A Scalable P2P RIA Crawling System with Partial Knowledge

Khaled Ben Hafaiedh*, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut

EECS, University of Ottawa, Ottawa, Ontario, Canada
`hafaiedh.khaled@uottawa.ca`, `{bochmann,gvj}@eecs.uottawa.ca`,
`vioonut@ca.ibm.com`
`http://ssrg.site.uottawa.ca/`

Abstract. Rich Internet Applications are widely used as they are interactive and user friendly. Automated tools for crawling Rich Internet Applications have become needed for many reasons such as content indexing or testing for correctness and security. Due to the large size of RIAs, distributed crawling has been introduced to reduce the amount of time required for crawling. However, having one controller may result in a performance bottleneck resulting from a single database simultaneously accessed by many crawlers. It may also be vulnerable to complete data loss if a node failure occurs at the storage unit. We present a distributed decentralized scheme for crawling large-scale RIAs capable of partitioning the search space among several controllers in which the information is partially stored, which allows for fault tolerance and for the scalability of the system. Our results are significantly better than for non-distributed crawling, and outperforms the distributed crawling using one coordinator.

Key words: Rich Internet Applications, Web Crawling, Web Application Modeling, Graph Exploration, Distributed Crawling, P2P networks.

1 Introduction

As the web has evolved towards dynamic content, modern web technologies gave birth to interactive and more responsive applications, referred to as Rich Internet Applications [4], which combine client-side scripting with new features such as AJAX (Asynchronous JavaScript and XML) [6] [2], allowing the client to asynchronously modify the currently displayed page. Exploring a RIA is referred to as *event-based crawling.* Automated *event-based crawling* [3] automatically explores all events by traveling each of the possible user-interactions within the given page, where each page is represented by its Document Object Model (DOM) [12].

In the context of RIAs, a study [1] has been conducted in a centralized environment to store states within a single coordinator. This central hub has the

* hafaiedh.khaled@uottawa.ca, bochmann, gvj@eecs.uottawa.ca, vioonut@ca.ibm.com

responsibility of partitioning the task of crawling RIAs among multiple crawlers. We address the scalability and resilience problems when crawling RIAs by distributing states in a P2P network composed of multiple coordinators, where each coordinator maintains a list of states and is associated with a set of crawlers.

The proposed decentralized architecture for crawling RIAs is challenging for two reasons: (1) Crawlers may have to go through a path of ordered states before exploring a new transition. If the states are partitioned among multiple coordinators, it is unsuitable to communicate with all coordinators that are associated with the states in this path. (2) Traversing a long path before executing a new state is costly. Some coordination between the coordinators 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.

To our knowledge, crawling large-scale RIAs over P2P networks while minimizing the cost (number of event executions), where coordinators maintain a partial knowledge of the application model has not been investigated yet. We make the following contributions:

– The distribution of responsibilities for the states among multiple coordinators in the underlying P2P network, where each coordinator maintains a portion of the application model, and a high number of crawlers are associated with each coordinator, which allows for scalability.
– Defining and comparing different knowledge sharing schemes for efficiently crawling RIAs in the P2P network.

The rest of this paper is organized as follows: Section 2 gives an overview of the distributed RIA crawling. Section 3 introduces the distributed P2P Architecture for Crawling RIAs with partial knowledge. The decentralized distributed greedy strategy and the P2P crawling protocol are also described. Section 4 introduces different knowledge sharing schemes for efficiently crawling RIAs. Finally, Section 5 describes the results of our simulation study and compares the efficiency of our exploration mechanisms. A conclusion is provided in the end of the paper with some future directions for improvements.

2 Overview of the Distributed Decentralized RIA Crawling

2.1 Traditional Web Crawling

The typical interaction between client and server in a traditional web application consists of sending a request for a URL so that the corresponding web page is returned in response. Thus, each web page is identified by its URL. Crawling a traditional web application consists of finding all its URLs [10]. Improving both scalability and efficiency of crawling traditional web applications may be achieved by partitioning the task of the crawl among multiple crawlers in a distributed network. Distributed crawling allows each crawler to explore only a portion of the search space by contacting one or more units, which are responsible for storing and distributing the graph exploration task, referred to as the coordinator.

