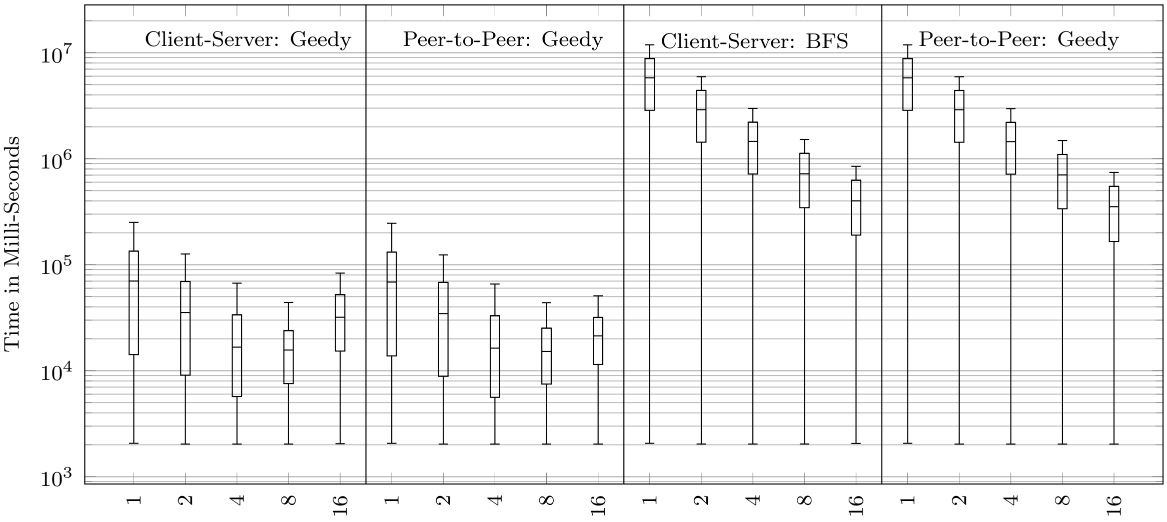


Figure 7.15: Cost of discovering Dyna-Table application states using different architectures

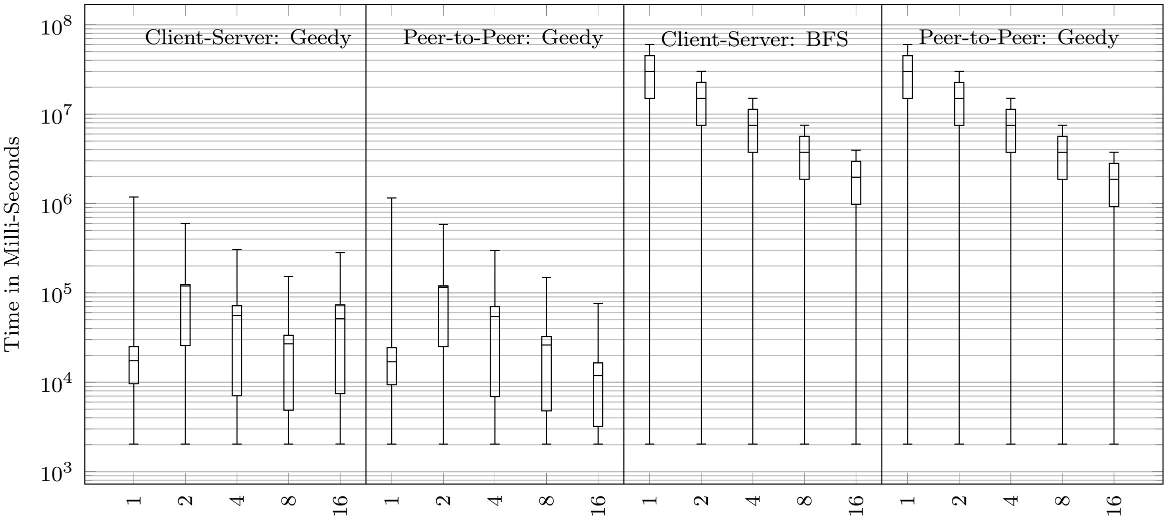


Figure 7.16: Cost of discovering Periodic-Table application states using different architectures

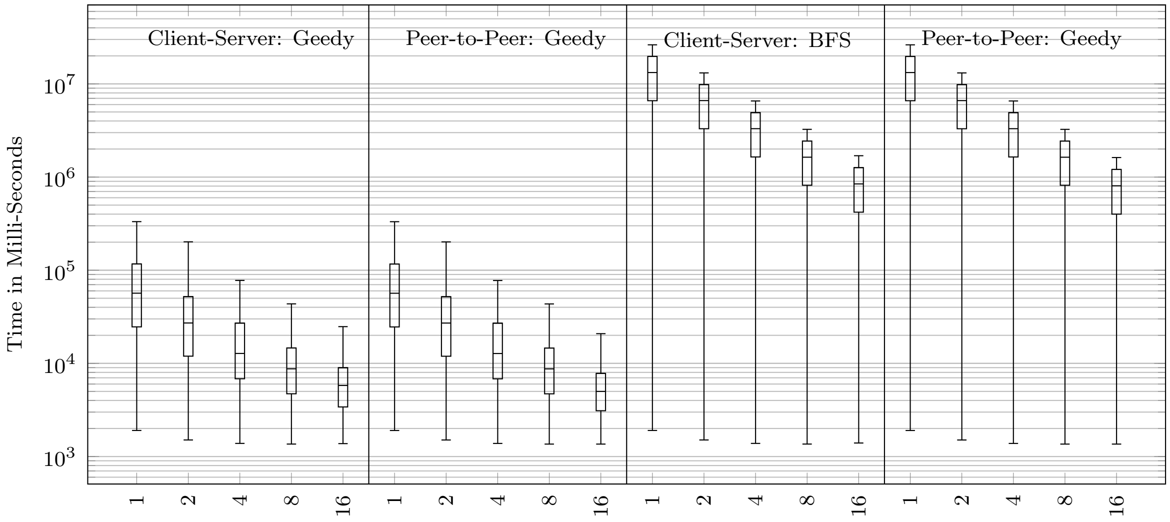


Figure 7.17: Cost of discovering Clipmarks application states using different architectures

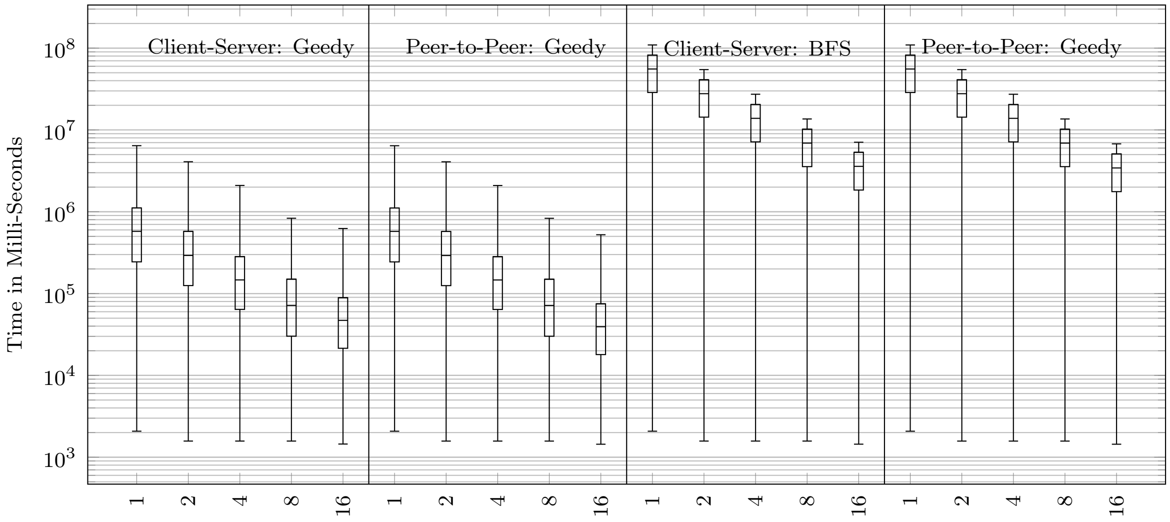


Figure 7.18: Cost of discovering Elfinder application states using different architectures

7.3.3 Discussion

Figures 7.7, 7.8, 7.9, 7.10, 7.11 and 7.12 show the time it takes to crawl Test-RIA, Altra-Mutual, Dyna-Table, Periodic-Table, Clipmarks and Elfinder web applications using different architectures. Table 7.3 summarizes these experimental results. As the figures show:

Table 7.3: Speedup achieved using peer-to-peer architecture with 20 nodes, compared to client-server architecture

Application Name	BFS Strategy Speedup	Greedy Strategy Speedup
Test-RIA	1.36	1.56
Altra-Mutual	4.90	6.58
Dyna-Table	1.30	3.20
Periodic-Table	1.18	8.28
Clip-Marks	1.15	1.42
elfinder	1.14	1.38

- Peer-to-Peer architecture scales better than the Client-Server architecture. The coordinator in Client-Server architecture becomes a bottleneck as the number of nodes increases and reaches 11 to 15 nodes.

- In both architectures, distributed greedy strategy outperforms distributed breath-first search strategy. This observation is compatible with the experimental results presented by Dincturk et al.[36].

- As the number of nodes increases, performance of Client-Server architecture suffers more in smaller web application, compared to the larger ones. This is the case, because in the larger applications, calculated tasks are more time consuming and the coordinator has more time to calculate next task to execute.

7.4 Detailed Experimental Results for the Peer-to-Peer Architecture

As explained in Section 7.3, the peer-to-peer architecture is superior to the client-server architecture. This section studies peer-to-peer architecture in more details. In addition to the greedy and breath-first search strategies, in this section we experiment with depth-first and probabilistic search strategies:

- Depth-First search strategy: In this strategy, nodes perform a depth-first search from the root. If during event execution the application graph changes a new depth-first search is began from the root. Similar to breath-first search strategy, JavaScript events on the page are partitioned based on their location in the DOM.

- Probabilistic strategy[38]: In this strategy nodes start by performing the probabilistic algorithm from the root. Similar to the greedy algorithm, after performing the first task, the node always uses the latest known application graph to calculate the next task to execute from the current state of the application. Unlike other strategies, in this strategy, JavaScript events are assigned to the nodes based on their type. Because an event type is always assigned to the same node, the node has the history of event execution. By having this history, the node can calculate the probability of reaching a new state by executing the event autonomously.

7.4.1 Time to finish crawl

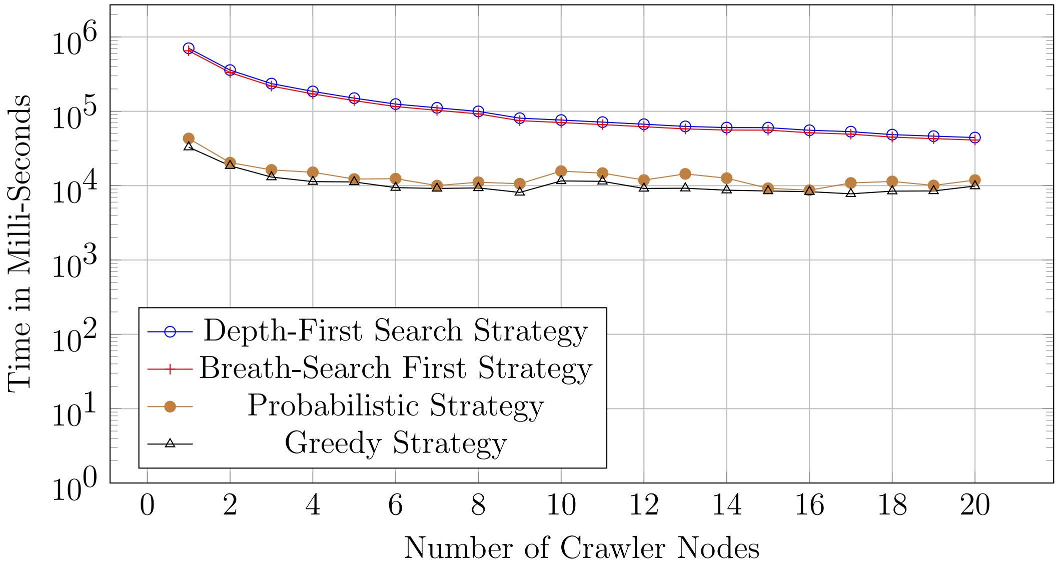


Figure 7.19: The total time to crawl the Test-RIA with multiple nodes in peer-to-peer architecture.

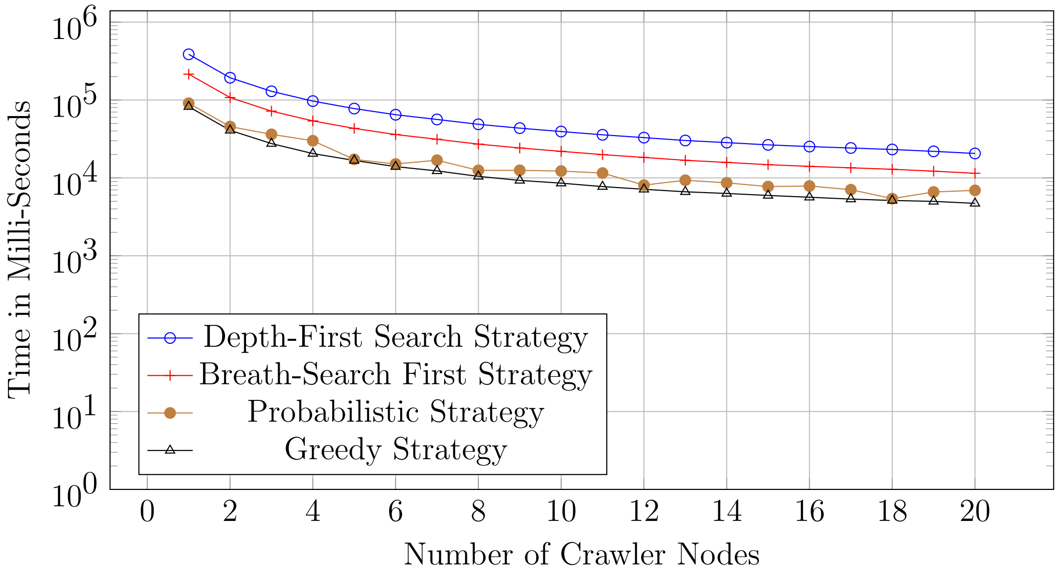


Figure 7.20: The total time to crawl the Altra-Mutual with multiple nodes in peer-to-peer architecture.

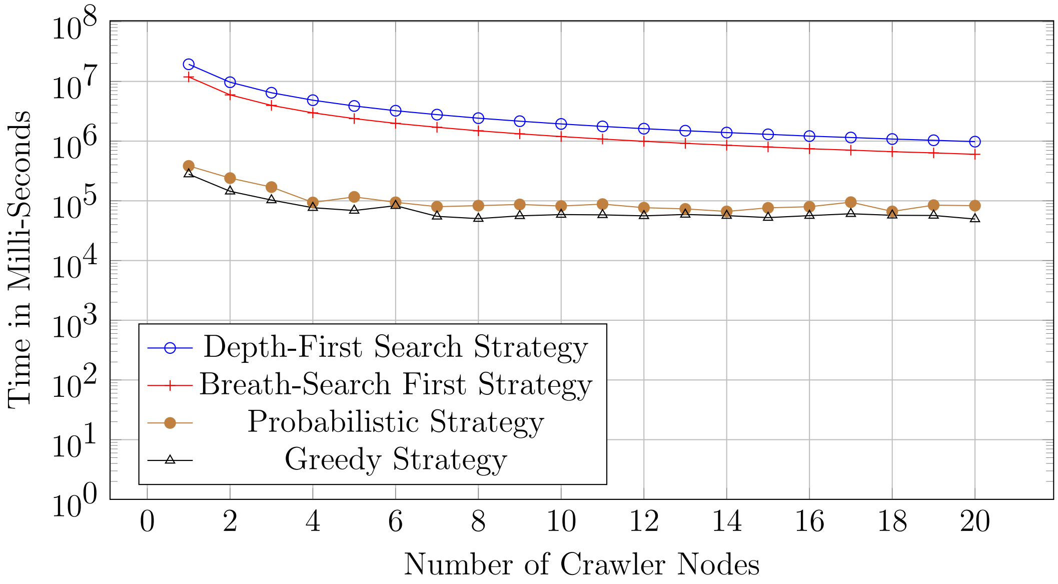


Figure 7.21: The total time to crawl the Dyna-Table with multiple nodes in peer-to-peer architecture.

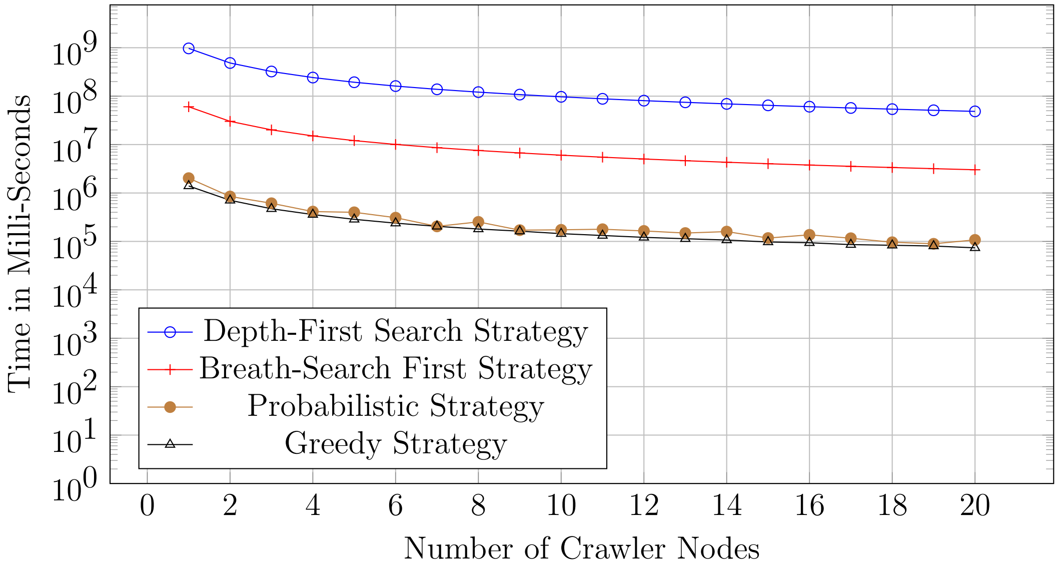


Figure 7.22: The total time to crawl the Periodic-Table with multiple nodes in peer-to-peer architecture.

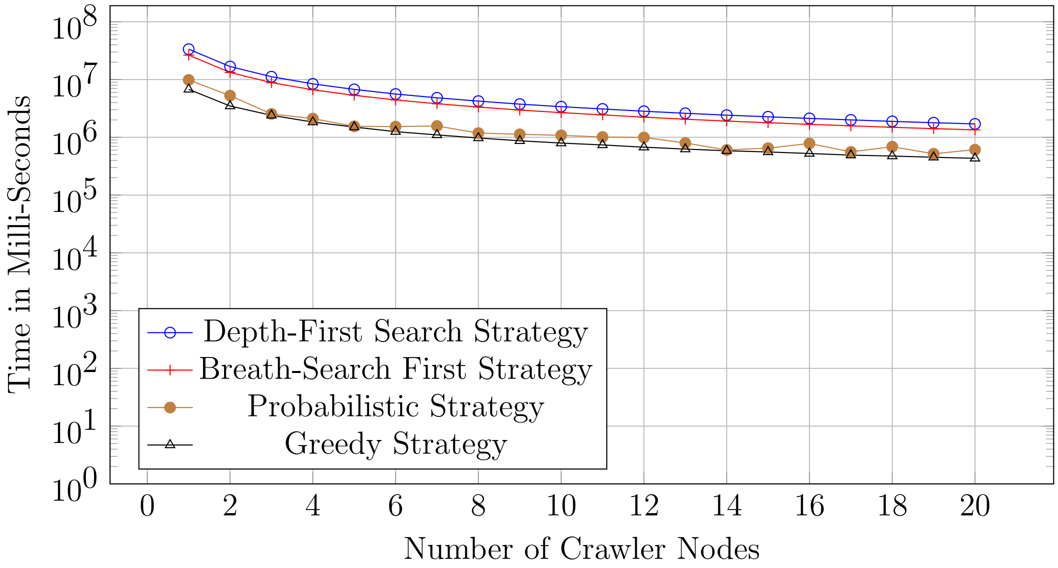


Figure 7.23: The total time to crawl the Clipmarks with multiple nodes in peer-to-peer architecture.

