A Scalable P2P RIA Crawling System with Fault Tolerance

Khaled Ben Hafaiedh

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for a Doctorate in Philosophy - Ph.D. degree in
Electrical and Computer Engineering

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

Abstract

Rich Internet Applications (RIAs) have been widely used in the web over the last decade as they were found to be responsive and user-friendly compared to traditional web applications. RIAs use client-side scripting such as JavaScript which allows for asynchronous updates on the server-side using AJAX (Asynchronous JavaScript and XML).

Due to the large size of RIAs and therefore the long time required for crawling, distributed RIA crawling has been introduced with the aim to decrease the crawling time. However, the current RIA crawling systems are not scalable, i.e. they are limited to a relatively low number of crawlers. Furthermore, they do not allow for fault tolerance in case that a failure occurs in one of their components. In this research, we address the scalability and resilience problems when crawling RIAs in a distributed environment and we explore the possibilities of designing an efficient RIA crawling system that is scalable and fault-tolerant. Our approach is to partition the search space among several storage devices (distributed databases) over a peer-to-peer (P2P) network where each database is responsible for storing only a portion of the RIA graph. This makes the distributed data structure invulnerable to a single point of failure. However, accessing the distributed data required by crawlers makes the crawling task challenging when the number of crawlers becomes high. We show by simulation results and analytical reasoning that our system is scalable and fault-tolerant. Furthermore, simulation results show that the crawling time using the P2P crawling system is significantly faster than the crawling time using both the non-distributed crawling system and the distributed crawling system using a single database.

Acknowledgments

I would like to express my deepest gratitude toward my supervisors Dr. Gregor von Bochmann (University of Ottawa), Dr. Guy-Vincent Jourdan (University of Ottawa) and Dr. Iosif Viorel Onut (IBM) for their academic support during my study, and for their helpful suggestions and advices. Thanks for converting my mistakes into lessons, pressure into productivity and skills into strengths. Your comments on my failures and your compliments on my performance, both motivate and inspire me to do better. This Ph.D thesis would not be possible without your guidance.

This work was in part supported financially by IBM Center for Advanced Studies (IBM CAS) and Natural Sciences and Engineering Research Council of Canada (NSERC). I would like to express my sincere gratitude for your financial support during the school session.

My deepest gratitude also goes out to my colleges at the Software Security Research Group and IBM for their continuous support and guidance. Very special thanks go to my colleges Muhammad Faheem, Salman Hooshmand, Emre Dincturk, Seyed M. Mirtaheri, Di Zou, Suryakant Choudhary, Sara Baghbanzadeh, Akib Mahmud, Ali Moosavi and Xinghao Xu. You showed me patience instead of anger, guidance instead of annoyance and understanding instead of intolerance. Thanks for being so supportive. I am proud of working with extraordinary colleagues like you. Thank you for making my work-life truly extraordinary.

I would also like to thank all the people at the University of Ottawa I have had the pleasure to work with.

The most special thanks go to my parents, my two sisters and their husbands who are always there for me and pushing me to work hard for my education.

Finally, I would like to thank all the people who made this possible.

Dedication

To the most loved people in my life

My parents Khaled Senior Hafaiedh and Nahla Senior Hafaiedh, my two sisters Nahla Junior Hafaiedh Chelli and Nouha Hafaiedh, and my two brothers-in-law Mourad Chelli and Slim Karray: Without you, my life would fall apart.

My beloved Dad, thank you for all the love, care and success for your family and children that you want to them more than you want it to you , thank you for teaching me to stand firm until the end, and never surrender, thank you for making me what I am today.

My beloved Mom, thank you for all the love you keep inside you, thank you for all the sweetness and tenderness that emerges from your humble soul. I always think of you.

My beloved Sister Nahloula, thank you for your support and encouragement for what I do. I wish you all the success for your professional career, I also wish you much love and happiness with your husband Mourad, your little princess Nour Nahla and for your familial and professional career.

My beloved Sister Nounou, thank you for always being there for me when I need it. You have always taken care of me since we have been living under the same roof in Canada. Your presence is very important to me. I wish you love, joy and happiness with your future husband Slim.

My five best Friends Ali Bouthiba (France), Taieb Annabi (Tunisia), Yassine Ghazouani (Morocco), Sadry Soudani (Canada) and Mohamed Fadhel Ben Rhouma (Tunisia), thank you for being a different friend than everyone else, different from the friends who are only there for the fun things. Thank you for doing all the things real best friends do. Thank you for your care, kindness and thoughtfulness I will not soon forget. You are like a five leaf clover, hard to find, and lucky to have.

Table of Contents

List of Tables

List of Figures

Nomenclature

List of Abbreviations

(in alphabetical order)

- **AJAX:** Asynchronous JavaScript and XML.
- **DHT:** Distributed Hash Table.
- **DOM:** Document Object Model.
- **P2P:** Peer-to-peer.
- **RIA:** Rich Internet Application.
- **URL:** Uniform Resource Locator.
- **XML:** Extensible Markup Language.

Mathematical Notation

(in alphabetical order)

- c**:** Average communication delay of a direct message between two nodes. (in milliseconds)

- e**:** Average time required for executing a new transition, which includes going through a path of ordered transitions before reaching the state with the next transition to be executed. (in milliseconds)
- e_{max}**:** Maximum time required for executing a new transition. (in milliseconds)
- k**:** Number of transitions in a RIA.
- n**:** Number of controllers in the P2P Crawling system.
- N_t**:** Number of transitions to be updated in the database per update period for the Combined strategy.
- m**:** Number of crawlers in the P2P Crawling system.
- p**:** Average processing time required for updating the database for the Redundancy strategy. (in milliseconds per transition)
- r**:** Number of back-up copies maintained by neighboring controllers.
- s**:** Number of states in a RIA.
- T**:** Total crawling time with normal operation, i.e. with no failures among controllers and crawlers.
- $time - out_{Crawler}$**:** Time-out set for an assigned transition to be executed by a given crawler. (in milliseconds)
- T_p**:** Update period for the Combined strategy. (in milliseconds)
- t_t**:** Average time required for executing a new transition. (in milliseconds)
- $\lambda_{DedicatedServers}$**:** Failure rate of dedicated servers, where nodes can only communicate with each other on a private network. (in failure per hour)

- λ_{P2P}**:** Failure rate of the P2P network, where nodes are publicly accessible from a P2P infrastructure. (in failure per hour)

- δ**:** Parameter describing the load of controllers.

Chapter 1

Introduction

1.1 Web Crawling

The typical purpose of a web crawler [23] is to systematically browse the World Wide Web, typically for the purpose of web indexing, security and accessibility testing. Moreover, web crawling potentially improves user experience by preventing the user from reaching irrelevant or undesirable content, unreachable pages, unavailable or outdated pages and even malicious pages. As the content on the internet is extremely large, web crawlers must apply a good crawling strategy to decide how the exploration proceeds, so that more relevant data is delivered to users during the allowed time slot.

1.2 Traditional Web Crawling

The typical interaction between the client and the server in a traditional web application consists of sending a request for a URL from the client to the server so that the corresponding web page is synchronously downloaded in response for each URL request. In traditional web crawling, the web pages are exclusively generated on the server-side. While requests are being processed on the server, users cannot interact with the client web page.

Traditional web crawlers usually crawl through a site a page at a time. The crawler starts from a given URL, downloads the corresponding page, extracts all the URLs embedded in the page and follows these URLs to other pages and so on, until all pages have been discovered.

In a traditional web application graph model, each URL is modeled as a vertex of the application graph and each link is modeled as an edge. Crawling a traditional web application is to find all its URLs where each web page is identified by one URL and has only a single state per URL. Any URL may be directly reached by exploring the corresponding link leading to this URL. The client would continuously crawl the links contained for each downloaded web page until all URLs of a given web domain are explored. Several studies have been conducted to improve the time required for crawling traditional web applications over the typical well-known search strategies such as the Breadth-First and the Depth-First strategies [58].

1.3 Distributed Traditional Web Crawling

Other directions for improvement have been considered by distributing the crawling task among multiple crawlers. In the concurrent environment, each crawler explores only a subset of the state space by contacting one or more units that are responsible for storing the application URLs and coordinating the exploration task among crawlers, referred to as controllers. Different approaches have been categorized to concurrently crawl traditional web applications [40]. We distinguish two types of distribution: The centralized and the decentralized distributions. In a centralized distribution, a single controller is responsible for storing a list of the newly discovered URLs and gives the instruction of loading each non-explored URL to an idle crawler [40] [96]. However, such a system has a single point of failure. If a failure occurs within the controller, all data collected by the crawlers is lost. Moreover, the single controller may become overloaded as the number of crawlers

increases and it would not be able to efficiently process the concurrent requests when the number of crawlers is high. Peer-to-peer traditional crawling systems have been introduced to continue the crawling task, possibly at a reduced level, rather than failing completely, by avoiding a single point of failure. In peer-to-peer [94] traditional crawling systems, the URLs are partitioned over several databases where each database is responsible for a set of URLs. In this system, crawlers can find locally the identifiers of a database by mapping the hash of each discovered URL information using the Distributed Hash Table (DHT) [107] [82], i.e. each URL is associated with a single database in the DHT. The crawler then stores this URL on its corresponding database, thereby associating each database with only a portion of the search space . Chord [39] [6] is one of the four original DHT protocols over structured networks, along with CAN [102], Tapestry [117] and Pastry [108]. These systems have been used in traditional web crawling in a faulty environment and are well known for their scalability and low latency. However, their performance may degrade when nodes are joining, leaving or failing, due to their tightly controlled topologies.

1.4 RIA Crawling

As the web has evolved towards dynamic content, modern web technologies gave birth to interactive and more responsive applications, referred to as RIAs, which combine client-side scripting with new features such as AJAX (Asynchronous JavaScript and XML) [47]. This allows the client to modify the currently displayed page without communicating with the server, resulting in a smoother functionality and more interactive experience. In a RIA, JavaScript functions allow the client to modify the currently displayed page and to execute JavaScript events in response to user input asynchronously using its Document Object Model (DOM) [31], without having the user to wait for a response from the server. These new features introduce new challenges to automate the crawling of RIAs [57] as they result in a graph of multiple states derived from each single URL.

In RIAs, states represent the distinct pages (DOM instances) in a RIA model [99], while transitions illustrate the possible ways to move from one page to another, without changing the URL. The triple $(SourceState, event, DestinationState)$ describes a transition of a RIA. Exploring a RIA is referred to as event-based crawling since a transition is usually executed by triggering a JavaScript event at the user interface. Automated event-based crawling consists of automatically invoking each of the possible user-interactions of a given page, starting from the initial page that follows from loading its corresponding URL.

Formally speaking, the task of crawling a RIA page consists of finding all its DOM states, starting from the original application URL, referred to as the initial DOM state. The typical function of a RIA crawler is to automatically execute JavaScript events starting from the initial DOM state, i.e. the crawler starts from a given URL, downloads the corresponding page and reaches the initial state, extracts all the JavaScript events embedded in the page and follows these events to other DOM states within the same URL and so on, until all DOM states have been discovered.

In order to ensure that all DOM states have been identified, the crawler may explore all transitions as it is not possible to know a priori whether the execution of an event will lead to an already discovered state or not [99]. In RIA crawling, a Reset consists of returning to the original URL page and re-executing the transitions that lead to a target state. Unlike traditional web crawling where any state may be directly reached by loading its corresponding URL, crawlers may have to go through a path of ordered states to move from their current state to a target state in RIA crawling since it is not possible to directly reach a target state, i.e. loading a URL in RIA crawling consists of returning to the original URL page, which requires re-executing the transitions that lead to a target state. This makes the RIA crawling challenging when compared to the traditional web crawling. Efficiency of crawling a RIA is to discover as much of the RIA states as quickly as possible in an automated an efficient manner, by minimizing the number of events executed and reducing the number of Resets performed before executing each new transition, until all

states are discovered [99] [19, P. 20].

1.5 Distributed RIA Crawling

Distributed crawling consists of running two or more crawlers concurrently in order to discover all RIA states. A distributed centralized scheme [74] for crawling RIAs has been recently introduced with the aim of reducing the required amount of time to crawl RIAs, by allowing each crawler to explore only a subset of a RIA simultaneously . In this system, all states are maintained by a single entity, the controller. This entity is responsible for storing information about the new discovered states including the non-executed events on each state. In this system, all crawlers are associated with the central controller and each crawler may communicate the information of the newly discovered state with the single controller upon executing a new transition. The initial designed prototype consisted of statically allocating the task executions among the crawlers so that each crawler is responsible for an equal and fixed number of transitions. However, this requires load balancing mechanisms to be applied as the time needed to explore each new transition is different from one another and is not predictable in real systems, which results in making some crawlers idle while some other ones are overloaded. One solution to overcome this issue consisted of dynamically allocating the task executions among crawlers in a way the single controller decides which transition to be executed next by each crawler. Even-though this strategy is more efficient, it introduces more message overhead due to the communication overhead between the crawlers and the central controller. Another major drawback of this system resides in its centralization. In fact, maintaining the RIA states within a single unit may be problematic for the following reasons: (1) Scalability: Preliminary analysis of experimental results [74] have shown that a controller can support up to only 20 crawlers before becoming overloaded. This is an important scalability issue that needs to be addressed when dealing with distributed RIA crawling. (2) Fault tolerance: A failure occurring within the single

controller will result in the loss of the entire graph under exploration. In real distributed systems, databases are vulnerable to different kinds of failures. If the system has no failure detection and recovery capabilities, the system may not be able to achieve the crawling task properly. In order to overcome these issues, a peer-to-peer architecture [76] for crawling RIAs has been proposed in where nodes do not rely on a centralized unit to collect the information required for the crawling. In this system, each node is responsible for crawling a set of transitions. Upon executing a new transition, the node may broadcast it to all other nodes, allowing them to find a better shortest path to execute the closest non-executed transition that is available under their scope, starting from their current state. This architecture is appealing due to its decentralization. However, it may introduce a high message overhead since nodes are required to share every newly executed transition with all other crawlers.

1.6 Motivation and Research Question

We address the scalability and resilience problems when crawling RIAs concurrently. We propose a scalable fault-tolerant P2P crawling system that is capable of partitioning the RIA states over multiple controllers, where each controller maintains a distinct subset of the RIA model. Such a partition needs to be structured in a way that allows crawlers to retrieve the required information from each of these controllers before executing a new transition when needed. Moreover, each controller is associated with a set of crawlers that can access other controllers in the system through it. The decentralized crawling of RIAs is challenging for two reasons: (1) Unlike traditional web crawling where any state may be directly reached by loading its corresponding URL, crawlers may have to go through a path of ordered states in RIA crawling before exploring a new transition on a given state since it is not possible to directly reach a state with a new transition to be explored. If the states are partitioned among multiple controllers, it is unsuitable to communicate with

all controllers that are associated with the states in this path. (2) Traversing a long path before executing a new transition is costly. Some coordination between the controllers needs to be performed to allow crawlers to execute new transitions while the length of the path to reach each of these transitions is minimized.

For scalability, we propose a RIA crawling system that is able to operate properly with a very large number of concurrent crawlers. One important consideration when designing this system is to not overload its crawlers and controllers by : (1) Distributing the search space among multiple controllers, therefore reducing the loads among controllers. (2) Efficiently partitioning the crawling task among the concurrent crawlers in a way the execution load remains balanced among crawlers.

Moreover, fault tolerance is achieved when decentralizing the system by avoiding a single point of failure. If a failure occurs in one of the controllers, only a relatively small part of the entire data is lost. A fault-tolerant system must be able to achieve two functions: (1) Detecting the faulty components and eliminating or isolating them from the system. (2) Recovering from failures by eliminating its side-effects, regenerating the lost data by using replication or other techniques and resume normal operation. Additional considerations need to be addressed if we want to design an efficient fault-tolerant system with an effective tradeoff between the amount of replicated data and the load on the databases. Attention has also to be paid on who is responsible for recovering the lost data, how to locate it and how to reach it efficiently with minimum communication overhead.

In this thesis, we aim to advance the current research on distributed crawling of RIAs by exploring the possibilities of designing a distributed P2P system for crawling large-scale RIAs capable of partitioning the RIA graph over several controllers, which allows in fault tolerance. Moreover, a set of crawlers may be associated with each controller allowing for the scalability of the system. Furthermore, we aim at making this system resilient and capable of achieving the distributed RIA crawling even when node failures occur. To our knowledge, efficiently crawling large-scale RIAs over P2P networks in a faulty environment

has not been investigated yet.

1.7 Overview and Organization

Two important problems that need to be addressed when designing a distributed system for crawling RIAs are the scalability and fault tolerance issues. We introduce different sharing schemes for efficiently crawling large-scale RIAs with a high number of crawlers while the cost is minimized (number of event executions and Resets performed), which allows for scalability. We also aim at designing different distributed architectures for crawling RIAs that have resilience capabilities.

This thesis is organized as follows. Chapter 2 introduces the existing work related to distributed RIA crawling. In this chapter, we first describe and compare the traditional web crawling with RIA crawling. We then introduce the distributed processing. We emphasize two architectures: The centralized and the decentralized distributed systems. Finally, the resilience problem and the different detection and recovery mechanisms are reviewed. In Chapter 3, we introduce the Scalable Distributed P2P RIA Crawling system with Partial Knowledge. The decentralized distributed greedy strategy and the P2P RIA crawling protocol are also described, along with different knowledge sharing schemes for efficiently crawling RIAs in a decentralized system. Chapter 4 describes our experimental results and compares the efficiency of the proposed distributed P2P RIA crawling strategies with partial knowledge. Chapter 5 introduces our proposed Fault-Tolerant RIA Crawling system. Different Data Recovery strategies for recovering lost data when node failures occur are introduced. Chapter 6 analytically evaluates the fault-tolerant RIA Crawling System. A conclusion is provided in Chapter 7 with some future directions for improvement.

Chapter 2

Literature Review

In this chapter, we first introduce the graph exploration problem related to web crawling. We then introduce both traditional and rich internet applications (RIAs) and we highlight the differences between them. We also give an overview of distributed processing and distributed databases. We emphasize the critical points and findings related to decentralized distributed systems. Existing work related to fault tolerance is also presented.

2.1 Web Crawling

2.1.1 Introduction to Web Graphs

A web graph is a graph model consisting of a certain number vertices and edges connecting these vertices in a graph. Several studies such as [114] and [53] have addressed the problem of modeling large-scale real graphs such as the World Wide Web graphs (WWW). They have demonstrated that there is a reverse power-law relationship for the proportion of vertices with a given degree when searching a randomly growing graph. In 2002, Aiello et al. [112] and Cooper et al. [20] addressed the problem of searching a randomly growing web graph by a random walk. They considered a model of search in which a process called

spider makes a random walk on the nodes of an undirected graph. As the spider is walking, the graph is growing, and the spider makes a random transition to whatever neighbors are available at the time. Both investigations share a common characteristic which can be described by the so-called power-law. In a power-law degree distribution, the fraction of vertices with degree d is proportional to $1/d - \alpha$ for some constant $\alpha > 0$. They also demonstrated that a web graph meets the power-law graph properties when it is randomly crawled, i.e. it has a power-law degree distribution.

2.1.2 Traditional Web Crawling

The role of crawlers is to collect web content. Given a set of seed URLs, the basic function of a traditional web crawler consists of downloading these URLs and extracting all hyperlinks contained in these URL pages, and iteratively downloads the web pages that follow from these hyperlinks. Olston et al. provides a survey [23] that outlines the important research areas related to traditional crawling such as defining page relevance metrics, maintaining content freshness, politeness, eliminating undesirable content, distributed crawling and so on.

2.1.2.1 Crawl Ordering

The goal of the crawl ordering is to maximize the coverage of the discovered URLs achieved over time by following some importance metric, i.e. important pages are downloaded first depending on the purpose of crawling. Different crawl ordering policies have been categorized based on which data retrieved from each URL they consider important:

- Batch crawling: It consists of exploring a crawl space until reaching a certain size or time limit, without containing duplicate occurrences of any page. Three main types of crawl ordering policies have been examined in the literature:

  - Breadth-First [17]: Pages are downloaded in the order in which they are first discovered in a Breadth-First search manner, i.e. all extracted links from each page are executed next. Breadth-First crawling is found to be appealing [77] due to its simplicity.
  - Prioritization by In-Degree [42] : Pages with the highest number of incoming hyperlinks from previously downloaded pages are downloaded first.
  - Prioritization by PageRank [43] [97] [95] [68]: Pages are downloaded in descending order of PageRank, where each PageRank is associated with a score based on the pages and links acquired by the crawler. PageRank scores may be updated incrementally [97] or periodically [95].

- Incremental Crawling [95] [45]: It consists of continuously exploring the crawling space and revisiting the discovered pages periodically to help keeping the content up-to-date. Unlike Batch crawling where multiple pages do not appear multiple times, the incremental crawling may contain duplicate pages with the same URL which are used to detect any change in their content during the crawling.

- Scoped Crawling [42] [71] [17]: It attempts to crawl pages that fall within a particular category. The search category may be defined according to a specific topic, geography, language, format, genre and so forth.

2.1.2.2 Page Freshness

One important consideration when crawling traditional web applications is to maintain freshness of old crawled content, i.e. the degree to which the downloaded pages remain up-to-date, relative to the current downloaded web copies. Several freshness models have been proposed, as follows:

- Binary Freshness Model [34]: Pages are evaluated by their change frequency. This

frequency can take two values: Either fresh or stale.

- Continuous Freshness Model [97]: The freshness of a page is evaluated based on the elapsed time between the last discovery of a change in the cached page and the discovery of a new change in the page.

- Content-based Freshness Model [24]: A page is divided into a set of content fragments, each with a corresponding weight that evaluates the fragments relevance.

2.1.2.3 Politeness

Another consideration in crawling is to achieve a high performance without slowing down the crawling. Overloading a server with high rate of requests is seen as being impolite and several politeness policies may be applied to avoid such situations:

- Periodic approach [118] [41] : It consists of putting a fixed delay between successive requests to the same server.

- Adaptive approach [77]: It consists of putting a changing delay that is proportional to the time it took to download the last page.

- Exclusion approach [73]: It consists of specifying what pages crawlers are allowed to download by excluding irrelevant pages from the crawling.

2.1.2.4 Eliminating Undesirable Content

There are many situations where automatic traditional crawling leads to undesirable content which can be wasteful or redundant. For example, different URLs may redirect or refer to the same content [120]. Different web pages may also have the same content although the elements are not in the same order [119]. Another source of duplication is mirroring

[59] of a web site, where parts of the same web site are provided on different hosts. Detecting this kind of behavior is important in crawling and allows crawlers to avoid these URLs without downloading them, thereby reducing the load on the crawler and diversifying the set of search results. Many techniques have been proposed to detect these URLs [119] [1] [7] [59] by introducing learning algorithms that can generate rules containing regular expressions. These rules can be used by crawlers to detect and remove unwanted URLs.

2.1.2.5 Distributed Traditional Crawling

Increasing the crawling throughput may be achieved by using multiple crawlers in parallel and partitioning the URL space such that each crawler is associated with a different subset of URLs. The coordination between crawlers is required to prevent downloading the same page by the concurrent crawlers. The coordination may be achieved either through a central coordination process [96] [111] that is responsible of coordinating the crawling task, or through a peer-to-peer crawling system [15] [107] [52] [13] [63] where crawlers employ some distributed hash table (DHT) schemes [107] [82] in order to assign different subsets of URLs to different crawlers.

2.1.3 Deep Crawling

Deep Web (or Hidden Web) addresses the problem of crawling some content that is accessible only by filling in HTML forms and cannot be reached by downloading hyperlinks. Most researchers address the problem of crawling Deep content by dividing it into three steps: Locating Deep Web content [64], selecting only the relevant content [83] [66] and extracting it [50] [101].

2.1.4 RIA Crawling

2.1.4.1 RIA Crawling Strategies with One Single Crawler

Different strategies [99] have been previously introduced with the aim of efficiently crawling RIAs. The basic and most known standard approaches are the Breadth-First (BF) [21] [93] and the Depth-First (DF) [9] strategies. Although BF and DF are able to completely explore RIAs, they are not efficient in most cases. One reason is that BF and DF explore events in a strict order, i.e. the crawler knows a priori which transition has to be executed next. Consequently, there is no flexibility on choosing another state among other available states with non-explored events. Furthermore, if no path leading to a target state is known to the crawler, it may return to the initial state to reach it, which may increase the number of Resets performed. Other strategies have been introduced to overcome the high number of events executed or Resets performed in BF and DF such as the greedy strategy [121]. This strategy is to explore a non-executed event from the current state if there is an available non-explored event. Otherwise, the crawler may use Dijkstra's shortest path algorithm [113] to find the closest state with non-executed events, starting from its current state. The greedy strategy has been suggested [121] for RIAs crawling rather than BF and DF for its flexibility and simplicity.

Moreover, model-based crawling has been introduced with the aim of reducing the number of events executed and the number of Resets performed, by predicting the underlying RIA model. In this context, the Hypercube [19] and the Menu[98] models have been proposed. The Hypercube model is based on the assumption that the events enabled in a given state are independent, i.e. executing them in different orders leads to the same state. On the other hand, the Menu model assumes that an event execution is independent of a given source state, and that executing it will always lead to the same destination state. Both Hypercube and Menu strategies outperform the Breadth-First and Depth-First search strategies. However, their efficiency highly depends on the model of the crawled

application and how much it fits these models. Another strategy for crawling RIAs has been recently introduced and addresses the problem of state space explosion when crawling RIAs, referred to as the Component-based crawling [10]. The strategy is based on the greedy strategy and consists of partitioning the DOM into independent components where each state represents a separate state, thereby reducing the state space effectively. Although it is more complex to implement, experimental results showed that this strategy significantly outperformed the current RIA crawling methods in terms of overall crawling time and was able to cover complete content of RIAs.