Different approaches have been used to concurrently crawl traditional web applications [7]. In a centralized environment, a single unit is responsible for storing a list of the newly discovered URLs and gives the instruction of loading each unexplored URL to an idle crawler [10]. An alternative has been proposed in a P2P [8] network in order to partition URLs among several coordinators by means of a Distributed Hash Table (DHT) [5] [14] where each coordinator maintains a portion of the application model so that there is no single point of failure.

2.2 RIA Crawling with one Single Crawler

In contrast to traditional web applications where each state represents a single URL, states represent the distinct pages within the same URL in a RIA model [3], while transitions illustrate the possible ways to move from one page to another. Consequently, a graph with a high number of states can be derived for each single URL in a RIA. Formally speaking, the task of crawling a RIA consists of finding all distinct states for each seed URL [3], where the initial state corresponds to the initial page that is reached from loading the seed URL. The triple $(SourceState, event, DestinationState)$ describes a transition in the RIA model, and *event* describes a possible user-interaction within a source state and leading to a destination state. The basic greedy strategy with a single crawler consists of exploring an event from the current state if there is any unexplored event. Otherwise, the crawler executes an unexplored event from another state by either performing a reset, i.e. returning to the initial state and retracing the steps that lead to this state [4], or by using a shortest path algorithm [13] to find the closest state with an unexecuted event without necessarily performing a reset.

2.3 Distributed Centralized RIA Crawling

A distributed centralized scheme [1] for crawling RIAs has been recently introduced with the aim of reducing the required amount of time to crawl RIAs, by allowing multiple crawlers to crawl a given RIA simultaneously. In this system, all states are maintained by a single entity, the coordinator. This entity is responsible for storing information about the new discovered states including the unexecuted events on each state. All crawlers are associated with the single coordinator. That is, each crawler may retrieve the required graph information by communicating with the single coordinator, and then executes a single unexecuted event from its current state if such an event exists, or may move to another state with some unexecuted events based on the information available in the database. A shortest path algorithm is performed by the coordinator to find the closest state with an unexecuted event of a given crawler.

However, maintaining the states within a single unit may be problematic for the following reasons: (1) Scalability: Preliminary analysis of experimental results [1] have shown that a coordinator can support up to only 20 crawlers without becoming overloaded. (2) Fault tolerance: A failure occurring within this unit may result in the entire loss of the graph under exploration.

2.4 Distributed Decentralized RIA Crawling

In this paper, we propose a P2P chordal ring structure [9] that is composed of multiple coordinators that are dispersed over the P2P network as shown in Fig.1. Moreover, a set of crawlers is associated with each coordinator, where crawlers and coordinators are independent processes running on different computers.

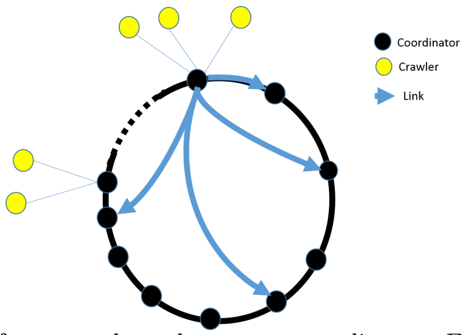

Fig. 1: Distribution of states and crawlers among coordinators: Each state is associated with one coordinator, and each crawler gets access to all coordinators through a single coordinator it is associated with.

3 Distributed P2P Architecture for Crawling RIAs with partial knowledge

3.1 The decentralized distributed greedy strategy:

In the P2P environment, states are partitioned among the coordinators. The coordinator responsible for storing the information about a state is contacted when a crawler reaches a new state. For each request, the coordinator returns in response a single event to execute on this state.