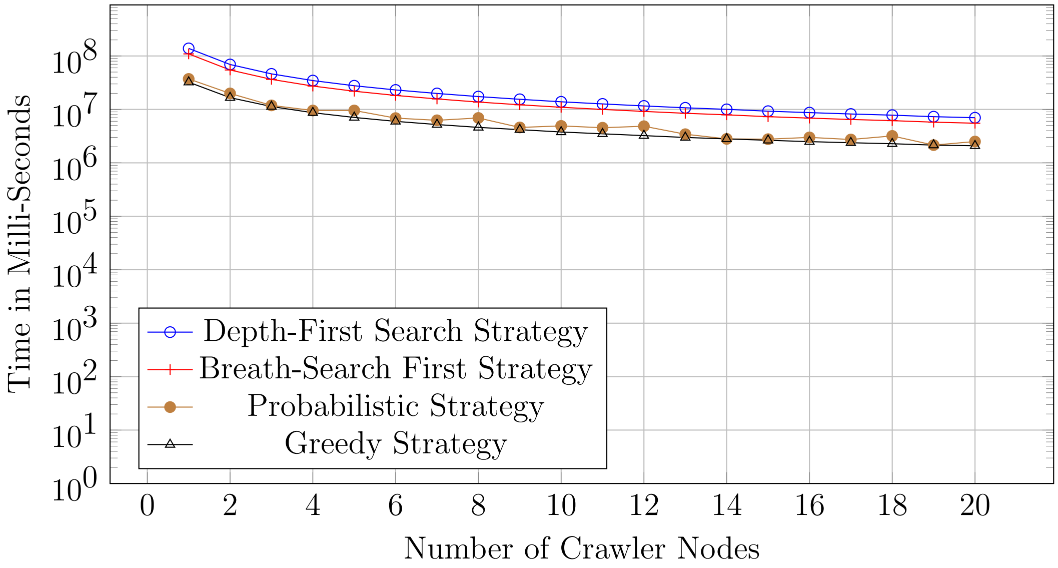


Figure 7.24: The total time to crawl the Elfinder with multiple nodes in peer-to-peer architecture.

7.4.2 Time to Discover New States

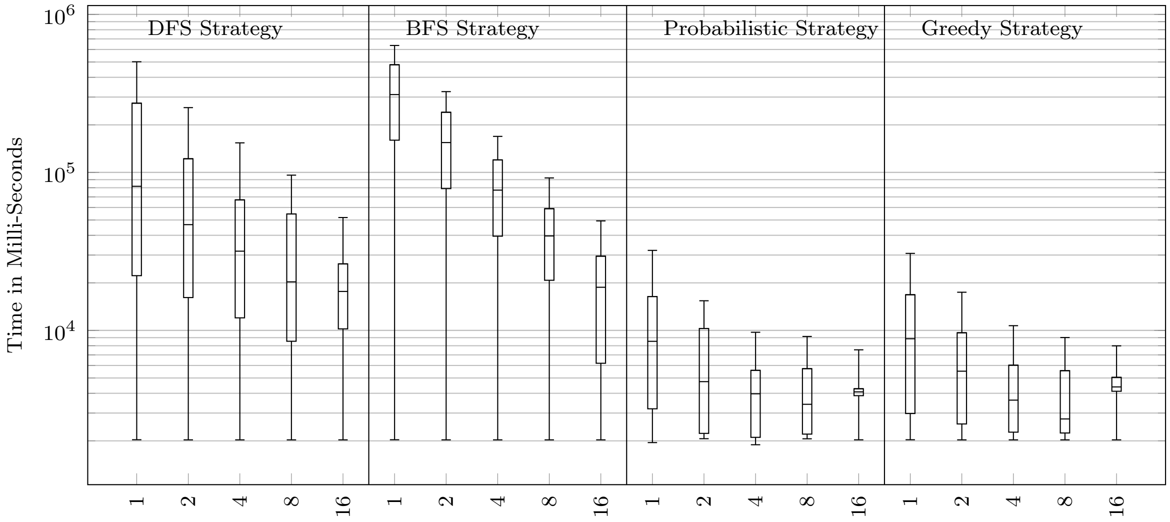


Figure 7.25: Cost of discovering Test-RIA application states using the peer-to-peer architecture

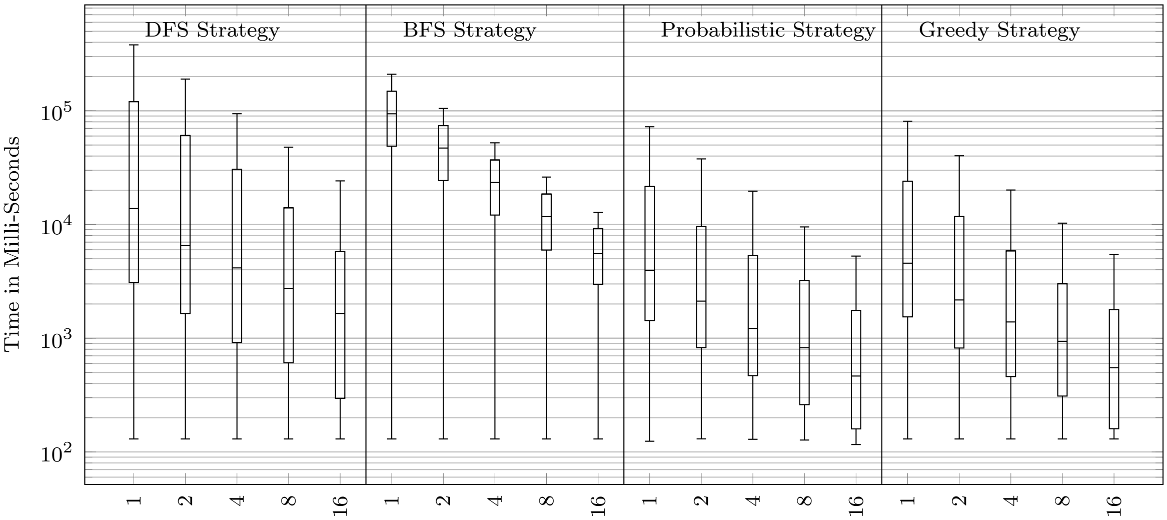


Figure 7.26: Cost of discovering Altra-Mutual application states using the peer-to-peer architecture

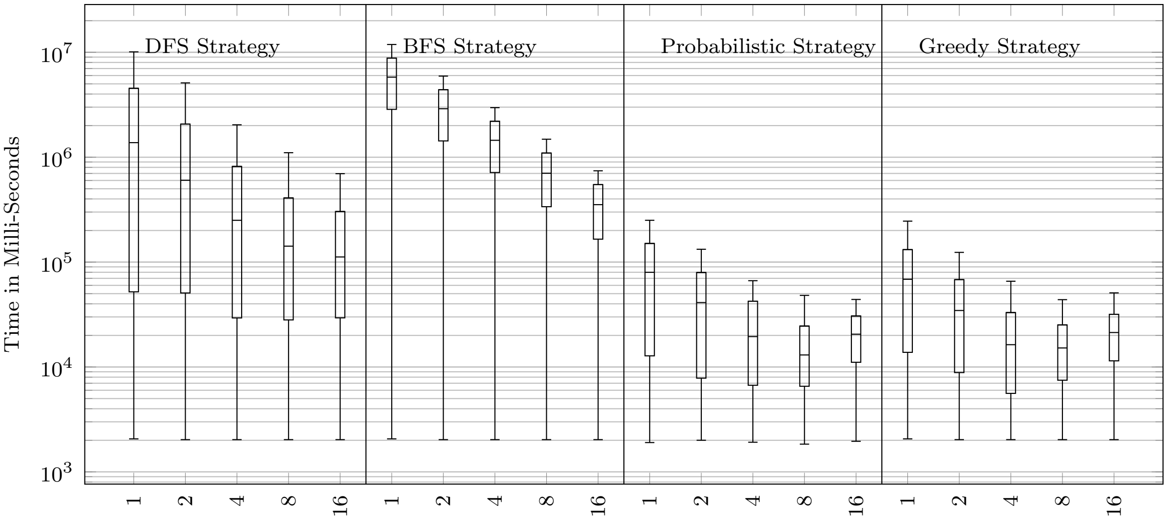


Figure 7.27: Cost of discovering Dyna-Table application states using the peer-to-peer architecture

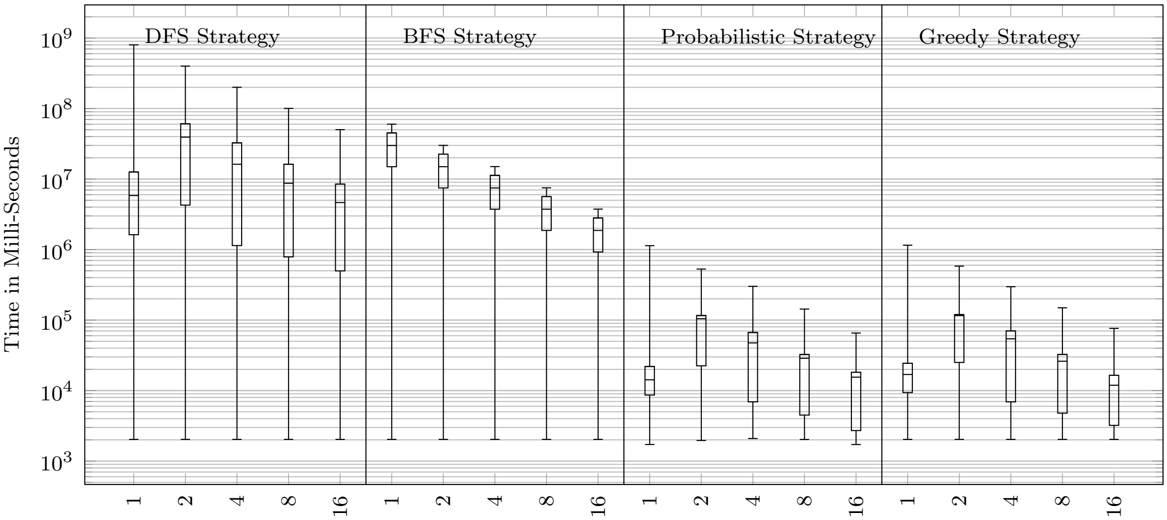


Figure 7.28: Cost of discovering Periodic-Table application states using the peer-to-peer architecture

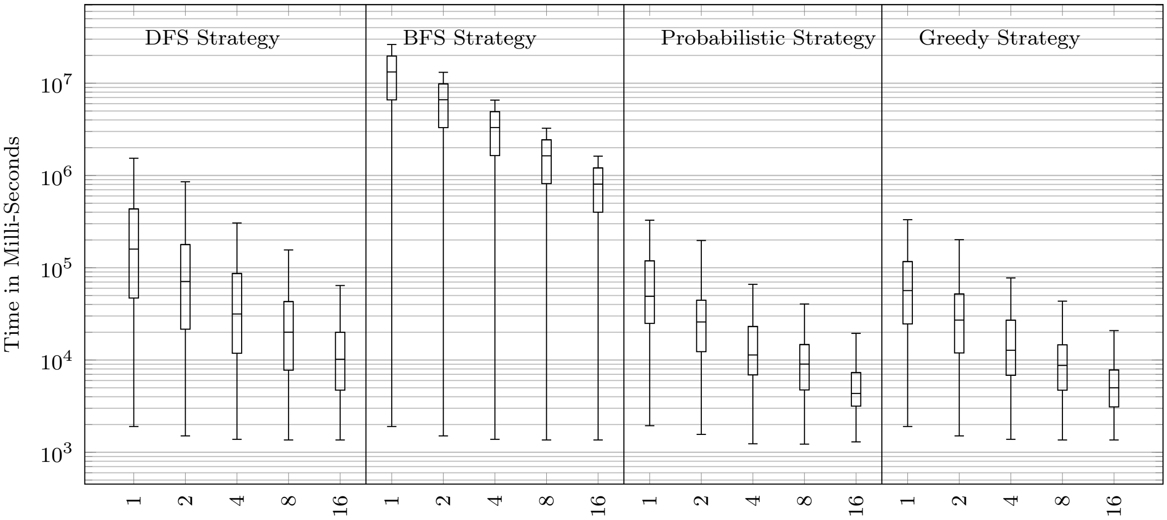


Figure 7.29: Cost of discovering Clipmarks application states using the peer-to-peer architecture

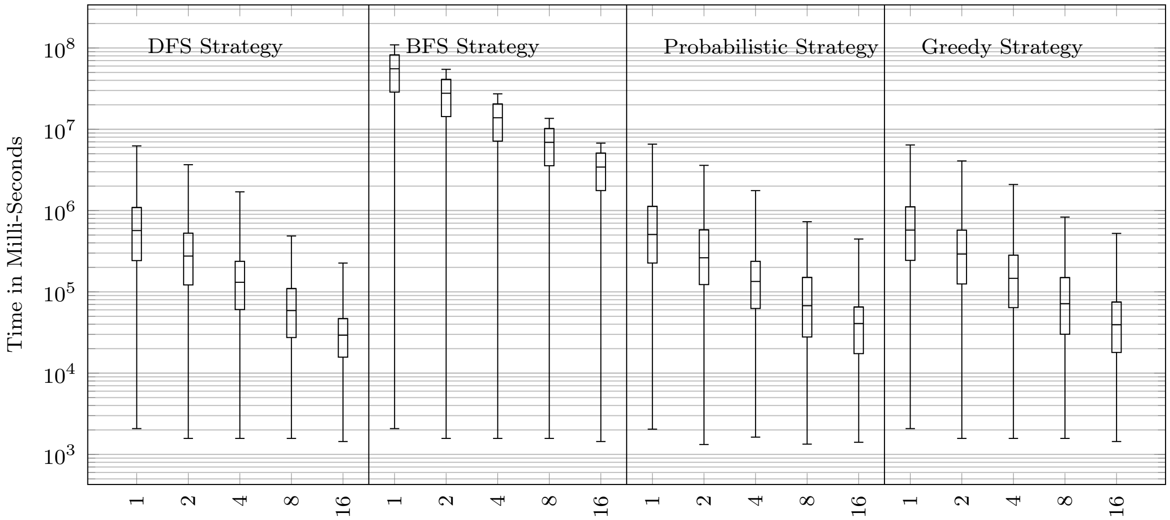


Figure 7.30: Cost of discovering Elfinder application states using the peer-to-peer architecture

7.4.3 Discussion

Figures 7.19, 7.20, 7.21, 7.22, 7.23 and 7.24 show the time it takes crawl Test-RIA, Altra-Mutual, Dyna-Table, Periodic-Table, Clipmarks and Elfinder web application using the peer-to-peer architecture with different crawling strategies. As the figures show, performance of crawling strategies is not degraded as number of nodes increases. Similar to the results presented by Dincturk[36], the greedy strategy outperforms other strategies.

Figures 7.25, 7.26, 7.27, 7.28, 7.29, and 7.30, show the time it takes to discover new application states using the peer-to-peer architecture. As the figures show, in most cases by increasing the number of crawler nodes, the time it takes to discover new applications states, reduces. This is not always the case, however, and in some cases (such as the case when crawling Periodic-Table using the probabilistic or greedy strategy), one node is more efficient than two nodes in discovering application states early. The greedy and the probabilistic strategies, look for the closest un-executed task. This happens since, in rare cases, partitioning algorithm allocates events, that lead to the discovery of new application states, to nodes that are busy executing events that do not lead to the discovery of new application states. This in-efficiency caused by the partition algorithm does not happen when crawling the web applications with one node, since all events are allocated to the node itself.

We expect that the peer-to-peer architecture scales well for the large applications. In the test examples presented, the last two test beds (i.e. Clipmarks and Elfinder), are good representations of what to expect when crawling large web applications with the peer-to-peer architecture. These two applications satisfy our assumption on number of events per page, and they have a large number of states. As our experimental results suggest, in such cases, the three strategies of greedy, breath-first and depth-first search, scale well on the peer-to-peer architecture, and a constant speedup (both on the time it takes to finish the crawl, and the time it takes to discover new application states) is observed as the number of nodes increases.

7.5 Conclusion

This chapter studied the relative performance of of the Client-Server and Peer-to-Peer architectures. To measure the performance of the crawlers against each other, all crawlers are implemented with MPI communication channel and the same programming languages. As the comparison between the performance of the crawlers show, the peer-to-peer architecture scales better than the client-server architecture. The chapter then experimented with peer-to-peer architecture in more details.

Next chapter talks about further increasing the efficiency of the crawlers by incorporating load-balancing mechanisms.

Chapter 8

Conclusion and Future Directions

RIAs are the new generation of web applications that shift part of the computation from the server to the client. This shift improves web applications by making them more interactive and user friendly. It further reduces the server workload and increases the scalability of the web applications. Inclusion of client-side events comes with a cost: Crawling web applications is no longer as easy as retrieving the contents of the application URLs.

To crawl RIAs, a web crawler has to run a virtual web browser, emulate a user, identify different states of the application, and execute client-side events on all states. The further complication is that, unlike traditional web applications where the state of the application is reachable by going to the corresponding URL, to reach a state of the application the crawler has to perform multiple client-side events before it reaches the target state. For a large web application this process is very time consuming.

In this thesis we addressed the problem of reducing the time it takes to crawl RIAs. We used principles in parallel processing and distributed computing to reduce the time it takes to crawl RIAs by using concurrent crawlers. We proposed different mechanisms and architectures to break down sequential crawling algorithm into smaller independent tasks and we used multiple nodes to crawl RIAs. To perform a crawl in parallel we introduced the concept of task partitioning in the context of RIAs crawling. We applied the concept to four different crawling strategies of breath-first, depth-first, greedy and probabilistic strategies. Experimental results are presented and running the greedy strategy on the peer-to-peer architecture achieved the best performance as the number of crawlers increases. This thesis then talks about theoretical improvements on the proposed algorithms in two fronts: load-balancing, and component-based crawling.

Some of the contributions of this thesis include:

- *Partitioning Algorithm*: To the best of our knowledge, this work is the first work that formally introduces partitioning algorithms in the context of RIA crawling. Previously introduced partitioning algorithms are not suitable for RIA crawling, since they partition the tasks at a higher granularity of URLs. In this thesis, we introduced autonomous and deterministic partitioning algorithms that assign tasks to the nodes almost equally.

- *Distributed Crawling Architectures for RIA*: This thesis is the first work to introduce distributed crawling architectures for RIAs. We formally described two crawling architectures: a peer-to-peer architecture and a client-server one. To design these two architectures, we measure various time consuming factors for crawling RIAs and engineered the two architectures to be practical and efficient.

- *Performance Measurement*: We thoroughly measured the performance of the introduced architectures in practice by implementing the crawlers and crawling six web applications with multiple nodes. The investigation of performances described in this thesis shows that the peer-to-peer architecture is more efficient than the client-server architecture and scales better. Our experimental results suggest that, given a large enough web application, the peer-to-peer architecture can scale up to 400 nodes, before the network become a bottleneck and the performance deteriorates.

- *Load-Balancing Algorithms*: To the best of our knowledge, this thesis is the first study that describes load-balancing algorithms in the context of RIA crawling. Previously introduced load-balancing algorithms are based for traditional web applications. Our newly introduced load-balancing algorithms take into account client-side events, and suits the RIA environment better.

- *Distributed component-based crawling*: This work is the first study that addresses distributed, recently introduced, component-based crawling. Component-based crawling defines states and events differently from other crawling strategies. Therefore, partitioning algorithms defined previously cannot be used by this algorithm. We suggest two approaches based on gate-keeper and handlers. The two described approaches are compatible with the component-based crawling, and can be used to parallelize the algorithm.

8.1 Future Directions

Some future directions of research are:

8.1.1 Cloud Computing

One of the implicit assumptions of this thesis is that number of computers available to crawl a RIA does not change during the crawl. This restriction hinders us from taking advantage of cloud computing and cloud elasticity. By relaxing this assumption, the crawler can better utilize resources available to it.

8.1.2 Fault Tolerance

Another assumption we made in this thesis is that all crawler nodes and the communication medium are reliable. To relaxing this assumption (i.e. assuming that some nodes may terminate unexpectedly or that the communication medium may fail to deliver some messages), it is necessary to find recovery mechanisms suitable for the environment.

8.1.3 Integrating into Traditional Distributed Crawlers

Crawling literature is rich in describing distributed solutions to crawl traditional web applications. To the best of our knowledge, this thesis is the first effort to crawl RIAs in parallel. An enhancement to this work is to consider RIAs that have a large number of URLs, and each URL is associated with a large number of states. In this scenario, there are as many seed states as the number of URLs, and a new partitioning algorithm is required that not only considers events, but also considers seed URLs.