2.1.4.2 Distributed RIA Crawling

A first study for distributed crawling of RIAs has been proposed in [93]. It extended some existing sequential algorithms for crawling traditional web applications by first extracting the URLs in the application, and then running independent crawling processes on the set of discovered URLs to perform event-based crawling using the Breath-First search strategy. The communication overhead is avoided by assigning each crawler to different URLs and allowing crawlers to execute JavaScript events on the subset of URLs they are responsible for, thereby eliminating the communication between crawlers.

Another research [9] proposed to distribute the RIA crawling task by using multiple threads for each discovered URL. These threats run simultaneously but share a common memory. In this system, a single thread is first initiated to perform a traditional web crawling. Then, for each URL that contains JavaScript events, multiple threads are initiated and perform a Depth-First search starting for this URL.

A distributed centralized crawling [74] has been recently introduced and consists of running multiple crawlers and sharing the search space of the RIA within one single database, referred to as controller. The crawlers explore only a subset of the application state and communicate with the controller to keep the current discovered graph up-to-date. In this

system, the crawling task is partitioned in a static way, i.e. all transitions on each state are associated to one of the crawlers, which has the responsibility of exploring them. Different load balancing schemes have been proposed to balance the workload among concurrent crawlers which have no equal processing powers, i.e. the execution of a sequence of transitions before reaching a non-executed event is different from a crawler to another. The basic motivation behind load balancing in this context is to forward the responsibility of executing transitions on a given state from overloaded crawlers to idle crawlers.

In order to eliminate the load balancing issue, another study [75] considers dynamically allocating transitions among the concurrent crawlers by the single controller. After every newly executed transition, the crawler communicates the information of its new current state with this controller, requesting the next transition to be executed. The controller applies the greedy strategy and locally finds the shortest path leading to a state with a non-executed event from the crawler's current state. This strategy provides an important improvement over the static approach [74] for distributed crawling of RIAs. However, it introduces more message overhead since crawlers have to communicate with the controller every time a new transition is executed.

Furthermore, a P2P architecture for crawling RIAs [76] has been introduced where crawlers do not rely on the single controller, they share every executed transition with all other crawlers in the network, allowing every crawler to locally find the shortest path leading to a state it is responsible for, starting from their current state. The main benefit of this architecture is to avoid a single point of failure by eliminating the use of the controller. However, a major drawback of this approach is the message overhead arising from broadcasting every transition among all crawlers in the P2P network.

2.2 Distributed Processing

A distributed system is a software system composed of more than one processing entity in a distributed network. Theses entities are independent from one another and communicate and coordinate their actions by passing messages in order to achieve a common task.

Timing Model

A distributed system can either be synchronous or asynchronous. A synchronous system assumes known bounds on message transmission delays between nodes as well as their execution rates. In other words, any message sent from one node to another is received and processed at the destination process within a bounded time. On the other hand, asynchronous systems have no timeliness assumptions, i.e. no assumptions about message transmission delays or execution rates. In this model, a message sent will eventually be received and processed, but with no guarantee on the reception time.

Distributed Architecture Paradigms

We distinguish two important paradigms in relation with distributed systems. The client-server paradigm and the peer-to-peer paradigm.

2.2.1 Client-Server Systems

A Client-Server system is a distributed system that is composed of two independent entities: An entity that is responsible of providing a service, called server, and entities that are responsible of using this service, called clients.

2.2.2 Peer-to-Peer Systems

In contrast to the traditional client-server architecture, a peer-to-peer system (P2P) [11] [94] is a networking system in which each peer acts as both the client and the server. Such

architecture insures that the system control is cooperatively maintained by all the peers with no hierarchical organization. The basic operations performed on a P2P system are the insertion, look-up and deletion of data items. P2P systems are found to be more scalable, robust and suitable for many applications.

P2P networks can be categorized into three classes [91]: Centralized, Decentralized and Unstructured and Decentralized and Structured.

2.2.2.1 Centralized

In the centralized P2P system, peers are responsible for coordinating the exchange of content of the entire network on a single storage device, the server. In this system, each peer has to update the server for each new item. Peers may then access an item by retrieving its corresponding access information from this server. Known as music exchange system, Napster [60] represents the first generation of P2P systems along with Publius [78]. Napster and Publius have an updated object directory that is maintained in the central Napster server. Upon logging in to the server, peers notify the directory with the list of files they maintain. The searching can be performed by issuing a query from a peer to the server to find which other peers hold their desired files. These two systems are simple and easy to deploy, but they have a single point of failure and they are not scalable, although they use several parallel servers to avoid a single point of failure.

2.2.2.2 Decentralized and Unstructured

These systems overcome a single point of failure by using a fully decentralized architecture with no centralized directory, offering a better resilience. However, there is no precise control over the network topology, which results in low query efficiency since the queries are executed hop-by-hop through the network until success/failure or timeout. An example of this type of of networks is Gnutella [37]. Gnutella uses the flooding approach for searching

data in the P2P network, where the query is communicated to all neighbors within a certain radius or constrained by some Time-To-Live (TTL) or hop-limit mechanisms. However, the flooding approach is not scalable since it generates large loads on the networked peers. Some directions [91] have been proposed to improve the latency of look-ups such as the dynamic TTL setting and the k-walker random walk.

2.2.2.3 Decentralized and Structured

Unlike the unstructured P2P networks where queries are unsure to be successful, the decentralized and structured P2P networks ensure that each query is guaranteed to success after some deterministic hops in a non-faulty environment. These structured topologies are usually constructed using distributed hashing table (DHT) [107] [82] techniques. The DHT service allows data to be placed not at random nodes but at specified peers that will make queries easier to satisfy. Moreover, the use of decentralized architectures using DHTs is appealing since they provide a low latency and are well known for their tolerance to node failures. In this context, different decentralized architectures using DHTs have been proposed over different structured topologies (Mesh [117], Ring [39], d-dimension Torus [102]). Although these P2P systems are found to be scalable and have no single point of failure, their performance may degrade as nodes join, leave or fail, due to their tightly controlled topologies. This requires some resilience mechanisms on top of each of these architectures (See section 2.3).

We introduce three decentralized architectures that use DHTs in the following section:

CAN [102]: CAN is a P2P systems that provides hash-table functionality for mapping data items to a point in dimensional space as its identifier around a d-dimensional Torus. Every region in this closed surface is associated with a peer that holds all keys whose IDs belong to this region. When a node is joining CAN, it randomly selects a point in the dimensional space. It then splits its region into two parts and takes one of them. When a

peer leaves CAN, the region it occupies and its associated data items are simply handed over to one of its neighbors. The routing and searching is performed by forwarding a query at each search/routing-hop to the region closest to the actual position of the wanted key. That is, the number of hops for each query is $O(\sqrt[d]{N})$ where d is the dimension of the Torus and N is the number of peers in CAN.

Plaxton [35]:

Plaxton is based on the mesh data structure. Plaxton allows peers to locate objects and route messages to them across an arbitrarily-sized overlay structure by maintaining pointers to peers whose IDs match the data items of a tree-like structure of ID prefixes up to a digit position. In order to provide a guaranteed peer from which a data item can be located, a root peer must be maintained for each data item. Queries are thus incrementally routed digit-by-digit from the right to the left until they reach the destination peer responsible for the given data item. Plaxton is scalable and has a simple fault-handling mechanism. However, its major limitation is that one requires global knowledge for assigning and identifying root peers, as well as their vulnerability to node failures. Two important variants of Plaxton have been introduced and allow for a better resilience: Pastry [108] and Tapestry [117].

Chord [33] [39] [115]:

Chord is a ring-based look-up service that stores key/value pairs in a distributed setting. Chord consists of storing the key/data item pair in the peer to which the key maps. Chord is based on consistent hashing [26]: It provides a unique mapping between an identifier space and a set of nodes by hashing every key to a unique node. Chord requires each node to keep a "finger table" storing up to m entries in a ring of N peers, where $m \leq log(N)$. The i^{th} entry of node n maintains the address of its $(2^{i-1})mod2^m$ subsequent nodes in the ring. This allows the look-ups to act like a binary search, where the searching space is reduced by half after each search/routing-hop. Therefore, the number of peers to be visited for

each query sent in a chord network of N nodes is $O(log(N))$. Chord is found to efficiently adapt as nodes join and leave the system even if the system is continuously changing. Moreover, Chord keeps the load among nodes balanced since the consistent hashing allows nodes to be associated with roughly the same number of keys when the nodes and keys are randomly chosen. The balanced load is maintained even when nodes join or leave the system since for every node joining or leaving, only a fraction of $O(1/n)$ of the keys are moved to a different location. Other important features that distinguish Chord from other P2P look-up protocols are its simplicity, scalability and performance [39]. However, maintaining these features at their highest levels requires a continuous maintenance to restore the topology as nodes join and leave the system. We outline different approaches for maintaining the Chord P2P system including joining and leaving nodes as well as the maintenance of Chord in the presence of failures required to reach a desired level of efficiency in Section 2.4.

2.3 Fault Tolerance

A fault is defined as an unwanted state transition of a process as a result of two distinct event classes [32]: Normal system operation and fault occurrences. A non-formal definition of fault tolerance [22] is the ability of a system to behave in a well-defined manner once faults occur. [2] provides a formal definition of a fault tolerance function as follows: *To preserve the delivery of expected services despite the presence of fault-caused errors within the system itself. Errors are detected and corrected, and permanent faults are located and removed while the system continues to deliver acceptable service.*

2.3.1 Types of Failure

2.3.1.1 Link Failure

A link failure [3] is a type of failure in which two linked nodes cannot communicate with one another. Different types of link failure may occur between two nodes: A message can be lost completely due to noise in the linking channel. A transmitted message may be corrupted and cannot reach its destination correctly. A link may also be broken temporarily causing all messages sent through it to be lost (packet loss). One way to overcome link failures is to attempt to retransmit lost messages through the same path several times until the message is successively received or through another path whose links are working properly. If the link failure is caused by a corrupted message, the destination node may detect the erroneous message using error detecting techniques [3]. In this context, Aguilera et al. [55] proposed several algorithms to detect link failures and introduces different solutions to handle them.

2.3.1.2 Software Failure

This type of faults are due to software errors, programming mistakes, non-handled exceptions or bugs. A software can be protected against failures by using software redundancy techniques [38]. N-version programming [2] is one of the software redundancy techniques known as fault masking [109]. It uses static redundancy in the form of independently written programs, each of which with a different version. These programs perform the same functions and their outputs are voted at special checkpoints.

2.3.1.3 Node Failure

- **Fail-Safe [18]:** A Fail-Safe fault is a type of fault in which processors malfunction but in a way that will cause no harm to the system, i.e. it will not respond in

any unpredictable way. Examples of Fail-Safe faults are omission faults (failures of receiving a request or failures of sending a response), timing failures and data out of bound. [89] categorizes Fail-Safe faults into four categories:

  - Permanent fault: This type of faults is always present in the system an never goes away.
  - Transient fault: This type of faults causes a component to malfunction for a limited period of time, and then goes away allowing the system to resume normal operation.
  - Intermittent fault: This type of faults appears from time to time but never goes away entirely; it can be active, causing a malfunction, or inactive.
  - Benign fault: This fault causes a unit to crash and go dead. Crash failures [22] are a good example of benign faults in which processors simply stop executing a task at a specific point in time without causing harm to the system.

- **Malicious Failure:** This is a type of faults in which processors may behave in unpredictable or malicious ways. This behavior may be due to malicious attacks or software errors. Malicious failures combine two classes of failures:

  - Symmetric: Failures are identical in all components of the system and they occur in a predictable way such as the omission faults.
  - Asymmetric: Failures are not identical and components may fail in arbitrary ways by processing requests incorrectly, corrupting their local state or by producing incorrect outputs. This class of failures is referred to as commission failures.

  Byzantine faults [67] are a good example of malicious failures which combine both omission and commission failures. When a Byzantine failure has occurred, the system may respond in any unpredictable way, unless it is designed to have Byzantine fault

tolerance. Several solutions to solve the byzantine failures have been introduced by Lamport et al. [67]. The trivial proposed solution is to solve the Byzantine Generals Problem by reaching a consensus among loyal generals, where generals of the Byzantine Empire's army must decide unanimously whether to attack some enemy army in the presence of traitors (faulty) among the loyal (non-faulty) generals. These traitors may act arbitrary in order to confuse the loyal generals, forcing a decision that is not consistent with the loyal general's desires. Byzantine fault tolerance can be achieved if the loyal generals succeed to make a consensus and reach an agreement on their strategy if a limited number of generals behave maliciously following the Byzantine fault model [67].

2.3.2 Fault Tolerance Strategies

Fail-Fast [48]:

The Fail-fast strategy belongs to the family of Fail-stop [48] fault tolerance where a program should either function correctly or it should detect a failure and stop operating. Fail-fast systems are designed to immediately detect and signal any failure and stop normal operation without attempting to recover from the failure. Fail-fast programs are made proactive by defensive programming, i.e. they allow for detecting failures early: They check all their inputs, intermediate results, outputs and stop normal operation in case any erroneous behavior occurs.

Fault Masking [109]: The fault masking consists of hiding the occurrence of faults and prohibiting even momentary erroneous results from being generated. The majority voting is a good example of this strategy. It consists of periodically performing a vote by three or more identical nodes on their outputs. The majority results are transmitted, and the minority results that are supposedly erroneous are discarded.

Reconfiguration [89]: This is a more popular strategy to tolerate faults in distributed

systems. It consists of eliminating faulty nodes from a system and restoring the system to some operational state. [89] divides the reconfiguration problem into three steps: Fault detection, Fault containment and Fault recovery. Fault detection is to detect and locate faulty nodes. Fault containment is the process of isolating and avoiding the propagation of errors throughout the system and eliminating any side effect that may occur after the detection of these faults. Finally, fault recovery consists of resetting the system to any of its operational states, allowing the system to resume normal operation.

2.3.3 Fault Tolerance Mechanisms

Several methods have been introduced for handling failures. [27] distinguishes two separate approaches: The proactive approach and the reactive approach.

Proactive Approach: In the proactive approach, the network configuration is maintained and pre-computed in a way to handle a given fault before the assignment of a job execution. When a failure occurs, the system makes minimum changes to handle it without having to make major change for its reconfiguration. The failure management in this case is usually straightforward and simple. However, one major drawback of this approach is the permanent usage of many resources to handle failures even though most of these resources are not always involved in the recovery.

Reactive Approach: The failure is handled after it has occurred and the system should only react to the failure as a response to it. Different studies have addressed this problem with the aim of minimizing the number of resources involved, allow the system for its reconfiguration only when a fault is detected. Such systems are usually more efficient than the proactive systems while their maintenance is more complex due to their dynamic adaption to failures.

Combined Approach: Both the proactive and the reactive approaches are considered. This is a more powerful approach since reactive and proactive techniques complement each

other.

2.3.4 Failure Detection Techniques

A major challenge when dealing with distributed systems is to identify faulty nodes so that they can be isolated or repaired accordingly. The correct nodes must collect information about the current execution or nodes that are involved in the distributed computation in order to detect failures. [16] categorized fault detection services into two main models: The Pull model and the Push model.

- Pull Model: In the pull model, nodes are responsible for sending periodic signals to a fault detector. This detector can identify a failing node if it does not receive a signal from this node after fixed period of time. Different fault-tolerant services use this model such as Condor-G [46] and BFT proactive recovery [70].

- Push Model: In this model, the fault detector is responsible for sending a signal to other nodes in the network. This detector can identify a failure if the node is not responsive. The membership detection mechanism is a good example of the push model and is implemented on several fault detection services such as Globus [69] and Net Solve [87]. In the membership agreement, each node periodically sends an *I am alive* message, referred to as a heartbeat [100]. Moreover, each node is listening for the heartbeat messages from other nodes in the distributed system. A node decides that another node has failed if it does not receive a heartbeat message from that server for a sufficiently long time out. Recently, Shukri A. et al [12] introduced a most popular technique for detecting failures using the heartbeat mechanism in very short intervals, allowing the system for quickly detect any failure.

  Unreliable fault detectors [88] are also a type of detectors in the push model and have the ability to suspect malicious nodes by making a decision on a suspected

node. The fault detector is called unreliable because it may suspect a correct node or it may fail to suspect a faulty node. Unreliable fault detectors are usually used to detect failures in asynchronous distributed systems that are subject to crash faults or Byzantine faults. Paul Stelling et al. [90] proposed a wide variety of techniques for detecting and correcting faults using unreliable fault detectors to detect node failures.

In the same context, [25] introduced a perfect failure detector that prevents wrong suspicions, i.e. non-faulty nodes are never suspected and faulty nodes are eventually suspected by everybody thereby preventing false alarms to occur. [25] showed that perfect failure detectors correctly detect node failures in asynchronous systems under the assumption that the communication is reliable (reliable communication delivery). Perfect failure detectors allow non-faulty nodes to suspect a faulty neighbor if it does not respond to an *areyoualive* message within twice the maximum round-trip time for any previous *areyoualive* message, assuming that messages are never lost and that the upper bound on message delay is known.

2.3.5 Task Recovery and Data Recovery Strategies

The task recovery is to maintain the state of a distributed task available to some other nodes, allowing the system to start from a given state when a failure occurs. The data recovery is to protect the database against data loss and reconstruct the database after data loss. We mention the following task and data recovery strategies:

Retry [100]: This is the simplest failure recovery technique and consists of replaying any erroneous task execution, hoping that the same failure will not occur in subsequent retries. The basic idea behind the Retry strategy is that if a failure occurs, then a task must restart from the beginning until it is successfully achieved.

Replication [100]: This consists of replicating the same task or data among several

nodes using backup copies, hoping that at least one of the replicated task would be successfully executed despite its failure on other nodes, or at least one of the replicated data remains available when data loss occurs on other nodes, respectively. One drawback of this technique is that the number of backup copies may dramatically increase and the backups management becomes very costly. A Fusion based technique [14] has been proposed to minimize the update overhead of replication by maintaining only a set of fused backup data structures which can be used to recover from a failure.

Message Logging [70]: All non-faulty nodes submit logging information about their current tasks and data to a reliable storage. Upon the detection of a failure, a diagnostic on these messages is performed in order to reset the system to a previous consistent global state.

Check-Pointing [81]: This consists of periodically storing the state of the application on a reliable storage. When a fault occurs, the application is resumed from the last checkpoint rather than restarting from the beginning. There are three types of Check-Pointing mechanisms [85]: Coordinated Check-Pointing, Uncoordinated Check-Pointing, and Communication-induced Check-Pointing.

- Coordinated Check-Pointing: The check-points are synchronized to ensure that the saved states are consistent with one another.
- Uncoordinated Check-Pointing: The scheduling of checkpoints is independently performed by different components at different time slots with no coordination of the check-point messages.
- Communication-induced Check-Pointing: Only few of the check-point messages are coordinated.

Redundancy [36], [79]: This is one of the popular methods to tolerate faults in a distributed system. Redundancy overcomes the drawback of Check-Pointing by making

multiple copies of each task or data on different nodes rather than a single one and using this redundancy feature when needed. [89] claims that redundancy is the key for fault tolerance: *There can be no FT without redundancy.* [110] distinguishes different types of redundancy: Time redundancy, space redundancy, and the combination of both (hybrid redundancy).

- Time Redundancy: The system exploits time redundancy by re-executing the same task on the same node periodically. Time redundancy is usually used to handle a certain type of failures that are not continuous and occurring at irregular intervals.
- Space Redundancy: Space redundancy consists of maintaining the same task or the same data on one or more different nodes, assuming that nodes fail independently. This kind of systems are used for a type of failures that are repetitive on the same node and thus needs to be permanently handled by other nodes in the system. The Primary-Backup approach [80] is applied in space redundancy.
- Hybrid Redundancy: In the case where some failure models require both time and space redundancy to be applied [55].

2.4 Maintenance of Chord

The maintenance of Chord addresses the problem of maintaining its distributed state as nodes fail, join or leave the system by properly updating the neighbor variables to maintain the topology. Since Chord is a continuously evolving system as nodes join and leave the system, it is required to continuously repair the overlay to ensure that the network remains connected and supports efficient look-ups.

Ideally, Chord can resolve all look-up queries with a complexity of $O(\log n)$ messages when the system is in the steady state, where n is the number of nodes in the system. By

steady state, we mean that the network have been reestablished correctly after a join, a leave or a failure. In real P2P networks, this performance is hard to maintain and may degrade in practice as nodes join, fail or leave the system arbitrarily. [84] [8] showed that all look-up queries can be performed with a high probability with $O(\log^2 n)$ when the system is continuously changing, by finding an alternative path through other nodes using the fingers, which guarantee that a node responsible for a key can always be found. Note that the fingers improve performance, but do not affect correctness [115], i.e. only the correct connectivity of the successor and predecessor nodes of a joining, failing or leaving node is required for correctness [115] under the assumption that the system is vulnerable to only fail-stop failures with perfect failure detection and reliable message delivery. Perfect failure detectors belong to the push model family introduced in Section 2.3.4 for periodically detecting failures, which ensure that faulty nodes are eventually detected by non-faulty nodes with no false alarms.

There are mainly two different approaches for maintaining Chord: The active approach and the passive approach. In the active approach, a node join consists of inserting a node in the network and updating the finger tables of other nodes in the network immediately after the join of the new node to reflect its addition. This is different from the passive approach which consists of only updating the successor and the predecessor of a joining node but leaving the finger tables of other nodes inaccurate. Since the fingers do not affect the correctness of Chord, these inaccuracies may be passively handled in the future and independently of the join operation, i.e. the finger updates of other nodes are handled periodically by all nodes in Chord using the idealization protocol [28]. To perform idealization, each node stores an extra predecessor pointer, used to record the closest predecessor of each node. This pointer is used to look-up the predecessor of a given node as required by the join, leave or fail operation. Moreover, the node leaving and node failing operations are handled similarly in the passive approach. Authors in [28] assume that a node may leave the network without notifying its neighbors. The idealization protocol allows for detecting

failures and to reestablish Chord along with updating periodically the finger table, the predecessors and the successors of all nodes in Chord to keep them up-to-date. However, in the active approach, the node leaving and the node failing operations are treated separately . [39] and [115] argue that these two operations should be handled separately since the leaving may occur more frequently than faults. Additionally, it appears simpler and more convenient for a node to initiate a leave protocol rather than waiting for other nodes to detect the disappearance of a node. In the following section, we briefly describe the node joining, leaving and failing operations for both the active and the passive approaches.

2.4.1 Active Approach

2.4.1.1 Joining Node

A node join using the active approach consists of inserting a node n_x with a unique $ID(n_x)$ between two successive nodes n_a and n_b in the ring such that $ID(n_a) < ID(n_x) < ID(n_b)$, so that the consistent hashing criteria is satisfied [26]. Moreover, since the insertion of the new node affects the finger table of other nodes, the finger table of some nodes have to be updated accordingly to reflect the addition of n_x.

We briefly describe the node joining protocol described in [115]. In order to start the join operation of a new node n_x, an arbitrary node n already existing in the ring must know the identity of the node n_x. We assume that the initiating node n knows the identity of n_x by an external mechanism. The active join operation of the node n_x consists of the following steps:

- Ask n to find the closet successor of n_x using $ID(n_x)$ denoted by n_b.
- Ask n to find the closet predecessor of n_x using $ID(n_x)$ denoted by n_a.
- Insert n_x between n_a and n_b and updating its successor and predecessor accordingly.

- Update the successor of n_a and the predecessor of n_b to reflect the addition of n_x.
- Transfer all values associated with the keys that the node n_x is now responsible for from n_a.
- Transfer the finger table of n_a to n_x and update the finger table of the predecessor n_a to reflect the addition of n_x.
- Finding and updating all nodes in the ring whose finger tables should refer to n_x.

2.4.1.2 Leaving Node

The leave operation of the node n_x may be actively performed by the node n_x and is described in [115] as follows:

- n_x finds its closet successor n_b using $ID(n_x)$.
- n_x finds its closet predecessor n_a using $ID(n_x)$.
- Update the successor of n_a and the predecessor of n_b to reflect the departure of n_x.
- Transfer all values associated with the keys that the node n_x was responsible for to n_a.
- Transfer the finger table of n_x to n_a.
- Remove n_x from Chord.
- Finding and updating all nodes in Chord whose finger tables were referring to n_x to reflect the removal of n_x.

2.4.1.3 Failing Node

Unlike the leaving node where a node can voluntary choose to leave Chord, a failing node may disappear from the network without notifying its neighbors. In the active approach, a node may choose to detect failures only when it actually needs to contact a neighbor. A node n may perform actively the repair operation upon detecting the disappearance of another node n_x in the network, i.e. the node n trying to reach n_x becomes aware that n_x is not responsive. Node n may then run a failure recovery protocol to recover from the failure of n_x using $ID(n_x)$ and to reestablish the ring. [115] has shown that the look-ups would be able to proceed by another path despite the failure with high probability.

Node n performs the repair protocol to recover from the failure of n_x as follows:

- n finds the closet successor of n_x denoted n_a using $ID(n_x)$.
- n finds the closet predecessor of n_x denoted n_b using $ID(n_x)$.
- Updating the successor of n_a and the predecessor of n_b.
- Finding and updating all nodes in the ring whose finger tables were referring to n_x to reflect the failure of n_x.

One drawback of the active approach is that only the finger table of some neighboring nodes are updated when a node joins, fails or leaves. When a node n joins or leaves Chord, not only nodes that were previously pointing to n must be updated but all other nodes preceding n that were pointing to any node succeeding n become inaccurate and therefore must be updated. The passive approach solves theses inaccuracies by running periodically a repair protocol by all nodes to maintain their finger tables up-to-date, which is more realistic since Chord is continuously changing in the real world. However, the passive approach may result in a considerable background traffic compared to the active approach due to the periodic maintenance of the ring.

2.4.2 Passive Approach

In the real world, nodes may join, fail or leave Chord arbitrary without notifying their neighbors. Authors in [28] suggest the use of a periodic and continuous maintenance of the ring in the background, i.e. nodes using the passive approach are not immediately updated when a join or a leave occurs, but a repair protocol runs periodically to restore the topology. This protocol, referred to as the idealization protocol [28] runs periodically and independently of the join and leave operations by every single node in the network which attempts to reconstruct its finger table, while only minimal operations for maintaining the connectivity of Chord are performed as nodes join, fail or leave the system. The goal of idealization is to support efficient look-ups by achieving the ideal state where a look-up query is resolved with $O(\log n)$.