However, if there is no event to execute on the current state of a visiting crawler, the coordinator associated with this state may look for another state with an unexecuted event among all states it is responsible for. Notice that maintaining a possible path from the initial state to a target state within the coordinator is necessary in RIA crawling as coordinators must be able to tell each visiting crawler how to reach a particular state starting from the initial state.

3.2 Protocol description:

Data-Structures:

- **State**: This represents a state of the application and has the following variables:
  - Integer *stateID*: The identifier of this 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*: (*unexecuted*, *assigned*, *executed*):
    1. *unexecuted*: 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 *myCoordinator*: The address of the coordinator that is associated with this crawler.
- **Coordinator**: Coordinators are responsible for storing states and coordinating the crawling task. Each coordinator has the following variables:
  - Address *myAddress*: The address of the coordinator.
  - Set $< State >$ *myDiscoveredStates*: The discovered states that belong to this coordinator.
  - String *URL*: The seed URL to be loaded when a reset is performed.

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 represents the content of the message.

Message types: We classify the message type with respect to the $messageInformation$ included in each message:

- **<u>Sent from a crawler to a coordinator</u>**:
  - **StateInfo(State currentState)**: This is to inform the coordinator about the current state of the crawler. The message is addressed by key using the ID of the crawler's current state, allowing the coordinator for finding an event to execute.
  - **AckJob(Transition executedTransition)**: Upon receiving an acknowledgment, the coordinator updates the list of unexecuted 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 coordinator finds an unexecuted event, or the same message is received back by the coordinator that is associated with this crawler, leading to entering the termination phase [11].
- **<u>Sent from a coordinator to a crawler</u>**:
  - **Start((URL)**: Initially, each crawler establishes a session with its associated coordinator. The coordinator 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 execute in the scope of the coordinator.
    - The URL is NULL but the *path* consists of one single transition: There is an event to execute 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.

The P2P RIA crawl protocol: Initially, each crawler receives a *Start* message from the coordinator 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 coordinator to find a new event to be executed from this state. The coordinator returns in response an *ExecuteEvent*

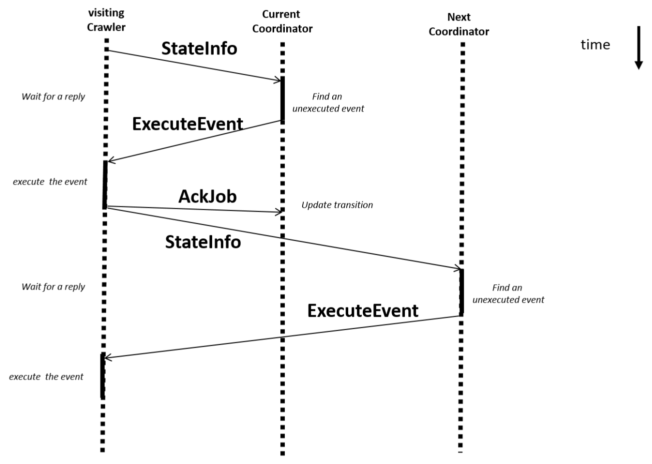

Fig. 2: Exchanged messages during the exploration phase.

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 coordinator 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 execute, it sends a *RequestJob* message to the coordinator it is associated with. This message is forwarded in the ring until a receiving coordinator finds a job or until the system enters a termination phase.

The following section defines the P2P RIA crawl protocol as executed by the coordinator and the crawler processes.

Coordinator process: UPON RECEIVING STATEINFO ($stateID, crawlerAddress, currentState$)

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

Coordinator process: UPON RECEIVING ACKJOB ($coordinatorAddress, crawlerAddress, executedTransition$)

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

Coordinator process: UPON RECEIVING REQUESTJOB ($coordinatorAddress, crawlerAddress, currentState$)

```
if $\exists\ s \in myDiscoveredStates\ \ and\ \ t \in s.transitions\ \ such\ \ that$
$t.status = unexecuted$ then
    $executionPath \leftarrow s.path + t$
    $t.status \leftarrow assigned$
    $path \leftarrow < URL, executionPath >$
    send $ExecuteEvent(crawlerAddress, myAddress, path)$
else
    forward $RequestJob\ \ to\ \ nextCoordinator$
end if
```