8.1.4 Relaxing Assumptions about RIAs

In this thesis we made several assumptions about the target RIAs. Two of the assumptions are particularly limiting: The assumption about determinism of the RIA, and the assumption about lack of external events. These assumptions are not particular to distributed crawling, but relaxing them can affect partitioning algorithms and thus distributed crawling.

A large number of applications do not satisfy these assumptions. User sessions, visitor counters, random advertisement widgets and news feeds are some examples that break the first assumptions. HTML5 web-sockets and Comet are examples that break the second assumption. Relaxing these assumptions may require redefining partitioning algorithms and finding new crawling architectures.

8.1.5 Impact of Architectural Parameters

In this thesis, we assumed that new transitions are broadcasted in the peer-to-peer architecture, as soon as they become available. Study of the impact of this assumption is missing from this thesis. More formally, the impact of the following two assumptions is missing:

- **Frequency of broadcasting**: To utilize the network better, it may be more efficient not to broadcast the new transitions as they become available, but to broadcast them in batches, or broadcast them in certain intervals.

- **Event sharing**: Not all transitions have a major impact on reducing the time it takes to crawl the RIA. Sharing only a sub-set of transitions, instead of all transitions, may not increase the time it takes to crawl the RIA, while it reduces network traffic.

Bibliography

[1] D.Turgay Altilar and Yakup Paker. Optimal scheduling algorithms for communication constrained parallel processing. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 197–206. Springer Berlin Heidelberg, 2002.

[2] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio 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.

[3] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio 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.

[4] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio 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.

[5] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Techniques and tools for rich internet applications testing. In *Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on*, pages 63 –72, sept. 2010.

[6] Amnon Barak. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13:4–5, 1998.

[7] J. Barbosa, J. Tavares, and A. J. Padilha. Linear algebra algorithms in heterogeneous cluster of personal computers. In *Proceedings of the 9th Heterogeneous*

Computing Workshop, HCW '00, pages 147–, Washington, DC, USA, 2000. IEEE Computer Society.

[8] Luciano Barbosa and Juliana Freire. Siphoning hidden-web data through keyword-based interfaces. In *SBBD*, pages 309–321, 2004.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[10] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[11] Zahra Behfarshad and Ali Mesbah. Hidden-web induced by client-side scripting: An empirical study. In *Proceedings of the International Conference on Web Engineering (ICWE)*, volume 7977 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2013.

[12] Kamara Benjamin. A strategy for efficient crawling of rich internet applications. Master's thesis, EECS - University of Ottawa, 2010. `http://ssrg.eecs.uottawa.ca/docs/Benjamin-Thesis.pdf`.

[13] Kamara Benjamin, Gregor von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif-Viorel Onut. A strategy for efficient crawling of rich internet applications. In *ICWE*, pages 74–89, 2011.

[14] Kamara Benjamin, Gregor Von Bochmann, Mustafa Emre Dincturk, Guy-Vincent Jourdan, and Iosif Viorel 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.

[15] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.*, 36(5):570–580, May 1987.

[16] Michael K. Bergman. The deep web: Surfacing hidden value. September 2001.

[17] Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. Automated security testing of web widget interactions. In *Proceedings of the the 7th joint meeting of the*

European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[18] Paul E Black. Fisher-yates shuffle. *Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., US National Institute of Standards and Technology*, 2005.

[19] Andrej Bogdanov and Emanuele Viola. Pseudorandom bits for polynomials. *SIAM J. Comput.*, 39(6):2464–2486, 2010.

[20] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Proc Australian World Wide Web Conference*, 34(8):711–726, 2002.

[21] Sergey Brin and Lawrence 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.

[22] Thang Nguyen Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *PPSC*, pages 445–452, 1993.

[23] Mike Burner. Crawling towards eternity: Building an archive of the world wide web. *Web Techniques Magazine*, 2(5), May 1997.

[24] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. Nfs over rdma. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, NICELI '03, pages 196–208, New York, NY, USA, 2003. ACM.

[25] Duen Horng Chau, Shashank Pandit, Samuel Wang, and Christos Faloutsos. Parallel crawling for online social networks. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 1283–1284, New York, NY, USA, 2007. ACM.

[26] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. Technical Report 2002-9, Stanford InfoLab, February 2002.

[27] S. Choudhary, E. Dincturk, Seyed Mirtaheri, G. v. Bochmann, G.-V. Jourdan, and V. Onut. Model-based rich internet applications crawling: "menu" and "probability" models. In *Journal of Web Engineering*, volume 13, pages 243 – 262, 2014.

[28] Suryakant Choudhary. M-crawler: Crawling rich internet applications using menu meta-model. Master's thesis, EECS - University of Ottawa, 2012. `http://ssrg.site.uottawa.ca/docs/Surya-Thesis.pdf`.

[29] Suryakant Choudhary, Emre Dincturk, Seyed. Mirtaheri, Guy-Vincent Jourdan, Gregor. Bochmann, and Iosif Onut. Building rich internet applications models: Example of a better strategy. In Florian Daniel, Peter Dolog, and Qing Li, editors, *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 291–305. Springer Berlin Heidelberg, 2013.

[30] Suryakant Choudhary, Mustafa Emre Dincturk, Gregor von Bochmann, Guy-Vincent Jourdan, Iosif-Viorel Onut, and Paul Ionescu. Solving some modeling challenges when testing rich internet applications for security. In *ICST*, pages 850–857, 2012.

[31] Suryakant Choudhary, Mustafa Emre Dincturk, Seyed M. Mirtaheri Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel 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.

[32] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[33] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, October 1989.

[34] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[35] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challenges in dynamic load balancing. *APPL. NUMER. MATH*, 52:2005, 2004.

[36] Mustafa Emre 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`.

[37] Mustafa Emre Dincturk, Suryakant Choudhary, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel Onut. A statistical approach for efficient crawling of rich internet applications. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *ICWE*, volume 7387 of *Lecture Notes in Computer Science*, pages 362–369. Springer, 2012.

[38] Mustafa Emre Dincturk, Suryakant Choudhary, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif-Viorel Onut. A statistical approach for efficient crawling of rich internet applications. In *ICWE*, pages 362–369, 2012.

[39] Stefan Dobrev, Rastislav Krlovi, and Euripides Markou. Online graph exploration with advice. In Guy Even and MagnsM. Halldrsson, editors, *Structural Information and Communication Complexity*, volume 7355 of *Lecture Notes in Computer Science*, pages 267–278. Springer Berlin Heidelberg, 2012.

[40] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'10, pages 111–131, Berlin, Heidelberg, 2010. Springer-Verlag.

[41] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. Ajax crawl: Making ajax applications searchable. In *ICDE*, pages 78–89, 2009.

[42] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong 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.

[43] Jenny Edwards, Kevin McCurley, and John Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 106–113, New York, NY, USA, 2001. ACM.

[44] Karl Entacher. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):61–70, 1998.

[45] José Exposto, Joaquim Macedo, António Pina, Albano Alves, and José 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.

[46] Ian Fette and Alexey Melnikov. The websocket protocol. 2011.

[47] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws. *J. Parallel Distrib. Comput*, 47:139–152, 1997.

[48] Rudolf Fleischer, Tom Kamphans, Rolf Klein, Elmar Langetepe, and Gerhard Trippen. Competitive online approximation of the optimal search ratio. In *In Proc. 12th Annu. European Sympos. Algorithms, volume 3221 of Lecture Notes Comput. Sci*, pages 335–346. Springer-Verlag, 2004.

[49] Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. *CoRR*, abs/0901.0131, 2009.

[50] Gianni Frey. Indexing ajax web applications. Master's thesis, ETH Zurich, 2007. `http://e-collection.library.ethz.ch/eserv/eth:30111/eth-30111-01.pdf`.

[51] Klaus-Tycho Frster and Roger Wattenhofer. Directed graph exploration. In Roberto Baldoni, Paola Flocchini, and Ravindran Binoy, editors, *Principles of Distributed Systems*, volume 7702 of *Lecture Notes in Computer Science*, pages 151–165. Springer Berlin Heidelberg, 2012.

[52] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[53] Vahid Garousi, Ali Mesbah, A. Betin Can, and Shabnam Mirshokraie. A systematic mapping study of web application testing. *Information and Software Technology*, 55(8):1374–1396, 2013.

[54] Stphane Genaud, Arnaud Giersch, and Frdric Vivien. Load-balancing scatter operations for grid computing. In *IN 12TH HETEROGENEOUS COMPUTING WORKSHOP (HCW2003). IEEE CS*, page 2004, 2003.

[55] Ashish Goel and Piotr Indyk. Stochastic load balancing and related problems. In *In FOCS*, pages 579–586, 1999.

[56] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the deep web. *Commun. ACM*, 50(5):94–101, May 2007.

[57] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[58] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[59] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2:219–229, 1999.

[60] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. Technical report, Daresbury Laboratory, 1995.

[61] James Wayne Hunt and M Douglas McIlroy. *An algorithm for differential file comparison.* Bell Laboratories, 1976.

[62] Frederick James. A review of pseudorandom number generators. *Computer Physics Communications*, 60(3):329–344, 1990.

[63] Nicholas T Karonis, Brian Toonen, and Ian Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[64] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, 20(1):359–392, 1998.

[65] Jon Kleinberg, Yuval Rabani, and Eva Tardos. Allocating bandwidth for bursty connections. *SIAM J. Comput*, 30:2000, 1997.

[66] Steve Lawrence and C. Lee Giles. Searching the world wide web. *SCIENCE*, 280(5360):98–100, 1998.

[67] Tapani Lehtonen. Scheduling jobs with exponential processing times on parallel machines. *Appl. Prob*, 25:752–762, 1988.

[68] 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.

[69] Yawei Li and Zhiling Lan. A survey of load balancing in grid computing. In *CIS*, pages 280–285, 2004.

[70] Stephen W. Liddle, David W. Embley, Del T. Scott, and Sai H. Yau1. Extracting Data behind Web Forms. *Lecture Notes in Computer Science*, 2784:402–413, January 2003.

[71] James Lo, Eric Wohlstadter, and Ali 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.

[72] Boon Thau Loo, Sailesh Krishnamurthy, and Owen Cooper. Distributed web crawling over dhts. Technical Report UCB/CSD-04-1305, EECS Department, University of California, Berkeley, 2004.

[73] Jianguo Lu, Yan Wang, Jie Liang, Jessica Chen, and Jiming Liu. An Approach to Deep Web Crawling by Sampling. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, 1:718–724, 2008.

[74] Kai Lu and Albert Y. Zomaya. A hybrid policy for job scheduling and load balancing in heterogeneous computational grids. In *Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, ISPDC '07, pages 19–, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Alessandro Marchetto and Paolo Tonella. Search-based testing of ajax web applications. In *Proceedings of the 2009 1st International Symposium on Search Based*

Software Engineering, SSBSE '09, pages 3–12, Washington, DC, USA, 2009. IEEE Computer Society.

[76] Alessandro Marchetto, Paolo Tonella, and Filippo 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.

[77] Joe Marini. *Document Object Model*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2002.

[78] Reto 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`.

[79] Oliver A. McBryan. Genvl and wwww: Tools for taming the web. In *In Proceedings of the First International World Wide Web Conference*, pages 79–90, 1994.

[80] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: new results on old and new algorithms. In *Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II*, ICALP'11, pages 478–489, Berlin, Heidelberg, 2011. Springer-Verlag.

[81] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, July 2009.

[82] Ali Mesbah, Engin Bozdag, and Arie van 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.

[83] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 210 –220, may 2009.

[84] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[85] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1):3:1–3:30, March 2012.

[86] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *TWEB*, 6(1):3, 2012.

[87] Amin Milani Fard and Ali 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.

[88] Seyed M Mirtaheri, Mustafa Emre Dinçtürk, Salman Hooshmand, Gregor V Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. A brief history of web crawlers. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 40–54. IBM Corp., 2013.

[89] Seyed M Mirtaheri, Di Zou, Gregor V Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. Dist-ria crawler: A distributed crawler for rich internet applications. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pages 105–112. IEEE, 2013.

[90] SeyedM. Mirtaheri, GregorV. Bochmann, Guy-Vincent Jourdan, and IosifViorel Onut. *PDist-RIA Crawler: A Peer-to-Peer Distributed Crawler for Rich Internet Applications*, volume 8787 of *Lecture Notes in Computer Science.* Springer International Publishing, 2014.

[91] SeyedM. Mirtaheri, Gregor von Bochmann, Guy-Vincent Jourdan, and IosifViorel Onut. Gdist-ria crawler: A greedy distributed crawler for rich internet applications. In Guevara Noubir and Michel Raynal, editors, *Networked Systems*, Lecture Notes in Computer Science, pages 200–214. Springer International Publishing, 2014.

[92] William F. Mitchell. Refinement tree based partitioning for adaptive grids. In *Proceedings of the 7 th SIAM Conference on Parallel Processing for Scientific Computing, SIAM*, pages 587–592, 1995.

[93] Ali Moosavi. Component-based crawling of complex rich internet applications. Master's thesis, EECS - University of Ottawa, 2014. `http://ssrg.site.uottawa.ca/docs/Ali-Moosavi-Thesis.pdf`.

[94] Alexandros Ntoulas. Downloading textual hidden web content through keyword queries. In *In JCDL*, pages 100–109, 2005.

[95] John Tinsley Oden, Abani Patra, and Yusheng Feng. Parallel domain decomposition solver for adaptive hp finite element methods. *SIAM journal on numerical analysis*, 34(6):2090–2118, 1997.

[96] Christopher Olston and Marc Najork. Web crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.

[97] Brake-N. Ionescu P. Smith D. Dincturk M.E. Mirtaheri S.M. Jourdan G.-V. Bochmann G.v. Onut, I.V. A method of identifying equivalent javascript events on a page, 2012. [Patent].

[98] I.V. Onut, K.A. Ayoub, P. Ionescu, G.v. Bochmann, G.V. Jourdan, M.E Dincturk, and S.M. Mirtaheri. Representation of an element in a page via an identifier. [Patent].

[99] Jourdan-G.-V. Bochmann G.v. Mirtaheri S.M. Onut, I.V. A method of partitioning the crawling space of a rich internet application for distributed crawling, 2012. [Patent].

[100] Elliott I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, MA, USA, 1972.

[101] D.F. Parkhill. *The challenge of the computer utility.* Number p. 246 in The Challenge of the Computer Utility. Addison-Wesley Pub. Co., 1966.

[102] Zhaomeng Peng, Nengqiang He, Chunxiao Jiang, Zhihua Li, Lei Xu, Yipeng Li, and Yong 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.

[103] John R. Pilkington, John R. Pilkington, Scott B. Baden, and Scott B. Baden. Partitioning with spacefilling curves. Technical report, 1994.

[104] Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, May 1990.

[105] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 129–138, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[106] Terry Ritter. The efficient generation of cryptographic confusion sequences. *Cryptologia*, 15(2):81–139, April 1991.

[107] Ronald Rivest. The md5 message-digest algorithm. 1992.

[108] Danny Roest, Ali Mesbah, and Arie van Deursen. Regression testing ajax applications: Coping with dynamism. In *ICST*, pages 127–136. IEEE Computer Society, 2010.

[109] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance, 2004.

[110] Alex Russell. Comet: Low latency data for browsers. *The Dojo Toolkit*, 2006.

[111] Vladislav Shkapenyuk and Torsten 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.

[112] Joaquim Silvestre. economies and diseconomies of scale. In John Eatwell, Murray Milgate, and Peter Newman, editors, *The New Palgrave: A Dictionary of Economics*. Palgrave Macmillan, Basingstoke, 1987.

[113] Horst D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[114] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[115] Chunqiang Tang, Zhichen Xu, and Mallik Mahalingam. psearch: Information retrieval in structured overlays, 2002.

[116] Valerie Taylor and Bahram Nour-omid. A study of the factorization fill-in for a parallel implementation of the finite element method. *Int. J. Numer. Meth. Engng*, 37:3809–3823, 1994.

[117] Shu Tezuka. Linear congruential generators. In *Uniform Random Numbers*, pages 57–82. Springer, 1995.

[118] Hsin tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Irlbot: Scaling to 6 billion pages and beyond, 2008.

[119] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[120] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM.

[121] RICHARD R. WEBER. Scheduling jobs with stochastic processing requirements on parallel machines to minimize makespan or flowtime. *Appl. Prob*, 19:167–182, 1982.

[122] Gideon Weiss. Approximation results in parallel machnies stochastic scheduling. *Annals of Operations Research*, 26:195–242, 1990.

[123] Stallings William and William Stallings. *Cryptography and Network Security, 4/E*. Pearson Education India, 2006.

[124] Chengzhong Xu, Burkhard Monien, Reinhard Lling, and Francis C. M. Lau. Nearest neighbor algorithms for load balancing in parallel computers, 1995.

[125] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[126] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

Appendix A

Load Balancing Approaches

The algorithms and architectures introduced in this thesis, assumed that all nodes have equal processing powers, and thus assigned them equal shares of work. Our distributed model risks to make a set of nodes bottleneck if nodes are not homogeneous. Also the static partitioning algorithm used in peer-to-peer architecture is not perfect: If the speed of nodes is known before the crawling begins, the partitioning algorithm can adopt to it. Further we assumed that the processing power of a node does not change during the course of the crawl. Non-homogeneous computing and elasticity of the computing resources available, are patterns frequent, but not popular in cloud-computing.

This appendix introduces mechanisms to share and transfer workload among the nodes. These algorithms, referred to as *load-balancing algorithms*, are used to decide how to assign tasks to the crawlers. Different assignments of tasks to the crawlers result in different workload and communication patterns among the nodes. Five load balancing approaches are introduced. Note the ideas presented are not implemented. We leave implementation of the presented ideas and experimenting with them to future studies. Due to the lack of implementation at this point, the contents of this appendix is not included in the thesis as a chapter.

The rest of this appendix is organized as follows:

- In Section A.1, we introduce the load-balancing algorithms used by the distributed traditional web-crawlers in the literature and explain how to apply them to rich internet applications.

- In Section A.2 we formally introduce the terms we use to construct our load-balancing algorithms.

- In Section A.3 we explain adaptation of two load-balancing algorithms introduced in the literature to rich internet application crawlers.
- In Section A.4 we introduce three new load-balancing algorithms.
- In Section A.5 we discuss the communication pattern among the nodes in different load-balancing approaches. Finally, In Section A.6 we conclude this appendix.

A.1 Introduction

Designing efficient and scalable parallel web crawlers has been the topic of extensive research for more than two decades[96]. Many efficient and scalable solutions are offered using distributed platforms[26]. Cho and Garcia-Molina[26] categorize distributed crawlers based on their communication patterns into three main categories:

- *Independent approach*: In this approach, nodes work independently, each with a separate set of seed URLs. Due to the lack of communication among the nodes, pages visited by different nodes may overlap.
- *Static approach*: In this approach a set of homogeneous nodes partition the search space among themselves, and operate without any centralized coordination unit. Upon encountering a task, the crawler examines the task and decides whether the task belongs to it or not. Depending on the architecture, should a task belong to another node, the finder node may or may not communicate the task to its owner[26].
- *Dynamic approach*: In this approach nodes rely on a centralized coordination unit that gathers all tasks and allocates them to the nodes. Upon discovering a new task, the node will inform the unit and should it be the first time the unit receives the task the unit adds the task to a task queue. Nodes then ask the unit for task, and the unit assigns the tasks to the probing nodes, and remove the assigned tasks from its queue. [25]

Due to the lack of communication, the independent approach does not guarantee absence of overlaps between the scope of different crawlers. Authors suggest mechanisms to reduce the overlaps to a certain degree with a certain amount of messages among the nodes. Since the purpose of this thesis is to discover all of the application states, we do not study this category.

We borrow the static and dynamic approaches described in the traditional web-crawling literature and adopt them to RIA crawling. In addition to these two approaches, three novel approaches are introduced. These novel task assignment algorithms are called *Hybrid Assignment*, *Adaptive Assignment*, and *Lazy-Adaptive Assignment*. The rest of this appendix introduces the tasks of RIA crawling more formally, then describes each of the load balancing approaches in detail.

A.2 Definitions

The following definitions are used in the rest of this appendix:

- E: The total number of events in all application states.
- N: Number of crawler nodes.
- i: Node identifier, where $1 \leq i \leq N$.
- T_e: The time it takes to execute event e.
- $T_{average}$: The average time it takes to execute an event.
- I_i: The average idle time of node i, expressed in the form of ratio of full crawling time.
- $I_{average}$: The average idle time of nodes, expressed in the form of ratio of full crawling time. We assume that this number is known from the previous runs, and we use this number to optimize some of the algorithms.