2.4.2.1 Joining Node

The node joining operation using the passive approach provides the minimum requirement for basic connectivity of the ring topology while the update of the finger table of all nodes is performed periodically using the idealization protocol [115] independently of the join operation. When a new node joins Chord, only the update of its direct neighboring nodes is required to maintain the topology, i.e. it suffices to maintain the correctness of the predecessor and the successor of a joining node [115]. Since the fingers only improve the performance, the idealization protocol allows for updating the fingers of each node periodically to keep them up-to-date. This allows older nodes to learn about newly joined nodes. In other words, the joining operation using the passive approach is the same as the joining operation using the active approach with the only difference that the finger table of each node is not updated actively, i.e. it is updated periodically using the idealization protocol that runs in parallel.

2.4.2.2 Leaving and Failing Node

In the passive approach, a node failing is handled similarly to the node leaving. In this case, a node may simply leave Chord without notifying its neighbors. A neighboring node may detect the disappearance of a given node by running continuously a repair protocol which verifies whether a neighbor is responsive or not. This is different from the active approach when a node may choose to detect failures only when it actually needs to contact a neighbor. [28] suggests to use the passive approach for detecting failures to avoid the risk that all node neighbors fail before the node notices any of the failures. The repair protocol that runs periodically by every node n is described as follows:

- Verify whether the predecessor of n is correct and update it otherwise.
- Verify whether the successor of n is correct and update it otherwise.
- Verify and update all pointers in the finger table maintained by n.

Handling Simultaneous Node Failures:

[115] introduces a fault-tolerant Chord with successive node failures. This may be achieved by letting each node maintain a list of r succeeding nodes following it in the ring rather than having a single pointer to its direct successor, where $r < n/2$ in a Chord of n nodes. The system tolerates less that $n/2$ node failures is due to the fact every node can reach at most half of the nodes in Chord using the highest pointer in its finger table. If half of the nodes fail simultaneously, the non-faulty nodes will not be able to reach any node on the other half of the ring, and Chord becomes disconnected. To allow multiple failures to occur simultaneously, every node periodically verifies if all its r succeeding nodes are alive. If a sequence of r succeeding nodes of a given node n fail simultaneously, n may fetch the successor list of the succeeding node following the last failing node in the sequence. All nodes preceding the first failing node in this sequence may update their successor list

accordingly. The protocol for handling multiple failures is performed periodically by every node as follows:

- Verify whether the predecessor of n is correct and update it otherwise.
- Verify whether the r successors of n are correct and update them otherwise.
- Verify and update all pointers in the finger table maintained by n.

2.4.2.3 Idealization

The idealization protocol consists of periodically updating the predecessor, the successors and the finger table of every node so that the look-ups remain efficient. It was shown that any idealization can be performed with $O(\log^2 n)$ messages with high probability and the system becomes ideal (all pointers are accurate) after $O(\log^2 n)$ rounds of idealization [39]. However, [28] distinguishes between weak idealization and strong idealization in Chords and showed that the system described in [39] is weakly ideal and may lead to routing inconsistencies. In a weakly ideal Chord, every node n maintains the following property: $n.Successor.Predecessor = n$. However, another node n_1 where $n < n1$ can be a predecessor of n such that $n < n1 < n.Successor$. This may lead to cycles in Chord where the same node can have more than one successor.

In other words, a weakly ideal Chord consists of a Chord topology in which successors might be incorrect, i.e. a node may have more than one successor. This does not guarantee the consistency of look-ups where nodes arbitrarily join, fail or leave the system, i.e. a search for the same query may lead to two different nodes, and therefore some of the data become unreachable from some other nodes. Solving this problem is to prevent cycles to occur within Chord when the system keeps changing. The system, referred to as strong ideal, handles the looping case to guarantee that every node has only a single successor. The following two properties are maintained in the strongly ideal idealization protocol:

- Chord Connectivity: For every node n in Chord, $n.Successor.Predecessor = n$.
- Preventing cycles: There are no nodes $n1$ and $n2$ such that $n1 < n2 < n1.Successor$

Authors in [28] consider that strong idealization guarantees the correctness of all look-ups in Chord as nodes concurrently join and leave the system, and that an arbitrary Chord network becomes strongly ideal with $O(n^2)$ rounds of strong idealization.

Chapter 3

Scalable Distributed P2P RIA Crawling with Partial Knowledge

In this chapter, we introduce a scalable distributed P2P RIA Crawling System [56] composed of multiple controllers, where each controller maintains a list of states and is associated with a set of crawlers. The contributions of this chapter are as follows:

- The distribution of responsibilities for the states among multiple controllers in the underlying P2P network, where each controller maintains a portion of the application model, thereby avoiding a single point of failure, which allows for partial resilience.
- Defining and comparing different knowledge sharing schemes for efficiently crawling RIAs in the P2P network.

The rest of this chapter is organized as follows: Section 3.1 gives an overview of the Distributed P2P Architecture for Crawling RIAs with partial knowledge [56]. The assumptions are described in Section 3.2. The decentralized distributed greedy strategy is introduced in Section 3.3. Section 3.4 describes the P2P crawling protocol. Section 3.5

introduces different knowledge sharing schemes for efficiently crawling RIAs. The message complexities of our exploration mechanisms are described in Section 3.6. Finally, a conclusion is provided in the end of this chapter.

3.1 Overview of the Distributed P2P RIA Crawling System

In this system, the P2P network is composed of a set of controllers, and each state is associated with a single controller. Moreover, a set of crawlers is associated with each controller, where crawlers are not part of the P2P network. Notice that both crawlers and controllers do not share a common memory storage, i.e. they are independent processes running on different computers. There are two types of working components in the P2P RIA crawling system, as shown in Figure 3.1:

Controller : The controller is responsible for storing states and coordinating the crawling task among the concurrent crawlers. We assume that controllers do not know the number of controllers in the network. Each controller maintains a unique identifier which is used to distinguish it among the controllers in the peer-to-peer network. In this system, states are partitioned into disjoint sets, each of which is handled by a distinct controller. Each state has a unique identifier that is used to identify the position of the controller that is responsible for it in the peer-to-peer system. Furthermore, a set of crawlers is associated with each controller.

Crawler : Crawlers are only responsible for executing JavaScript events in a RIA and are not part of the P2P network. Each crawler is associated with one of the controllers in the P2P network and gets access to all controllers in the P2P system through the controller it is associated with. After executing an event, the crawler may find locally the identifier of the controller of its current state by mapping the hash of its current state information

and contacts its corresponding controller using the underlying P2P network.

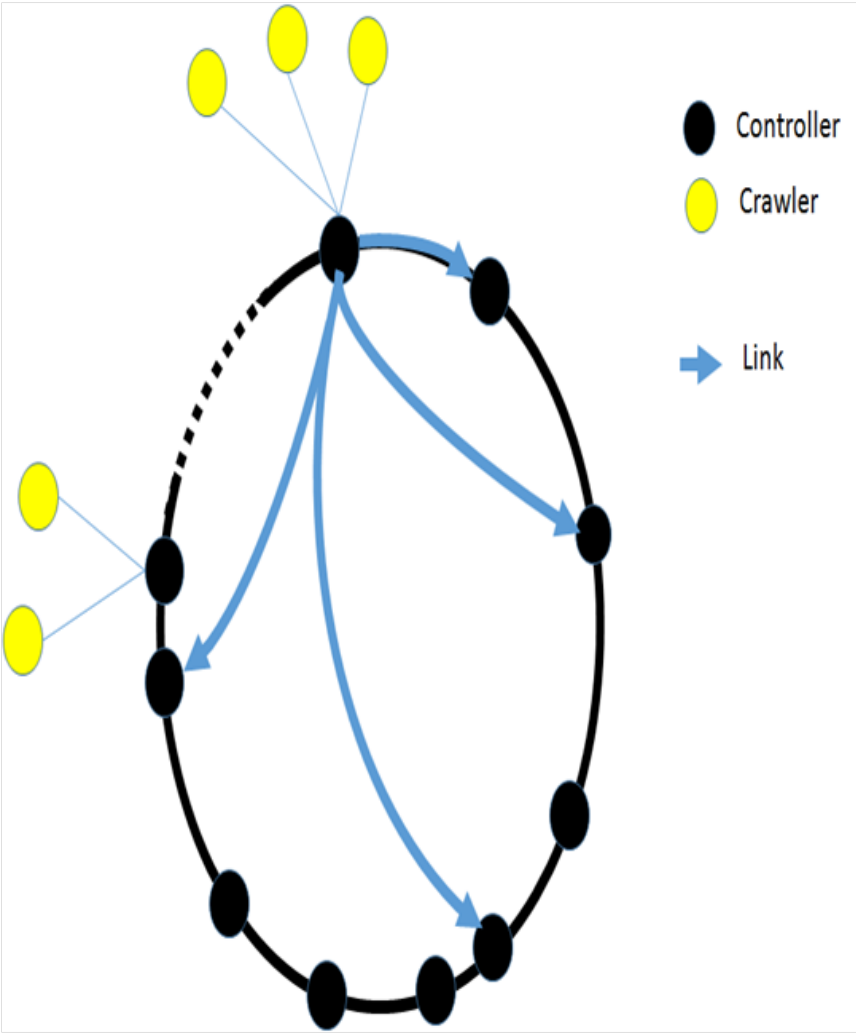

Figure 3.1: Distribution of states and crawlers among controllers: Each state is associated with one controller, and each crawler gets access to all controllers through a single controller it is associated with.

3.2 Assumptions

Joining and Leaving Controllers :

In the P2P crawling system described in Figure 3.1, controllers may join the P2P network arbitrary when the P2P crawling system starts. The operation of a controller n_x joining the P2P network consists of inserting n_x with a unique $ID(n_x)$ between two

successive controllers n_a and n_b, where n_a is responsible for the states in the interval $[ID(n_a), ID(n_b)]$, such that $ID(n_a) < ID(n_x) < ID(n_b)$, and transferring all states in the interval $[ID(n_x), ID(n_b)]$ from n_a to n_x. We assume that the joining controller n_x knows the identity of at least one existing controller in the P2P network through some external mechanism so that the join operation can be performed. On the other hand, when a controller n_y between two controllers n_c and n_d leaves the P2P network, the following actions are performed: Removing n_y from the network, reconnecting n_c and n_d, and transferring all states associated with the keys that controller n_y was responsible for from n_y to n_c.

Joining and Leaving Crawlers :

Crawlers may join and leave the P2P crawling system arbitrary during the crawl. We assume that a joining crawler knows the address of the controller it associates with through some external mechanism. Moreover, since crawlers are only responsible for executing an assigned job, i.e. they do not store any relevant information about the state of the RIA, a leaving crawler may simply leave the system arbitrary by communicating with the single controller it is associated with, assuming that some other crawlers will remain crawling the RIA.

Notice that for the RIA crawling to progress, there must be at least one controller and one crawler that are able to achieve the RIA crawling in a finite amount of time.

RIA Model :

The RIA model is composed of states and transitions where each state and each transition has a unique identifier in the RIA. The unique identifier of a state may be derived by hashing the content of the DOM page. On the other hand, a transition may be uniquely identified by hashing the DOM page the transition belongs to, the XPath which specifies the position of the transition in the DOM page, along with the information provided by the Javascript event to be executed in the corresponding transition. Moreover, we assume that all RIA states are reachable from the seed URL and all transitions are deterministic and

are executed in a finite amount of time. By deterministic, we mean that a event executed from a given source state will always lead to the same target state if it is executed more than once. Finally, we assume that loops are allowed in the RIA model where an event that is executed from a given source state may lead to the same state with the same state identifier, i.e. $ID_{SourceState} = ID_{DestinationState}$. However, since it is not possible to know a priori if an executed event will lead to the same destination state, each transition must be executed by the crawler to ensure that all states will be discovered.

3.3 The Greedy Strategy

The greedy strategy is to explore an event from the current state if there is any non-explored event. Otherwise, the crawler may execute an non-explored event from the closest state to its current state, until all transitions are traversed.

In the centralized RIA crawling system introduced in [75], all states are maintained by a single entity, called a controller, and is responsible for storing information about the new discovered states including the available events on each state. After the execution of a new transition, the crawler retrieves the required graph information by communicating with the single controller, and executes a single available event from its current state if such an event exists, or moves to another state with some available events based on the information available in the single database. When all transitions have been explored, crawlers may move to the termination stage to make sure there is no remaining job. If so, the crawl is achieved and the global termination is reached.

In order to eliminate the use of the single controller, a P2P RIA crawling system [76] has been proposed where crawlers share information about the RIA crawling among other crawlers directly, without relying on the single controller. In this system, each crawler is responsible for exploring transitions on a subset of states from the entire RIA graph model by associating each state to a different crawler. In order to find the shortest path from their

current state to the next transition to explore, crawlers are required to broadcast every newly executed transition to all other crawlers. Although this approach is appealing due to its simplicity, it may introduce a high message overhead due to the sharing of transitions in case the number of crawlers is high.

In the P2P RIA crawling system we propose, each state is associated with a single controller, allowing each controller to maintain a partial knowledge of the RIA graph model. In this system, the controller responsible for storing the information about a newly reached state is contacted when a crawler executes a new transition. For each request, the controller returns in response a single event to be executed on this state. However, if there is no event to be executed on the current state of a visiting crawler, the controller associated with this state may look for another state with a non-executed event among the states it is responsible for. Notice that maintaining a possible path from a source state to a target state within the controller is necessary in RIA crawling as controllers must be able to tell a visiting crawler how to reach a particular state starting from the crawler's current state.

Furthermore, a visited controller may forward the request for executing a job to its succeeding controller in the ring if there are no events to be executed on the states the visited controller is responsible for. This operation is repeated until a subsequent controller finds an event to be executed on one of the states it maintains, or until the request is received back by the visited controller, allowing for initiating the termination phase, as described in Section 3.4.5.

3.4 Protocol Description

The P2P RIA crawling is performed as follows, as shown in Figure 3.2: Initially, each crawler receives a *Start* message from the controller it is associated with, which contains the seed URL. Upon receiving the message, the crawler loads the URL and reaches the

initial state. The crawler then sends a $StateInfo$ message using the ID of its current state as a key, requesting the receiving controller to find a new event to be executed from this state. The controller returns in response an $ExecuteEvent$ message with an event to be executed or without any event. If the $ExecuteEvent$ message contains a new event to be executed, the crawler executes it and sends an acknowledgment for the executed transition. It has reached a new state and sends a new $StateInfo$ message to the controller which is associated with the ID of the new current state as a key. In case a crawler receives an $ExecuteEvent$ message without an event to be executed, it sends a $RequestJob$ message to the controller it is associated with. This message is forwarded in the ring until a receiving controller finds a job or until the system enters a termination phase.

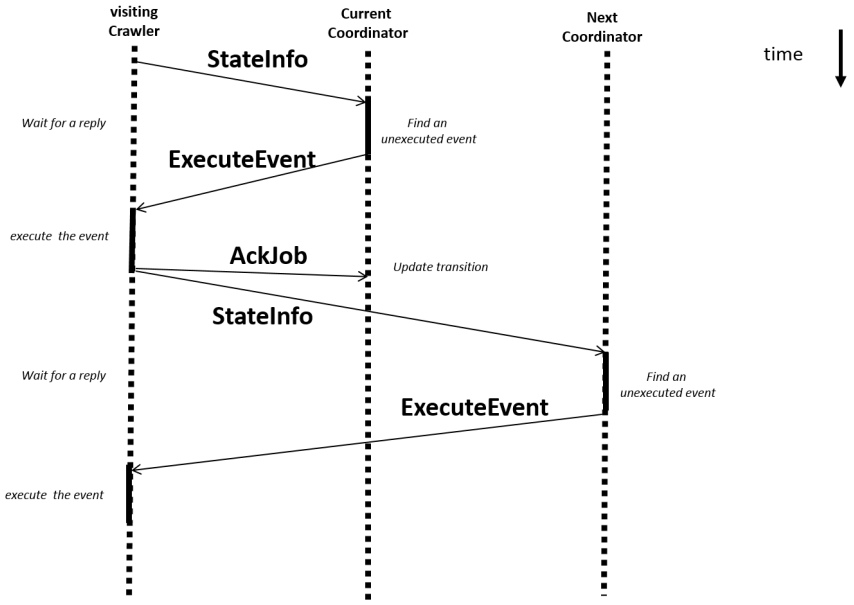

Figure 3.2: Exchanged messages during the exploration phase.

3.4.1 Data-Structures

- **State**: This represents a state of the application and has the following variables:
  - Integer $stateID$: The identifier of the state, which may be obtained by hashing

the information of the state.

  - Set $< Transition > myTransitions$: The set of transitions that can be executed from this state.

  - (initial URL, Sequence $< Transition >$) *path*: A pair of the initial URL and a sequence of transitions describing a path to this state from the initial state.

- **Transition**: This represents a transition of the application and has the following variables:

  - Enumeration *status* ($non - executed$, $assigned$, $executed$):

    1. $non - executed$: This is the initial status of the transition.
    2. $assigned$: A transition is assigned to a crawler.
    3. $executed$: The transition has been executed.

  - Integer $eventID$: The identifier of the JavaScript event on this transition.

  - Integer $destStateID$: The identifier of the destination State of this transition. It is null if its status is not $executed$.

Processes: We describe the processes involved during the crawl.

- **Crawler**: Crawlers are only responsible for executing JavaScript events in a RIA. Each crawler has the following variables:

  - Address $myAddress$: The address of the crawler.

  - Address $myController$: The address of the controller that is associated with this crawler.

- **Controller**: Controllers are responsible for storing states and coordinating the crawling task. Each controller has the following variables:

  - Address $myAddress$: The address of the controller.

- Set $< State >$ $myDiscoveredStates$: The discovered states that belong to this controller.
- String URL: The seed URL to be loaded when a Reset is performed.

3.4.2 Exchanged Messages

The following section describes the different type of messages that are exchanged between controllers and crawlers during the crawl. Each message type has the form *(destination, source, messageInformation)*

- **destination:** This identifies the destination process. It is either an address , or an identifier, as follows:
  - **AdressedByAddress**: This is when a message is sent directly to a known destination process.
  - **AdressedByKey**: It is a message forwarded to the appropriate process using the DHT look-up based on the given identifier in the P2P network.
- **source:** It maintains the address of the sending process.
- **messageInformation:** It consists of the message type and some parameters that represent the content of the message.

3.4.2.1 Message Types

We classify the message type with respect to the $messageInformation$ included in the message as follows:

- **Sent from a crawler to a controller**:

- **StateInfo(State currentState)**: This is to inform the controller about the current state of the crawler. The message is addressed by key using the ID of the crawler's current state, allowing the controller to find an event to be executed.
- **AckJob(Transition executedTransition)**: Upon receiving an acknowledgment, the controller updates the list of non-executed events by setting the status of the newly executed event to *executed*. The destination state of this transition is updated accordingly.
- **RequestJob(State currentState)**: *RequestJob* is a message sent by an idle crawler looking for a job after having received an ExecuteEvent message without an event to be executed. This message is forwarded around the ring until a receiving controller finds a non-executed event, or the same message is received back by the controller that is associated with this crawler, leading to entering the termination detection phase (see Section 3.4.5).

- **Sent from a controller to a crawler**:
  - **Start((URL)**: Initially, each crawler establishes a session with its associated controller. The controller sends a Start message in response to the crawler to start crawling the RIA.
  - **ExecuteEvent((initial URL, Sequence** $< Transition >$**)** *path*): This is an instruction to a crawler to execute a given event. The message includes the execution path, i.e. the ordered transitions to be executed by the crawler, where the last transition in the list contains the event to be executed. Furthermore, the message may contain a *URL*, which is used to tell the crawler that a Reset is required before processing the *executionPath*. The following four cases are considered:
    - Both the URL and the *path* are NULL: There is no event to be executed in

the scope of the controller.

* The URL is NULL but the *path* consists of one single transition: There is an event to be executed from the current state of the crawler.
* The URL is NULL but the *path* consists of a sequence of transitions: It is a path from the crawler's current state to a new event to be executed.
* The URL is not NULL and the *path* consists of a sequence of transitions: A Reset path from the initial state leading to an event to be executed.

We refer to a message execution from a controller to a crawler with an event to be executed as a Positive ExecuteEvent message, while a Negative ExecuteEvent message has no event to be executed

3.4.3 The P2P RIA Crawling Protocol

The following section defines the P2P RIA crawl protocol in more detail as executed by the controller and the crawler processes.

Controller process: Upon Receiving StateInfo $(stateID, crawlerAddress, currentState)$

Local variables:
$executionPath \leftarrow \emptyset$
$path \leftarrow < URL, \emptyset >$

1: **if** $stateID \notin myDiscoveredStates$ **then**
2: $\quad add\ currentState\ to\ myDiscoveredStates$
3: **end if**
4: **if** $\exists\ t \in currentState.transitions\ such\ that\ t.status = non-executed$ **then**
5: $\quad executionPath \leftarrow t$
6: $\quad t.status \leftarrow assigned$
7: $\quad URL \leftarrow \emptyset$
8: **else if** $\exists\ s \in myDiscoveredStates\ and\ t' \in s.transitions\ such\ that\ t'.status = non-executed$ **then**
9: $\quad executionPath \leftarrow s.path + t'$
10: $\quad t'.status \leftarrow assigned$
11: **end if**
12: $path \leftarrow < URL, executionPath >$
13: **send** $ExecuteEvent(crawlerAddress, myAddress, path)$

Controller process: UPON RECEIVING ACKJOB
$(controllerAddress, crawlerAddress, executedTransition)$

1: *Get t from myDiscoveredStates.transitions such that t.eventID = executedTransition.eventID*
2: $t.status \leftarrow executed$

Controller process: UPON RECEIVING REQUESTJOB
$(controllerAddress, crawlerAddress, currentState)$

Local variables:
$executionPath \leftarrow \emptyset$
$path \leftarrow < URL, \emptyset >$

1: **if** $\exists$ $s \in myDiscoveredStates$ *and* $t \in s.transitions$ *such that* $t.status = non - executed$ **then**
2: $executionPath \leftarrow s.path + t$
3: $t.status \leftarrow assigned$
4: $path \leftarrow < URL, executionPath >$
5: **send** $ExecuteEvent(crawlerAddress, myAddress, path)$
6: **else**
7: **forward** *RequestJob to nextController*
8: **end if**

Crawler process: UPON RECEIVING START
(URL)

Local variables:
$currentState \leftarrow \emptyset$

1: $currentState \leftarrow load(URL)$
2: $currentState.path \leftarrow \emptyset$
3: **for all** $e \in currentState.transitions$ **do**
4: $e.status \leftarrow non - executed$
5: **end for**
6: **send** $StateInfo(stateID, myAddress, currentState)$

Crawler process: UPON RECEIVING EXECUTEEVENT
$(crawlerAddress, controllerAddress, executionPath)$

1: **if** $executionPath \neq \emptyset$ **then**
2: **if** $URL \neq \emptyset$ **then**
3: $currentState \leftarrow load(URL)$
4: $currentState.path \leftarrow \emptyset$
5: **end if**
6: **while** executionPath.hasNext **do**
7: $currentState \leftarrow process(executionPath.next)$
8: **end while**
9: **send** $AckJob(controllerAddress, myAddress, executionPath.last)$
10: $currentState.path \leftarrow executionPath$
11: **for all** $e \in currentState.transitions$ **do**
12: $e.status \leftarrow non - executed$
13: **end for**
14: **send** $StateInfo(stateID, myAddress, currentState)$
15: **else**
16: **send** $RequestJob(nextController, myAddress, currentState)$
17: **end if**

3.4.4 Handling Traditional and RIA Crawling Simultaneously

The proposed P2P RIA crawling system can easily handle both RIA and traditional web crawling simultaneously since an initial state of a RIA is equivalent to a downloaded URL in traditional web crawling. Since RIA crawlers have the feature of executing a hyperlink by loading a given URL when performing a Reset is required, a crawler may simply move from a state in one URL to another state in a different URL by loading the new URL when a Reset path with a different URL is returned in response from the visited controller by means of an *ExecutedEvent* message, i.e. when a controller contacted by a visiting crawler returns in response a Reset path with a URL that is different from the current URL of the crawler, prompting the visiting crawler to execute a Javascript event or a hyperlink from a different URL page. Notice that the contacted controller must have previously discovered at least one RIA state on a different URL from the URL described in the $StateInfo$ message sent by the visiting crawler. However, for the crawling to be consistent in the case when multiple URLs are derived from the original URL in a RIA, a crawler may only move from a URL page to another if one of the following two criteria is satisfied: (1) The contacted controller responsible for finding a new transition to be executed from the crawler's current state cannot find a new transition to be executed on the current URL page of the crawler. However, the controller has previously discovered a state with non-executed events from a different URL. (2) The cost of the best computed execution path from the crawler's current state to another state in the same URL page is higher than the cost of performing a Reset in order to execute a new transition on a state that is in a different URL from the current URL of the visiting crawler.

3.4.5 Termination Detection

The distributed termination problem is to detect whether a computation within a distributed system has terminated. Taking this fundamental problem to the field of dis-

tributed RIA crawling consists of reaching a termination phase where all crawlers and controllers reach the same final state, i.e. all transitions have been executed, and that this state is not susceptible to change in the future.