Crawler process: UPON RECEIVING START (URL)

```
$currentState \leftarrow load(URL)$
$currentState.path \leftarrow \emptyset$
for all $e \in currentState.transitions$ do
    $e.status \leftarrow unexecuted$
end for
send $StateInfo(stateID, myAddress, currentState)$
```

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

```
if $executionPath \neq \emptyset$ then
    if $URL \neq \emptyset$ then
        $currentState \leftarrow load(URL)$
        $currentState.path \leftarrow \emptyset$
    end if
    while executionPath.hasNext do
        $currentState \leftarrow process(executionPath.next)$
    end while
    send $AckJob(coordinatorAddress, myAddress, executionPath.last)$
    $currentState.path \leftarrow executionPath$
    for all $e \in currentState.transitions$ do
        $e.status \leftarrow unexecuted$
    end for
    send $StateInfo(stateID, myAddress, currentState)$
else
    send $RequestJob(nextCoordinator, myAddress, currentState)$
end if
```

4 Choosing the next event to explore from a different state

If no event can be executed from the current state of a given crawler, the coordinator that is maintaining this state may look for another state with some unexecuted events, depending on its available knowledge about the executed transitions. In a non-distributed environment, the crawler may have access to all the executed transitions, which allows for the use of a shortest path algorithm to find the closest state with unexecuted events, starting from the current state. However, in the distributed environment, sharing the knowledge about executed transitions may introduce a message overhead and increase the load on the coordinators. Therefore, there is a trade-off between the shared knowledge which improves the choice of the next event to execute, and the message overhead in the system. We introduce different approaches with the aim to reduce the overall time required to crawl RIAs.

Global-Knowledge: This is a theoretical model used for comparison purpose in which we assume that all coordinators have instant access to a globally shared information about the state of knowledge at each coordinator.

Reset-Only: A crawler can only move from a state to another by performing a Reset. In this case, the coordinator 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 an unexecuted event. Note that Reset-Only approach is a simple way for concurrently crawling RIAs.

Shortest path based on local knowledge: In this case, a visited coordinator may use its local transitions knowledge to find the shortest path from the crawler's current state leading to the closest state with an unexecuted event the coordinator is responsible for. Unlike the Reset-Only approach where only one path from a URL to the target state is stored, coordinators 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 the shortest path from the current state of the crawler to a state with an unexecuted event. Since the knowledge is partial, this may often lead to a reset path even though according to global knowledge, there exists a shorter direct path to the same state.

Shortest path based on shared knowledge: In this case, the transitions of the $StateInfo$ message are locally stored by intermediate coordinators when the message is forwarded through the DHT. Therefore, all forwarding coordinators in the chordal ring, i.e. intermediate coordinators receiving a message that is not designated to them, may also update their transitions knowledge before forwarding it to the next coordinator. This way, the transitions knowledge is significantly increased among coordinators with no message overhead.

Forward exploration: One drawback of the shortest path approach is the distribution of states among coordinators, i.e. each state is associated with a single coordinator in the network. Consequently, shortest paths can be only computed to states the visited coordinator is responsible for. An alternative consists of globally finding the optimal choice based on the breadth-first search.

The forward exploration search is initiated by the coordinator and 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 coordinators, and so on. The coordinator maintains two sets of states for each forward exploration query: The first, referred to as $statesToVisit$ is used to tell a receiving coordinator what are the states to visit next, while the second set, referred to as $visitedStates$ is used to prevent loops, i.e. states that have been already explored by the forward exploration. Additionally, each state to visit has a history path of ordered transitions from the root state to itself, called $intermediatePath$. This path is used to tell a visiting crawler how to reach a particular state with an unexecuted event from its current state.