In addition to the above-mentioned terms, previously defined terms of s, e, S and E_S are also used in this appendix. For definition of these terms see Section 6.1.3.

A.3 Adapting static and dynamic approaches to RIA crawling

Originally, static and dynamic approaches are described in the context of distributed crawling of traditional web applications. In this section, we first briefly explain the approaches as they are used in the context of traditional web crawler, then explain how to adopt them to crawling RIAs.

A.3.1 Static

In this approach nodes are peers. When a node discovers a new URL, it calculates locally and deterministically the node responsible to crawl the URL and send a message to the node responsible informing it about the URL. Often hash functions are used to map URLs to the set of crawling nodes.

In the context of traditional web crawling, partitioning of the work happens on URL granularity. A URL can be associated with many states in a RIA, and thus it is not practical to use URLs to partition the search space in static approach. Two other ways to partition to search space are: by application states, and by events. The latter approach is superior to the former approach. In the former approach a node responsible to crawl a state has to execute all events in that state. As such it has to keep going back to the state. After each event execution, the node may end up in another state and the probability that the new state belongs to the node is low. Thus there is no work to be done in the new state, and the node simply has to find its way back to a state that belongs to it.

The latter approach (i.e. partitioning by events) does not have this shortcoming: No matter at what state the node ends up after executing an event, there is a good chance there are events in this state that belongs to the node. As such we adopt the static approach to RIA crawling by partitioning the search space based on events. In this approach: when a node discovers a new application state, it propagates it to the other nodes. Upon learning about a new state, the node runs a local partitioning algorithm and based on the output of the algorithm, the node finds the tasks that belong to it in this state. All the tasks are performed; while the partitioning algorithm ensures that no task is performed by more than one crawler.

The static approach avoids overlaps and scales well, grateful to its peer-to-peer architecture. This approach, however, merely assumes that all nodes are equivalent in their processing power. Should a task take a long time to complete and a node becomes a bottleneck, this approach has no means to alleviate the problem. Dist-RIA Crawler and the peer-to-peer architecture are constructed following this approach.

A.3.2 Dynamic

In the context of traditional web application crawling, a single server keeps track of all discovered URLs. Crawler units contact this server and ask for URLs to crawl. The server returns a set of URLs to these units and flags them as visited. For the same reason explained in the previous section, to adopt this approach to RIA crawling, URLs can be used by the centralized unit. Similarly, assigning an state to a crawler on fly may not be efficient, since the crawler has to keep executing long chain of events to go back to the state.

Similar to the static approach, this approach can be adopted to RIA crawling by partitioning the search space based on the events. In the adopted approach, a centralized unit keeps track of all events. Idle nodes contact this coordination unit and request tasks. The coordination unit returns a task to the probing node. In essence a task is an un-executed event. It is however, necessary sometimes to execute a path of events before reaching to a state that hosts the un-executed event. In these cases a task is composed of a chain of events, concatenated by the un-executed event. After the task is returned to the probing node, the centralized unit removes the event from its list of un-executed events.

The dynamic approach avoids overlaps among the nodes and it achieves a perfect load balancing: working nodes only ask for new tasks when they are free, thus no working node becomes a bottleneck. This approach, however, is not scalable since the centralized unit may become a bottleneck, as the number of nodes increases. Client-Server architecture is constructed based on this approach.

A.4 New Load Balancing Approaches

In an attempt to take advantages of the dynamic and static approaches, we propose two new approaches, which conceptually have elements from both dynamic and static approaches. These two new approaches are called *hybrid* and *adaptive* approaches. A third approach, called *lazy-adaptive* approach is introduced subsequently and combines the two.

A.4.1 Hybrid approach

Assuming we have an estimate of the percentage of the time a node is idle in the static approach, the hybrid approach is a simple mechanism to balance the load by reserving and keeping some of the tasks for idle nodes. In the hybrid approach, like in the static approach, nodes locally determine the portion of the work that belongs to them, and communicate the rest of the work to the responsible nodes. In addition, this approach incorporates a load balancing mechanism to transfer certain tasks among the nodes.

In the hybrid approach, there is a coordination station in the system with no crawling power. This node merely acts as a storage unit, and it is allocated a portion of the tasks. This portion, referred to as *reserved* tasks, is not allocated to any node in particular. When a crawling node becomes idle, it will contact the coordination station and asks for some of the reserved tasks. This approach is thus a hybrid of static and dynamic approaches: some of the tasks are allocated statically, and some are allocated dynamically; hence the name "hybrid".

The coordination unit keeps track of the reserved tasks. The unit creates a reserved task per reserved event, and waits for an idle node to request it. The hybrid approach uses the following algorithm to determine which events are considered reserved: The ratio of events that are reserved in each state is called *reserved ratio* (RR). When the coordination station learns about a new state s, it calculates the number of reserved events in s (called *reserved tasks in s* or RT_S) by multiplying reserved ratio by E_S. The coordination station then takes the last RT_S events in s and create one reserved task per event. The partitioning algorithm on the crawling nodes, uses the same formula to determine the reserved events. After identifying the reserved events, the partitioning algorithm exclude these events and partitions the rest of the events for crawling.

When a node becomes idle, it contacts the coordination station, and the coordination station returns one of the reserved tasks to the probing node. The returned reserved task is removed from the list of tasks in the coordination station. This process continues until all of the nodes are in the idle state, and the coordination station has no reserved task left.

Assuming on average each node is idle $I_{average}$ percent of the time and busy $1 - I_{average}$ percent of the times, RR can be calculated as:

$$RR = MIN(1, I_{average} \times N)$$

By using this formula, the approach tries to set aside enough events that can be allocated to idle nodes and make them busy. This formula shows how the hybrid approach is equal to the static and dynamic approaches given different average idle times:

- If on average nodes are idle more than $\frac{1}{N}$ percent of the times, the formula becomes $RR = 1$ and, in effect, the load balancing algorithm becomes the dynamic approach where all tasks are stored on the coordinator, and it the coordinator dispatches the jobs as nodes become idle.

- If there are no idle times, then the formula becomes $RR = 0$ and, in effect, the load balancing algorithm becomes the static approach where the nodes calculate the tasks locally and autonomously, without any centralized unit to dispatch tasks to idle nodes.

A.4.2 Adaptive approach

The hybrid approach is a simple method to reserve some of the tasks for future idle nodes. This method assumes that knowledge of the average idle times is available prior to crawl. If the knowledge of the idle times is not available, the hybrid approach may not have a good performance. In this section we introduce an adaptive approach, an approach that learns about the idle times and relative speed of different nodes as the crawl proceeds and based on the gathered information dynamically modifies the partitioning algorithm.

More formally, the adaptive approach is similar to the static approach, with the following change: As the crawling proceeds, this approach adjusts the portion of tasks assigned to each node. 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. Further, the approach is similar to the dynamic approach in that a coordinator allocates the tasks to the nodes at the time of their discovery, except that tasks are not assigned equally, but assigned based on the perceived computational speed of the node.

In this approach, every time a new state is discovered the coordinator partitions the the tasks in the state and assigns different portions of tasks to different nodes. The purpose of this assignment is to drive all nodes to finish together. The portion of tasks 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 tasks 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.

Adaptive partitioning algorithm

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

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

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

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

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

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

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:

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

Equations A.4 can be re-written using equation A.3:

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

$$\tag{A.9g}$$

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

Limitations of the Adaptive Approach

The adaptive approach has two limitations:

1. It only works if there are enough events in a newly discovered state s to rescue every bottleneck node. In other words, if there are not enough events in s, and the workload gap between the nodes is large, the above-mentioned method fails to assign enough jobs to all idle nodes and make them busy so that all nodes finish together.

2. The approach assign jobs to the nodes as soon as they are discovered. The method assumes that the past performance of the crawler is an indication of its future performance and based on this assumption it calculates the speed of the crawler. This assumption creates a shortcoming: If a node starts with a fast pace, and get assigned a large amount of tasks, then its computational processing power shrinks and it becomes a bottleneck. Similarly, if a node starts with a small computational power, get assigned a small amount of tasks, then its computational power increases, it will be idle until new states are discovered.

The second challenge is addressed by the Lazy-Adaptive approach.

A.4.3 Lazy-Adaptive approach

The adaptive approach allocates newly discovered tasks to the nodes based on the current bottlenecks, and does not maintain any mechanism to deal with future bottlenecks or idleness. This can be a problem, if all states are discovered early in the crawl, all tasks are assigned to the node, and later in the crawl some nodes become bottleneck. Since no new state is discovered, the strategy cannot balance the load between nodes.

As an enhancement to the adaptive approach we introduce another load balancing approach: the *Lazy-Adaptive* approach. This approach is adaptive in assigning workloads to the nodes. To do this, it introduces a delay in releasing the workloads to the nodes, and uses this delay as a mechanism to gain knowledge about future bottlenecks.

This approach is similar to the adaptive approach in using the same formula to assign tasks to the nodes. However, unlike the adaptive approach where new tasks are assigned to the nodes as soon as they become available, in the Lazy-Adaptive approach new tasks are held by the coordinator and dispatched to the nodes only at certain points in time.

The coordinator postpones assigning assigning and dispatching tasks to the nodes as much as possible. In order to decide when to release the held tasks, the coordinator calculates the number of un-executed tasks that each node has left. As long as all nodes are busy, there is no need for the coordinator to dispatch jobs to the nodes. If there is a node with only one task, the coordinator runs the algorithm to assign tasks and dispatch new tasks to the nodes.

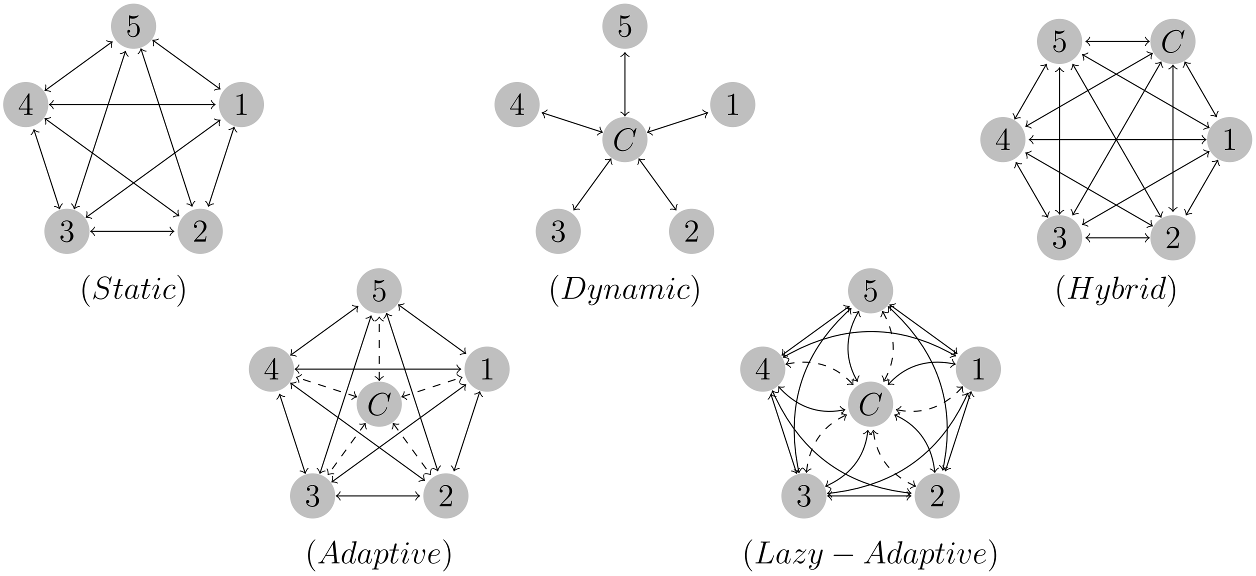
A.5 Communication Pattern

The communication patterns of static, dynamic, adaptive, hybrid, and lazy-adaptive approaches are depicted in Figure A.1. In this figure, solid lines represent transfer of tasks communication among the nodes. Dashed lines, on the other hand, represent coordination and administration communications. As the figure depicts:

- In the static approach, nodes act autonomously and communicate discovered states among one another.
- In the dynamic approach, a centralized coordinator assigns work to the nodes.
- In the hybrid approach, nodes act semi-autonomously. Part of the discovered tasks go to a coordination station. Free nodes retrieve these tasks from the coordination station and execute them.
- In the adaptive approach tasks are released to all the nodes as soon as they are discovered and no task is reserved. However, a centralized unit specifies the task distribution among the nodes.
- The lazy-adaptive approach combines the adaptive and the hybrid approaches. In this approach, a centralized coordination station is used to both, adjust task distribution, and decide when to release new tasks to the nodes.

A.6 Conclusion

This appendix studies five load-balancing algorithms. Two of the load-balancing algorithms introduced in the literature are static and dynamic approach. Based on the static and dynamic approaches, a new Hybrid approach was introduced that incorporates elements from both approaches. Another new approach called Adaptive approach was introduced that learns and adapt to the computational speed of the nodes. Finally, to take into account fluctuations in the computational speed of the nodes, a lazy version of the adaptive algorithm, called lazy-adaptive approach, was introduced. The newly introduced approaches can be used to better take advantage of cloud-computing.



($Static$) ($Dynamic$) ($Hybrid$)

($Adaptive$) ($Lazy - Adaptive$)

Figure A.1: Static, dynamic, adaptive, hybrid and lazy-adaptive load balancing approaches .

Appendix B

Distributed Component-Based Crawling of RIAs

The algorithms and architectures introduced in this thesis, assumed that the web crawler explores all events on all states of the application. One of the major challenges in the field of RIA crawling is the problem of state explosion. Real-life RIAs often use independent widgets in their user interface. Interacting with one widget does not effect another widget. Examples of independent widgets on the client interface are the Facebook chat page and day rectangles in Google calendar. As the user chats with a friend, the interaction with the chat window does not effect other widgets in the page. Similarly, as the user interact with a given day widget, it is often the case that other days are not modified.

Moosavi[93] studies component-based crawling of RIAs. The introduced algorithm identifies different components in the page. The strategy then interacts with identified widgets independently and updates its knowledge of widgets if required. Due to the additional assumption of independence, this strategy out-performs other strategies such as the greedy and probabilistic strategy by a far margin. This appendix studies the distribution of the component-based crawling of RIAs and analyses the complexity and overhead introduced by distribution. Main contributions of this appendix are two partitioning algorithms described that can be used in the context of component-based crawling. Albeit highly practical, due to the additional assumption, this strategy does not always guarantee finding all states of the application. Therefore, contents of this appendix are not included as a chapter in the thesis.

The rest of this appendix is organized as follow: In Section B.1, we briefly introduce component-based crawling of RIAs. In Section B.2, we explain how to distribute the algorithm and run it in parallel over multiple nodes. In this section we describe the two partitioning algorithms. Finally, In Section B.3, we conclude this appendix.

B.1 Introduction to Component-Based RIA Crawling

Component-based crawling of RIAs differs from other crawling strategies in two major ways:

- **State**: In all crawling strategies, except the component-based strategy, the application state is defined as the client-side state of the application. Client-side state of the application is often defined as the state of DOM. Unlike other strategies, the component-based crawling strategy defines components in the page. The assumption of the strategy is that it can detect independent components in the page. Each component itself can have different states. Therefore, component-based crawling strategy does not define state of the application as state of DOM, but only deals with the components in the page.

  In the component-based crawling strategy, components are detected by calculating the difference[1] between the DOMs before execution of an event and the DOM after it. The algorithm first finds set of DOM elements that differ between the two DOMs. After finding the nodes, the most immediate ancestor of all the changed nodes is calculated. The subtree of this ancestor is considered to be a new component. This new component is represented by the *Xpath* of the ancestor node. The list of discovered Xpaths is stored in a database, and is used to verify weather the discovered component is a new component, or the crawler has seen it before.

- **Application Graph**: With the exception of component-based strategy, in all crawling strategies, the application graph is composed of vertices and edges. Nodes corresponds to the application states and edges corresponds to the application events. By representing the application graph this way, from any source state, after executing an event, only one target state is reachable.

  The component-based strategy defines the application graph differently. Unlike the other strategies, in the component-based strategy, a node in the application

[1] For more details on the *Diff* algorithm see [61].

graph does not correspond to the state of DOM, but to the components found in the application. In the component-based crawling, the execution of an event can change multiple components. In other words, executing an event from a component can cause emergence of multiple separate components in the application graph. As the result, multiple outgoing edges, representing the same event execution, can exist from a node in the application graph to different nodes. Figures B.1 and B.2 visualize this difference between application graph in a non-component-based crawling strategy and the component-based crawling strategy:

– As the Figure B.1 shows, the application graph in a non-component-based crawling strategy corresponds to a deterministic finite state machine. In this model given a source state and a event to execute, the crawler always reaches the same target state.

– In the case of component-based crawling strategy, as it is depicted in Figure B.2, vertices in the application graph do not represent single states of DOM, but they are components of the DOM. In other words, the application graph created during component-based crawling is not a state machine, and at any given point multiple vertices of the graph may be present on the DOM. Therefore, one event execution from a vertex can lead to multiple vertices (e.g. executing event e_h from state 8).

B.2 Distributed Component-Based RIA Crawling

The previously described distributed RIA crawler architectures (i.e. Dist-RIA, client-server architecture, and peer-to-peer architecture) use the partitioning algorithm to partition the task of executing JavaScript events among themselves. The partitioning algorithm maps JavaScript events to the crawling nodes such that each event is assigned to one and only one crawler. The partitioning algorithm cannot be used by the component-based distributed RIA crawler, since the definition of state differs in component-based crawling from our previous definition. In the component-based crawling algorithm, the crawler does not execute every JavaScript event in every state of the DOM. Therefore, the previously defined partitioning algorithms are not useful for component-based crawling. This section introduces two new algorithms to distribute a component-based crawling algorithm.

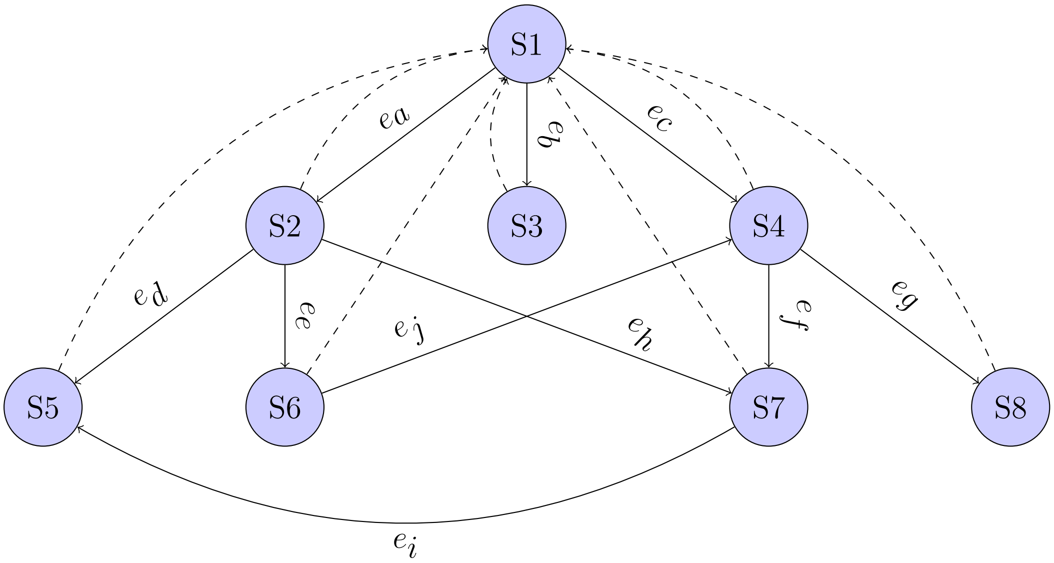


Figure B.1: Application graph in a non-component-based crawling strategy: solid lines represent events, and dashed lines represent a reset.

B.2.1 Partitioning based on component handlers

As explained earlier, in component-based crawling strategy, components are identified by their XPath. One approach to parallelize the algorithm is to allocate each component to one crawling node, or define a *handler* for each component. The component handler is the node responsible to crawl a component. Every time a node discovers a new component, it calculates the component XPath. Hash of the XPath is used to map the component to its handler. If the node itself is the handler, it continues to crawl the component, if not it will abandon the component. In both cases, weather the component handler is the node itself or not, the node broadcasts the information about the new component to other nodes.