Misra [51] introduced an algorithm for detecting termination of distributed computations using markers. In Misra's algorithm, a marker visits all the processes in the network and checks to see if they are passive or active. Since the messages are in transit, the marker cannot assert that the computation has terminated if it finds all processes to be passive after one round of visits. For the special case of a network in which processes are arranged in the form of a ring (every process has a unique predecessor from which it can receive messages and a unique successor to which it can send messages), the marker can assert that the computation has terminated if it finds after two rounds of visits that every process has remained continuously passive since the last visit of the marker to that process. The marker turns a process white when it leaves a passive process. A process changes to black if it becomes active. If the marker arrives at a white process, it can claim that the process has remained continuously passive since the marker's last visit. The marker detects termination if it visits N white processes, where N is the number of processes in the ring. Misra's termination algorithm [51] is applied in this study with the following additional considerations: (1) Markers are messages of type $CheckTerm$ and are used to check whether all controllers have no jobs to assign to a crawler. That is, a controller that receives a $CheckTerm$ messages will mark it white if and only if it has no jobs to assign to the visiting crawler. The message will then be forwarded to the next controller in the P2P system. (2) Since executing a single event is not immediate and may take an unpredictable amount of time, it is possible that a controller has assigned all its jobs but did not receive all acknowledgments back from the crawlers executing these jobs, signaling the entire execution of an event. Consequently, the termination may be reached without executing some events. Therefore, a controller that receives a $CheckTerm$ message from a crawler must reject it, i.e. it turns a process black, if it has some jobs to assign to the

visiting crawler or if not all assigned events are acknowledged.

A trivial solution for handling acknowledgments during the termination phase consists of maintaining a counter by each controller for the assigned jobs it is responsible for, called $assignedJobsCounter$. Initially, $assignedJobsCounter$ is set to zero. When a controller assigns a new job to a visiting crawler, $assignedJobsCounter$ is incremented. However, when an acknowledgment for a given job execution is received, $assignedJobsCounter$ is decremented. A controller who has no jobs to assign to idle crawlers accepts a $CheckTerm$ message and forwards it to its neighbor if and only if $assignedJobsCounter$ is 0. This way, every controller ensures that the termination is not reached before all controllers have received acknowledgments for their assigned jobs.

The termination detection may be initiated by one or more idle crawlers. For simplicity, we restrict the task of checking termination to a single crawler. This can be achieved by performing a leader election among crawlers. Two steps are considered in order to elect one of the crawlers in the P2P system to initiate the termination phase: (1) First, the controller with the highest ID among all controllers in the P2P system is elected. This can easily be applied in the ring as controllers are ordered in the clockwise direction in the underlying P2P system. (2) The crawler with the highest ID among all crawlers that are associated with this controller is elected to initiate the termination. Notice that idle crawlers other than the leader will keep asking for jobs from different controllers until they receive a given task, or until the termination is reached. The termination is reached when the initiating crawler receives its own *CheckTerm* message in two rounds, signaling that all controllers have accepted twice the termination *CheckTerm* message without interruption, i.e. The *CheckTerm* message is marked white: All controllers have no remaining jobs to assign to idle crawlers and all their assigned jobs have been acknowledged. The crawler then declares global termination by forcing all crawlers and controllers in the P2P system to terminate.

3.5 Choosing the Next Event to Explore from a Different State

If no event can be executed from the current state of a visiting crawler, the controller that maintains this state may look for another state with some non-executed events without necessary performing a Reset, depending on its available knowledge about the graph under exploration. Moving from a state to another usually consists of going through a path of ordered states before reaching a target state. Reducing the cost of such a path is challenging for distributed RIA crawling for two reasons: (1) State distribution: Each state is associated with a single controller in the network. It may be unsuitable to communicate with all controllers on the path to find the closet non-executed event. (2) Transition knowledge required: Moving from a state to another usually consists of following a path of ordered transitions before reaching the state, which requires a prior knowledge about the executed transitions. In a non-distributed environment, the crawler may have access to all the executed transitions, which allows for finding the closest state with non-executed events, starting from the current state. However, in the distributed environment, sharing the knowledge about executed transitions may introduce a high message overhead and may produce bottlenecks on some controllers if the number of crawlers is high. Typically, sharing more transitions results in raising the overall number of messages in the crawling system. Therefore, there is a trade-off between the shared knowledge which improves the choice of the next event to be executed and the message overhead in the system. We introduce in the following different approaches with the aim to reduce the overall time required to crawl RIAs by executing as few transitions as possible , while the message overhead and the number of Resets performed are minimized.

3.5.1 Global-Knowledge

The Global-Knowledge scheme consists of sharing all executed transitions among all controllers in the system. That is, for each executed transition by a visiting crawler, the controller responsible for the reached state, upon receiving its state information, may broadcast the newly executed transition to all controllers in the network. This means that the RIA information is replicated in all controllers. Although not realistic in our setting, the Global-Knowledge scheme allows all controllers to have instant access to a globally shared information about the state of knowledge at each controller. This may introduce a high message overhead and may produce bottlenecks on controllers due to the repetitive update of the application graph among all controllers. Note that this approach is considered for comparison only and would give the same number of event executions as the single controller in the centralized crawling system [75].

3.5.2 Reset-Only

With this scheme, a crawler can only move from a state to another by performing a Reset. In this case, the controller returns an execution path, starting from the initial state, allowing the visiting crawler to load the seed URL and to traverse a Reset path before reaching a target state with a non-executed event. In order to reduce the number of transitions to be traversed from the initial state to a target state, dynamic updates of Reset paths may be applied. This allows each controller to compare the size of the visiting crawler path and update it if necessary by only maintaining the shortest known path from the initial state to every target state the controller is responsible for. Note that the Reset-Only approach is a simple way for concurrently crawling RIAs. However, this approach results in a high number of Resets performed, which may increase the time required to crawl a given application (cost).

3.5.3 Local-Knowledge

With the Local-Knowledge scheme, a visited controller may use its local transitions knowledge to find a short path from the crawler's current state leading to a state with a non-executed event. This local knowledge consists of the states the controller is responsible for and the executed transitions on these states, along with all executed transitions provided within the path of each visiting crawler. Unlike the Reset-Only approach where only one path from a URL to the target state is stored, controllers store all executed transitions with their destination states and obtain then a partial knowledge of the application graph. This local knowledge is used to find a short path from the crawler's current state to a state with a non-executed event based on the available knowledge of the controller. Since the knowledge is partial, this may often lead to a Reset path even though according to global knowledge, there exists a short direct path to the same state.

Notice that the dynamic updates of Reset paths are also maintained by a given controller when visited, similarly to the Reset-Only scheme, and are used as an optional choice in case the cost of the computed short path is higher than the cost of performing a Reset to reach the same target state. That is, when a visiting crawler communicate its path to a newly reached state, the controller updates its knowledge by adding all transitions on this path. The controller then locally finds a short path from the crawler's current state to the closest state with a non-executed event and returns it to the visiting crawler in response if the cost of executing this path is shorter than the costs of executing a possible path to the target state after performing a Reset. If no such a path is found, the controller may force the visiting crawler to perform a Reset, similarly to the Reset-Only approach.

3.5.4 Shared-Knowledge

With the Shared-Knowledge scheme, the transitions contained in the $StateInfo$ messages are stored by the intermediate controllers when the message is forwarded through the

underlying P2P system. This way, the transitions knowledge of controllers is significantly increased without introducing any message overhead compared to the Local-knowledge scheme. Therefore, the controllers will be able to find better short paths.

3.5.5 Original Forward Exploration

Two important drawbacks of the short path approach are the partial knowledge of controllers: (1) Since each state is associated with a single controller in the network, short paths can be only computed toward states the visited controller is responsible for. If other neighboring states to a crawler's current state exist with a non-executed event and belong to a controller other than the visited controller, this controller cannot choose an event to be executed from one of these states. (2) Short paths found may not be optimal since they are based on the knowledge available to the controller. An alternative consists of globally finding the optimal choice based on the Breadth-First search by forwarding the exploration to the controllers that are associated with the neighboring states of the crawler's current state rather than locally finding a non-executed event from one of the states each controller is responsible for.

The Original Forward Exploration search is initiated by the visited controller and consists of distributively performing a Breadth-First search: It begins by inspecting all neighboring states from the current state of the crawler if there are no available events on its current state. For each of the neighbor states in turn, it inspects their neighbor states which were unvisited by communicating with their corresponding controllers, and so on. The controller maintains two sets of states for each Forward Exploration query: The first set, called $statesToVisit$, is used to tell a receiving controller which states are to be visited next. On the other hand, the second set, called $visitedStates$, is used to prevent loops, i.e. states that have been already discovered by the Forward Exploration. Additionally, each state to be visited has a history path of ordered transitions from the crawler's current

state to itself, called *intermediatePath*.

Initially, when a visited controller receives a $StateInfo$ message from a crawler, it will pick a non-executed event from the crawler's current state. If no non-executed event is found, the controller waits for acknowledgments for the assigned transitions that have not been acknowledged yet, by putting the current Forward Exploration query along with all subsequent Forward Exploration queries on that state to a list called *parkedQueries*. Once all transitions have been acknowledged, the controller picks all destination states of the executed transitions on this state and adds them to the set $statesToVisit$. The *intermediatePath* from the crawler's current state to each of these state is updated by adding the corresponding transition to this path. This controller then picks the first state in the list. It first adds it to the set *visitedStates* to avoid loops, and then sends a Forward Exploration message containing both $statesToVisit$ and *visitedStates* to the controller responsible for these states. When a controller receives the Forward Exploration message, it checks if there is a non-executed event from the current state. If not, it adds the destination states of the transitions on that state at the beginning of the list $statesToVisit$ after verifying that these destination states are not in the set *visitedStates* and that all transitions have been acknowledged on this state. It will then pick the last state in the list $statesToVisit$ and send again a Forward Exploration message which will be received by the controller that is responsible for that state.

We note that globally performing a distributed Breadth-First search is appealing since it allows for completely removing the termination detection phase introduced in Section 3.4.5, i.e. when a state with no non-executed events is reached when performing a global Breadth-First search starting from the initial state of the RIA and its neighbors have already been visited and have no non-executed events, the termination is directly reached. This can be achieved by adding the initial state to the list of $statesToVisit$ when the cost of executing the next transition with a global Breadth-First search starting from the crawler's current state is equal to the cost of performing a Reset and initiating a global Breadth-First search

starting from the initial state of the RIA. Three cases arise from this approach: (1) The cost of executing the next transition with a global Breadth-First search starting from the crawler's current state is less than the cost of performing a Reset and performing a global Breadth-First search from the initial state, i.e. the number of transitions to be traversed from the crawler's current state are less than the cost of performing a Reset and traversing a number of transitions before reaching a state with a non-executed transition: The controller allows the visiting crawler to execute this transition starting from the crawler's current state without performing a Reset. (2) The cost of executing the next transition by performing a Reset and initiating a global Breadth-First search from the initial state is less than the cost of executing the next transition with a global Breadth-First search starting from the crawler's current state: The controller allows the visiting crawler to execute the next transition by performing a Reset and a global Breadth-First search starting from the initial state of the RIA. (3) The controller cannot find neither a transition to be executed with a global Breadth-First search starting from the crawler's current state nor a transition to be executed by performing a Reset and a global Breadth-First search from the initial state of the RIA: The controller can claim that the global Breadth-First search is terminated without finding an event to be executed from the initial state since the search for the next transition to be executed includes the global Breadth-First search starting from the initial state, which proves termination of the crawling task. Therefore, the termination phase of Section 3.4.5 is not required.

The following algorithm describes the Original Forward Exploration protocol, as executed by the controller process (*Algorithm.UponReceivingForwardExploration*). Additionally, line 4 to line 13 of *Algorithm.UponReceivingStateInfo* are replaced by *Algorithm.UponReceivingForwardExploration*, allowing for initiating the Forward Exploration operation by a controller that is receiving a new StateInfo message. Finally, *Algorithm.UponReceivingAckJob* is updated, allowing for processing each of the parked ForwardExploration messages that are waiting for all assigned events on a state to

be acknowledged.

Controller process: UPON RECEIVING FORWARDEXPLORATION
$(controllerAddress, crawlerAddress, currentState,$
$sourceController, statesToVisit, visitedStates)$

```
Local variables:
executionPath ← ∅
path ←< URL, ∅ >
nextState ← ∅
parkedFlag ← false
 1: if ∃ t ∈ currentState.transitions such that t.status = non − executed then
 2:     executionPath ← currentState.intermediatePath + t
 3:     t.status ← assigned
 4:     URL ← ∅
 5:     path ←< URL, executionPath >
 6:     send ExecuteEvent(crawlerAddress, myAddress, path)
 7: else if ∄ t ∈ currentState.transitions such that t.status = assigned then
 8:     for all t ∈ currentState.transitions do
 9:         if t.destinationState ∉ visitedStates then
10:             nextState.intermediatePath ← currentState.intermediatePath + t
11:             statesToVisit ← t.destinationState + statesToVisit
12:         end if
13:     end for
14: else if ∃ t ∈ currentState.transitions such that t.status = assigned then
15:     push ForwardExploration(controllerAddress, crawlerAddress, currentState,
        sourceController, statesToVisit, visitedStates) to parkedQueries
16:     parkedFlag ← true
17: end if
18: if !parkedFlag then
19:     if statesToVisit ≠ ∅ then
20:         nextState ← statesToVisit.last
21:         remove statesToVisit.last
22:         push nextState to visitedStates
23:         send ForwardExploration(nextState.controllerAddress, crawlerAddress, nextState,
            sourceController, statesToVisit, visitedStates)
24:     else
25:         send ExecuteEvent(crawlerAddress, myAddress, ∅)
26:     end if
27: end if
```

3.5.6 Locally Optimized Forward Exploration

One drawback of the Original Forward Exploration approach is that a controller repetitiously sends queries that are started from a given state to the controllers associated with all neighboring states to this state, in order to reach the closest state with a non-executed event, even though these controllers did not find an event to be executed on their states previously. One way to overcome this issue is to make controllers remember the controller where the last query that was started from the same state has stopped, i.e. the state in which the last Forward Exploration query succeeded to find a non-executed transition. This

Controller process: UPON RECEIVING ACKJOB
$(controllerAddress, crawlerAddress, executedTransition)$

```
1: Get t from myDiscoveredStates.transitions such that
   t.eventID = executedTransition.eventID
2: t.status ← executed
3: if ∄ t ∈ currentState.transitions such that t.status = assigned then
4:     for all ForwardExploration(controllerAddress, crawlerAddress, currentState,
       statesToVisit, visitedStates, messageKnowledge) ∈ parkedQueries do
5:         receive ForwardExploration(controllerAddress, crawlerAddress, currentState,
           statesToVisit, visitedStates, messageKnowledge)
6:     end for
7: end if
```

way, this controller is directly contacted during a subsequent query from the same state. When a contacted controller finds a non-executed transition on a target state, it sends a $RememberMe$ message containing the sets of $nextStatesToVisit$ and $nextVisitedStates$ leading to that state to the original controller, i.e. the controller from where the Forward Exploration was initiated, allowing it to move directly to this state whenever a new Forward Exploration query is initiated from the same state. The parameters of the next received Forward Exploration query to the same state $(statesToVisit, visitedStates)$ will be updated with the new stored parameters $nextStatesToVisit, nextVisitedStates)$ respectively, thereby allowing controllers to start from where the last query from the same state has stopped. This results in reducing the message overhead by eliminating repetitions of different Forward Exploration queries from the same state to neighboring controllers with no non-executed events on their target states.

3.5.7 Globally Optimized Forward Exploration

In the Locally Optimized Forward Exploration scheme introduced in Section 3.5.6, the controller responsible for the state from where a Forward Exploration query is initiated remembers the target state where the previous query has found a state with a non-executed event, so that the controller can forward the next query that is starting from the same state directly to the controller responsible for the target state. This way, the exploration is resumed from where the previous query has stopped. One drawback of this scheme

is that controllers only remember the states where the last query has stopped, not the intermediate states explored in the path of the query. If two distinct queries are initiated from two states that are associated with two different controllers, it is possible that the queries explore some same intermediate states before reaching a state with a non-executed event.

In order to prevent different controllers from visiting intermediate states that have already been visited by other controllers during the Forward exploration and have no non-executed events, controllers may share during the Forward Exploration their knowledge about all executed transitions on the intermediate states that have been already explored by the Forward Exploration and has no event to be executed, with other controllers in the network. If the executed transitions on the next intermediate state to be explored by the Forward Exploration are available to the visited controller, i.e this controller is aware that other controllers in the network have already visited this intermediate state and there is no benefit from visiting this state again, the Forward Exploration jumps over this state and reaches directly the destination states of each of the transitions on it. This allows for preventing the intermediate states with no non-executed events that have been already explored by the Forward Exploration, from getting visited again. The knowledge sharing of executed transitions that have been already seen by the Forward Exploration is made by means of the *messageknowledge* parameter included in each of the Forward Exploration queries. Therefore, the sharing of additional knowledge about intermediate states that have been already explored by other controllers during the Forward Exploration but have no event to be executed is performed during the Forward Exploration with no message overhead. Notice that all executed transitions must be acknowledged on each visited intermediate state before they can be shared, i.e. for each reached intermediate state, a controller can only jump over a visited intermediate state if and only if all transitions have been executed on the intermediate state and their destination states are known to the controller associated with this state. This approach is called the Globally Optimized

Forward Exploration scheme.

The following figure describes the Globally Optimized Forward Exploration protocol, as executed by the controller process upon receiving a *ForwardExploration* message.

Controller process: UPON RECEIVING FORWARDEXPLORATION
($controllerAddress, crawlerAddress, currentState,$
$statesToVisit, visitedStates, messageKnowledge$)

Local variables:
$executionPath \leftarrow \emptyset$
$path \leftarrow < URL, \emptyset >$
$nextState \leftarrow \emptyset$
$noJumping \leftarrow false$

1: $transitionsKnowledge \leftarrow messageKnowledge + transitionsKnowledge$
2: **if** $\exists\ \ t \in currentState.transitions\ \ such\ \ that\ \ t.status = non-executed$ **then**
3: $executionPath \leftarrow currentState.intermediatePath + t$
4: $t.status \leftarrow assigned$
5: $URL \leftarrow \emptyset$
6: $path \leftarrow < URL, executionPath >$
7: **send** $ExecuteEvent(crawlerAddress, myAddress, path)$
8: **else if** $\nexists\ \ t \in currentState.transitions\ \ such\ \ that\ \ t.status = assigned$ **then**
9: **for all** $t\ \ \in currentState.transitions$ **do**
10: $transitionsKnowledge \leftarrow t + transitionsKnowledge$
11: **end for**
12: **for all** $t \in currentState.transitions\ \ such\ \ that\ \ t.status = executed$ **do**
13: **if** $t.destinationState \notin visitedStates$ **then**
14: $t.destinationState.intermediatePath \leftarrow currentState.intermediatePath + t$
15: $statesToVisit \leftarrow t.destinationState + statesToVisit$
16: **end if**
17: **end for**
18: **while** $statesToVisit \neq \emptyset$ **or** $!noJumping$ **do**
19: $nextState \leftarrow statesToVisit.last$
20: **remove** $statesToVisit.last$
21: **push** $nextState\ \ to\ \ visitedStates$
22: **if** $nextState.transitionsKnowledge \neq \emptyset$ **then**
23: **for all** $t\ \ \in nextState.transitionsKnowledge$ **do**
24: **if** $t.destinationState \notin visitedStates$ **then**
25: $t.destinationState.intermediatePath \leftarrow nextState.intermediatePath + t$
26: $statesToVisit \leftarrow t.destinationState + statesToVisit$
27: **end if**
28: **end for**
29: **else**
30: $noJumping \leftarrow true$
31: **send** $ForwardExploration(nextState.controllerAddress, crawlerAddress, nextState,$
$statesToVisit, visitedStates, transitionsKnowledge)$
32: **end if**
33: **end while**
34: **if** $statesToVisit = \emptyset$ **and** $!noJumping$ **then**
35: **send** $ExecuteEvent(crawlerAddress, myAddress, \emptyset)$
36: **end if**
37: **else if** $\exists\ \ t \in currentState.transitions\ \ such\ \ that\ \ t.status = assigned$ **then**
38: **push** $ForwardExploration(controllerAddress, crawlerAddress, currentState,$
$statesToVisit, visitedStates, messageKnowledge)\ \ to\ \ parkedQueries$
39: **end if**

3.6 Message Complexities

The message complexity is measured in terms of the maximum number of transmitted messages that may be required by each of the different sharing schemes during the crawling phase, i.e. upper bound. Moreover, the lower bound corresponds to the special case where a minimum number of messages is required. We use the following notation: k is the total number of transitions in the RIA, n is the number of controllers and s is the total number of states in the RIA. We assume a non-faulty environment in this section where a message from a source node x to a destination process y reaches y in a finite amount of time with no message loss. We are interested in the scalability of the proposed P2P RIA crawling system in respect to the number of controllers. Therefore, both the number of states and the number of executed transitions are also important scaling factors and are therefore considered in this analysis. Additionally, we assume that the time for a message communication is much smaller than the time for executing an event in a RIA. We distinguish two types of message: The search messages in the P2P network that require $log(n)$ real messages, and direct messages that require one real message.

Reset-Only:

For each newly executed transition, the $StateInfo$ message is search message that is forwarded to the appropriate controller in the P2P system, resulting in a $log(n)$ number of real messages. Additionally, there is an additional initiating $StateInfo$ direct message sent from the crawler to the controller it is associated with before getting access to other controllers in the P2P network. The maximum number of sent messages is given by:

$$M_1 \leq k(log(n) + 1)$$

k is the total number of transitions

n is the number of controllers

Upon receiving the $StateInfo$ message, the controller sends an $ExecuteEvent$ direct

message back to the original crawler from where the $StateInfo$ message was sent, resulting in k additional direct messages:

$$M_2 \leq k$$

The receiving crawler then executes the transition and sends an $AckJob$ direct message back to the controller associated with the source state of the newly executed transition:

$$M_3 \leq k$$

The maximum number of messages sent during the crawling phase for the Reset-Only scheme is given by:

$$M_{Reset-Only_{Exploration}} \leq M_1 + M_2 + M_3 \equiv k(log(n) + 3)$$

The message complexity of the Reset-Only scheme is therefore given by:

$$C_{Reset-Only_{Exploration}} = O(k \log n)$$

For the termination detection phase, the message complexity of Misra's termination algorithm [51] that is applied in this study is given by:

$$C_{Termination} = O(n \log n)$$

Since the total number of executed transitions in a RIA is higher than the number of controllers, the complexity of the Reset-Only approach during both the exploration and the termination phase is the following:

$$C_{Reset-Only} = O(k \log n) + O(n \log n) = O(k \log n)$$

Note that this is the minimum communication requirements for crawling a RIA in a P2P network and all subsequent approaches have a similar or worse complexity than the Reset-Only approach even-thought they may outperform it in terms of the cost and crawling time.

Shortest-Path schemes:

The shortest-path schemes have the same complexity as the Reset-Only scheme. When the Local-knowledge scheme is applied, controllers may locally find a short path to a target state they are responsible for by using the executed transitions on these states, without exchanging extra messages. On the other hand, when the Shared-knowledge scheme is applied, all forwarding controllers in the chordal ring may also update their transitions knowledge before the $StateInfo$ search message is forwarded, resulting in a better transitions knowledge with no message overhead. Therefore, the complexity of both the Local-knowledge and the Shared-knowledge schemes is equal to the complexity of the Reset-Only scheme.

Original Forward Exploration:

The Original Forward Exploration consists of two steps: (1) Minimum requirements for crawling a RIA using the P2P system, which is equal to the complexity of the Reset-Only scheme. (2) Performing the distributed Breadth-First search starting from the crawler's current state. It consists of sequentially sending a search message to explore all neighboring states of the crawler's current state until it finds an event to be executed from a neighboring state. For every newly executed event, a controller may at most visit all RIA states before reaching a non-executed event using the distributed Breadth-First search, resulting in a maximum of $s(log(n))$ messages sent per newly executed transition, where s is the total number of states in the RIA, as follows:

$$M_2 \leq ks(log(n)).$$

That is, the maximum number of messages sent for the Original Forward Exploration scheme is given by:

$$M_{Original-Forward-Exploration} \equiv k(log(n)+3) + ks(log(n)) \equiv k(log(n)+3+(s(log(n))))$$

Therefore, the complexity of the Original Forward Exploration approach is given by:

$$C_{Original-Forward-Exploration} = O(ks \log n)$$

Locally Optimized Forward Exploration:

The Locally Optimized Forward Exploration consists of remembering states from where the last Forward Exploration query has stopped, allowing for preventing a controller to visit states it has already visited with no non-executed events, which clearly reduces the number of messages sent when performing a Breadth-First search compared the basic Forward Exploration approach. Additionally, remembering states from where the last Forward Exploration query has stopped consists of sending an additional $RememberMe$ direct message for each newly executed transition. The minimum number of messages sent during the Breadth-First search for the Locally Optimized Forward Exploration is reached when the RIA under exploration follows the Breadth-First model, i.e. a controller receiving a forward exploration query always finds a non-executed event from a state that is reached by the last forward exploration. Therefore, the minimum number of messages sent during the Breadth-First search for the Locally Optimized Forward Exploration is given by:

$$M_2 > k(log(n)+1)$$

Therefore, the lower and upper bounds complexity of the Locally Optimized Forward Exploration are as follows:

$$k(log(n)+3) + k(log(n)+1) < M_{Locally-Optimized-Forward-Exploration} < M_{Original-Forward-Exploration}$$

$$2k(log(n)+4) < M_{Locally-Optimized-Forward-Exploration} < M_{Original-Forward-Exploration}$$

Globally Optimized Forward Exploration: The Globally Optimized Forward Exploration scheme is an optimization to the Locally Optimized Forward Exploration scheme and consists of sharing additional knowledge about intermediate states that have been already explored by other controllers during the Forward Exploration but have no event to be executed, which allows for reducing the number of messages sent when globally performing the Breadth-First search. Since the sharing is performed during the Forward Exploration with no message overhead, the controller maintaining the next state to be visited by the global search is sequentially updated. Therefore, the updates only depend on the structure of the RIA graph, which makes the message complexity of the Globally Optimized Forward Exploration scheme comparable to the message complexity of the Locally Optimized Forward Exploration.

3.7 Conclusion

One goal of this thesis is to address the scalability problem when crawling large-scale RIAs concurrently. We conducted a simulation study of a P2P RIA crawling system [56] composed of multiple controllers, where each controller maintains a partial knowledge of the RIA model, and each controller is associated with a set of crawlers, allowing for scalability by avoiding the system bottleneck that occurs when the single controller is simultaneously accessed by a high number of crawlers [74]. Furthermore, we proposed different sharing schemes for efficiently crawling large-scale RIAs, including the Reset-Only, the Local-knowledge, the Shared-knowledge and the Forward Exploration schemes.