Initially, when a visited coordinator receives a $StateInfo$ message from a crawler, it will pick an unexecuted event from the crawler's current state. If no unexecuted event is found, the coordinator picks all destination states of the transitions on that 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 coordinator 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 its appropriate coordinator. When a coordinator receives the forward exploration message, it checks if there is an unexecuted 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 coordinator that is responsible for that state.

In order to prevent different coordinators from visiting states that have already been visited and has no unexecuted events, coordinators may share during the forward exploration their knowledge about all executed transitions on these states, with other coordinators in the network. This allows for preventing the states with no unexecuted event that have been already explored, from getting visited again. The knowledge sharing of executed transitions is made by means of the $messageknowledge$ parameter included in each of the breadth-first search queries. The $messageknowledge$ is updated with the variable $transitionsKnowledge$ that is maintained by each coordinator upon receiving a $ForwardExploration$ message. Notice that all executed transitions must be acknowledged on each visited state before they are added to the $transitionsKnowledge$ variable, i.e. for each reached state, a coordinator can only jump over a visited state if and only if all transitions have been executed on that state and are known to a given coordinator.

The following figure describes the forward Exploration protocol, as executed by the coordinator process upon receiving a $ForwardExploration$ message. The line 4 to line 13 of $UponReceivingStateInfomessage$ are replaced by $UponReceivingForwardExplorationmessage$, allowing for initiating the Forward Exploration.

5 Evaluation

5.1 Simulation:

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 [15].

Coordinator process: UPON RECEIVING FORWARDEXPLORATION ($coordinatorAddress, crawlerAddress, currentState, statesToVisit, visitedStates, messageKnowledge$)

```
$transitionsKnowledge \leftarrow messageKnowledge + transitionsKnowledge$
if $\exists\ t \in currentState.transitions$ such that $t.status = unexecuted$ then
    $executionPath \leftarrow currentState.intermediatePath + t$
    $t.status \leftarrow assigned$
    $URL \leftarrow \emptyset$
    $path \leftarrow < URL, executionPath >$
    send $ExecuteEvent(crawlerAddress, myAddress, path)$
else
    if $\nexists\ t \in currentState.transitions$ such that $t.status = assigned$ then
        for all $t \in currentState.transitions$ do
            $transitionsKnowledge \leftarrow t + transitionsKnowledge$
        end for
    end if
    for all $t \in currentState.transitions$ such that $t.status = executed$ do
        if $t.destinationState \notin visitedStates$ then
            $t.destinationState.intermediatePath \leftarrow currentState.intermediatePath + t$
            $statesToVisit \leftarrow t.destinationState + statesToVisit$
        end if
    end for
    $noJumping \leftarrow$ false
    while $statesToVisit \neq \emptyset$ or $noJumping =$ false do
        $nextState \leftarrow statesToVisit.last$
        remove $statesToVisit.last$
        push $nextState$ to $visitedStates$
        if $nextState.transitionsKnowledge \neq \emptyset$ then
            for all $t \in nextState.transitionsKnowledge$ do
                if $t.destinationState \notin visitedStates$ then
                    $t.destinationState.intermediatePath \leftarrow nextState.intermediatePath + t$
                    $statesToVisit \leftarrow t.destinationState + statesToVisit$
                end if
            end for
        else
            $noJumping \leftarrow True$
            send $ForwardExploration(nextState.coordinatorAddress, crawlerAddress, nextState,$
            $statesToVisit, visitedStates, transitionsKnowledge)$
        end if
    end while
    if $statesToVisit = \emptyset$ and $noJumping =$ false then
        send $ExecuteEvent(crawlerAddress, myAddress, \emptyset)$
    end if
end if
```

5.2 Test-Applications:

The first real large-scale application we consider is the JQuery-based AJAX file browser [1] RIA, which is an AJAX-based file explorer. It has 4622 states and 429654 transitions with a reset cost of 12. The second and largest tested real large-scale application is the Bebop [2] RIA. It consists of 5082 states and 468971 transitions with a reset cost of 3. 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.