Algorithm 2 shows the handler-based crawling algorithm ran by each crawler node. Following procedures are used by this algorithm:

- getTask: This procedure chooses a task to execute. This procedure gets pending tasks as input. Moosavi[93] suggests using a greedy algorithm to pick the next task to execute. In order to find new components faster during crawl, the probabilistic strategy could be used instead of the greedy strategy.
- GoToTargetDOM: This procedure runs multiple breath-first search algorithms from the vertices that represent components in the current DOM, and stops when it

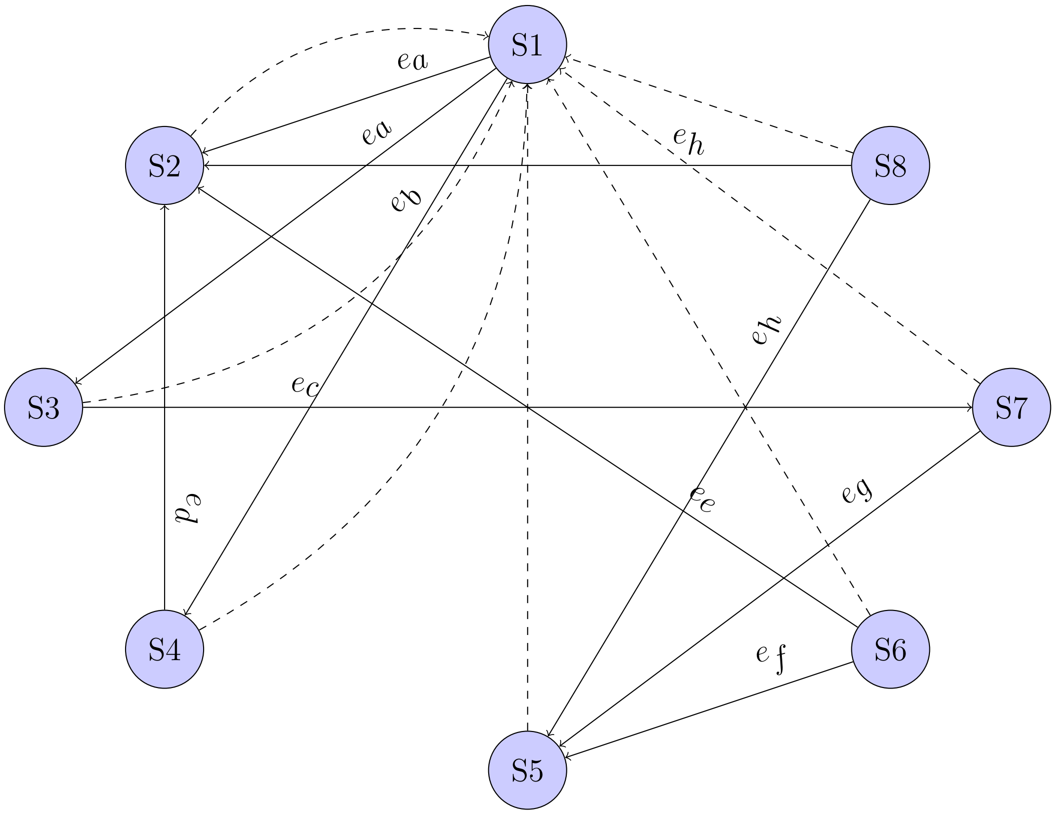


Figure B.2: Application graph in a component-based crawling strategy: solid lines represent events, and dashed lines represent a reset.

Algorithm 2 Task execution algorithm based on component handlers

```
TaskToExecute ← getTask(currentState, pendingTasks)
GoToTargetDOM(TaskToExecute)
domBefore ← GetDOM()
ExecuteTask(TaskToExecute)
domAfter ← GetDOM()
component ← GetComponent(domBefore, domAfter)
componentHandler ← GetComponentHandler(component)
BroadCastComponent(TaskToExecute, component)
```

finds a path to the target component. After finding the shortest path, the procedure executes the task. When the procedure returns the control to the caller function, the crawler has access to the target state.

- GetDom: This procedure returns the DOM of the application.
- GetComponent: Given two DOMs, this procedure finds the most immediate ancestor of all changed DOM elements in second DOM, and returns the ancestor node XPath.
- BroadCastComponent: Given a component and the executed task, this procedure broadcasts the information about the executed task and the discovered component.
- NotifyHandler: Given a component and its handler, this procedure sends a synchronous message to the handler informing it of the component.

B.2.2 Partitioning based on component gate-keepers

By using the component handler to partition the work, the node simply delegate the responsibility of crawling the component to the handler. This simple strategy results in low number of network messages. Unfortunately, it is not necessarily the case that the handler is readily at a state of DOM where the component is present. The handler has to run a search algorithm to find the set of events it has to execute before it reaches the target component. It then has to execute the events to reach a state where it can interact with the component.

Another approach to partition the execution of JavaScript events is to define *gate-keeper* for each component. The gate-keeper of the component is not necessarily responsible to crawl the component itself, rather it is responsible to ensure that the component is crawled by one and only one node. When a crawler node discovers a new component, it calculates the XPath of the component and base on the hash of the XPath calculates the gate-keeper of the component. After finding the component gate-keeper, the crawler node contacts the gate-keeper and informs it of the component. In response, the gate-keeper responds with one of the following messages:

- The component is already assigned: This message notifies the node that the component was assigned to another node. Upon receiving this message, the crawler abandons the component and moves on to other tasks.

- The component is not assigned: This message notifies the node that this component is not assigned to any other node. At this point the node decides to take over the responsibility of crawling the component based on its workload. If the node is not busy with other tasks, it takes over the responsibility of crawling the component by sending a request to the gate-keeper. If the gate-keeper approves, the node will gain the responsibility to crawl the component.

In gate-keeper-based partitioning algorithm, the node that request the responsibility to crawl the component from handler, is already at a state where it has access to the component. Therefore, (if permission is granted) the node can simply interact with and crawl the component without executing any JavaScript events to reach to the component. Based on our experiments, it is often the case that execution of the JavaScript events is the bottle-neck in a distributed crawling algorithm. Therefore, we envision that the component gate-keeper-based partitioning algorithm has a better chance to outperform handler-based partitioning algorithm. If the network becomes the bottleneck, however, the handler-based strategy can outperform the gate-keeper-based strategy.

Algorithm 3 shows the crawling algorithm ran by each crawler node. getTask, GoToTargetDOM, GetDom, GetComponent, BroadCastComponent and NotifyHandler procedures are similar to B.2.1. New procedures used in are:

- GetComponentGateKeeper: Given a component, this function calculates the gate-keeper for the component by taking the hash of its XPath and map the hash to a node.

- RequestIfAssigned: Given a component and its gate-keeper, this procedure sends a synchronous message to the component gate-keeper and asks the gate-keeper if the component has been assigned.

- RequestForAssignment: Given a component gate-keeper, this procedure sends a synchronous message to the component gate-keeper and request to take over the responsibility to crawl the component.

- AddComponentToPendingTasks: This method adds the component to pending tasks, so it would be crawled eventually.

Algorithm 3 Task execution algorithm based on component handlers

```
TaskToExecute ← getTask(currentState, pendingTasks)
GoToTargetDOM(TaskToExecute)
domBefore ← GetDOM()
ExecuteTask(TaskToExecute)
domAfter ← GetDOM()
component ← GetComponent(domBefore, domAfter)
BroadCastComponent(TaskToExecute, component)
componentGateKeeper ← GetComponentGateKeeper(component)
if GetWorkLoad() < WorkLoadThreshold then
    handlerRespond ←
        RequestIfAssigned(componentGateKeeper, component)
    if handlerRespond is ComponentIsNotAssigned then
        assignmentRespond ←
            RequestForAssignment(componentGateKeeper, component)
        if handlerRespond is Not AlreadyAssigned then
            AddComponentToPendingTasks(component, pendingTasks)
        end if
    end if
end if
```

B.3 Conclusion and Future work

This appendix introduced a distributed component-based crawler for RIAs. Two algorithms to partition the search space based on the component handler and the component gate-keeper were introduced. We envision that the former algorithm is better suited to shift the burden away from JavaScript execution, while the latter algorithm is better tuned to shift the burden away from the network; however, this needs to be verified in future work.

A future direction to this appendix is combining the two partitioning algorithms proposed. The proposed algorithms tend to either reduce the cost of executing tasks at the cost of network, or vice versa. The combination of the two algorithms can help reducing the network traffic as well as the burden on executing JavaScript events.

Appendix C

Cost of Discovering Application States in the Client-Server Architecture

C.1 Breath First Search Strategy

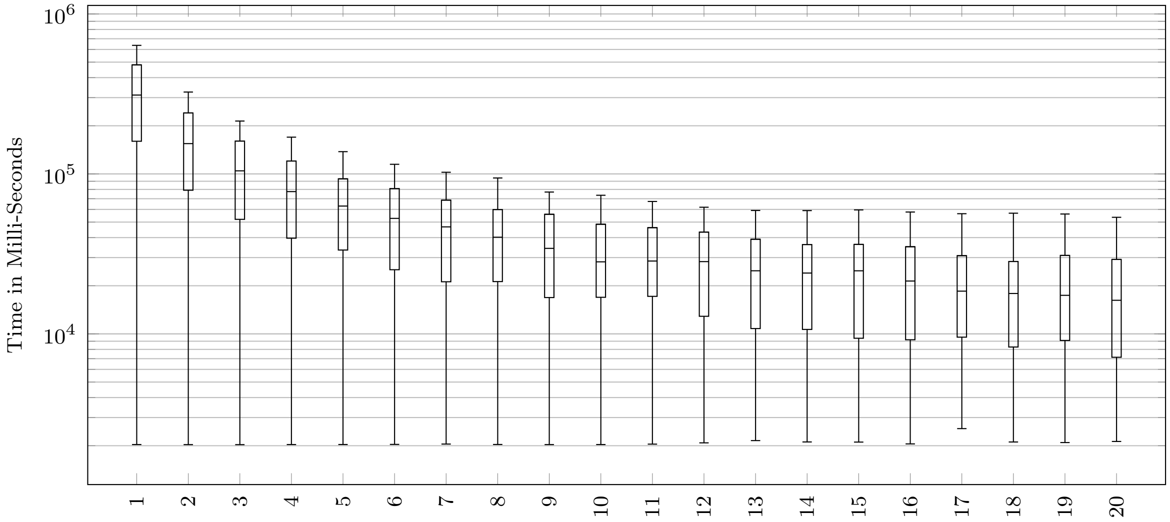


Figure C.1: Client-Server Architecture: Cost of discovering Test-RIA application states

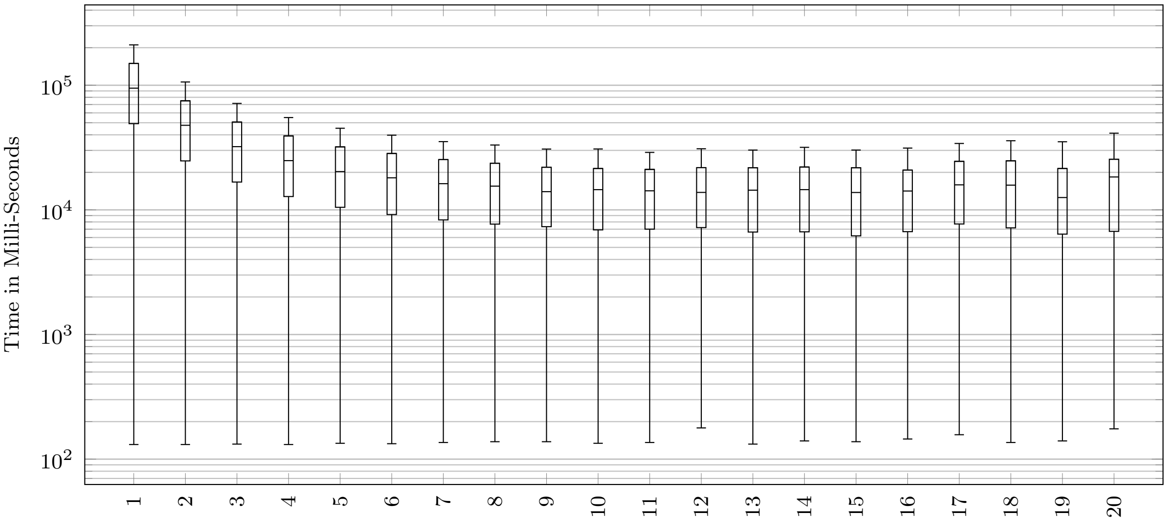


Figure C.2: Client-Server Architecture: Cost of discovering Altoro-Mutual application states

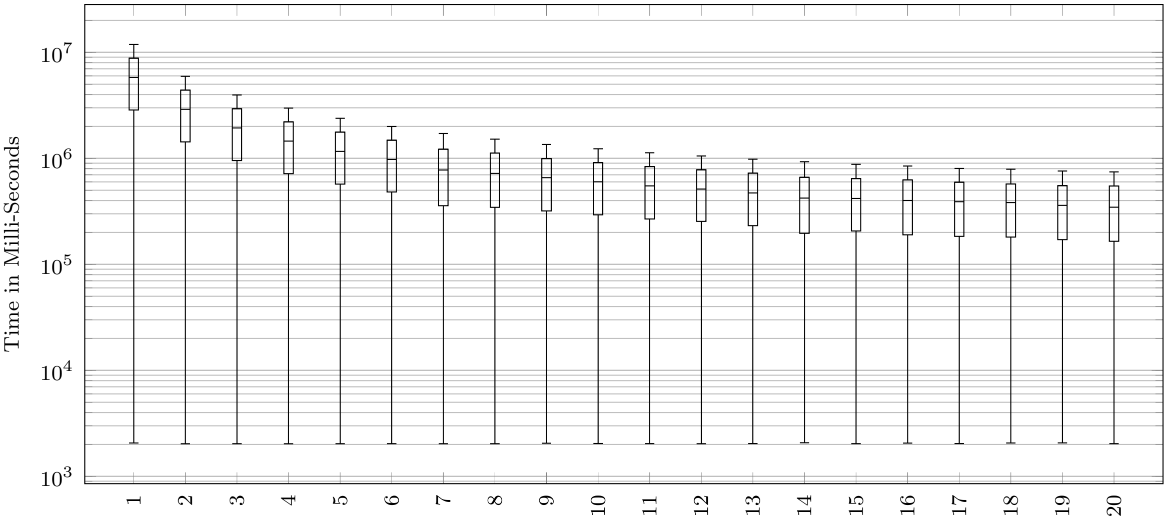


Figure C.3: Client-Server Architecture: Cost of discovering Dyna-Table application states

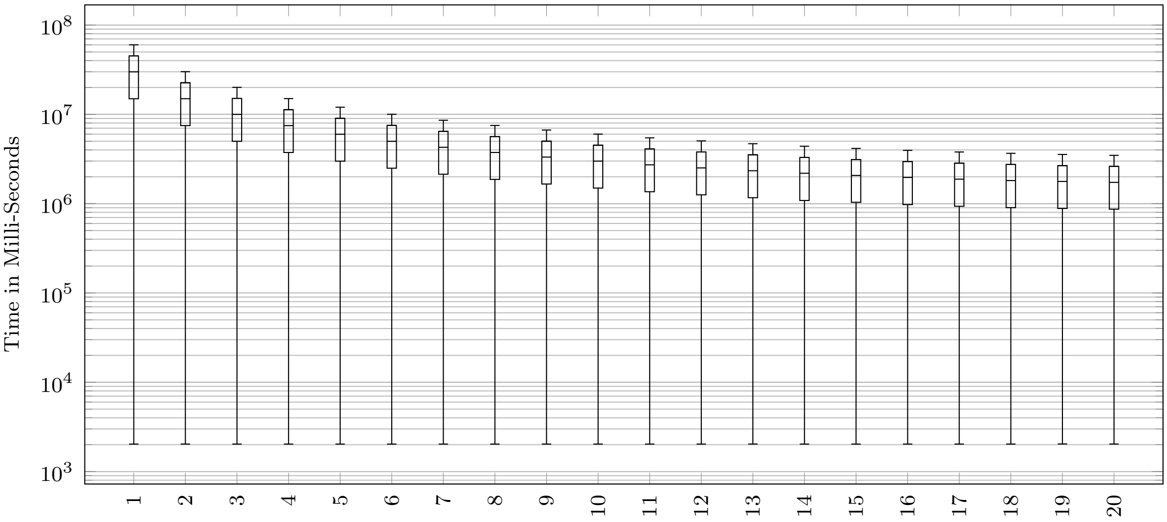


Figure C.4: Client-Server Architecture: Cost of discovering Periodic-Table application states

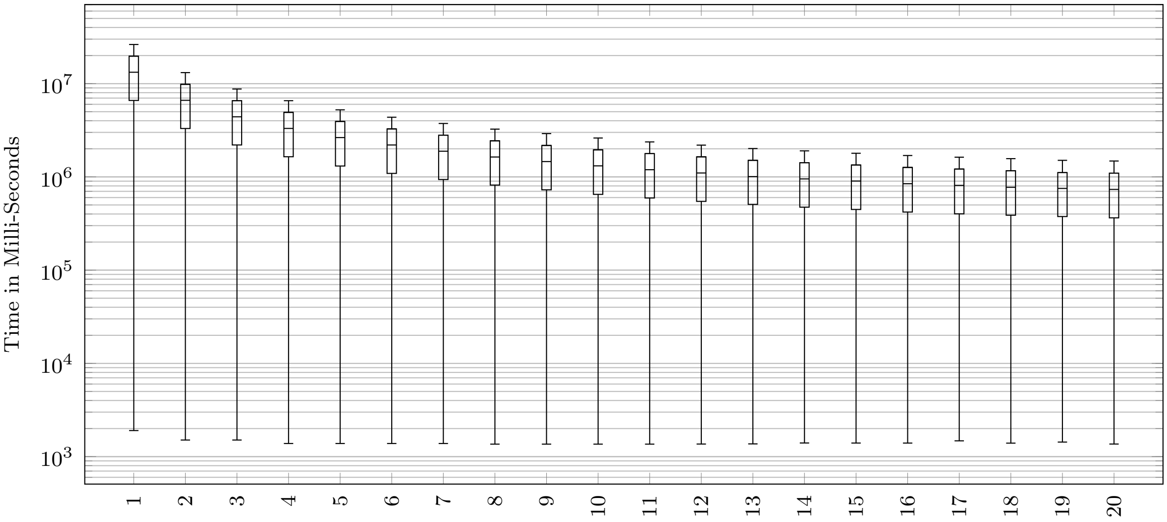


Figure C.5: Client-Server Architecture: Cost of discovering Clipmarks application states

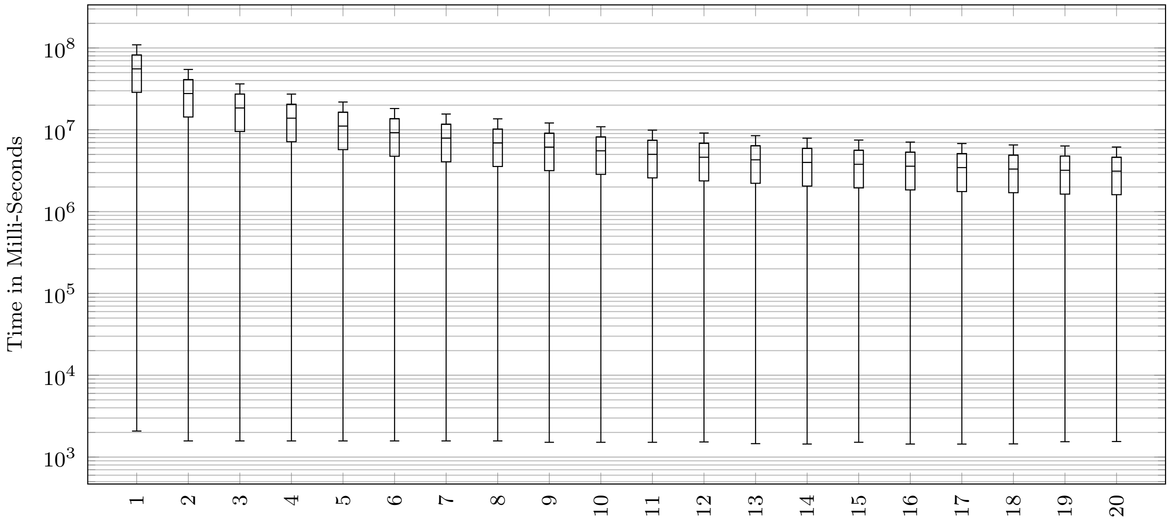


Figure C.6: Client-Server Architecture: Cost of discovering Elfinder application states