Additionally, we proposed two variants of the original forward exploration approach. The first variant, called the Locally Optimized Forward Exploration strategy, consists of reducing the message overhead when performing the Forward Exploration by allowing controllers to remember the last state reached by the distributed Breadth-First that was performed from each state the controller is associated with. As a consequence, controllers

can start all subsequent Breadth-First queries from the state where the last Forward Exploration has stopped.

The second variant of the Forward Exploration, called the Globally Optimized Forward Exploration strategy, consists of sharing all transitions on states that have been already visited by the controllers during the Forward Exploration but have no event to be executed, with other controllers. This way, other controllers may jump over these states, by directly reaching the destination states on these transitions without having to revisit their states again. The sharing of these transitions is performed within the Forward Exploration queries with a little overhead.

Chapter 4

Experimental Results of the Scalable Distributed P2P RIA Crawling with Partial Knowledge

In this chapter, we introduce the simulation results related to the distributed P2P RIA crawling with partial knowledge [56] described in Chapter 3.

We first introduce the developed software and the test-applications used in the simulation. We then compare the performance in crawling time of the different sharing schemes, followed by a performance comparison of the Original Forward Exploration strategy and its variants. Furthermore, an in-depth analysis of the exchanged messages is presented. We also give an in-depth analysis of the Forward-Exploration strategy. Finally, a conclusion is provided at the end of this chapter.

4.1 Implementation

The simulation software that we developed is written in the Java programming language using the Kepler Service Release 1 of the Eclipse software development environment. For

the purpose of simulation, we used the Java SSIM simulation package [5].

SSIM is an object-oriented utility library written in Java that provides discrete event process-based simulation. SSIM is available free to users since 2003. SSIM implements a simulator to run reactive discrete-time processes. These processes execute actions in terms of discrete execution steps performed at given times in response to an event where events describe a piece of information exchanged between two processes through the simulator. During the execution of an action, a process may schedule other future actions for itself, or it may signal events to other processes, which will respond by processing the signaled events at the given further time. These processes are defined by the interface Process and must be implemented by a simulated process. The SSIM library defines the basic interface of a process, and provides the main simulation scheduler, including methods for creating, starting, and stopping processes, and for scheduling events or signaling other events. For instance, a process can start an action immediately once being created or can stop an action being executed. It can also execute an action in response to an event signaled to its process by another process or execute an action in response to a timeout.

This simulation is performed using Java programming language on an Intel(R) Core(TM) i7 CPU running at 2.67 GHz. The system is of type 64-bit operating system with an installed memory (RAM) of 8.00 GB. For figures and charts, We used the MATLAB numerical computing environment. Developed by MathWorks, MATLAB allows plotting of functions and data, creation of user interfaces and interfacing with programs written in other languages , including C, C++ , Java and Fortan. Notice that the experimental results illustrated in this study are based only on the execution environment described above. Using a different execution environment may introduce different results but do not affect the consistency of this study, i.e. the experimental results may be different but they are expected to remain comparable to the results obtained in this study.

4.2 Test-Applications

The first real large-scale RIA we consider is an AJAX-based ClipMarks [1] RIA. ClipMarks consists of 2,663 states and 355,201 transitions and the Reset cost is the equivalent of 32 transition executions. The second real large-scale application considered is the JQuery-based AJAX file browser [2] RIA, which is an AJAX-based file explorer. The number of states and transitions of the file browser depends on the system content. The AJAX-based file explorer we consider has 4,622 states and 429,654 transitions with a Reset cost that is equivalent to 12 transition executions. The third and largest tested real large-scale application is the Bebop [3] RIA. It consists of 5,082 states and 468,971 transitions with a Reset cost that is equivalent to 3 transition executions. Notice that in an effort to minimize any influence that may be caused by considering events in a specific order, the events at each state are randomly ordered for each crawl.

4.3 Comparing the crawling time of the different sharing schemes

This section presents the simulation results of crawling the test-applications using our simulation software. Based on preliminary analysis of experimental results [74], a controller can support up to 20 crawlers without becoming overloaded. For each of the test-applications, we plot the simulated time (in seconds) for an increasing number of controllers from 1 to 20, with steps of 5, while the number of crawlers is constant and set to 20 crawlers. In this simulation, we plot the cost in time required for crawling each of the test-applications and we compare the efficiency of the proposed schemes to the Global-knowledge scheme

[1] http://www.clipmarks.com/ (Local version: http://ssrg.eecs.uottawa.ca/clipmarks/)

[2] http://www.abeautifulsite.net/blog/2008/03/jquery-file-tree/ (Local version: http://ssrg.eecs.uottawa.ca/seyed/filebrowser/)

[3] http://www.alari.ch/people/derino/apps/bebop/index.php/ (Local version: http://ssrg.eecs.uottawa.ca/bebop/)

where all controllers have instant access to a globally shared information about the state of knowledge at all controllers. Notice that the Global-knowledge scheme is unrealistic in our setting and is used only for comparison.

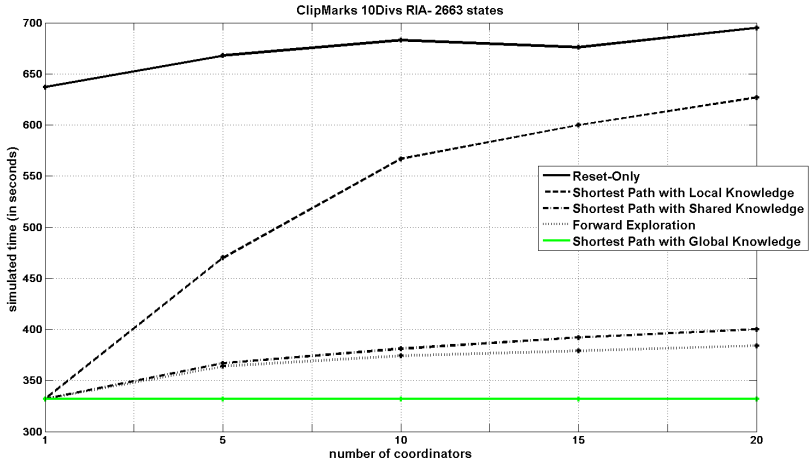

Figure 4.1: Comparing different sharing schemes for crawling the ClipMarks RIA.

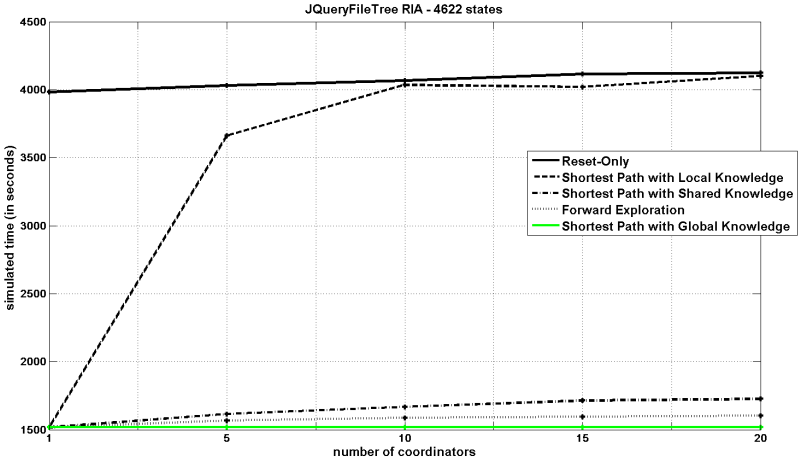

Figure 4.2: Comparing different sharing schemes for crawling the JQuery file tree RIA.

The worst performance is obtained with the Reset-Only strategy, followed by the Local-Knowledge scheme. This is due to the high number of Resets performed as well as the partial knowledge compared to all other strategies. Our simulation results also show that the Local-Knowledge scheme converges towards the Reset-Only strategy as the number of

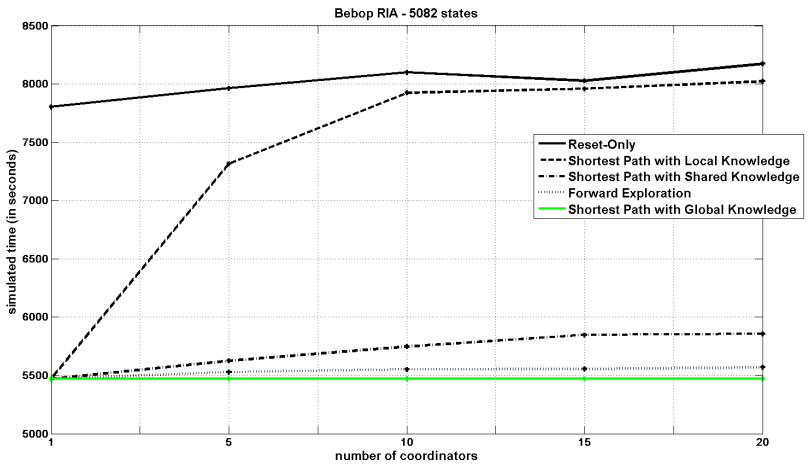

Figure 4.3: Comparing different sharing schemes for crawling the Bebop RIA.

controllers increases, which is due to the low partial knowledge available on each controller when the number of controllers is high.

The Shared-Knowledge scheme comes in the second position and significantly outperforms both the Reset-Only and the Local-Knowledge schemes as controllers have more knowledge about the application graph. However, it is worst than the Forward Exploration strategy due to its partial knowledge.

For all applications, the best performance is obtained with the Forward Exploration strategy. This strategy has performed significantly better than the Reset-Only and the Local-Knowledge schemes and it slightly outperformed the Shared-Knowledge scheme. This is due to the fact that short paths can be only computed toward states the visited controller is responsible for, while the Forward Exploration strategy consists of finding globally the optimal choice based on the distributed Breadth-First search. However, the message overhead introduced by the Forward Exploration scheme is larger than for the Shared-Knowledge scheme.

We conclude that the Reset-Only and the Local-Knowledge schemes are bad strategies. On the other hand, the Forward Exploration strategy is near optimal compared to the Global-knowledge scheme which makes it the best choice for RIA crawling in a decentralized

P2P environment. The Shared-Knowledge scheme is the second best choice being only approximately 10 % worst than for the Global-knowledge scheme for a large number of controllers.

Our simulation results show that the simulated time for all schemes increases as the number of controllers increases, which explains the difficulty of decentralizing the crawling system.

4.4 Comparing the different variants of the Forward Exploration scheme to the Shared-Knowledge scheme

We crawled the Bebop, the JQuery File Tree and the ClipMarks RIAs with 5 controllers and 100 crawlers, and we compared in Table 4.1, Table 4.2 and Table 4.3 the efficiency of crawling these RIAs with the Forward Exploration strategy. Each table illustrates the total cost (in number of executed transitions), the total number of exchanged messages, and the simulated crawling time (in milliseconds) between the Shared-Knowledge, the Locally Optimized Forward Exploration and the Globally Optimized Forward Exploration Schemes, respectively.

In all applications, the total cost of both the Locally Optimized Forward Exploration and the Globally Optimized Forward Exploration schemes is significantly decreased compared to the Shared-Knowledge scheme. This is due to the global search performed by the Forward Exploration which allows to find the shortest path from the crawler's current state compared to the Shared-Knowledge scheme where a short path can only reach states the visited controller is responsible for.

Additionally, both the Locally Optimized Forward Exploration and the Globally Optimized Forward Exploration schemes have more message overhead compared to the Shared-Knowledge scheme. This is due to the Forward Exploration messages that are sent toward neighboring controllers of the crawler's current state, along with the RememberDepth messages that are sent to the controllers from where the Forward Exploration operations have started, which allows controllers to start from where they stopped. We also observe that the number of Forward Exploration messages with the Globally Optimized Forward Exploration scheme is significantly reduced compared to the Locally Optimized Forward Exploration scheme as controllers share transitions on states that have been already visited by other controllers with no non-executed events, which allows for avoiding the repetitive task of revisiting neighboring states that have been already visited by other controllers

during the previous Forward Exploration operations.

In terms of crawling time, the Globally Optimized Forward Exploration outperforms both the Locally Optimized Forward Exploration and the Shared-Knowledge schemes, allowing the global search by the Forward Exploration to reach its highest performance by avoiding as much as possible the repetition of the work that have been already performed by other controllers to reach the same states with no non-executed events.

Furthermore, our measurements show that the number of messages for these two variants of the Forward Exploration are much higher than for the Shared-Knowledge scheme, but a very small difference in total cost and simulation time. This high number of messages may have an impact in network bottleneck due to the extra work performed by the Forward Exploration to find globally the optimal choice using the distributed Breadth-First search. However, the Forward Exploration does not guarantee a good improvement in cost and simulation time over the Shared-Knowledge scheme. This improvement mainly depends on the RIA application and the number of controllers involved during the crawling. As we increase the number of controllers, the Shared-Knowledge scheme performs worse than the Forward Exploration scheme as controllers have more difficulty to share their knowledge. Thus, we believe that the Forward Exploration scheme would significantly outperform the Shared-Knowledge scheme when the number of controllers is very high.

Table 4.1: Comparing the different variants of the Forward Exploration scheme with the Shared-Knowledge scheme for crawling Bebop RIA.

Strategy	Cost (Transitions)	Updating depth messages	Breadth-First messages	Total number of messages	Crawling Time (ms)
Shared-Knowledge	3116092	0	0	1795184	110812
Locally Optimized Forward Exploration	3059602	304599	1296928	3360286	110259
Globally Optimized Forward Exploration	3059544	249233	448539	2472503	109289

Table 4.2: Comparing the different variants of the Forward Exploration scheme with the Shared-Knowledge scheme for crawling the JQuery File Tree RIA.

Strategy	Cost (Transitions)	Updating depth messages	Breadth-First messages	Total number of messages	Crawling Time (ms)
Shared-Knowledge	1953017	0	0	2151770	31220
Locally Optimized Forward Exploration	1890958	334619	8900600	10904140	39005
Globally Optimized Forward Exploration	1890599	272327	396030	2477898	30402

Table 4.3: Comparing the different variants of the Forward Exploration scheme with the Shared-Knowledge scheme for crawling the ClipMarks RIA with 10 divisions.

Strategy	Cost (Transitions)	Updating depth messages	Breadth-First messages	Total number of messages	Crawling Time (ms)
Shared-Knowledge	430117	0	0	1556324	6646
Locally Optimized Forward Exploration	414869	28507	366347	2094430	6953
Globally Optimized Forward Exploration	414345	20323	33516	1560460	6462

4.5 In-depth analysis of the exchanged messages

We analyzed the different types of exchanged messages during the crawling of our largest RIAs with 5 controllers and 100 crawlers. In an effort to easily distinguish between the different types of messages involved at the beginning, in the middle and at the end of the crawling, we divided the distributed crawling task into 20 phases, where each phase corresponds to the execution of (1/20) of the total number of newly executed transitions. Notice that the used number of phases is considered for in-depth analysis only and does not affect the results obtained in this study. We consider the Bebop RIA as an example. In Bebop RIA, the total number of newly executed transitions is 468,971. Therefore, each phase corresponds to the execution of approximately 23,449 transitions. The following figures show the number of exchanged messages when crawling the Bebop RIA with the Shared-Knowledge, the Locally Optimized Forward Exploration and the Globally Optimized Forward Exploration schemes, respectively.

Moreover, a high number of Request Job messages are sent during the last crawling phase (before reaching the termination) for crawling the RIA with the Shared-Knowledge scheme (Figure 4.4). However, since the Forward Exploration consists of globally finding the shortest path to a state with a non-executed event, Request Job messages are eliminated in both the Locally Optimized Forward Exploration and the Globally Optimized Forward Exploration (Figure 4.5 and Figure 4.6) since any state can be globally reached by the Forward Exploration.

Moreover, the number of $Forward - Exploration$ and $RememberMyDepth$ messages with the Globally Optimized Forward Exploration (Figure 4.6) are significantly less than the ones with the Locally Optimized Forward Exploration scheme (Figure 4.5). This is due to the global sharing of the transitions from states that have been already visited with no event to execute with other controllers, which allows for globally preventing these states from getting visited again.

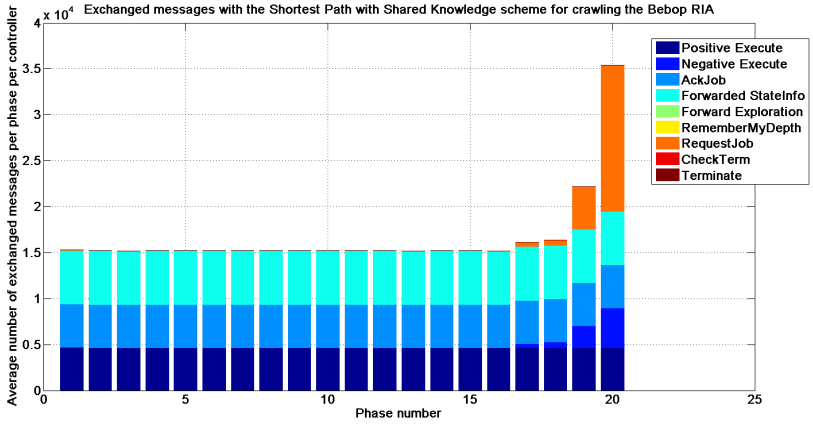

Figure 4.4: Average number of exchanged messages per newly explored transition with the Shared-Knowledge scheme for crawling the Bebop RIA with 5 controllers and 100 crawlers.

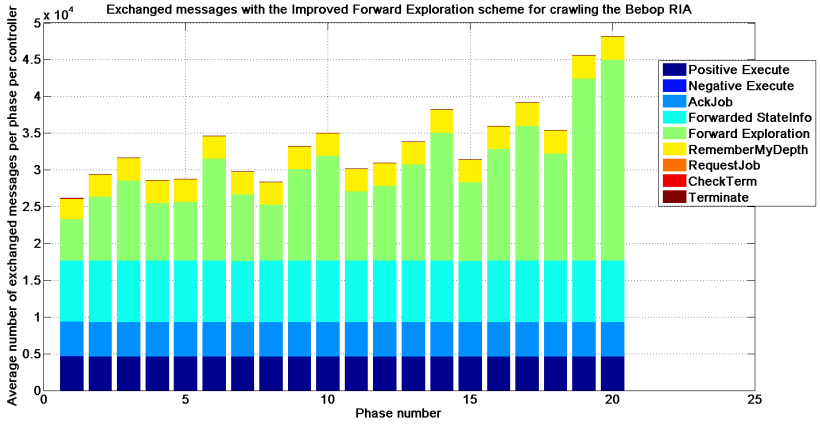

Figure 4.5: Average number of exchanged messages per newly explored transition with the Locally Optimized Forward Exploration scheme for crawling the Bebop RIA with 5 controllers and 100 crawlers.

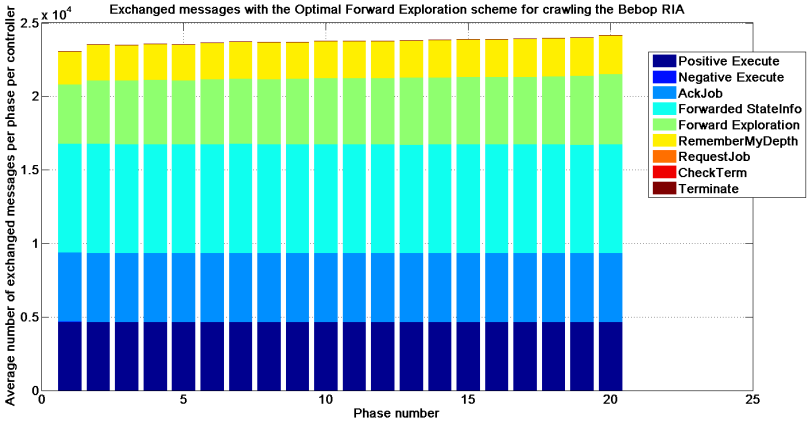

Figure 4.6: Average number of exchanged messages per newly explored transition with the Globally Optimized Forward Exploration scheme for crawling the Bebop RIA with 5 controllers and 100 crawlers.

4.6 In-depth analysis of the Forward-Exploration approach: Non-executed events found in different depths during the Forward Exploration operation

The following figure illustrates the number of non-executed events found in different depths using the Forward Exploration scheme with 5 controllers and 100 crawlers. Note that the depth in which the non-executed events are found is necessarily the same for the Forward Exploration scheme and its variants since they all perform the same Breadth-First search to reach states with non-executed events, while the only difference between these variants is the reduction of the number of messages that are sent when performing the distributed Breadth-First search.

In the following figure, each depth corresponds to the distance of a non-executed event found in a neighboring state from the crawler's current state which is reached by the Forward Exploration. Moreover, the Reset Path executions are non-executed events chosen by a visited controller that cannot be reached by the Forward Exploration scheme, starting from the crawler's current state. Additionally, a non-executed event found from a Request Job message corresponds to an assigned event to an idle crawler.

For all applications, most of the non-executed events are found in lower depths and thus are close to the crawler's current state. The highest depths are reached as we approach the end of the crawling. The figure below shows the non-executed events found in different depths using the Forward Exploration scheme for crawling the Bebop RIA with 5 controllers and 100 crawlers.

The following table shows the number and percentage of non-executed events found in different depths using the Forward Exploration scheme with 5 controllers and 100 crawlers for crawling the ClipMarks, the JQuery File Tree and the Bebop RIAs. The last row of the table shows the compared average size of the Reset path execution if the Forward

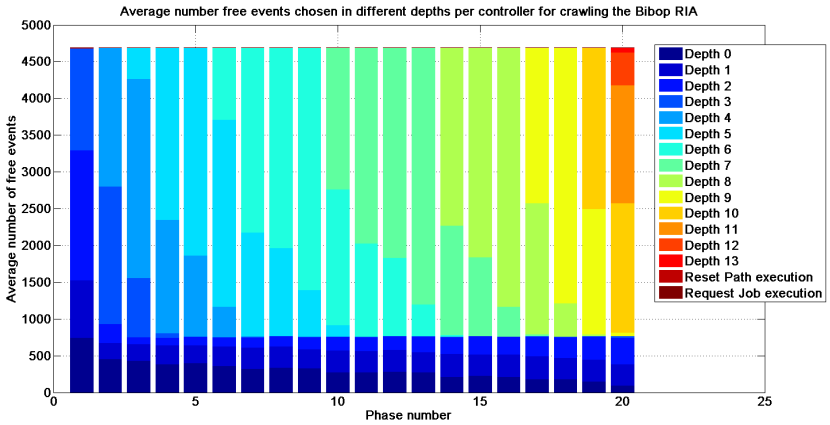

Figure 4.7: Transitions chosen in different depths per phase per controller for crawling the Bebop RIA.

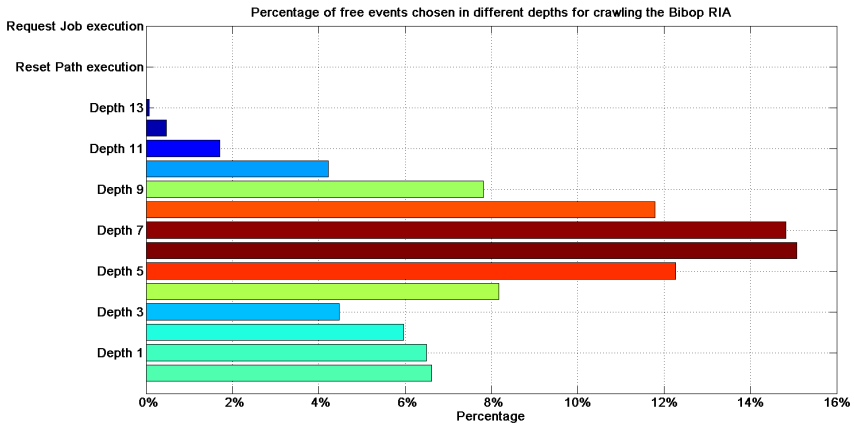

Figure 4.8: Percentage of transitions chosen in different depths during the crawl of the Bebop RIA.

Exploration scheme is not applied, i.e. by applying the Shared-Knowledge scheme for choosing a non-executed event from a state the visited controller is responsible for. When crawling the ClipMarks RIA, more than 96 % of the non-executed events are found in a depth that is less than 2 transitions, where the compared average size of the Reset path execution, if the Forward Exploration scheme is not applied, is 2 transitions. For the JQuery File Tree RIA, around 90 % of the non-executed events are found in a depth that is less than 7 transitions, where the compared average size of the Reset path execution is 7 transitions. When crawling the Bebop RIA, around 74 % of the non-executed events are found in a depth that is less than the compared average size of the Reset path execution of 8 transitions. Therefore, most of the non-executed events when crawling all RIAs using the Forward Exploration are found with a better short path compared to the short path found with the previous schemes. This is due to the global search performed by the Forward Exploration, which makes it a good choice for crawling RIAs.

Additionally, since states are distributed among controllers in the P2P Crawling System, the size of the short path executions from states the controller is responsible for may increase as we increase the number of controllers if the Forward Exploration scheme was not applied. The reason is that controllers may not find the shortest path from the crawler's current state to a non-executed event on a state they are responsible for due to the partial knowledge they maintain. Since the Forward Exploration consists of globally reaching events on neighboring states even though these states are associated with other controllers, it is guaranteed that controllers find the shortest path to a non-executed event from a neighboring state, in contrast to the other approaches where a controller can only choose an event from a state it is responsible for. This makes the Forward Exploration scheme a better choice than the previous approaches as the number of controllers increases. We conclude that the Forward Exploration scheme scales with the number of controllers for crawling large-scale RIAs. However, it may introduce more messages due the communication delay required by the Breadth-First search between controllers to globally find the shortest path

to a non-executed event from a neighboring state using the Forward Exploration scheme.

Table 4.4: Number and Percentage of non-executed events found in different depths using the Forward Exploration scheme with 5 controllers and 100 crawlers for crawling the ClipMarks, the JQuery File Tree and the Bebop RIAs.