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

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

5.3 Results and Discussion:

This section presents the simulation results of crawling the test-applications using our simulation software. Based on our preliminary analysis of experimental results, a coordinator 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 coordinators 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 where all coordinators have instant access to a globally shared information about the state of knowledge at all coordinators. Notice that the Global Knowledge scheme is unrealistic in our setting and is used only for comparison.

The worst performance is obtained with the Reset-Only strategy, followed by the Shortest Path with Local Knowledge strategy. 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 Shortest Path with Local Knowledge strategy converges towards the Reset-Only strategy as the number of coordinators increases, which is due to the low partial knowledge available on each coordinator when the number of coordinators is high.

The Shortest Path based on shared Knowledge strategy comes in the second position and significantly outperforms both the Reset-Only and the Shortest Path based on Local Knowledge strategies as coordinators 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 Shortest Path based on Local Knowledge strategies and it slightly outperformed the Shortest Path based on shared Knowledge strategy. This is due to the fact that shortest paths can be only computed toward states the visited coordinator is responsible for, while the Forward Exploration strategy consists of finding globally the optimal choice based on the distributed breadth-first search.

We conclude that the Reset-Only, the Shortest Path based on Local Knowledge and the Shortest Path based on shared Knowledge strategies are bad strategies, while the Forward Exploration is the best choice for RIA crawling in a decentralized P2P environment.

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

5.4 Scalability of our approach:

The following section illustrates the expected performance when we have 20 crawlers per coordinator, assuming that a coordinator can support up to 20 crawlers without becoming a bottleneck. The behavior of the crawling system

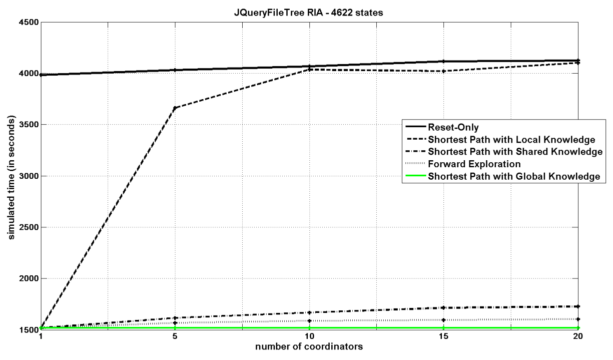

Fig. 3: Comparing different sharing schemes for crawling the JQuery file tree RIA.

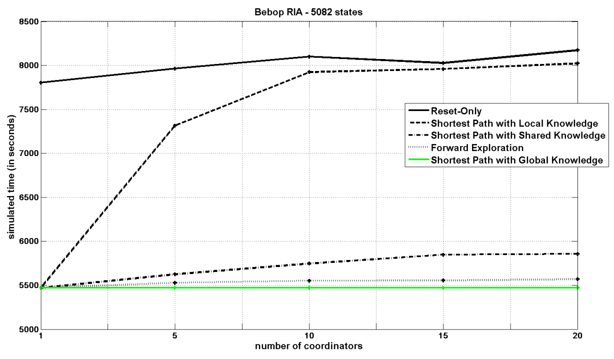

Fig. 4: Comparing different sharing schemes for crawling the Bebop RIA.

is similar across our test-applications. Therefore, we demonstrate the system scalability using the largest test-application we have, which is the Bebop RIA.

We consider the strategy with the best performance, which is the Forward Exploration strategy and we plot the simulated time (in seconds) for an increasing number of coordinators from 1 to 5, with 20 crawlers for each coordinator. Our simulation results show that the crawling time decreases near optimally as we increase the number of crawlers, which is consistent with our expectations. We conclude that our system scales with the number of crawlers when the coordinators are not overloaded.

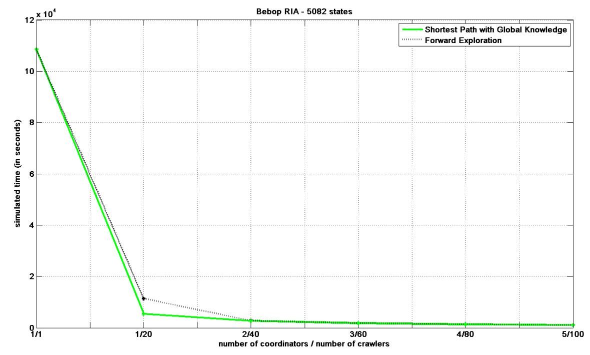

Fig. 5: System scalability for crawling the Bebop RIA.