C.2 Greedy Strategy

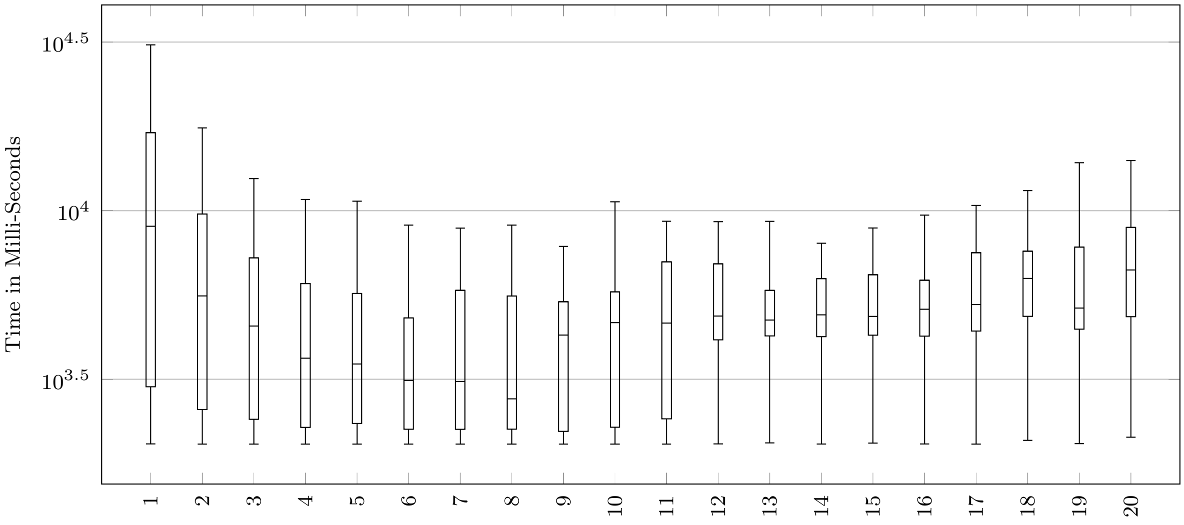


Figure C.7: Client-Server Architecture: Cost of discovering Test-RIA application states

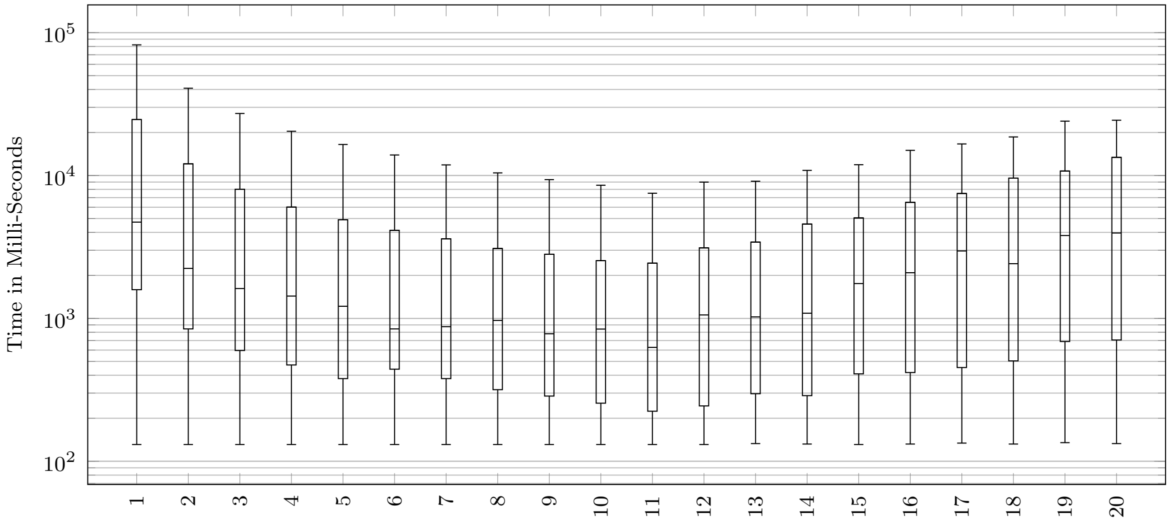


Figure C.8: Client-Server Architecture: Cost of discovering Altoro-Mutual application states

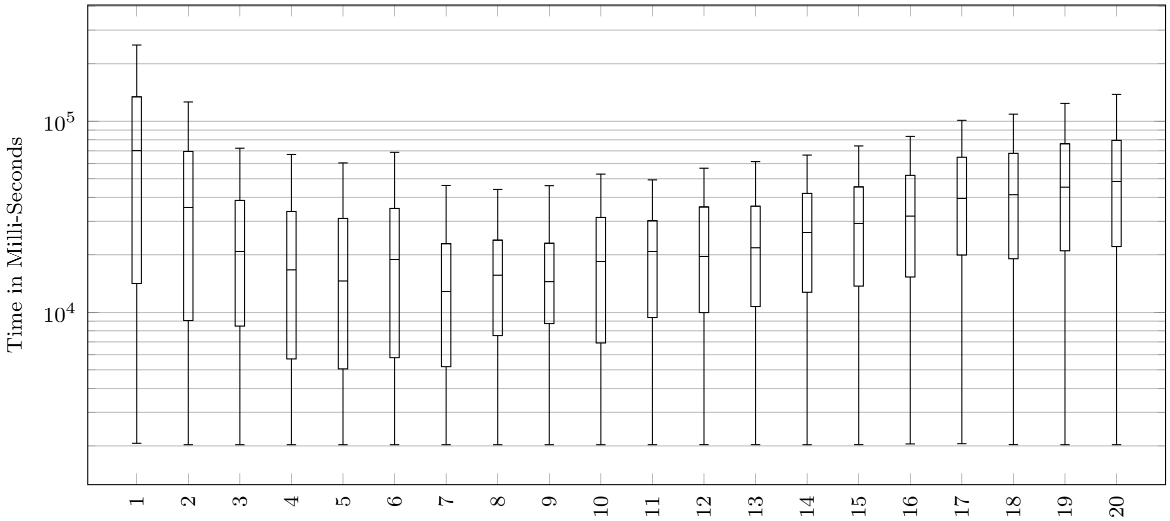


Figure C.9: Client-Server Architecture: Cost of discovering Dyna-Table application states

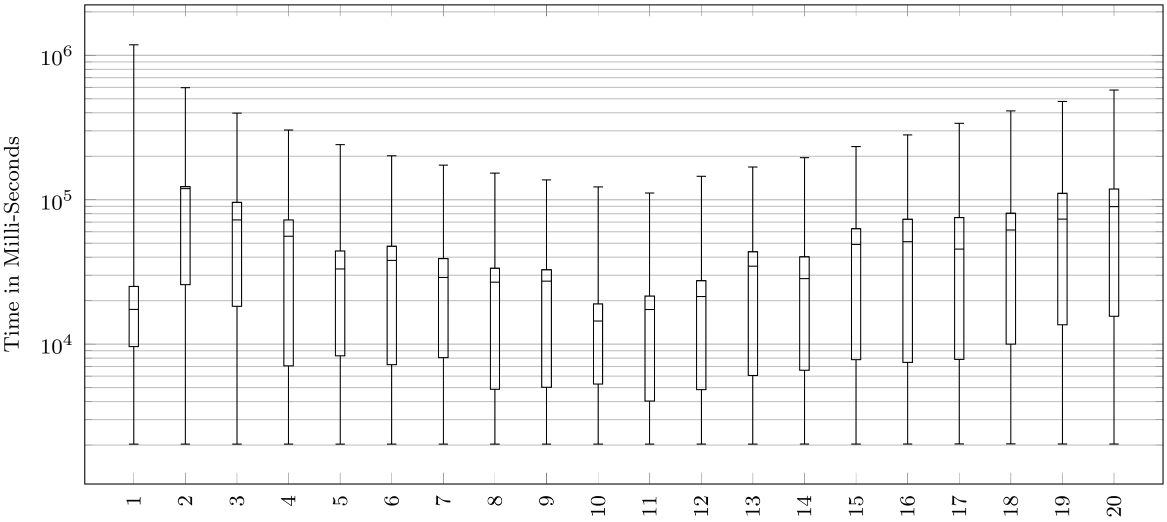


Figure C.10: Client-Server Architecture: Cost of discovering Periodic-Table application states

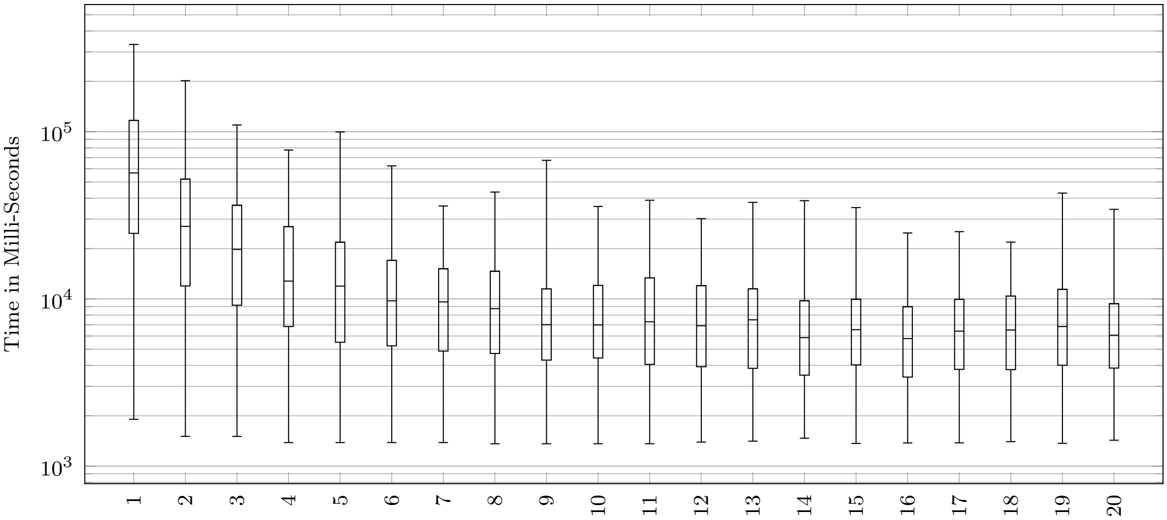


Figure C.11: Client-Server Architecture: Cost of discovering Clipmarks application states

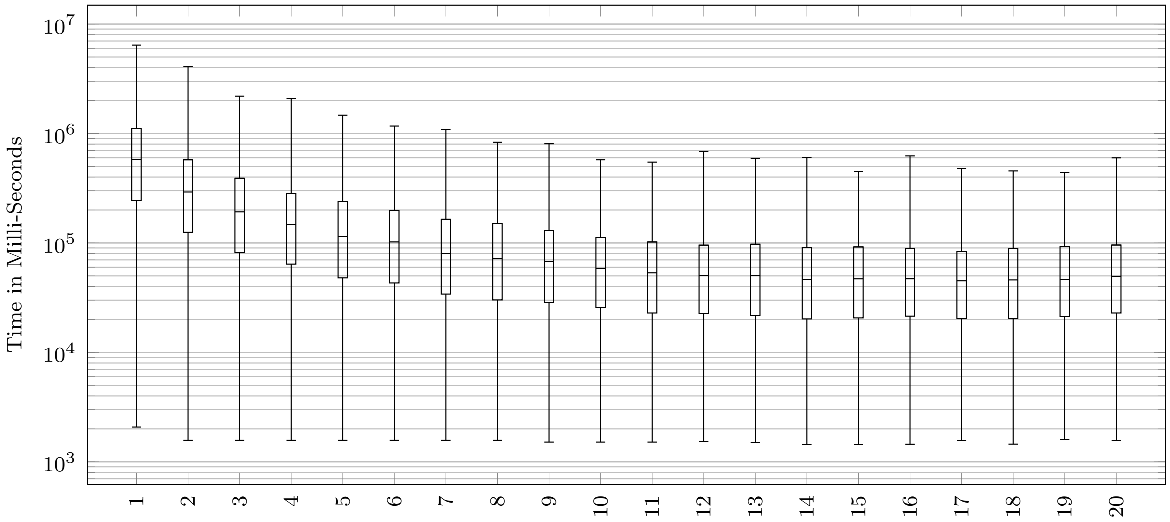


Figure C.12: Client-Server Architecture: Cost of discovering Elfinder application states

Appendix D

Cost of Discovering Application States in the Peer-to-Peer Architecture

D.1 Breath First Search Strategy

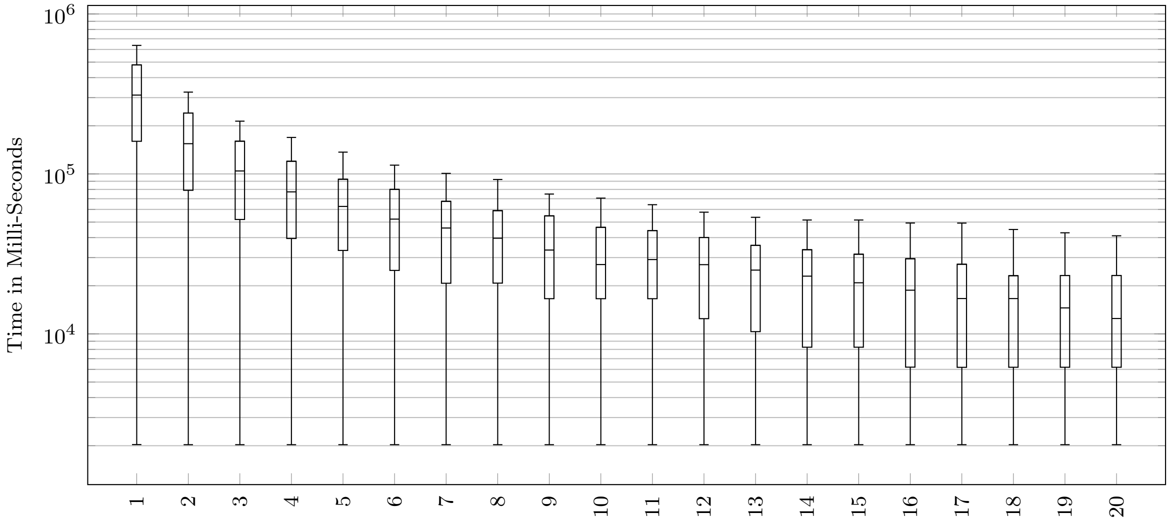


Figure D.1: Client-Server Architecture: Cost of discovering Test-RIA application states

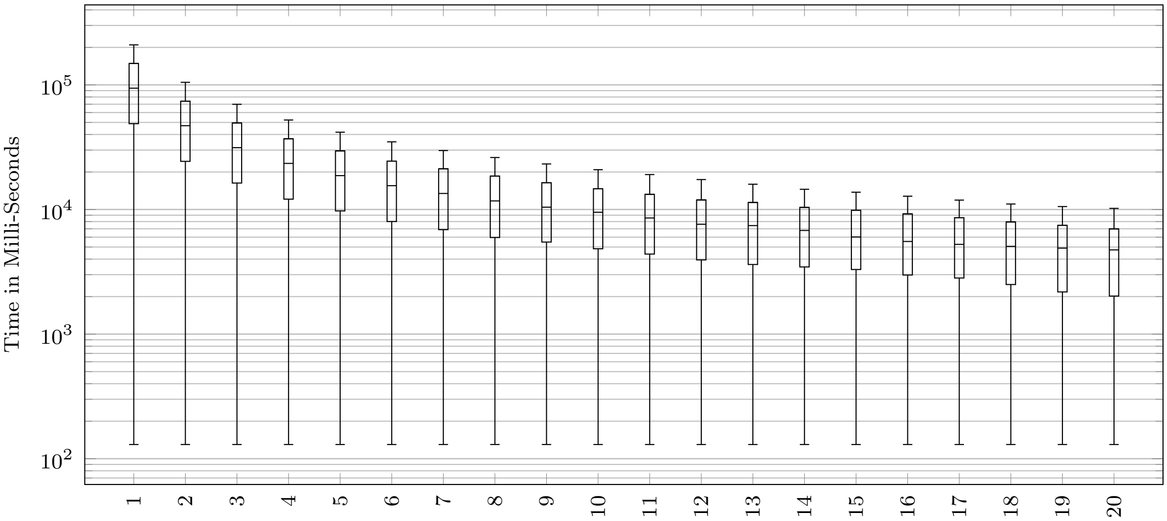


Figure D.2: Client-Server Architecture: Cost of discovering Altoro-Mutual application states

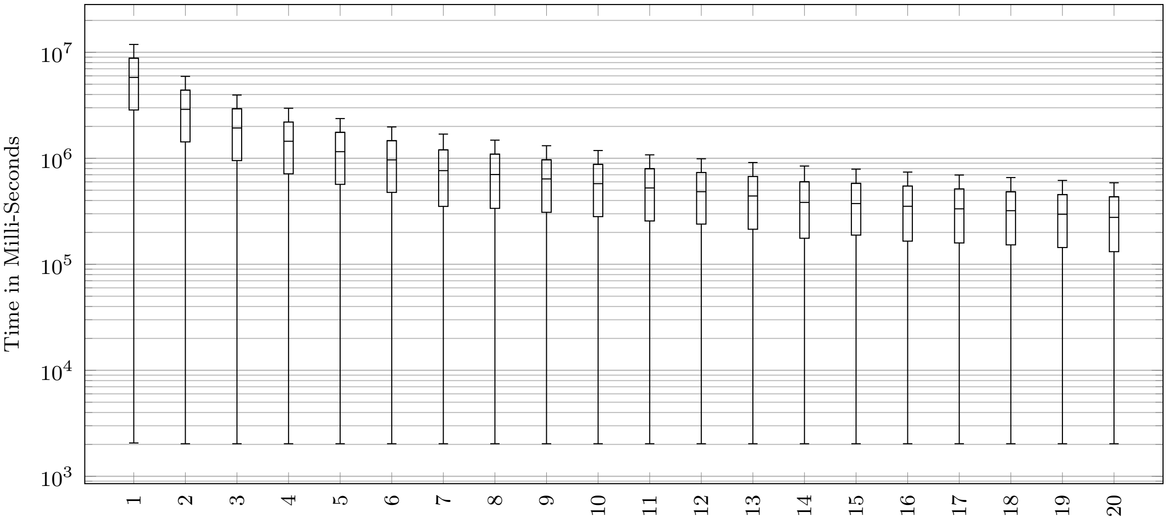


Figure D.3: Client-Server Architecture: Cost of discovering Dyna-Table application states

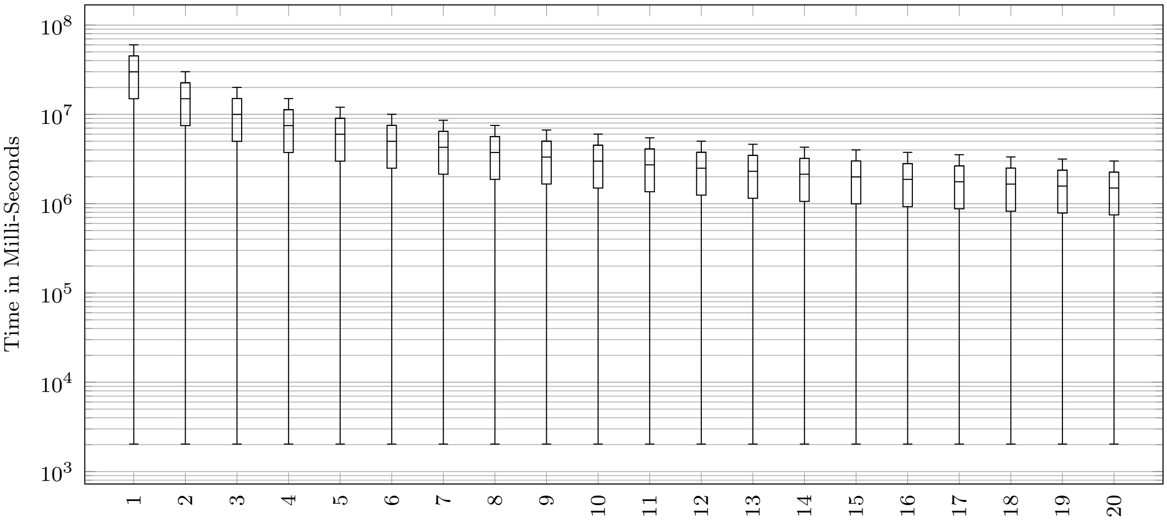


Figure D.4: Client-Server Architecture: Cost of discovering Periodic-Table application states