Depth	ClipMarks with 10 divisions	JQuery File Tree	Bebop
0	321903 90.6256 %	47911 11.1511 %	31018 6.6141 %
1	19723 5.5526 %	50482 11.7495 %	30500 6.5036 %
2	11979 3.3725 %	66169 15.4005 %	27970 5.9641 %
3	878 0.2472 %	77181 17.9635 %	21017 4.4815 %
4	6 0.0017 %	62722 14.5983 %	38307 8.1683 %
5	25 0.0070 %	47239 10.9947 %	57569 12.2756 %
6	17 0.0048 %	33082 7.6997 %	70721 15.0800 %
7	0 0 %	16661 3.8778 %	69510 14.8218 %
8	0 0 %	12541 2.9189 %	55278 11.7871 %
9	0 0 %	7327 1.7053 %	36665 7.8182 %
10	0 0 %	3989 0.9284 %	19782 4.2182 %
11	0 0 %	1754 0.4082 %	8020 1.7101 %
12	0 0 %	857 0.1995 %	2220 0.4734 %
13	0 0 %	441 0.1026 %	328 0.0699 %
14	0 0 %	220 0.0512 %	0 0 %
15	0 0 %	94 0.0219 %	0 0 %
Execution of a transition on another state	452 0.1273 %	881 0.2050 %	57 0.0122 %
Request Job execution	218 0.0614 %	103 0.0240 %	9 0.0019 %
Average size of Reset Path execution	2	7	8

4.7 Conclusion

In this chapter, we compared the different sharing schemes introduced in Chapter 3 through simulation. Simulation results showed that the Shared-Knowledge scheme is efficient, simple and scalable, while the Reset-Only and Local-Knowledge schemes do not scale with the number of crawlers. Additionally, the Globally Optimized Forward Exploration strategy is near optimal compared to the ideal setting and outperforms the Reset-Only, the Local-Knowledge, the Shared-Knowledge, the Original Forward Exploration and the Locally Optimized Forward Exploration schemes. This is due to its ability to globally finding the shortest path with little overhead, compared to all other strategies. This makes the Forward Exploration a good choice for general purpose crawling in a decentralized P2P environment, followed by the Shared-Knowledge scheme. Moreover, the Globally Optimized Forward Exploration outperformed the Original Forward Exploration, the Locally Optimized Forward Exploration and the Shared-Knowledge schemes by avoiding as much as possible the repetition of the work that have been already done by other controllers to reach the same states with no non-executed events.

Chapter 5

Fault-Tolerant RIA Crawling System

In this chapter, we address the resilience problem when using the proposed P2P RIA crawling system introduced in Chapter 3 when both crawlers and controllers are vulnerable to node failures. By fault tolerance, we mean that the non-faulty crawlers and controllers will still be able to achieve the RIA crawling, knowing that some crawlers and controllers may fail at an arbitrary time during the crawling. We introduce three recovery mechanisms for crawling RIAs in a faulty environment: The Retry, the Redundancy and the Combined mechanisms.

5.1 Assumptions

- The unreliable chordal ring network is composed of a set of controllers, and a set of crawlers is associated with each of these controllers where both crawlers and controllers are vulnerable to Fail-stop failures, i.e. they may fail but without causing harm to the system. We also assume a perfect failure detection and reliable message delivery which allows nodes to correctly decide whether another node has crashed or not. This prevents false suspicions of failures, i.e. a node appears failed when it is actually alive.

- Crawlers can be unreliable as they are only responsible for executing an assigned job, i.e. they do not store any relevant information about the state of the RIA. Therefore, a failed crawler may simply disappear or leave the system without being detected, assuming that some other non-faulty crawlers will remain crawling the RIA. However, for the RIA crawling to progress, there must be at least one non-faulty crawler that is able to achieve the RIA crawling in a finite amount of time. We also assume that a joining crawler knows the address of the controller it is associated with through some external mechanism.

5.2 Solutions

In the fault-tolerant P2P RIA crawling system, crawlers and controllers must achieve two goals in parallel: Maintaining the ring topology and performing the fault-tolerant RIA crawling. The maintenance of Chord consists of maintaining the ring topology as nodes join and leave the network and repairing the ring when failures occur, independently of the RIA crawling. On the other hand, the RIA crawling must be able to achieve the intended crawling task despite the permanent change of the Chord structure as nodes join, leave or fail using a data-recovery mechanism. We discuss these two operations separately. We first introduce the maintenance of the Chord structure, including the failure detection and recovery techniques. We then introduce the fault-tolerant RIA crawling protocol and the different data-recovery mechanisms.

5.2.1 Chord Maintenance

Controllers maintain the topology of the P2P RIA crawling system and are responsible for storing information about the RIA crawling. If a controller fails, the connectivity of the Chord structure is affected and some controllers become unreachable from other

controllers. Since Chord is a continuously evolving system, it is required to continuously repair the overlay to ensure that the ring remains connected and supports efficient look-ups. The maintenance of the Chord structure consists of maintaining its topology as controllers join and leave the network and repairing the ring when failures occur among controllers independently of the RIA crawling.

There are mainly two different approaches for maintaining the Chord structure when failures occur as introduced in Section 2.4 : The active and the passive approaches. In this study, we use the passive approach for maintaining the Chord structure where less than $n/2$ successive nodes may fail simultaneously, under the assumption that the system is vulnerable to only fail-stop failures with perfect failure detection and reliable message delivery.

5.2.2 Fault-Tolerant Crawling Protocol

A major problem we address in this section is to make the proposed P2P RIA crawling system described in Chapter 3 resilient to node failures, i.e. to allow the system to achieve the RIA crawling when both crawlers and controllers may fail. The fault-tolerant crawling system is required to discover all states of a RIA despite failures, so that the entire RIA graph is explored. In the P2P crawling system, controllers are responsible for storing part of the discovered states. If a controller fails, the set of states maintained by the controller is lost. For the P2P crawling system to be resilient, controllers are required to apply a data recovery mechanism so that lost states and their transitions can be eventually recovered after the reestablishment of the ring. For the data recovery to be consistent, i.e. all lost states can be recovered when failures occur, each newly reached state by a crawler must be always stored by the controller the new state is associated with before the transition leading to the state is assumed to be executed. If a new state is not stored by the controller it is associated with, the controller performing a data-recovery will not be aware about the

state and the data-recovery becomes inconsistent if the state is lost. As a consequence, the state becomes unreachable by crawlers and the RIA graph cannot be fully explored.

In the P2P RIA crawling system introduced in Chapter 3, an acknowledgment for an assigned transition consisted of a crawler informing the controller responsible for the transition about the destination state that follows from the transition execution, as shown in Figure 3.2. However, in a faulty environment, a crawler may fail after having sent the result of a transition execution to the previous controller and before contacting the next controller. As a consequence, the destination state of the executed transition may never be available to the next controller and data-recovery of the state cannot be performed. For the P2P crawling system to be resilient, every newly discovered state must be stored by the next controller before the executed transition is updated by the previous controller. Therefore, we introduce a change to the P2P crawling system described in Chapter 3 to make it fault-tolerant, as shown in Figure 5.1: When the next controller responsible for a newly reached state by a crawler is contacted, the controller stores the newly discovered state and forwards the result of the transition execution, i.e. an *AckJob* message, to the previous controller. As a consequence, the controller responsible for the transition can only update the destination state of the transition after the newly reached state is stored by the next controller. Moreover, the fault-tolerant P2P system requires each assigned transition by a controller to be acknowledged before a given time-out. When the time-out expires due to a failure, the transition is reassigned by the controller to another crawler at a later time.

The data recovery mechanisms allow for either recovering lost states a failed controller was responsible for, reassigning all transitions on the recovered states to other crawlers and rebuilding the RIA graph model, or for making back-up copies of the RIA information on neighboring controllers when a newly reached state or an executed transition is available to a controller so that crawlers can resume crawling from where a failed controller has stopped, as introduced in the following section.

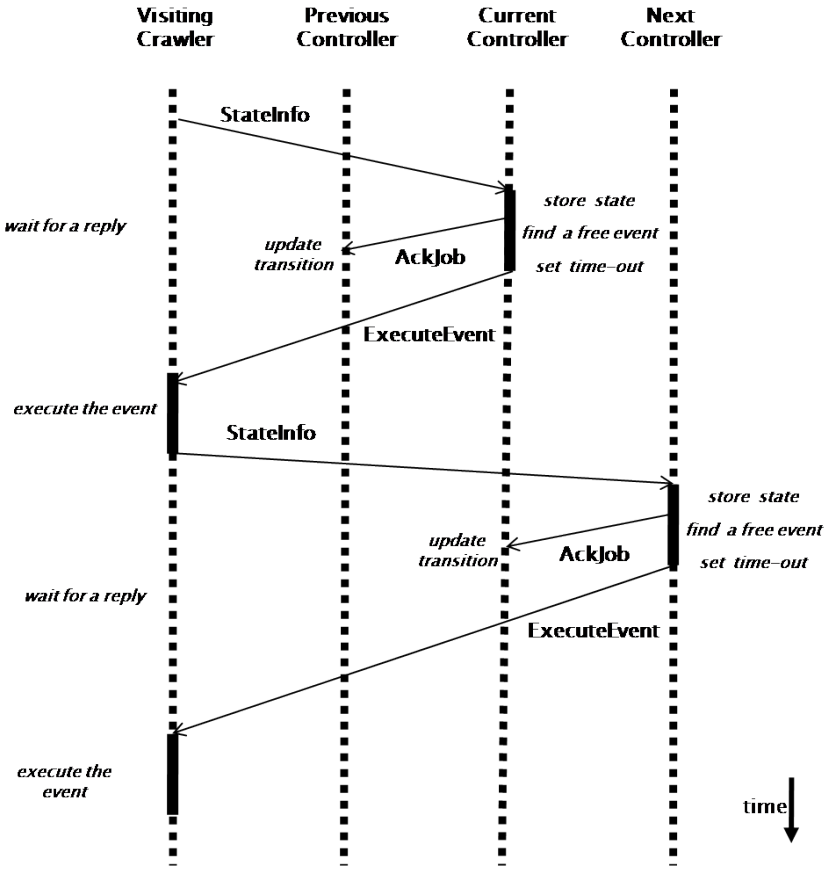

Figure 5.1: The Fault-Tolerant P2P RIA Crawling during the exploration phase.

5.3 Crawling Data Recovery Mechanisms

We introduce three data recovery mechanisms to achieve the RIA crawling task properly despite node failures, which are based on existing data recovery mechanisms introduced in the literature in Section 2.3.5, as follows:

5.3.1 Retry Strategy

The Retry strategy [100] consists of replaying any erroneous task execution, hoping that the same failure will not occur in subsequent retries. The Retry Strategy may be applied to the P2P RIA crawling system by re-executing all lost jobs a failed controller was responsible for. When a controller becomes responsible for the set of states a faulty controller was responsible for, the controller allows crawlers to explore all transitions on these states again. However, since all states held by the failed controller disappear, the new controller may not have the knowledge about the states the failed controller was responsible for and therefore can not reassign them. To overcome this issue, each controller that inherits responsibility from a failed controller may collect lost states from other controllers.

The state collection operation consists of forwarding a message, called *CollectStates* message, which is sent by a controller replacing a failed one. The message goes around the ring and allows all other controllers to verify if the ID of any destination state of executed transitions they maintain belongs to the set of states the sending controller is responsible for; such state will be appended to the message. This can be performed by including the starting and ending keys defining the set of state IDs the sending controller is responsible for as a parameter within the *CollectStates* message. A controller receiving its own *CollectStates* message considers the transitions on the collected states as non-explored. A situation may arise during the state collection operation where a lost state that follows from a transition execution is not found by other controllers. In this case, a controller responsible for a transition leading to the lost state must have also failed. The

transition will be re-executed and the controller responsible for the destination state of the transition will be eventually contacted by the executing crawler and therefore becomes aware about the lost state. For the special case where the initial state can be lost, a transition leading to the initial state may not exists in a RIA. As a consequence, the *CollectStates* message may not be able to recover the initial state. To overcome this issue, a controller that inherits responsibility from a failed controller always assumes that the initial state is lost and asks a visiting crawler to load the $SeedURL$ again in order to reach the initial state. The controller responsible for the initial state is then contacted by the crawler and becomes aware about the initial state.

5.3.2 Redundancy Strategy

The Redundancy Strategy is a strategy based on Redundant Storage [100] and consists of maintaining back-up copies of the set of states that are associated with each controller, along with the set of transitions on each of these states and their status, on the successors of each controller. Notice that a back-up copy of states is not cached by a neighboring controller. It is stored as a copy in the database in a distinct set of states called *backUpStates* to distinguish it from the discovered states in the set of *myDiscoveredState* the controller is responsible for. The main feature of this strategy is that states that were associated with a failed controller and their transitions can be recovered from neighboring controllers, which allows for reestablishing the situation that was before the failure i.e. the new controller can start from where the failed controller has stopped. This strategy consists of immediately propagating an update from each controller to its r back-up controllers in the ring when a new relevant information is received, where r is the number of back-up controllers that are associated with each controller, i.e. a newly discovered state or a newly executed transition becomes available to the controller. When a newly reached state is stored by a controller, the controller updates its back-up controllers with the new state before sending an acknowledgment to the previous controller. This ensures that every discovered state

becomes available to the back-up controllers before the transition is acknowledged. Note that the controller responsible for the new state must receive an acknowledgment of reception from all back-up controllers before sending the acknowledgment. On the other hand, each executed transition that becomes available to the previous controller is also updated among back-up controllers before the result of the transition is locally updated by the previous controller. In case some of the r succeeding controllers fail simultaneously , the lost states along with their executed transitions remain available to at least one of the $(r + 1)$ controllers that are maintaining back-up copies [115]. Furthermore, when a controller fails, the list of succeeding controllers maintained by each controller may change. If a controller notices a change on its list of successors, it may update the new controllers in this list with all states it is associated with, along with the executed transitions on these states so that the back-up copies become available to its new successors.

5.3.3 Combined Strategy

One drawback of the Redundancy strategy is that an update is required for each newly executed transition received by a controller. This may be problematic in RIA crawling since the number of transitions is usually much higher than the number of states. The Combined Strategy overcomes this issue by periodically copying the executed transitions a controller maintains so that if the controller fails, a portion of the executed transitions remains available to the back-up controller, and the lost transitions that have not been copied have to be re-executed again. The advantage of using the Combined data recovery strategy is that all executed transitions maintained by a controller are copied one time at the end of each update period rather than copying every newly executed transition when the result of the transition execution becomes available to a controller, as introduced by the Redundancy Strategy. Note that the state collection operation used by the Retry strategy is required by the Combined Strategy since not all states are recovered when a failure occurs.

Chapter 6

Analytical Evaluation of the Fault-Tolerant RIA Crawling System

In this chapter, we compare the efficiency of the Retry, the Redundancy and the Combined data recovery strategies during the crawling phase as crawlers and controllers fail. We are mainly interested in the overhead introduced by a node failure for each of the data recovery strategies, under the assumptions introduced in Section 5.1. We use the following notation: t_t is the average required time required for executing a new transition, T is the total crawling time with normal operation, k is the total number of transitions in the RIA, c is the average communication delay of a direct message between two nodes, n is the number of controllers, m is the number of crawlers, s is the total number of states in the RIA and e is the average time required for executing a new transition which includes going through a path of ordered transitions before reaching the state with the next transition to be executed. Moreover, since the recovery of Chord is performed in parallel and is independent of the RIA crawling, we ignore the delay introduced by the $log^2(n)$ rounds of idealization and we assume that queries are resolved with only $log(n)$ messages after a short period of time after the failure of a controller. We also assume that there are no simultaneous failures of successive controllers, which means that only one back-up copy is

maintained by each controller, i.e. r is equal to 1.

6.1 Crawling Time with Normal Operation

The RIA crawling time with normal operation, i.e. with no failures, using the P2P Crawling System introduced in Fig. 5.1 is approximated as follows:

For each newly executed transition, a $StateInfo$ search message is forwarded to the appropriate controller in the P2P system, resulting in a delay of $c.log(n)$ units of time per transition. Additionally, there is an additional initiating $StateInfo$ real message sent from the crawler to the controller it is associated with before getting access to other controllers in the P2P network. This results in a total of $c.(log(n) + 1)$ messages for each $StateInfo$ message sent. Upon receiving the $StateInfo$ message, the controller stores the newly reached state and then sends an acknowledgment back to the previous controller, allowing the receiving controller to update the destination state of the executed transition. A new transition to be executed is also returned back to the visiting crawler, resulting in one additional real message. Furthermore, the controller sets a time-out to the executing crawler called $time - out_{Crawler}$ in order to detect failing crawlers that do not return messages. To prevent false alarms when executing a new transition which takes a longer time than usual, we set the value of $time - out_{Crawler}$ to twice the maximum round-trip time for the $ExecuteEvent$ message, i.e. $time - out_{Crawler} = 2(c + e_{max})$, where e_{max} is the maximum time required for executing a new transition by a crawler. If the time-out expires before the crawler has sent an acknowledgment back to the visited controller, the transition is reassigned to another crawler at a later time. In this section, we are interested in the delay of executing a new transition without failures among crawlers, i.e. the average delay for executing a new transition with normal operation is equivalent to e units of time. The crawling time with a failing crawler is described in Section 6.4. Assuming that crawlers do not fail during normal operation, the receiving crawler executes the assigned

transition, resulting in an average delay of e units of time. The crawler finally forwards the information about the newly reached state to the next controller.

Therefore, the delay of executing a new transition with normal operation, called t_t, for a crawling system composed of n controllers and one crawler is given by:

$$t_t = c.(log(n) + 2) + e \text{ units of time}$$

6.2 Processing Time per Message Type

In order to evaluate the impact of the message processing time on the crawling performance, we perform a simulation study on experimental data-sets with a crawling system composed of 100 controllers and 1000 crawlers in the execution environment introduced in Section 4.1. We measure the processing time of messages involved during the crawling and we compare the processing time of the Search, ExecuteEvent, Acknowledgment and Backup update messages, assuming that controllers are underloaded, as shown in Fig. 6.1.

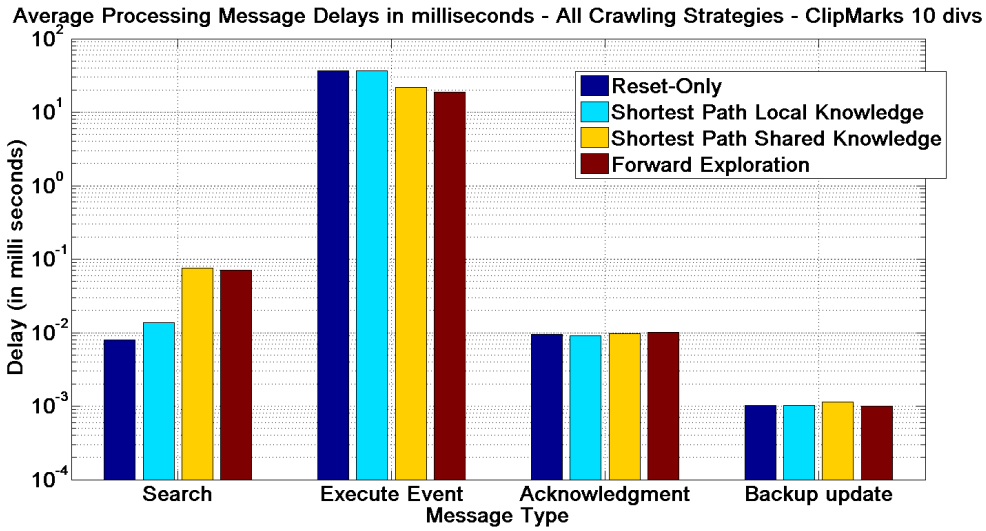

Figure 6.1: Average processing time per message type in milliseconds for a crawling system composed of 100 controllers and 1000 crawlers - ClipMarks 10 divs.

- Search Message: Fig. 6.1 shows that the search message with the Reset-Only scheme

has the lowest processing time, followed by the Local-knowledge scheme. This is due to the ability of the Reset-Only and Local-knowledge schemes to find non-executed events locally based on their local knowledge, which usually leads to a Reset, along with a long path of ordered transitions before reaching the target state. On the other hand, the Shared-knowledge and the Forward-Exploration schemes take more time to find a non-executed event by finding a shortest path based on their shared knowledge or by globally performing a distributed Breath-First search respectively , usually not performing a Reset.

However, since the ExecuteEvent message processing time is usually much higher than the Search message processing time (at least 100 times higher), the processing time of the Search message has a low impact on the overall crawling performance.

- Acknowledgment Message: The processing time of the Acknowledgment message is comparable in all crawling strategies since it consists of updating the executed transition with the newly available destination state independently of the crawling strategy. Notice that the Acknowledgment Message processing time is significantly faster than both the Search and the ExecuteEvent messages since the controller only updates an executed transition with the destination state rather than searching for a new event or executing an assigned event respectively.

- Back-up Update Message: The processing time of the back-up update message is comparable in all crawling strategies since it consists of storing a back-up transition on the database of a back-up controller independently of the crawling strategy. Furthermore, Fig. 6.1 shows that the back-up update message processing time is slightly faster than the Acknowledgment message processing time. This is due to the fact that back-up transitions are only updated on the database of a back-up controller while processing an acknowledgment consists of finding the source state of the executed transition before storing the result of the transition execution.

Based on the measurements of Fig. 6.1, the back-up update Message processing time is significantly faster than the processing time of all other messages involved during the crawling (at least 10 times higher). Therefore, the processing time of the back-up update message has an insignificant impact on the crawling performance when the underloaded controllers concurrently perform the back-up update operation. Moreover, Fig. 6.1 shows that the back-up update message processing times are comparable for all crawling strategies. We conclude that the fault tolerance overhead introduced by the Redundancy strategy is independent of the crawling strategy.

6.3 Failure Rate

We assume a resource allocation of interconnected computers for crawling RIAs and we distinguish between two categories of system architectures with corresponding failure characteristics: (1) P2P node failures, where nodes are publicly accessible from a P2P infrastructure. (2) Dedicated servers of Non-P2P context, where nodes can only communicate with each other on a private network.

We also assume a distribution function of node failures according to a Poisson process [65] where failures may occur randomly, continuously and independently at a constant average rate λ_f during the crawling phase.

6.3.1 P2P Node Failures

P2P node failures are measured based on the average online time of a peer remaining active in a P2P network before the peer disappears due to a disconnection (connectivity root cause), i.e. a node that disappears from the P2P network is assumed to have failed. The Median Session Time [103] of a node is a measurement metric of churn in peer-to-peer networks and is defined as the elapsed time between the time when the node joins the

network and the time when the node subsequently leaves the network. The Average Online Time is also found in the literature and has a similar definition as the median session time.

Measurement studies have shown that the Median Session Time ranges from as long as an hour to as short as a few minutes depending on the peer-to-peer structure deployed, as shown in Table 6.1:

Table 6.1: Observed average session times in various peer-to-peer systems.

Study Reference	Peer-to-peer Structure	Measured Median Session Time	Failure Rate
[103], [92], [4]	structured overlays (Bamboo, Overnet, Kademlia, Chord)	less than 60 minutes	1 failure per hour
[104]	unstructured overlays (Gnutella and Napster)	less than 60 minutes	1 failure per hour
[44]	unstructured overlays (Gnutella and Napster) (Gnutella and Napster)	less than 10 minutes	6 failures per hour
[86]	unstructured overlays (Kazaa) (Kazaa)	less than 10 minutes	6 failures per hour
[105]	unstructured overlays (FastTrack) (FastTrack)	less than 1 minute	60 failures per hour

Table 6.1 shows that different measurements of the median session time of a node in a structured P2P network are found to be approximately 1 hour [103] [92] [4]. However, the median session time of a node in unstructured P2P networks is much shorter due to their lack of structure. [103] argues that a structured peer-to-peer network built on a DHT should be robust for session times of at most 1 hour. In this study, we assume that the failure rate λ_f is equal to the average failure rate of a node in structured P2P overlay networks λ_{P2P}, which is equivalent to 1 failure per hour.

6.3.2 Failures of Dedicated Servers

Failures of dedicated servers are categorized based on multiple root causes in a private network composed of one or more interconnected nodes. There are mainly five high level root causes for failures of dedicated servers: Hardware, software, network failures (connec-

tivity), human and environment failures (power outages or A/C failures). Various studies measured the node failure rates in different private networks using three metrics: The length of the measurements, the number of computers deployed in the private network and the root causes of the failure. The failure rate measurements with all root causes (hardware, software, connectivity, human and environment) are shown in Table 6.2, as follows:

Table 6.2: Observed average node failure rates in various private networks.

Study Reference	Test Length	Number of Nodes	Number of Failures	Failure Rate
[48], [49]	3 years	4,000 (Machines in Tandem systems)	800	7.61 e-6 failures per hour
[72]	6 months	70 (Windows NT mail servers)	1,100	3.588 e-3 failures per hour
[29]	3-6 months	3,000 (Machines in Internet services)	501	5.080 e-5 failures per hour
[30]	8 months	7 (Machines in VAX systems)	364	8.898 e-3 failures per hour
[116]	22 months	13 (VICE file servers)	300	1.436 e-3 failures per hour
[61]	3 years	2 (IBM machines of 370/169 mainframes)	456	8.67 e-3 failures per hour
[62]	1 year	395 (Nodes in machine room)	1285	3.711 e-4 failures per hour
[106]	1-36 months	70 (Nodes in university and Internet services)	3200	3.383 e-3 failures per hour
[54]	4 months	503 (Nodes in corporate environment)	2127	1.447 e-3 failures per hour

The average failure rate $\lambda_{Dedicated-Servers}$ of all measurements introduced in Table 6.2 is equivalent to $3.095e-3$ failure per hour. Notice that the average failure rate in the P2P context is approximately 1000 times higher than for the dedicated servers.