6 Conclusion

We have presented a new distributed decentralized scheme for crawling large-scale RIAs by partitioning the search space among several controllers that share the information about the explored RIA. This allows for fault tolerance and scalability. Simulation results show that the Forward Exploration strategy is near optimal and outperforms the Reset-Only, the Shortest Path based on Local Knowledge and the Shortest Path based on Shared Knowledge strategies. This is due to its ability to globally find a shortest path, compared to all other strategies that are based on partial knowledge. This makes Forward Exploration a good choice for general purpose crawling in a decentralized P2P environment. However, there is still some room for improvement: We plan to study the system behavior when controllers become bottlenecks. We also plan to apply other crawling strategies besides the greedy strategy.

Acknowledgments. This work is supported in part by IBM and the Natural Science and Engineering Research Council of Canada.

References

1. Mirtaheri, S.M., Zou, D., Von Bochmann, G., Jourdan, G.V., 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 2013), France, Compiegne (2013)

2. Zhang, X., Wang, H.: AJAX Crawling Scheme Based on Document Object Model. In: Fourth International Conference on Computational and Information Sciences (ICCIS), pp. 1198–1201, China, Chongqing (2012)
3. Choudhary, S., Dincturk, M.E., Mirtaheri, S.M., Moosavi, A., Von Bochmann, G., Jourdan, G.V., Onut, I.V.: Crawling Rich Internet Applications: The State of the Art. In: Conference of the Center for Advanced Studies on Collaborative Research, pp. 146–160, Toronto, Markham (2012)
4. Benjamin, K., Von Bochmann, G., Dincturk, M.E., Jourdan, G.V., Onut, I.V.:Some Modeling Challenges when Testing Rich Internet Applications for Security. In: First International workshop on modeling and detection of vulnerabilities, France, Paris (2010)
5. Xiao, X., Zhang W.Z., Zhang, H.L., Fang B.X.: A Forwarding-Based Task Scheduling Algorithm for Distributed Web Crawling over DHTs. In: Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS), pp. 854–859, China, Shenzhen (2009)
6. Paulson, L.D.: Building rich web applications with Ajax. In: Computer publication of the IEEE Computer Society, vol. 38, pp. 14–17, (2005)
7. Cho, J., Garcia-Molina, H.: Parallel crawlers. In: Proceedings of the 11th international conference on World Wide Web, WWW, vol. 2, Hawaii, Honolulu (2002)
8. Schollmeier, R.: A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In: Proceedings of the IEEE 2001 International Conference on Peer-to-Peer Computing (P2P2001), Sweden, Linkping (2001)
9. Stoica, I., Morris, R., Karger, D., Frans Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proceedings of ACM SIGCOMM 2001, California, San Deigo (2001)
10. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: Proceedings of the 7th international conference on World Wide Web, WWW, vol. 7, Australia, Brisbane (1998)
11. Misra, J.: Detecting termination of distributed computations using markers. In: PODC'83, Proceedings of the second annual ACM symposium on Principles of distributed computing, vol. 22, pp. 290–294, NY, New York (1983)
12. Marini, J.: Document object model: processing structured documents. McGraw-Hill/Osborne, (2002)
13. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, pp. 269-271, (1959)
14. Loo, B.T., Owen, L., Krishnamurthy, C.S.: Distributed web crawling over DHTs. Technical Report, (2004)
15. Carzanig, A., Rutheford, M.: SSim, a simple Discrete-event Simulation Library, University of Colorado,Antonio.Carzaniga, Matt.Rutherford@usi.ch, http://www.inf.usi.ch/carzaniga/ssim/index.html, 2003.