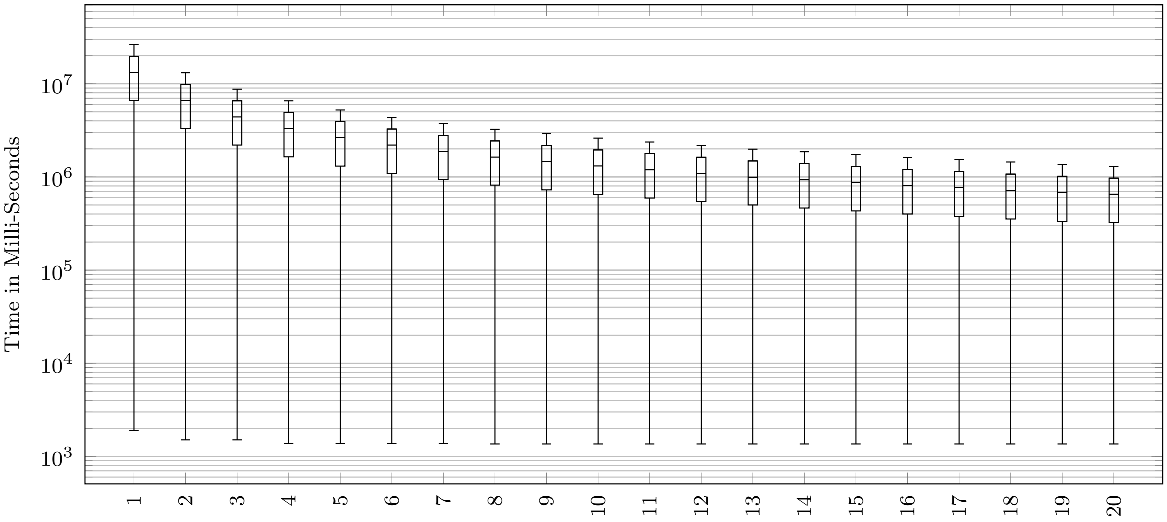


Figure D.5: Client-Server Architecture: Cost of discovering Clipmarks application states

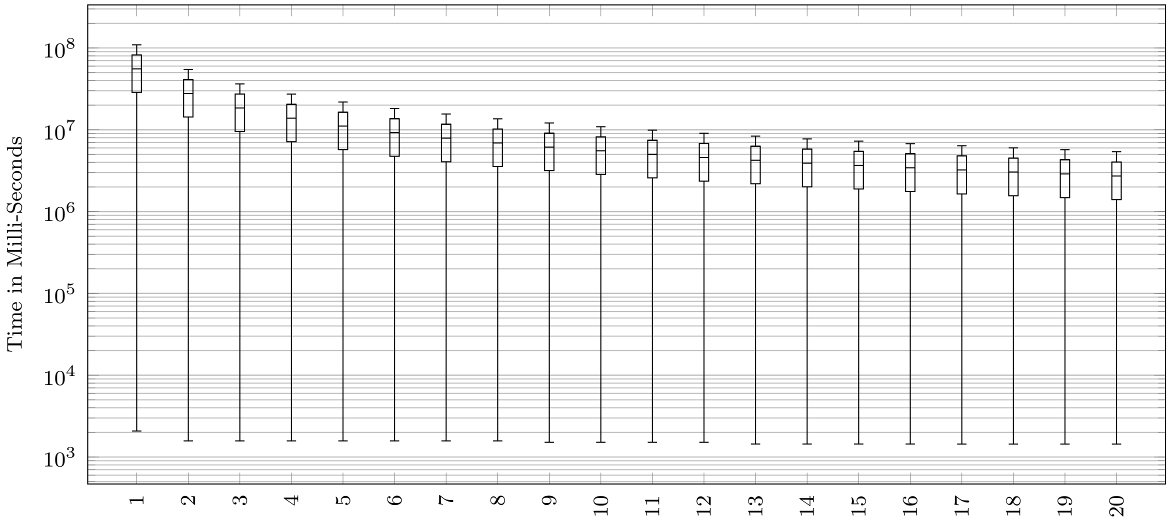


Figure D.6: Client-Server Architecture: Cost of discovering Elfinder application states

D.2 Depth First Search Strategy

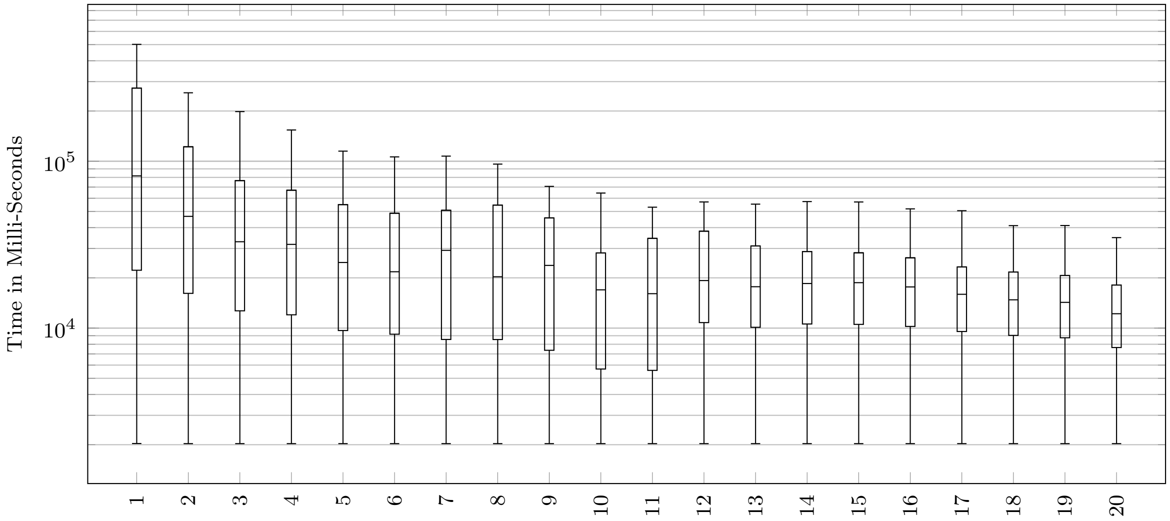


Figure D.7: Client-Server Architecture: Cost of discovering Test-RIA application states

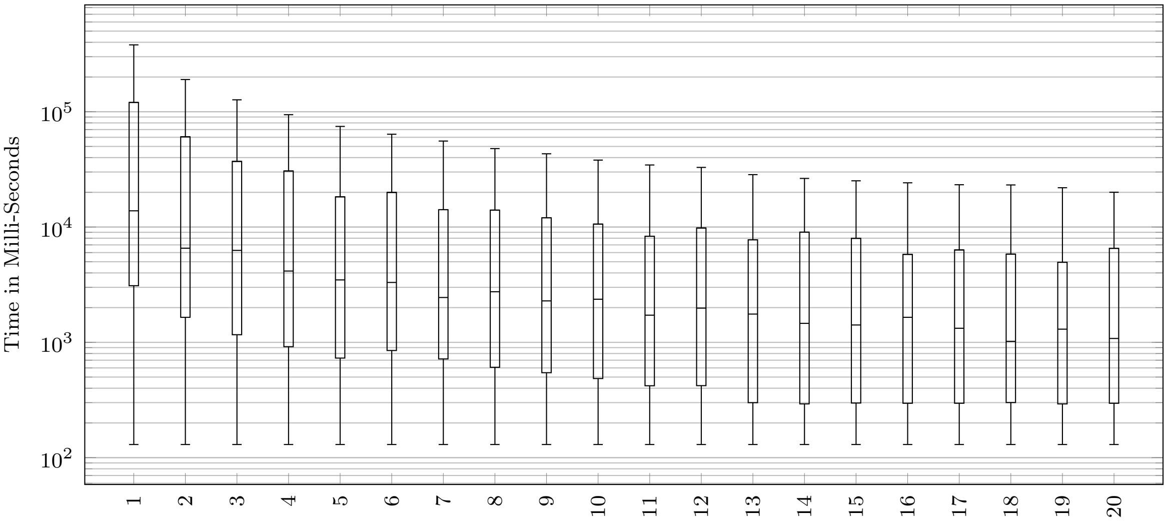


Figure D.8: Client-Server Architecture: Cost of discovering Altoro-Mutual application states

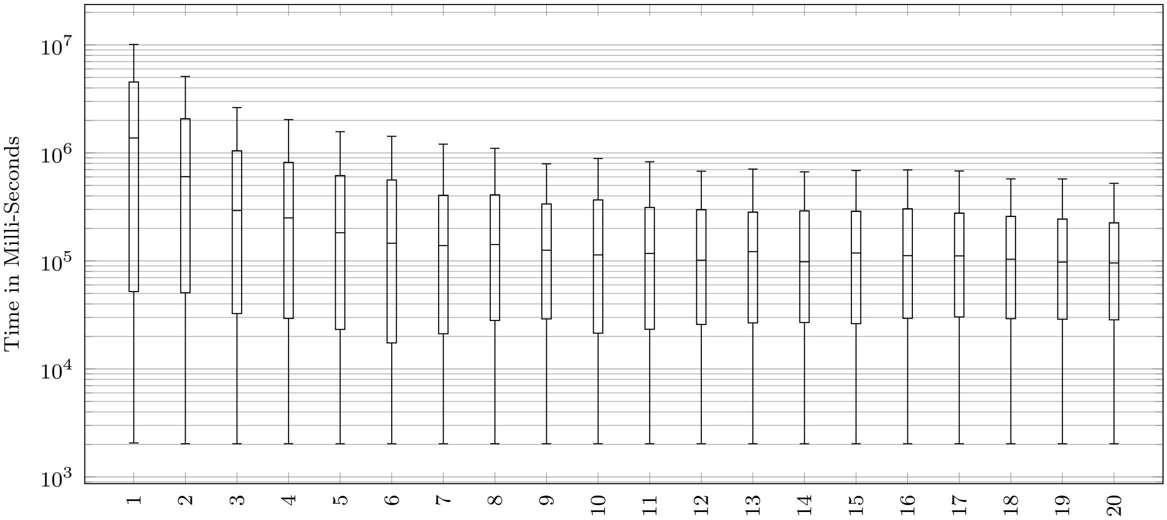


Figure D.9: Client-Server Architecture: Cost of discovering Dyna-Table application states

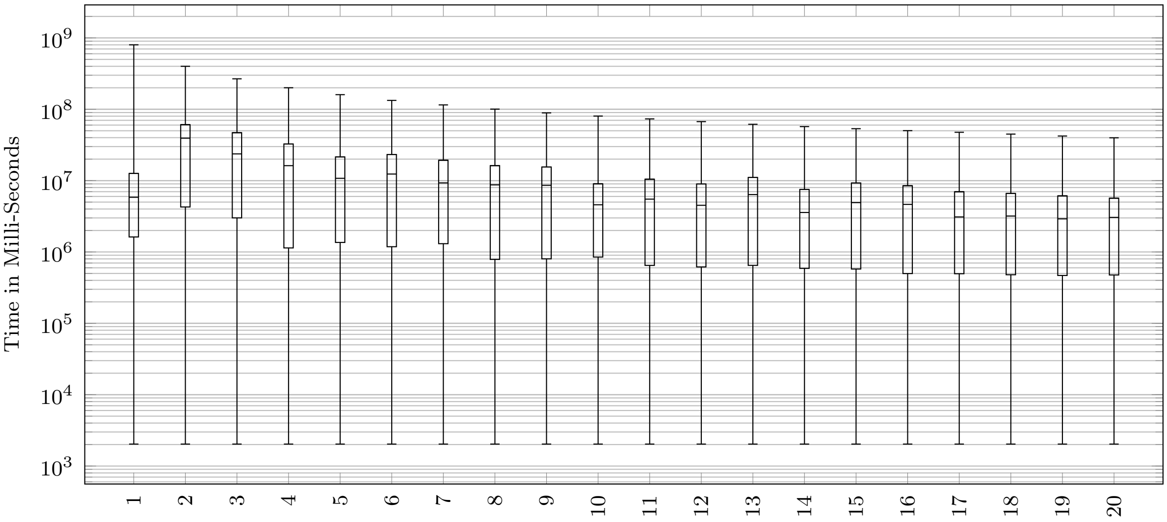


Figure D.10: Client-Server Architecture: Cost of discovering Periodic-Table application states

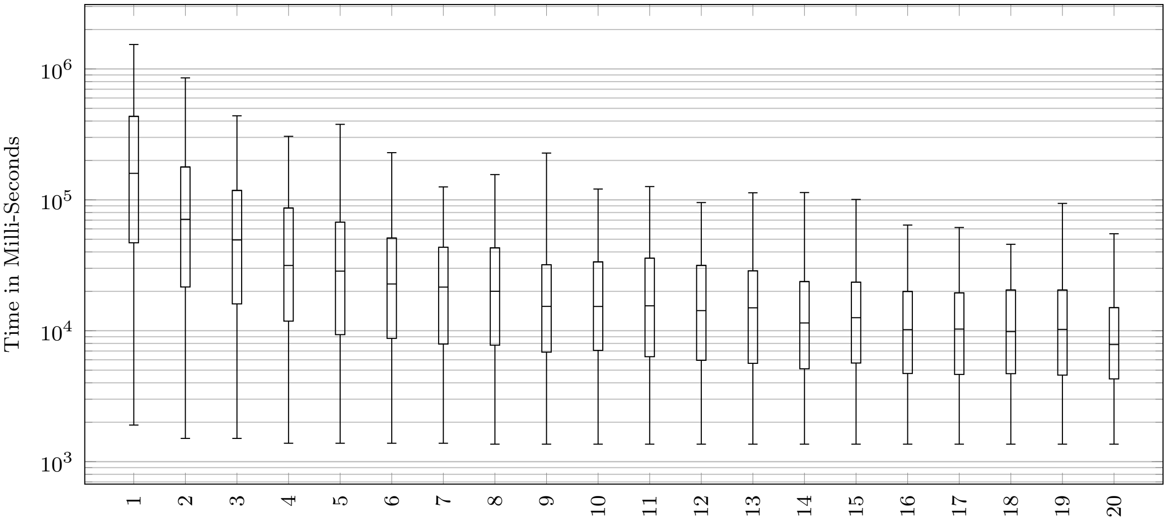


Figure D.11: Client-Server Architecture: Cost of discovering Clipmarks application states

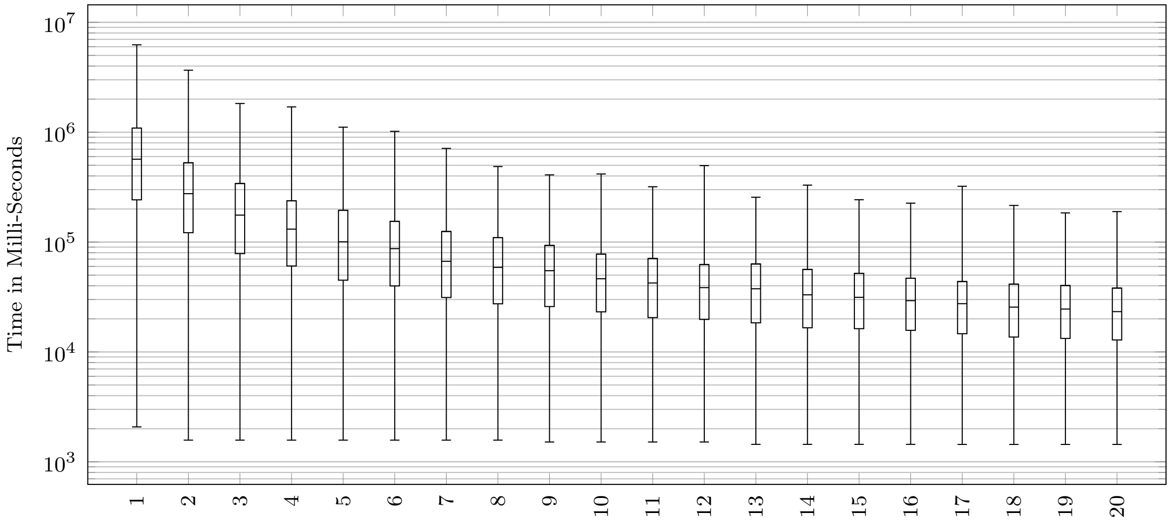


Figure D.12: Client-Server Architecture: Cost of discovering Elfinder application states

D.3 Greedy Strategy

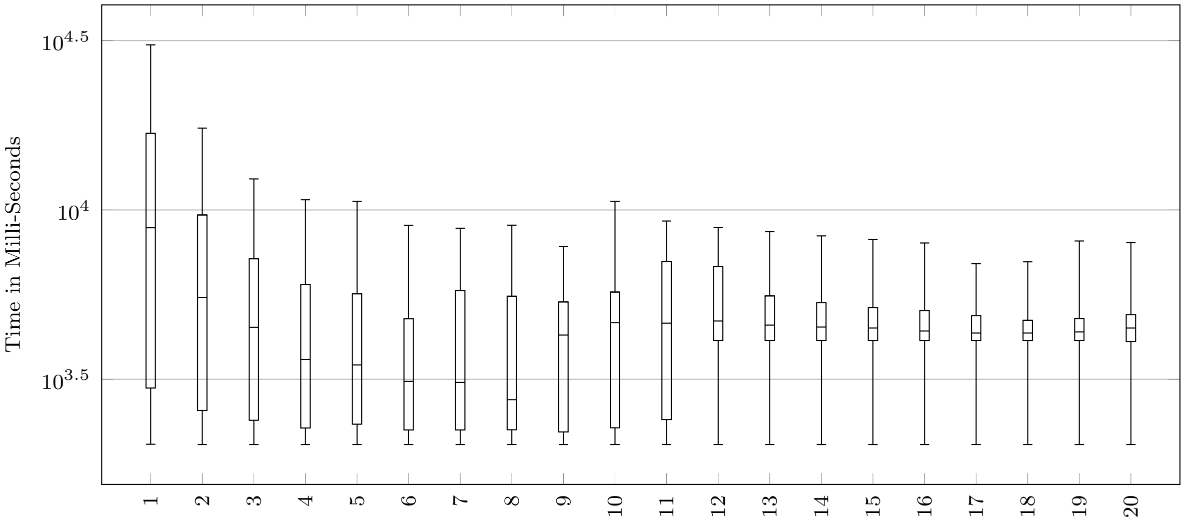


Figure D.13: Client-Server Architecture: Cost of discovering Test-RIA application states

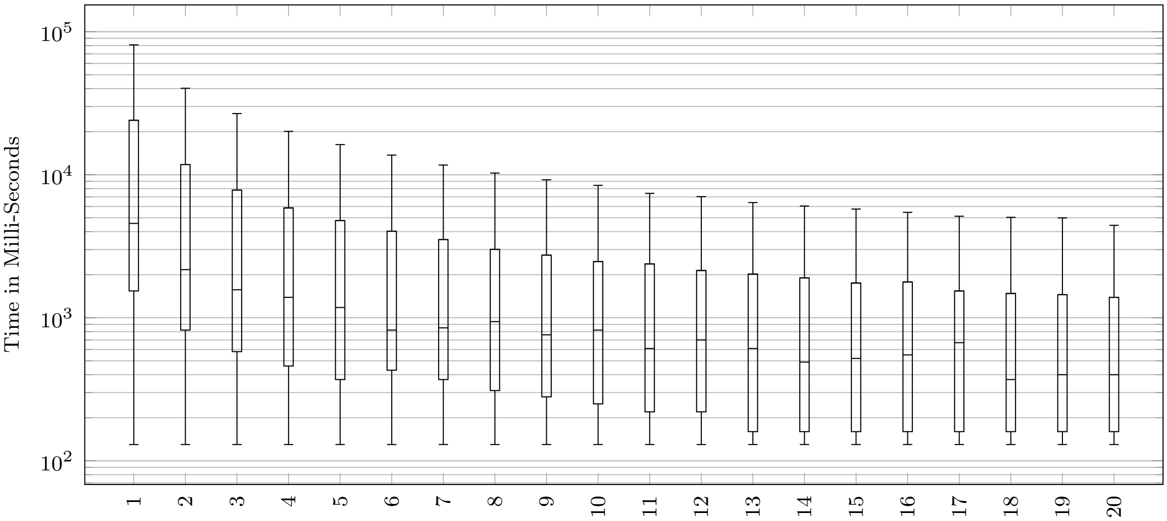


Figure D.14: Client-Server Architecture: Cost of discovering Altoro-Mutual application states