6.4 Failing Crawlers

A controller that has assigned a new job to a visiting crawler becomes aware that the crawler is non responsive when the time-out of the assigned transition, called $time - out_{Crawler}$ has expired, independently of the data-recovery mechanism applied. If a crawler fails before the result of a transition is received by its appropriate controller, the transition is reassigned to another crawler at a later time. Therefore, each failure of a crawler during the execution of a new job introduces a delay of the $time - out_{Crawler}$ plus one transition execution, where $time - out_{Crawler}$ is equivalent to $2(c + e_{max})$ units of time. Notice that the crawling performance after the failure has occurred is reduced since only remaining crawlers will be active for exploring next transitions. The probability of having a failing crawler depends on the total crawling period, which varies from one RIA to another. We consider the situation when a single crawler fails during the total crawling period. With a number of executed transitions K_t at time t before the crawler fails during a transition execution, the total crawling time is $((K_t.t_t)/m) + ((time - out_{Crawler} + t_t)/(m-1)) + (((K - K_t).t_t)/(m-1))$ units of time. Clearly, the time of occurrence of failures during the total crawling period has an impact on the crawling performance: If a crawler fails at the beginning of the crawling period, the system performance is slightly degraded since only remaining $(m-1)$ crawlers will be active for exploring almost k transitions, with a decline of $1/m$ on the time performance. On the other hand, if a crawler fails at the end of the crawling period, the impact is negligible since crawlers have already explored most of the transitions before the failure occurred. Assuming that a crawler fails in the middle of the crawling period, i.e. k_t is equal to $k/2$, the overhead introduced by a failed crawler is $1/(2.m)$.

6.5 Failing Controllers with Low Load

Preliminary analysis of experimental results [75] have shown that a controller can support up to 20 crawlers before becoming a bottleneck. In this section, we assume that each controller is associated with at most 20 crawlers so that controllers are not overloaded. The delay introduced by each data recovery mechanism, when a controller fails, is as follows:

6.5.1 Retry Strategy

When a controller fails, all states associated with the controller are lost and all transitions on these states have to be re-executed. The impact of the time when a controller fails during the total crawling time is important when the Retry strategy is performed. Since states are randomly distributed among controllers, the number of transitions to be re-executed when a controller fails is of the order of $1/n$ at the beginning of the crawling period, and a percentage of $1/n$ at the end of the crawling period. Assuming that a controller fails in the middle of the total crawling period T, the delay introduced by the failure of a controller is equivalent to $\lambda_f.T/(2.n)$. Additionally, the state collection operation results in a delay of $c.(n-1)$ units of time before the message is received back by the neighbor responsible for the recovered states, which is very small compared to the first delay and could be neglected. Note that the performance of making a choice by the neighbors for the next job to be executed may decrease during the state collection operation since the controller will not have the knowledge about states it is newly associated with. Therefore, the overhead of the Retry strategy is equivalent to $(\lambda_f.T)/(2.n)$.

6.5.2 Redundancy Strategy

In the Redundancy Strategy, the update operations are performed concurrently. When a controller fails, all states associated with the controller along with the executed transitions on these states are recovered by the Redundancy strategy. To do so, each result of a newly executed transition that becomes available to a controller is updated on its successor before the transition is locally updated. However, since the next controller responsible for sending the result of the executed transition is not required to wait for the transition to be acknowledged before finding a job for the visiting crawler, the delay introduced by the transition update operation is very short and therefore can be ignored. Notice that controllers may possibly become a bottleneck due to the additional processing messages if the number of transitions is high, i.e. due to the update of all newly executed transitions among the back-up controllers. However, this possibility is ignored in the following.

Finally, a controller noticing a change on its list of successors due to a failed neighbor updates its new successor with all states and transitions the controller maintains and waits for an acknowledgment of reception from the back-up controller before proceeding, resulting in one additional update operation per failure to be performed with a delay of $2c$ units of time, assuming that the size of the message is relatively small. Notice that the update operation delay increases as the size of the data included in the message increases. The overhead of the Redundancy strategy is given by $(2.c)/(t_t)$.

6.5.3 Comparison of Retry and Redundancy Strategies when Controllers are Underloaded

Fig. 6.2 compares the overhead of the Retry and the Redundancy strategies with respect to the P2P node failure failure λ_f when controllers are not overloaded. Fig. 6.2 shows that the Redundancy strategy significantly outperforms the Retry strategy as the number of failures increases and is as better as the Retry strategy at the failure rate in the P2P context

λ_{P2P} (Red Line in Fig. 6.2). We conclude that the Redundancy strategy outperforms the Retry strategy when controllers are underloaded. Notice that this conclusion holds true under the condition that each controller is associated with at most 20 crawlers, so that controllers remain underloaded. In the case that more that 20 crawlers are associated with each controller, controllers may become a bottleneck and the Redundancy strategy may not remain efficient compared to the Retry strategy, due to the repetitive back-up update of every executed transition required for redundancy, i.e. processing backup updates by controllers would result in a high delay when controllers are overloaded, which could have a negative impact on the crawling performance, and would possibly exceed the delay introduced by the Retry strategy.

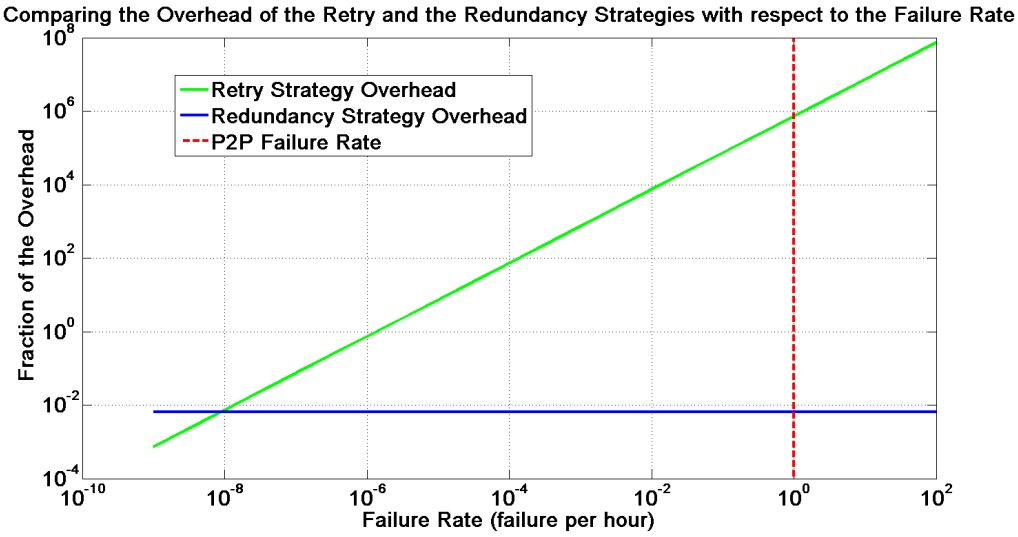

Figure 6.2: Comparing the Overhead of the Retry and the Redundancy strategies with respect to the failure rate, assuming that controllers are not overloaded.

6.6 Combined Strategy at relatively High Load

The Combined Data Recovery Strategy consists of periodically copying the executed transitions a controller maintains so that, if the controller fails, a portion of the executed transitions remains available in the back-up controller, and lost transitions that have not

been copied have to be re-executed again. The advantage of using the combined data recovery strategy when controllers are relatively overloaded is that all executed transitions maintained by a controller are copied together at the end of each update period rather than copying every newly executed transition separately when the result of the transition execution becomes available to a controller, as introduced by the Redundancy Strategy. Notice that the update operations using the Combined Strategy are performed concurrently between back-up controllers, i.e. the update operations are processed in parallel.

Let N_t be the number of executed transitions maintained by a given controller per update period. The update period, i.e. the time required for executing N_t transitions, called T_p, is given by:

$$T_p = N_t.t_t \text{ units of time} \tag{6.1}$$

We are interested in the additional delay introduced by the Combined Strategy compared to the update period T_p. The Overhead introduced by the Combined Strategy is defined as follows:

$$\text{Overhead} = \frac{\text{Additional delay in one update period}}{\text{Normal Operation delay in one update period}}$$

The overhead introduced for fault handling using the combined data recovery strategy includes two parts: The redundancy management and the retry processing operations. We aim to minimize the sum of the two operations which depends on two parameters: The update period T_p and the failure rate λ_f. We ask the following question: What is the value of T_p that minimizes the Combined Strategy Overhead given the failure rate λ_f.

6.6.1 Redundancy Management Delay

We measure by simulation the processing time required for updating the database with back-up transitions using the simulation software introduced in Section 4.1. In this simulation, we plot the average delay required for processing the back-up updates with an increasing number of transitions when crawling the test-applications introduced in 4.2 with a crawling system composed of 100 controllers and 1000 crawlers. Notice that only one back-up copy is maintained by each controller in this simulation, i.e. r is equal to 1.

Let p be the delay required for processing the update of backup transitions.

Fig. 6.3 shows the measurement of the processing delay p introduced when updating the database in milliseconds, with an increasing number of transitions from 1 to 1000, with steps of 100 (excluding the communication delay for sending and receiving an acknowledgment back by the backup controller). The best Fit Line in Fig. 6.3 (Red Line) corresponds to the overhead of the Redundancy strategy with respect to the number of transitions to be updated.

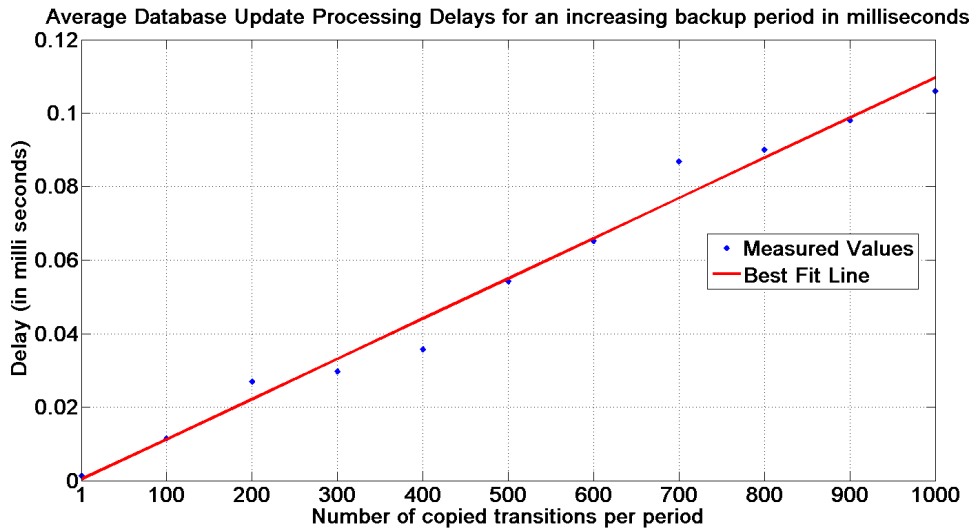

Figure 6.3: Measurements of the processing delay p for updating the database for an increasing number of copied transitions.

Based on the processing time measurements of Fig. 6.3, we obtain the linear equation

$Overhead_{Redundancy}$ as a function of the number of copied transitions per update period N_t, as follows:

$$Overhead_{Redundancy} = 0.0001094.N_t + 0.00030433 \text{ (in milliseconds)} \tag{6.2}$$

The curve of $Overhead_{Redundancy}$ corresponds to the delay required for processing the update of backup transitions called p. The delay required for processing one back-up copy is $T_p.p/t_t$ units of time, where p is shown in Fig. 6.3. Moreover, there is an additional communication delay required for sending the backup copy and receiving the acknowledgment back from the back-up controller of $2.c$ units of time. Therefore, the total delay introduced by the redundancy management operation at the end of each period, called T_{bp}, is given by:

$$T_{bp} = \frac{T_p.p}{t_t} + 2.c \tag{6.3}$$

Notice that the redundancy update operations are performed periodically and therefore are independent of the failure rate λ_f.

6.6.2 Retry Processing Delay

The Retry Processing operation consists of re-executing, after a failure, the lost transitions that were executed after the last redundancy update operation. We assume that failures among controllers occur on average in the middle of the update period. Given the failure rate λ_f, the failure probability of a given controller is $\lambda_f.T_p$. In this case, on average N_t transitions must be executed again, which takes $T_p/2$ units of time.

$$T_{rp} = \frac{\lambda_f.T_p^2}{2} \tag{6.4}$$

6.6.3 Total Overhead introduced by the Combined Strategy

The overhead introduced by the redundancy management and the retry processing operations is given by:

$$Overhead_{CombinedStrategy} = \frac{\text{Additional delay in one period}}{\text{Normal Operation delay in one period}} = \frac{T_{bp} + T_{rp}}{T_p}$$

$$= \frac{\frac{T_p.p}{t_t} + 2.c + \frac{\lambda_f.T_p^2}{2}}{T_p}$$

$$Overhead_{CombinedStrategy} = \frac{\lambda_f.T_p}{2} + \frac{2.c}{T_p} + \frac{p}{t_t} \tag{6.5}$$

The minimum value of T_p corresponds to an update period with only one transition execution, i.e. $T_p = t_t$. On the other hand, the maximum value of T_p corresponds to an update period with an average of k/n transition execution, where k/n is the average maximum number of transitions that can be maintained by each controller, i.e. $T_p = k.t_t/n$.

6.6.4 The value of T_p to minimize the Combined Strategy Overhead

At the minimum value of $Overhead_{CombinedStrategy}$, we have:

$$dOverhead_{CombinedStrategy}/dT_p = 0$$

which implies

$$\frac{\lambda_f}{2} - \frac{2.c}{T_p^2} = 0$$

and

$$T_p = \sqrt{\frac{2.c}{\frac{\lambda_f}{2}}}$$

That is, the minimum value of T_p is given by:

$$T_p = 2\sqrt{\frac{c}{\lambda_f}} \tag{6.6}$$

The value of T_p with the minimum Combined Strategy Overhead, as a function of the failure rate λ_f, is shown in Fig. 6.4.

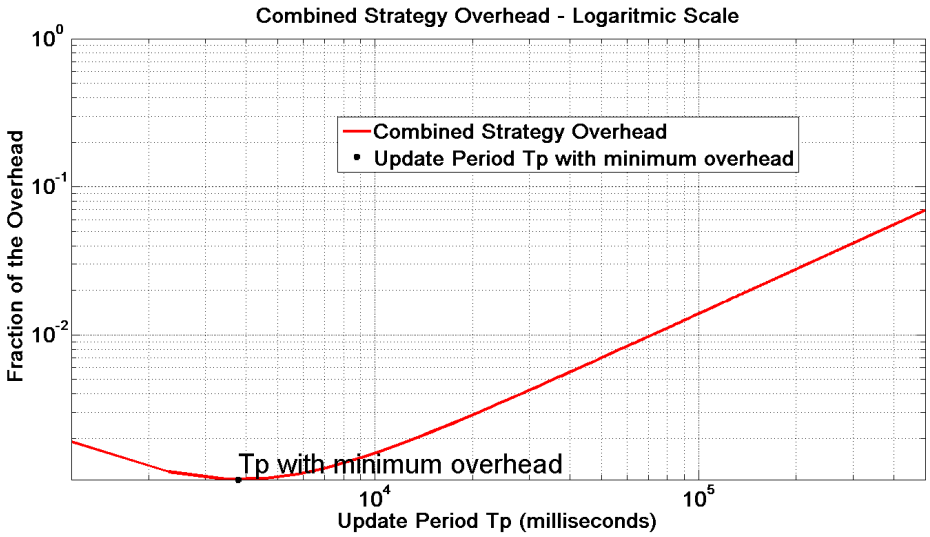

Figure 6.4: Minimum Overhead of the Combined Strategy.

Clearly, the value of T_p with minimum overhead is inversely proportional to the failure rate λ_f, as shown in Equation 6.6. If λ_f is low, T_p is high, i.e. many transitions are executed before the next update operation, allowing for prioritizing the Retry Strategy over the Redundancy Strategy, hoping that failures are unlikely to occur in the future.

In contrast, if λ_f is high, T_p becomes low and a few transitions are executed before the next update operation, allowing for prioritizing the Redundancy Strategy over the Retry Strategy since failures are likely to occur in the future.

6.7 Impact of Extreme High Load on the Performance of the Combined Strategy

We aim to evaluate the impact of the load of controllers on the performance of the Combined strategy when controllers are overloaded. We ask the following question: Given the average failure rate of a node in the P2P overlay networks λ_{P2P} and the processing time for updating the database p, how does the high load of controllers affect the performance of the Combined strategy ?

In order to evaluate the impact of the Combined strategy on the crawling performance when controllers are overloaded, we measure the average delays for sending and receiving back-up messages in the P2P Crawling System during the crawling phase.

Let t_{Send} and $t_{Receive}$ be the processing delays for sending and receiving a back-up message respectively. Based on our measurements, the average processing time for sending a message t_{Send} is in the order of $10^{-}3$ milliseconds, while the average processing time for receiving a message $t_{Receive}$ is in the order of $10^{-}4$ milliseconds, when controllers are underloaded.

Additionally, let δ be a parameter describing the load of controllers. The factor $1/(1-\delta)$ describes the factor by which the performance of controllers is degraded where $0 \leq \delta \leq 1$. Notice that a very small value of δ means that the controllers are underloaded, while the controllers are considered highly overloaded when δ is very close to 1.

In order to include the impact of the delay resulting from the sending and receiving back-up messages on the crawling performance when controllers are overloaded, we assume that t_{Send} and $t_{Receive}$ are directly proportional to the factor $1/(1-\delta)$. We add t_{Send} and $t_{Receive}$ to the processing time p and we multiply their sum by the factor $1/(1-\delta)$ in the overhead introduced by the Combined data-recovery strategy in Equation 6.5, as follows:

$$Overhead_{CombinedStrategy-HighLoad} = \frac{\lambda_f.T_p}{2} + \frac{2.c}{T_p} + \frac{(p + t_{Send} + t_{Receive})}{t_t.(1-\delta)} \tag{6.7}$$

Notice that the high load of controllers δ when sending, receiving and processing back-up messages may also increase the total overhead of the Combined Strategy. However, this possibility is ignored in the following.

We compare the combined data-recovery overhead in the P2P Crawling System during the crawling phase when the controllers are highly overloaded for different values of δ, as shown in Fig. 6.5.

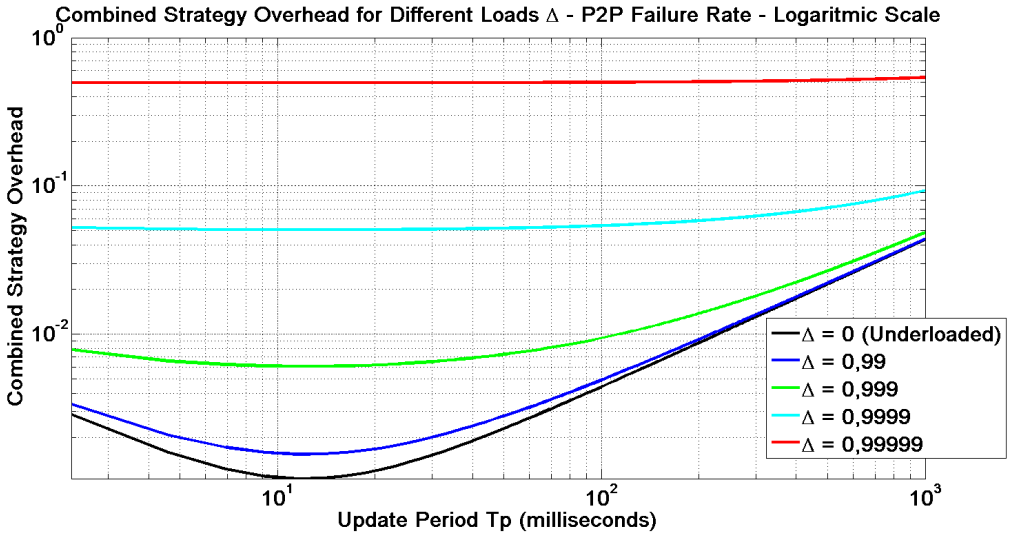

Figure 6.5: Comparison of the combined data-recovery overhead in the P2P Crawling System for different values of δ.

Fig. 6.5 shows that the performance of the Combined strategy significantly decreases as the load on controllers increases. However, Fig. 6.5 indicates that the Combined strategy converges towards the Redundancy strategy as the load on controllers increases. For extremely large values of δ, i.e. controllers are extremely overloaded with $\delta \geq 0.9999$, the Combined strategy and the Redundancy strategy are comparable. We conclude that the Combined strategy is appealing for crawling RIA in a faulty environment when controllers are not extremely overloaded.

6.8 Comparison of the Data Recovery Mechanisms

Analytical results show a high delay related to the Retry and the Combined strategies compared to the Redundancy strategy when controllers are underloaded. This is due to the re-execution of the same task when a controller fails while the Redundancy strategy allows for a faster recovery with an insignificant and constant overhead, i.e. the delay introduced by the Redundancy strategy remains insignificant and constant as the number of failing controllers increases. However, the Redundancy Strategy may not remain efficient in the case when controllers are overloaded. This is due to the high processing time introduced by the back-up update operations. In fact, one major drawback of the Redundancy strategy is that controllers may become a bottleneck since an update operation is required for each newly executed transition. The Combined strategy overcomes this issue by periodically copying the executed transitions a controller maintains so that if the controller fails, a portion of the executed transitions remains available in the back-up controller, which allows for significantly reducing the number of updates performed, thereby reducing the impact of the possible bottleneck on the crawling performance when the controllers are relatively overloaded. This makes the Combined strategy a good choice for crawling RIAs in a faulty environment when controllers are relatively overloaded. However, when the controllers are extremely overloaded, the Combined strategy is not as better as the Redundancy strategy.

Chapter 7

Conclusion and Future Directions

7.1 Conclusion

In this research, we addressed the scalability and resilience problems when crawling RIAs in a distributed environment. First, we proposed a scalable P2P crawling system for crawling RIAs [56]. Our approach is to partition the RIA model that results from the crawling over several storage devices called controllers in a peer-to-peer (P2P) network, and a set of crawlers is associated with each controller, which allows for scalability. Moreover, the responsibilities for the RIA states were distributed among these controllers in the underlying P2P network, where each controller maintains a portion of the application model, thereby avoiding a single point of failure. We also defined different knowledge sharing schemes for efficiently crawling RIAs in the P2P network: the Global Knowledge, Reset-Only, Local-Knowledge, Shared-Knowledge, Original Forward Exploration, Locally Optimized Forward Exploration and Globally Optimized Forward Exploration sharing knowledge schemes.

We conducted a simulation study to compare the efficiency of the sharing schemes by crawling real large-scale RIAs using the proposed P2P crawling system [56]. Simulation results showed that the Shared-Knowledge scheme, despite its simplicity, is efficient and scalable compared to the Reset-Only and Local-Knowledge schemes which did not scale

with the number of controllers. Additionally, the Globally Optimized Forward Exploration strategy was near optimal compared to the ideal setting and outperformed the Reset-Only, the Local-Knowledge, the Shared-Knowledge, the Original Forward Exploration and the Locally Optimized Forward Exploration schemes. We conclude that the Forward Exploration scheme is a good choice for general purpose crawling in a decentralized P2P environment, followed by the Shared-Knowledge scheme.

Moreover, we integrated a fault-tolerant scheme to the scalable P2P RIA crawling system assuming that crawlers and controllers are vulnerable to fail-stop failures, and we modified the system architecture accordingly, allowing the proposed P2P RIA crawling system to resume crawling RIAs despite failures. Additionally, we introduced three data recovery mechanisms for crawling RIAs in an unreliable environment: The Retry, the Redundancy and the Combined mechanisms and we showed how to adapt the recovery mechanisms to the existing crawling strategies. We evaluated the performance of the recovery mechanisms and their impact on the crawling performance through analytical reasoning. Our analysis showed that the Redundancy strategy with parallel back-up update operations is optimal and significantly outperforms the Retry strategy when controllers are underloaded. However, the Redundancy strategy was vulnerable to produce bottlenecks on controllers due to the update of every single transition. In the case that controllers are relatively overloaded, the Combined strategy outperformed the Redundancy strategy by periodically copying the executed transitions a controller maintains rather than copying every executed transition, so that if the controller fails, a portion of the executed transitions remains available in the back-up controller, i.e. by prioritizing the Retry Strategy over the Redundancy Strategy, which allows for significantly reducing the number of updates performed compared to the Redundancy strategy. Consequently, the impact of possible bottlenecks on the crawling performance is significantly reduced. However, our analysis showed that the Combined strategy is not as good as the Redundancy strategy when controllers are extremely loaded.

7.2 Contributions

The contributions of the thesis apply to the problem of crawling Rich Internet Applications using concurrent processing in a system of distributed computers. The main contributions are the following:

- Scalability: A scalable system where a high number of crawlers may be associated with each controller, without having a central bottleneck that may result from a single database simultaneously accessed by all crawlers.
- Partial Resilience: The distribution of responsibilities among multiple controllers in the underlying P2P network, where each controller maintains a portion of the application model, thereby avoiding a single point of failure, which allows partial resilience.
- Knowledge Sharing: Defining and comparing the performance of different knowledge sharing schemes for efficiently crawling RIAs in the P2P network:
  - Global Knowledge scheme
  - Reset-Only scheme
  - Local-Knowledge scheme
  - Shared-Knowledge scheme
  - Original Forward Exploration scheme
  - Locally Optimized Forward Exploration scheme
  - Globally Optimized Forward Exploration scheme
- Termination Detection: Defining a distributed termination detection algorithm for crawling RIAs in a P2P network.