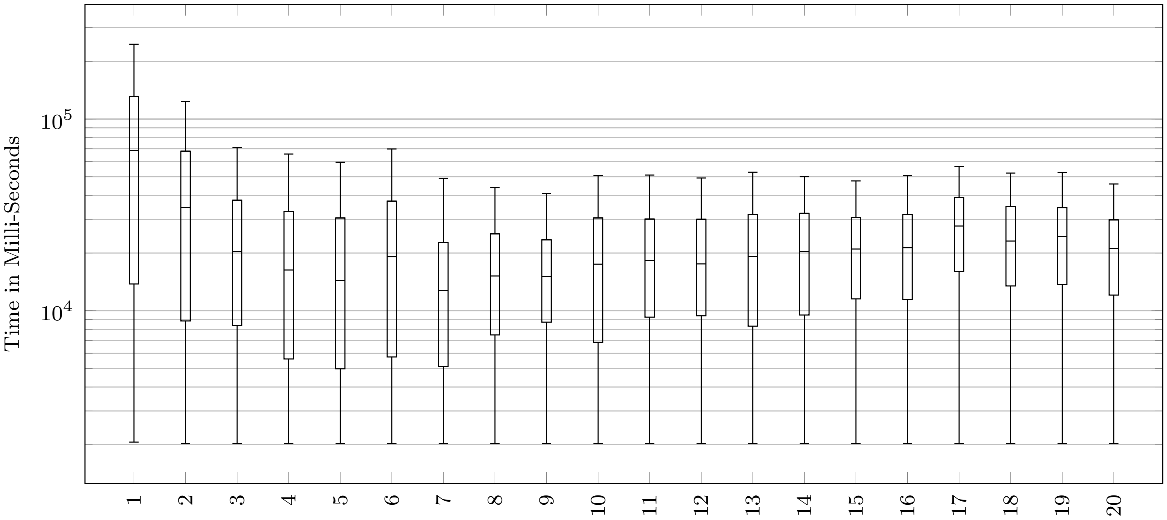


Figure D.15: Client-Server Architecture: Cost of discovering Dyna-Table application states

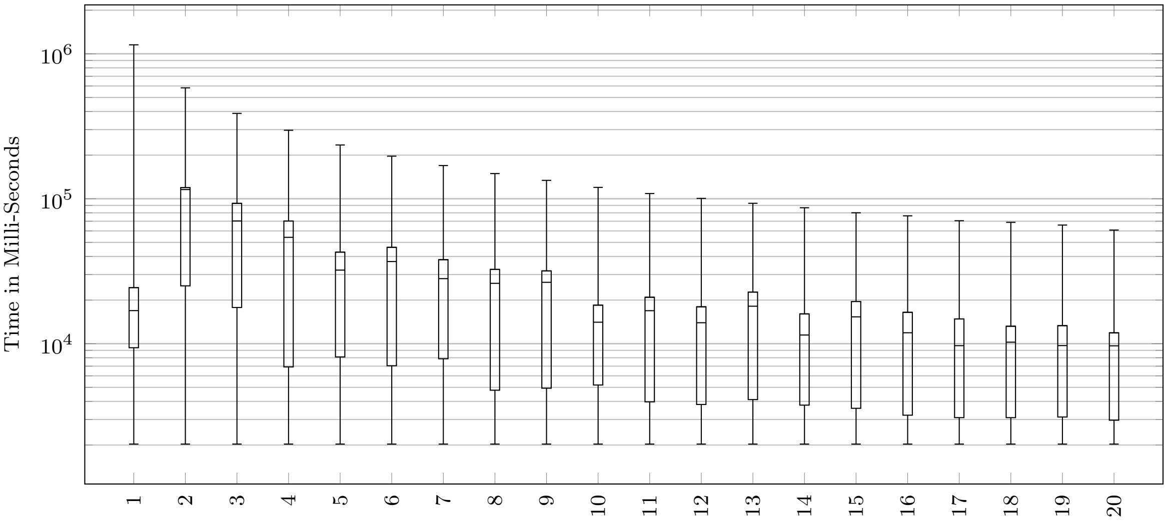


Figure D.16: Client-Server Architecture: Cost of discovering Periodic-Table application states

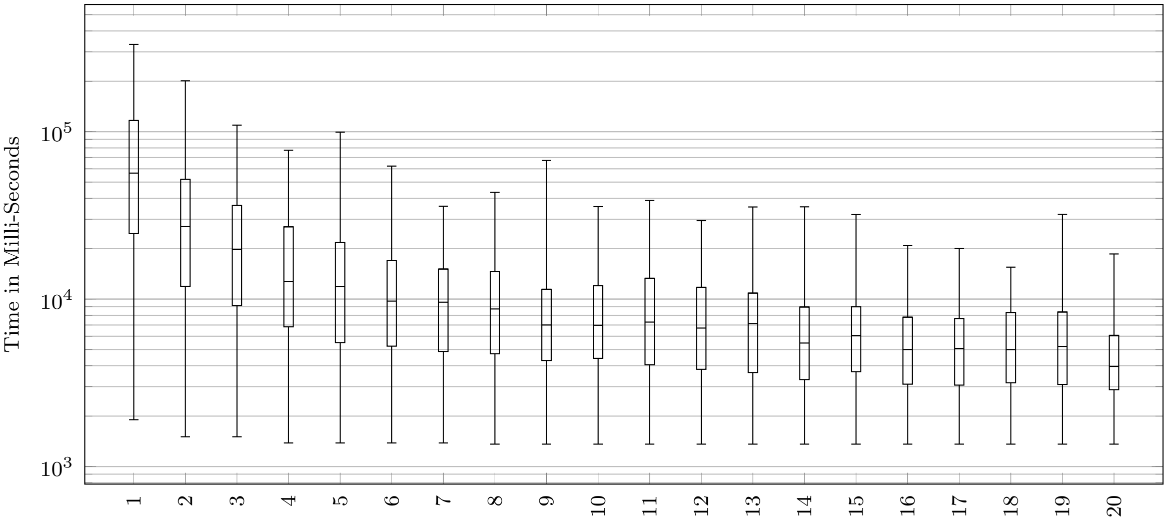


Figure D.17: Client-Server Architecture: Cost of discovering Clipmarks application states

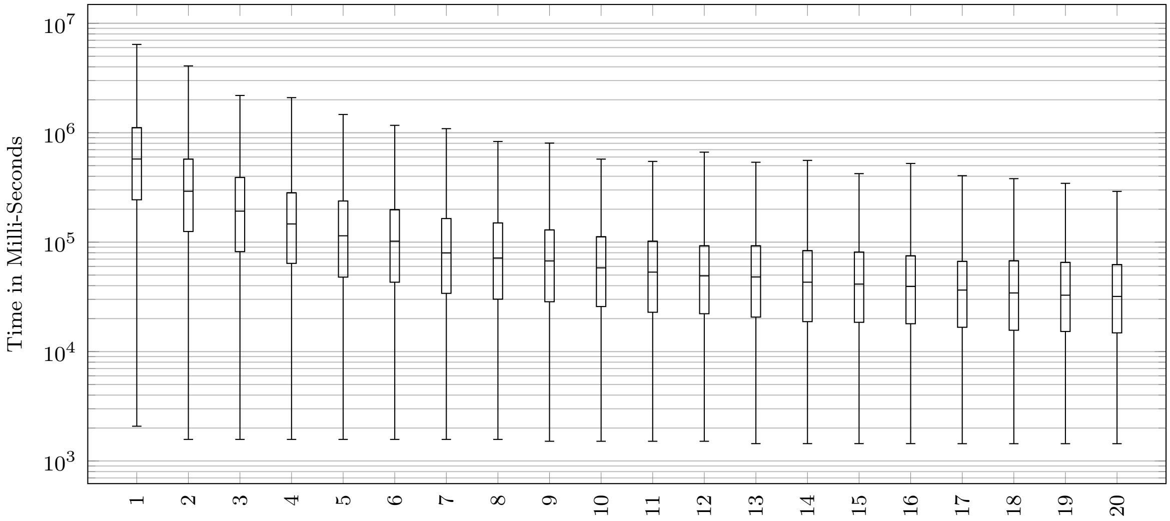


Figure D.18: Client-Server Architecture: Cost of discovering Elfinder application states

D.4 Probabilistic Strategy

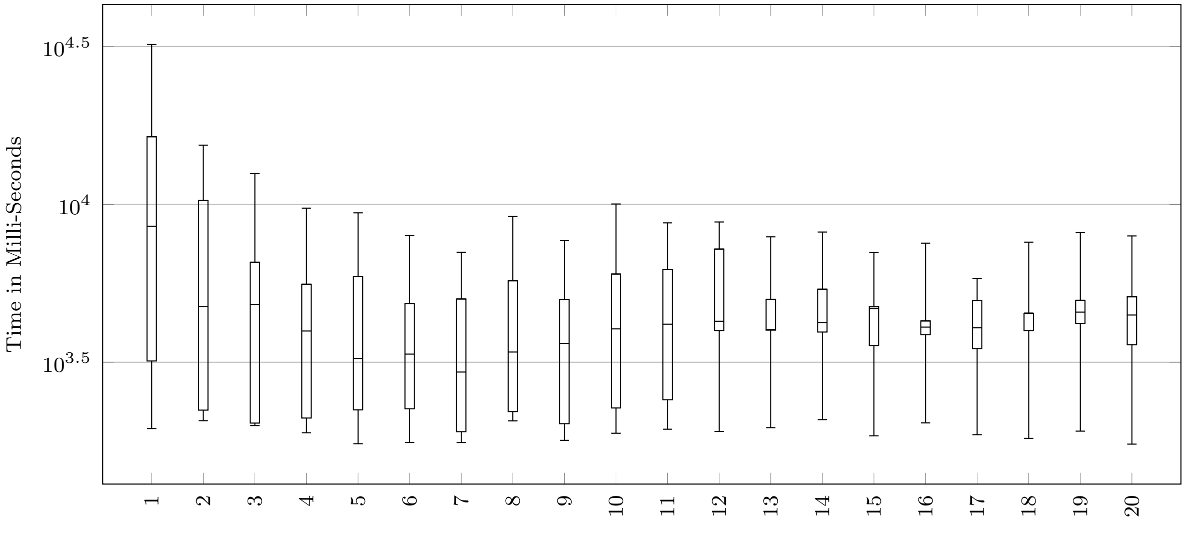


Figure D.19: Client-Server Architecture: Cost of discovering Test-RIA application states

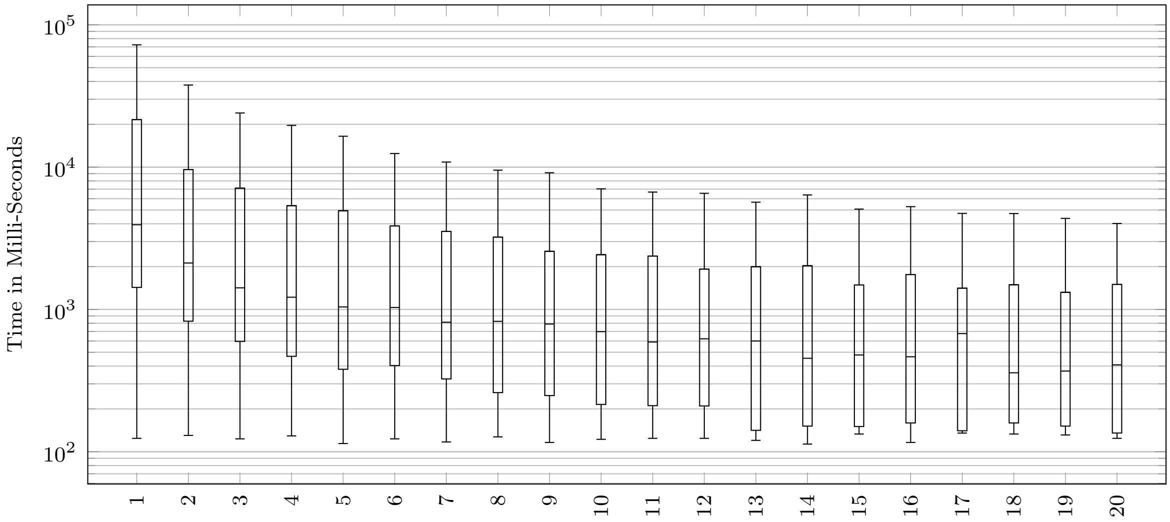


Figure D.20: Client-Server Architecture: Cost of discovering Altoro-Mutual application states

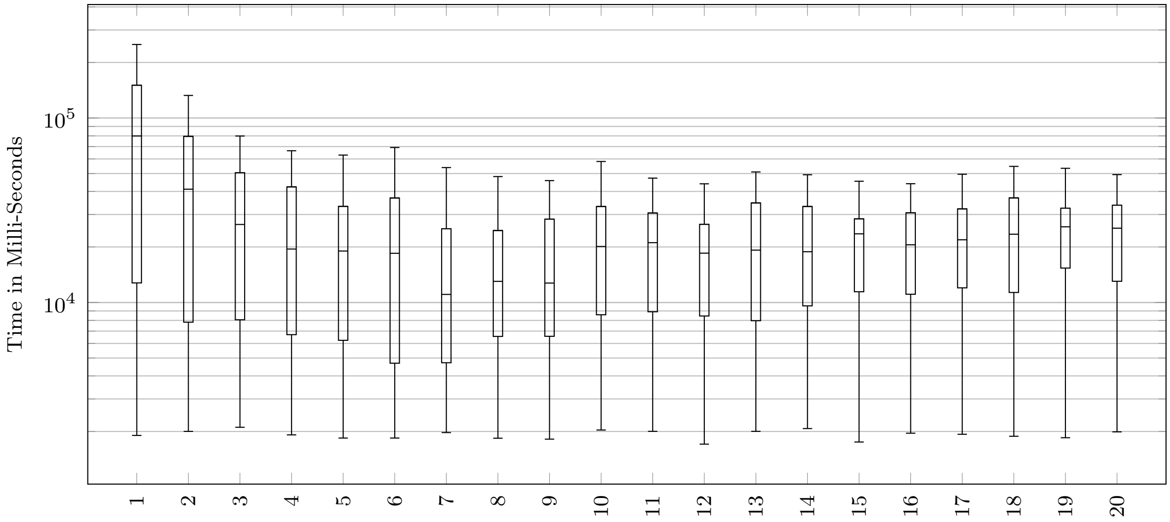


Figure D.21: Client-Server Architecture: Cost of discovering Dyna-Table application states

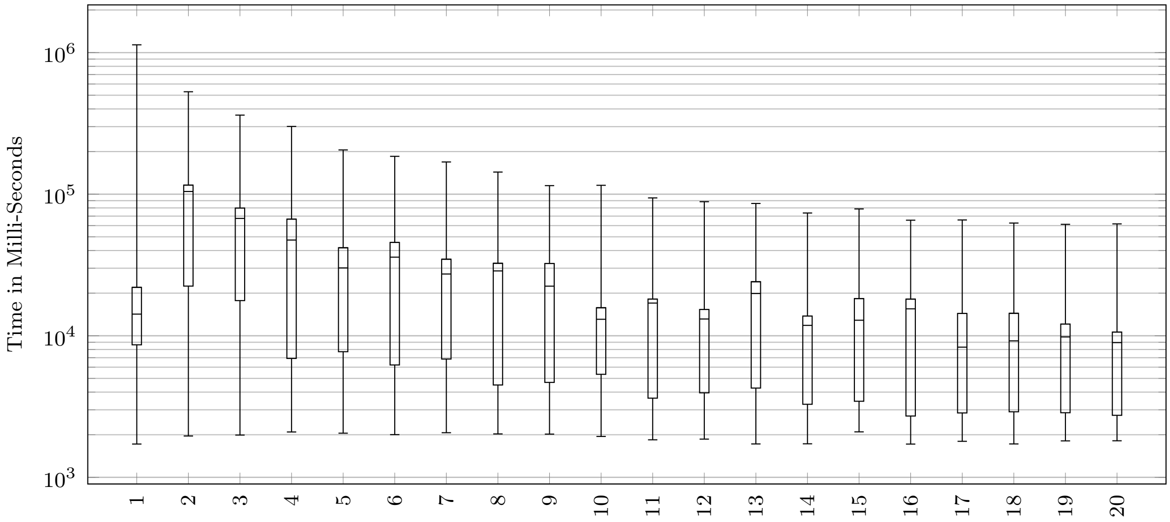


Figure D.22: Client-Server Architecture: Cost of discovering Periodic-Table application states

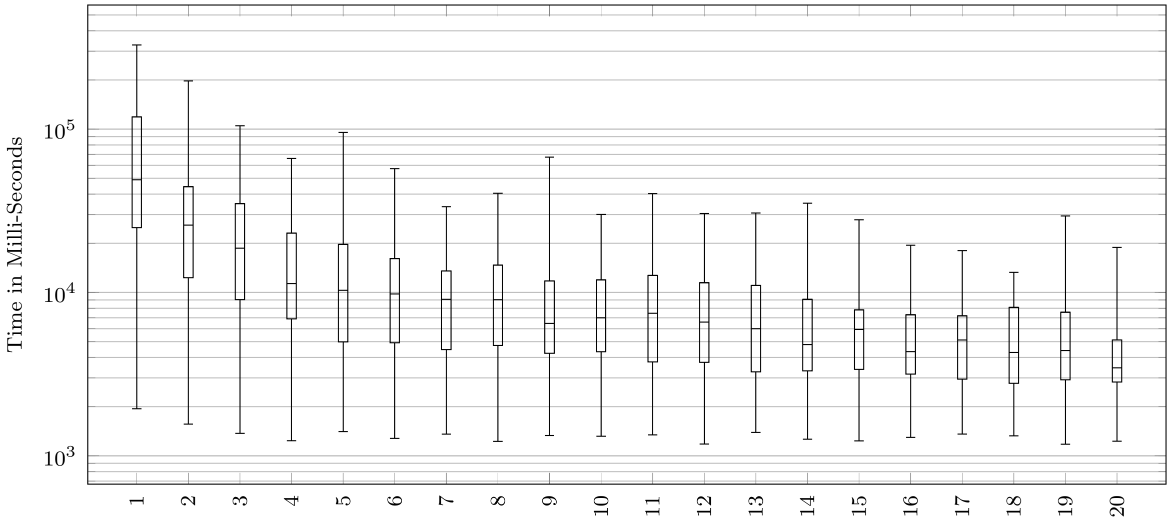


Figure D.23: Client-Server Architecture: Cost of discovering Clipmarks application states

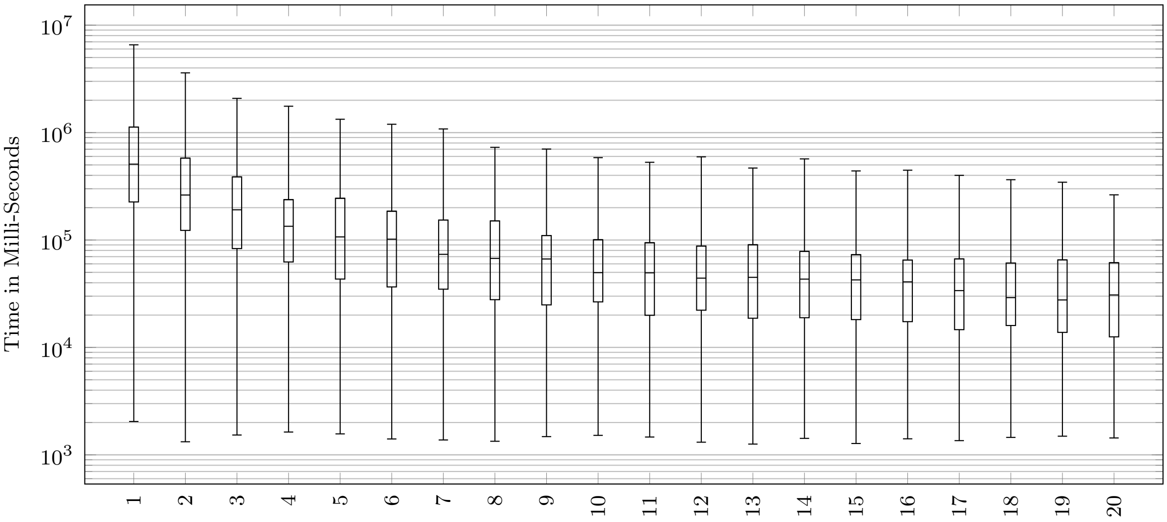


Figure D.24: Client-Server Architecture: Cost of discovering Elfinder application states