- Fault Tolerance: Defining a fault-tolerant RIA crawling system that is able to achieve the crawling task despite node failures.
- Data-Recovery of RIAs: Defining and comparing different Data Recovery mechanisms for crawling RIAs in a faulty environment:
  - Retry Data Recovery mechanism
  - Redundancy Data Recovery mechanism
  - Combined Data Recovery mechanism

7.3 Future Directions

Some future directions of this research are:

- Applying other crawling strategies besides the greedy strategy to the RIA crawling system, such as the menu model, the component-based model and the probabilistic strategy to the fault-tolerant RIA crawling system.
- Dynamic Adaptive Combined Strategy: In this thesis, the proposed Combined Strategy consisted of periodically copying the executed transitions a controller maintains rather than copying every executed transition with the aim of avoiding the possible bottleneck on back-up controllers that may occur when the Redundancy strategy is applied. The combined strategy could be improved by periodically evaluating the load of crawlers and controllers, and dynamically prioritizing the Retry strategy or the Redundancy strategy accordingly, i.e. if the crawlers are most likely to remain overloaded compared to the controllers, the system automatically prioritizes the Redundancy strategy over the Retry strategy, which allows for moving the future load from crawlers to controllers. On the other hand, if the controllers are most likely

to remain overloaded compared to the crawlers, the system automatically prioritizes the Retry strategy over the Redundancy strategy.

- Evaluating the impact of the data recovery strategies on the crawling performance when controllers are overloaded through simulation studies.

References

[1] Agarwal A., Koppula H. S., Leela K. P., Chitrapura K. P., Garg S., GM P. K., Haty C. Roy A., and Sasturkar A. Url normalization for de-duplication of web pages. In *Proceedings of the 18th International Conference on Information and knowledge management, ACM CIKM 09, New York, NY, USA*, pages 1987–1990, 2009.

[2] Avizienis A. The n-version approach to fault-tolerant software. In *IEEE Transactions on Software Engineering, Piscataway, NJ, USA*, volume 11, pages 1491–1501, December 1985.

[3] Bernstein P. A., Hadzilacos V., and Goodman N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, January 1987.

[4] Binzenhofer A., Kunzmann G., and Henjes R. A scalable algorithm to monitor chord-based p2p systems at runtime. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium, IEEE IPDPS 06, Rhodes Island, Greece*, April 2006.

[5] Carzanig A. and Rutheford M. *SSim, a simple Discrete-event Simulation Library.* University of Colorado, Technical Report, http://www.inf.usi.ch/carzaniga/ssim/index.html, 2003.

[6] Crainiceanu A., Linga P., Machanavajjhala A., Gehrke J., and Shanmugasundaram J. P-ring: an efficient and robust p2p range index structure. In *Proceedings of International Conference on Management Of Data, ACM SIGMOD 07, New York, NY, USA*, pages 223–234, 2007.

[7] Dasgupta A., Kumar R., and Sasturkar A. De-duping urls via rewrite rules. In *Proceedings of the 14th International Conference on Knowledge discovery and data mining, ACM SIGKDD 08, New York, NY, USA*, pages 186–194, 2008.

[8] Fiat A. and Saia J. Censorship resistant peer-to-peer content addressable networks. In *Proceedings of the thirteenth annual symposium on Discrete algorithms, ACM SODA 02, Philadelphia, PA, USA*, pages 94–103, 2002.

[9] Mesbah A., Van Deursen A., and Lenselink S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. In *ACM Transactions on the Web (TWEB), New York, NY, USA*, volume 6, March 2012.

[10] Moosavi A. Component-based crawling of complex rich internet applications. Master's thesis, School of Information Technology and Engineering, Faculty of Engineering, University of Ottawa, Ottawa, Ontario, Canada, 2014.

[11] Nascimento M. A. Peer-to-peer: Harnessing the power of disruptive technologies. In *ACM SIGMOD Record, New York, NY, USA*, pages 57–58, June 2003.

[12] Shukri A., Noor M., and Deris M. M. Distributed dynamic failure detection. In *Journal of Software*, volume 9, pages 1342–1347, May 2014.

[13] Singh A., Srivatsa M., Liu L., and Miller T. Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web. In *SIGIR 03 Workshop on Distributed Information Retrieval, Toronto, Canada*, volume 2924, pages 126–142, August 2003.

[14] Balasubramanian B. and Garg V. K. Fused data structures for handling multiple faults in distributed systems. In *Proceedings of the 31st International Conference on Distributed Computing Systems, IEEE ICDCS 11, Minneapolis, Minnesota, USA*, pages 677–688, June 2011.

[15] Bamba B., Liu L., Caverlee J., Padliya V., Srivatsa M., Bansal T., Palekar M., Patrao J., Li S., and Singh A. Dsphere: A source-centric approach to crawling, indexing and searching the world wide web. In *Proceedings of the IEEE 23rd International Conference on Data Engineering, Istanbul, Turkey*, pages 1515–1516, April 2007.

[16] Nazir B. and Khan T. Fault tolerant job scheduling in computational grid. In *IEEE International Conference on Emerging Technologies, Peshawar, Pakistan*, pages 708–713, November 2006.

[17] Pinkerton B. Finding what people want: Experiences with the web crawler. In *Proceedings of the 2nd International Conference on World Wide Web, ACM WWW 94,Geneva, Switzerland*, May 1994.

[18] Rutherford D. B. *What do you mean - It is fail-safe.* Local transit, Vancouver, British Columbia, Canada, 1992.

[19] Benjamin C. A strategy for efficient crawling of rich internet applications. Master's thesis, School of Information Technology and Engineering, Faculty of Engineering, University of Ottawa, Ottawa, Ontario, Canada, 2010.

[20] Cooper C. and Frieze A. Crawling on simple models of web graphs. In *Internet Mathematics*, volume 1, pages 57–90, 2003.

[21] Duda C., Frey G., Kossmann D., Matter R., and Zhou C. Ajax crawl: Making ajax applications searchable. In *Proceedings of the International Conference on Data Engineering, IEEE ICDE 09, Shanghai, China*, pages 78–89, March 2009.

[22] Gartner F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. In *ACM Computing Surveys, CSUR 99, New York, NY, USA*, volume 31, pages 1–26, March 1999.

[23] Olston C. and Najork M. Web crawling. In *Foundations and Trends in Information Retrieval, Hanover, MA, USA*, volume 4, pages 175–246, March 2010.

[24] Olston C. and Pandey S. Recrawl scheduling based on information longevity. In *Proceedings of the 17th International Conference on World Wide Web, ACM WWW 08, New York, NY, USA*, pages 437–446, 2008.

[25] Chandra T. D. and Toueg S. Unreliable failure detectors for reliable distributed systems. In *Journal of the ACM, JACM 96, New York, NY, USA*, volume 43, pages 225–267, March 1996.

[26] Karger D., Lehman E., Leighton F., Levine M., Lewin D., and Panigrahy R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual Symposium on Theory of Computing, ACM STOC 97, New York, NY, USA*, pages 654–663, May 1997.

[27] Kavila S. D., Raju G. S. V. P., Satapathy S. C., Machiraju A., Kinnera G. V. L., and Rasly K. A survey on fault management techniques in distributed computing. In *Proceedings of the International Conference on Frontiers of Intelligent Computing: Theory and Applications, FICTA 13, Bhubaneswar, Odisa, India*, volume 199, pages 593–602, 2013.

[28] Liben-Nowell D., Balakrishnan H., and Karger D. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st Symposium on Principles of Distributed Computing, ACM PODC 02, New York, NY, USA*, pages 233–242, July 2002.

[29] Oppenheimer D., Ganapathi A., and Patterson D. A. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th International Conference*

on USENIX Symposium on Internet Technologies and Systems, ACM USITS 03, Berkeley, CA, USA, volume 4, pages 1–1, 2003.

[30] Tang D., Iyer R. K., and Subramani S. S. Failure analysis and modeling of a vax cluster system. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing, IEEE FTCS 20, Digest of Papers, Newcastle Upon Tyne, UK*, pages 244–251, June 1990.

[31] Le Hgaret P. et al. Document object model (dom). http://www.w3.org/DOM/, January 2005.

[32] Cristian F. A rigorous approach to fault-tolerant programming. In *IEEE Transactions on Software Engineering*, volume 11, pages 23–31, January 1985.

[33] Dabek F., Brunskill E., Kaashoek M. F., Karger D., Morris R., Stoica I., and Balakrishnan H. Building peer-to-peer systems with chord, a distributed lookup service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, IEEE HOTOS 01, Washington, DC, USA*, pages 81–86, May 2001.

[34] Coffman E. G., Liu Z., and Weber R. R. Optimal robot scheduling for web search engines. In *Journal of Scheduling.*, volume 1, pages 15–29, 1998.

[35] Plaxton C. G., Rajaraman R., and Richa A. W. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ninth Annual Symposium on Parallel Algorithms and Architectures, ACM SPAA 97, New York, NY, USA*, pages 311–320, June 1997.

[36] Abawajy J. H. Fault-tolerant scheduling policy for grid computing systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium, IEEE IPDPS 04*, April 2004.

[37] Rasti Amir H., Stutzbach D., and Rejaie R. On the long-term evolution of the two-tier gnutella overlay. In *Proceedings of the 25th IEEE International Conference on Computer Communications, IEEE INFOCOM, Barcelona, Spain*, pages 1–6, May 2006.

[38] Koren I. and Krishna C. *Fault Tolerant Systems, 1st Edition*. Elsevier Science Inc, March 2007.

[39] Stoica I., Morris R., Karger D., Frans Kaashoek M., and Balakrishnan H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the International Conference on Applications, technologies, architectures, and protocols for computer communications, ACM SIGCOMM 01, New York, NY, USA*, volume 31, pages 149–160, October 2001.

[40] Cho J. and Garcia-Molina H. Parallel crawlers. In *Proceedings of the 11th International Conference on World Wide Web, ACM WWW 02, New York, NY, USA*, volume 2, pages 124–135, 2002.

[41] Cho J. and Garcia-Molina H. Effective page refresh policies for web crawlers. In *ACM Transactions on Database Systems, ACM TODS 03, New York, NY, USA*, volume 28, pages 390–426, December 2003.

[42] Cho J., Garcia-Molina H., and Page L. Efficient crawling through url ordering. In *Proceedings of the 7th International Conference on World Wide Web, ACM WWW 98, Brisbane, Australia*, volume 30, pages 161–172, April 1998.

[43] Cho J. and Schonfeld U. Rankmass crawler: A crawler with high personalized pagerank coverage guarante. In *Proceedings of the 33rd International Conference on Very Large Data Bases, ACM VLDB 07*, pages 375–386, 2007.

[44] Chu J., Labonte K. S., and Levine B. N. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of ITCom, Scalability and Traffic Control in IP Networks*, 2002.

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

[46] Frey J., Tannenbaum T., Livny M., Foster I., and Tuecke S. Condor-g: A computation management agent for multi-institutional grids. In *Cluster Computing, Hingham, MA, USA*, volume 5, pages 237–246, July 2002.

[47] Garrett J. J. Ajax: A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, February 2005.

[48] Gray J. Why do computers stop and what can be done about it. In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[49] Gray J. A census of tandem system availability between 1985 and 1990. In *IEEE Transactions on Reliability*, volume 39, pages 409–418, October 1990.

[50] Madhavan J., Ko D., Kot L., Ganapathy V., Rasmussen A., and Halevy A. Google's deep-web crawl. In *Proceedings of the 34th International Conference on Very Large Data Bases, ACM VLDB Endowment, New York, NY, USA*, volume 1, pages 1241–1252, August 2008.

[51] Misra J. Detecting termination of distributed computations using markers. In *Proceedings of the second Annual Symposium On Principles of Distributed Computing, ACM PODC 83, New York, NY, USA*, volume 22, pages 290–294, 1983.

[52] Padliya V. J. and Liu L. Peercrawl: A decentralized peer-to-peer architecture for crawling the world wide web. In *Georgia Institute of Technology Technical Report*, 2006.

[53] Wu J. and Watts D. J. Small worlds: The dynamics of networks between order and randomness. In *ACM SIGMOD 02 Record, New York, NY, USA*, volume 31, pages 74–75, December 2002.

[54] Xu J., Kalbarczyk Z., and Iyer R. K. Networked windows nt system field failure data analysis. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 178–185, December 1999.

[55] Aguilera M. K., Chen W., and Toueg S. Failure detection and consensus in the crash recovery model. In *Distributed Computing*, volume 13, pages 99–125, April 2000.

[56] Ben Hafaiedh K., Von Bochmann G., Jourdan G. V., and Onut I. V. A scalable peer-to-peer ria crawling system with partial knowledge. In *Proceedings of the 2nd International Conference on Networked Systems, NETYS 14, Marrakesh, Morocco*, pages 185–199, May 2014.

[57] Benjamin K., Von Bochmann G., Dincturk M. E., Jourdan G. V., and Onut I. V. Some modeling challenges when testing rich internet applications for security. In *Proceedings of the 1st International workshop on modeling and detection of vulnerabilities, MDV 10, Paris, France*, April 2010.

[58] Benjamin K., Von Bochmann G., Dincturk M. E., Jourdan G. V., and Onut I. V. A strategy for efficient crawling of rich internet applications. In *Proceedings of the 11th International Conference on Web engineering, ICWE 11, Paphos, Cyprus*, July 2011.

[59] Bharat K. and Broder A. Mirror, mirror on the web: A study of host pairs with replicated content. In *Proceedings of the 8th International Conference on World Wide Web, ACM WWW 99, New York, NY, USA*, volume 31, pages 1579–1590, May 1999.

[60] Gummadi P. K., Saroiu S., and Gribble S. D. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. In *Proceesings of the Computer Communication Review, ACM SIGCOMM 02, New York, NY, USA*, volume 32, pages 82–82, January 2002.

[61] Iyer R. K., Rossetti D. J., and Hsueh M. C. Measurement and modeling of computer reliability as affected by system activity. In *ACM Transactions on Computer Systems, ACM TOCS 86, New York, NY, USA*, volume 4, pages 214–237, August 1986.

[62] Sahoo R. K., Sivasubramaniam A., Squillante M. S., and Zhang Y. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings of the International Conference on Dependable Systems and Networks, IEEE DSN 04*, pages 772–781, July 2004.

[63] Zhu K., Xu Z., Wang X., and Zhao Y. A full distributed web crawler based on structured network. In *Lecture Notes in Computer Science*, volume 4993, pages 478–483, 2008.

[64] Barbosa L. and Freire J. An adaptive crawler for locating hidden-web entry points. In *Proceedings of the 16th International Conference on World Wide Web, ACM WWW 07, New York, NY, USA*, pages 441–450, 2007.

[65] Devroye L. Non-uniform random variate generation. In *Springer-Verlag, New York, NY, USA*, pages 392–401, 1986.

[66] Gravano L., Garcia-Molina H., and Tomasic A. The effectiveness of gloss for the text database discovery problem. In *Proceedings of the International Conference on*

Management of Data, ACM SIGMOD 94, New York, NY, USA, volume 23, pages 126–137, June 1994.

[67] Lamport L., Shostak R., and Pease M. The byzantine generals problem. In *ACM Transactions on Programming Languages and Systems, ACM TOPLAS 82, New York, NY, USA*, volume 4, pages 382–401, July 1982.

[68] Page L., Brin S., Motwani R., and Winograd T. The pagerank citation ranking: Bringing order to the web. Standford University Technical Report, Stanford, United States, December 1997.

[69] Affaan M. and Ansari M. A. Distributed fault management for computational grids. In *Proceedings of the Fifth International Conference on Grid and Cooperative Computing, IEEE GCC 06, Hunan, China*, pages 363–368, October 2006.

[70] Castro M. and Liskov B. Practical byzantine fault tolerance and proactive recovery. In *ACM Transactions on Computer Systems, ACM TOCS 02, New York, NY, USA*, volume 20, pages 398–461, November 2002.

[71] Hersovici M., Jacovi M., Maarek Y. S., Pelleg D., Shtalhaim M., and Ur S. The shark-search algorithm - an application: Tailored web site mapping. In *Proceedings of the 7th International Conference on World Wide Web, ACM WWW 98, Amsterdam, The Netherlands*, volume 30, pages 317–326, April 1998.

[72] Kalyanakrishnam M., Kalbarczyk Z., and Iyer R. Failure data analysis of a lan of windows nt based computers. In *Proceedings of the IEEE 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Switzerland*, pages 178–187, October 1999.

[73] Koster M. A standard for robot exclusion. http://www.robotstxt.org/orig.html., 1994.

[74] Mirtaheri S. M., Zou D., Von Bochmann G., Jourdan G. V., and Onut I. V. Dist-ria crawler: A distributed crawler for rich internet applications. In *Proceedings of the 8TH International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 13, Compiegne, France*, October 2013.

[75] Mirtaheri S. M., Von Bochmann G., Jourdan G. V., and Onut I. V. A greedy distributed crawler for rich internet applications. In *Proceedings of the 2nd International Conference on Networked Systems, NETYS 2014, Marrakesh, Morocco*, May 2014.

[76] Mirtaheri S. M., Von Bochmann G., Jourdan G. V., and Onut I. V. Pdist-ria crawler: A peer-to-peer distributed crawler for rich internet applications. In *Proceedings of the 15th International Conference of WISE, Lecture Notes in Computer Science, Thessaloniki, Greece*, volume 8787, pages 365–380, October 2014.

[77] Najork M. and Wiener J. L. Breadth-first crawling yields high-quality pages. In *Proceedings of the 10th International Conference on World Wide Web, ACM WWW 01, New York, NY, USA*, pages 114–118, 2001.

[78] Waldman M., Rubin A. D., and Cranor L. F. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th International Conference on USENIX Security Symposium, ACM SSYM 00, Berkeley, CA, USA*, volume 9, pages 5–5, 2000.

[79] Babar N. and Taimoor K. Fault tolerant job scheduling in computational grid. In *Proceedings of the 2nd International Conference on Emerging Technologies, IEEE ICET 06, Peshawar, Pakistan*, pages 708–713, November 2006.

[80] Budhiraja N., Marzullo K., Schneider F. B., and Toueg S. *The primary-backup approach.* Distributed systems, 2nd Edition, ACM Press, Addison-Wesley Publishing Co. New York, NY, USA, 1993.

[81] Hussain N., Ansari M. A., Yasin M. M., Rauf A., and Haider S. Fault tolerance using parallel shadow image servers (psis) in grid based computing environment. In *Proceedings of the International Conference on Emerging Technologies, IEEE ICET 06, Peshawar, Pakistan*, pages 703–707, November 2006.

[82] Boldi P., Codenotti B., Santini M., and Vigna S. Ubicrawler: a scalable fully distributed web crawler. In *Journal Software, Practice and Experience, John Wiley and Sons, Inc. New York, NY, USA*, volume 34, pages 711–726, July 2004.

[83] Callan J. P., Lu Z., and Croft W. B. Searching distributed collections with inference networks. In *Proceedings of the 18th Annual International Conference on Research and Development in Information Retrieval, ACM SIGIR 95, New York, NY, USA*, pages 21–28, 1995.

[84] Druschel P. and Rowstron A. Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems, IEEE HOTOS 01*, pages 65–70, 2001.

[85] Egwutuoha I. P., Levy D., Selic B., and Chen S. A survey of fault tolerance mechanisms and checkpoint-restart implementations for high performance computing systems. In *Journal of Supercomputing, Hingham, MA, USA*, volume 65, pages 1302–1326, September 2013.

[86] Gummadi K. P., Dunn R. J., Saroiu S., Gribble S. D., Levy H. M., and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM SOSP 03, New York, NY, USA*, volume 37, pages 314–329, December 2003.

[87] Jalote P. *Fault Tolerance in Distributed Systems.* Prentice Hall, 1st edition, 1994.

[88] Kihlstrom K. P., Moser L. E., and Melliar-Smith P. M. Solving consensus in a byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems, OPODIS 97*, 1997.

[89] Latchoumy P. and Sheik Abdul Khader P. Survey on fault tolerance in grid computing. In *International Journal of Computer Science and Engineering Survey, IJCSES 11*, volume 2, November 2011.

[90] Stelling P., Foster I., Kesselman C., and Lee C. A fault detection service for wide area distributed computations. In *Proceedings of the seventh International Symposium on High Performance Distributed Computing, IEEE HPDC 98, Chicago, IL, USA*, pages 268–278, July 1998.

[91] Lv Q., Cao P., Cohen E., Li K., and Shenker S. Search and replication in unstructured peer-to-peer networks. In *Proceedings of 16th International Conference on Supercomputing, ACM ICS 02, New York, NY, USA*, pages 84–95, June 2002.

[92] Bhagwan R., Savage S., and Voelker G. M. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems, IPTPS 03*, February 2003.

[93] Matter R. Ajax crawl: Making ajax applications searchable. Master's thesis, Swiss Federal Institute of Technology in Zurich, ETH Zurich, 2008. http://e-collection.library.ethz.ch/eserv/eth:30709/eth-30709-01.pdf.

[94] Schollmeier R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the IEEE first International Conference on Peer-to-Peer Computing, Linkoping, Sweden*, pages 101–102, August 2001.

[95] Abiteboul S., Preda M., and Cobena G. Adaptive on-line page importance computation. In *Proceedings of the 12th International Conference on World Wide Web, ACM WWW 03, New York, NY, USA*, pages 280–290, 2003.

[96] Brin S. and Page L. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on World Wide Web, ACM WWW 07, Amsterdam, The Netherlands*, volume 30, pages 107–117, April 1998.

[97] Chien S., Dwork C., Kumar R., Simon D. R., and Sivakumar D. Link evolution: Analysis and algorithms. In *Proceedings of the International Conference on Distributed Systems Platforms (Middleware), ACM FIP 03, Heidelberg, Germany*, volume 1, pages 277–304, November 2003.

[98] Choudhary S. M-crawler: Crawling rich internet applications using menu meta-model. Master's thesis, School of Information Technology and Engineering, Faculty of Engineering, University of Ottawa, Ottawa, Ontario, Canada, 2012.

[99] Choudhary S., Dincturk M. E., Mirtaheri S. M., Moosavi A., Von Bochmann G., Jourdan G. V., and Onut I. V. Crawling rich internet applications: The state of the art. In *Conference of the Center for Advanced Studies on Collaborative Research*, pages 146–160, 2012.

[100] Hwang S. and Kesselman C. A flexible framework for fault tolerance in the grid. In *Journal of Grid Computing*, volume 1, pages 251–272, September 2003.

[101] Raghavan S. and Garcia-Molina H. Crawling the hidden web. In *Proceedings of the 27th International Conference on Very Large Data Bases, ACM VLDB 01, San Francisco, CA, USA*, pages 129–138, 2001.

[102] Ratnasamy S., Francis P., Handley M., and Karp R. Shenker S. A scalable content-addressable network. In *Proceedings of the International Conference on Applications,*

technologies, architectures, and protocols for computer communications, ACM SIGCOMM 01, New York, NY, USA, volume 31, pages 161–172, October 2001.

[103] Rhea S., Geels D., Roscoe T., and Kubiatowicz J. Handling churn in a dht. In *Proceedings of the International Conference on USENIX Annual Technical Conference, ACM ATEC 04, Berkeley, CA, USA*, pages 10–10, 2004.

[104] Saroiu S., Gummadi P. K., and Gribble S. D. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the International Conference on Multimedia Computing and Networking, MMCN 02, San Jose, CA, USA*, January 2002.

[105] Sen S. and Wang J. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the IEEE-ACM Transactions on Networking*, volume 12, pages 219–232, April 2004.

[106] Heath T., Martin R. P., and Nguyen T. D. Improving cluster availability using workstation validation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems, ACM SIGMETRICS 02, New York, NY, USA*, volume 30, pages 217–227, June 2002.

[107] Loo B. T., Cooper O., and Krishnamurthy S. *Distributed web crawling over DHTs.* UC Berkeley Technical Report, 2004.

[108] Rowstron A. I. T. and Druschel P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *FIP-ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, volume 22, pages 329–350, January 2001.

[109] Saridakis T. A system of patterns for fault tolerance. In *Proceedings of the European Conference on Pattern Languages of Programs, EuroPLoP 02, Kloster Irsee, Germany*, pages 535–582, July 2002.

[110] De Florio V. and Blondia C. A survey of linguistic structures for application-level fault tolerance. In *ACM Computing Surveys, ACM CSUR 08, New York, NY, USA*, volume 40, April 2008.

[111] Shkapenyuk V. and Suel T. Design and implementation of a high performance distributed web crawler. In *Proceedings of the 18th IEEE International Conference on Data Engineering, San Jose, CA, USA*, pages 357–368, March 2002.

[112] Aiello W., Chung F., and Lu L. Random evolution in massive graphs. In *Handbook of Massive Data Sets*, volume 4, pages 97–122, 2002.

[113] Dijkstra E. W. *A note on two problems in connexion with graphs.* Numerische Mathematik 1, 1959.

[114] Hoeffding W. Probability inequalities for sums of bounded random variables. In *Journal of the American Statistical Association*, volume 58, pages 13–30, March 1963.

[115] Li X., Misra J., and Plaxton C. G. Concurrent maintenance of rings. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, volume 19, pages 126–148, October 2006.

[116] Lin T. T. Y. and Siewiorek D. P. Error log analysis: Statistical modeling and heuristic trend analysis. In *IEEE Transactions on Reliability*, volume 39, pages 419–432, October 1990.

[117] Zhao B. Y., Huang L., Stribling J., Rhea S. C., Joseph A. D., and Kubiatowicz J. D. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE Journal on Selected Areas in Communications, Piscataway, NJ, USA*, volume 22, pages 41–53, September 2006.

[118] Baeza yates R. and Castillo C. Balancing volume, quality and freshness in web crawling. In *Soft Computing Systems - Design, Management and Applications, Santiago, Chile*, pages 565–572, 2002.

[119] Bar-Yossef Z., Keidar I., and Schonfeld U. Do not crawl in the dust: Different urls with similar text. In *Proceedings of the 16th International Conference on World Wide Web, ACM WWW 07, New York, NY, USA*, pages 111–120, 2007.

[120] Broder A. Z., Glassman S. C., Manasse M. S., and Zweig G. Syntactic clustering of the web. In *Proceedings of the 6th International Conference on World Wide Web, ACM WWW 97, Amsterdam, The Netherlands*, volume 29, pages 1157–1166, September 1997.

[121] Peng Z., He N., Jiang C., Li Z., Xu L., Li Y., and Ren Y. Graph-based ajax crawl: Mining data from rich internet applications. In *Proceedings of the International Conference on Computer Science and Electronic Engineering, IEEE ICCSEE 12, Hangzhou, China*, volume 3, pages 590–594, March 2012.