Fault Tolerant P2P RIA Crawling

Khaled Ben Hafaiedh[(✉)], Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut

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

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. Distributed RIA crawling has been introduced with the aim of decreasing the crawling time due to the large size of RIAs. However, the current RIA crawling systems do not allow for tolerance to failures that occur in one of their components. In this paper, we address the resilience problem when crawling RIAs in a distributed environment and we introduce an efficient RIA crawling system that is fault tolerant. Our approach is to partition the RIA model that results from the crawling over several storage devices in a peer-to-peer (P2P) network. This makes the distributed data structure invulnerable to the single point of failure. We introduce three data recovery mechanisms for crawling RIAs in an unreliable environment: The Retry, the Redundancy and the Combined mechanisms. We evaluate the performance of the recovery mechanisms and their impact on the crawling performance through analytical reasoning.

Keywords: Fault tolerance · Data recovery · Rich internet applications · Web crawling · Distributed RIA crawling · P2P Networks

1 Introduction

In a traditional web application, each web page is identified by its URL. The basic function of a crawler in traditional web applications consists of downloading a given set of URLs, extracting all hyperlinks contained in the pages that follow from loading these URLs, and iteratively downloading the web pages that follow from these hyperlinks. Distributed traditional web crawling has been introduced to reduce the crawling time by distributing the work among multiple crawlers. In a 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, called controllers. In a centralized distributed system, the single controller is responsible for storing a list of the newly discovered URLs and gives the instruction of loading each unexplored URL to an idle crawler [6]. However, this system has a single point

© Springer International Publishing AG 2016
P.A. Abdulla and C. Delporte-Gallet (Eds.): NETYS 2016, LNCS 9944, pp. 32–47, 2016.
DOI: 10.1007/978-3-319-46140-3_3

of failure. P2P traditional crawling systems have been introduced to avoid the single point of failure and to continue the crawling task in case of a node failure, possibly at a reduced level, rather than failing completely. In this system, the URLs are partitioned over several controllers in which each controller is responsible for a set of URLs. Crawlers can find locally the identifiers of a database by mapping the hash of each discovered URL information using the Distributed Hash Table (DHT) [11], i.e. each URL is associated to a single controller in the DHT. P2P systems [4] have been used in traditional web crawling and are well-known for their decentralization and scalability.

As the web has evolved towards dynamic content, modern web technologies allowed for interactive and more responsive applications, called Rich Internet Applications (RIAs), which combine client-side scripting with new features such as AJAX (Asynchronous JavaScript and XML) [13]. 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, without having the user to wait for a response from the server. A RIA model [15] is composed of states and transitions, where states describe the distinct pages (DOM instances) and transitions illustrate the possible ways to move from one page to another by triggering a JavaScript event at the user interface.

The triple *(SourceState, event, DestinationState)* describes a transition in a RIA model where *event* refers to the triggered JavaScript event, *SourceState* refers to the page where the event is triggered and *DestinationState* refers to the next page that follows from triggering the event. The status of a RIA transition can take the following values: *free*, *assigned* or *executed*. A *free* transition refers to the initial status of the transition where the destination state is not known. An *assigned* transition refers to a transition that has been assigned to a crawler, and an *executed* transition is a transition that has been explored, i.e. the destination state is known. The task of crawling a RIA application consists of finding all the RIA states, starting from the original application URL. In order to ensure that all states have been identified, the crawler has to explore all transitions as it is not possible to know a priori whether the execution of a transition will lead to an already explored state or not [15]. This introduces new challenges to automate the crawling of RIAs as they result in a large number of states derived from each single URL. In RIA crawling, a Reset consists of returning to the original page by loading the RIA URL, called *SeedURL*. Efficiency of crawling a RIA is to find all RIA states as quickly as possible by minimizing the number of events executed and Resets [15]. The greedy strategy has been suggested by Peng et al. [16] for crawling RIAs due to its simplicity. The basic greedy strategy with a single crawler consists of exploring an event from the crawler's 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 [15], or by using a shortest path algorithm [1] to find the closest state with a free event without performing a Reset.

A distributed decentralized scheme for crawling large-scale RIAs was recently introduced by Ben Hafaiedh et al. [19]. It is based on the greedy strategy and consists of partitioning the search space among several controllers over a chordal ring [5]. In this system, RIA states are partitioned over several controllers in which each controller is responsible for only a subset of states. Crawlers can find locally the identifiers of a controller by searching for the controller responsible for a given state by means of the state identifier, i.e. by mapping the hash of each discovered state information using the Distributed Hash Table (DHT), which allows for avoiding the single point of failure. In this system, the RIA crawling performs as follows, as introduced in Fig. 1: The controller responsible for storing the information about a state (Current Controller) is contacted when a crawler reaches a new state by sending a search message, called *StateInfo* message. The *StateInfo* message consists of the information about the newly reached state along with all transitions on this state. Initially, the status of each transition is *free* and the destination state of the transition is not known by the controller. For each *StateInfo* message sent, the controller returns in response a new event to be executed on this state by sending a message, called *ExecuteEvent* message. 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 free event among all the states it is responsible for. Upon sending an *ExecuteEvent* message, the controller updates the status of the transition to *assigned*. The crawler then executes the assigned transition and sends the result of the execution back to the visited controller by means of an *AckJob* message. Upon receiving an *AckJob* message, the controller updates the destination state of the transition and changes the status of the transition to *executed*. The controller responsible for storing the information about the newly reached state (Next Controller) is then contacted by the crawler.

However, this system is not fault tolerant, i.e. lost states and transitions are not recovered when failures occur at the crawlers and controllers. In this paper, we address the resilience problem when using the proposed P2P RIA crawling system introduced by Ben Hafaiedh et al. [19] when controllers and crawlers are vulnerable to node failures, and we show how to make the P2P crawling system fault tolerant. Moreover, we introduce three recovery mechanisms for crawling RIAs in a faulty environment: The Retry, the Redundancy and the Combined mechanisms. Notice that the proposed RIA fault tolerance handling could be applied to any structured overlay network. However, the network recovery may depend on the structured overlay applied. The rest of this paper is organized as follows: The related work is described in Sect. 2. Section 3 introduces the fault tolerant P2P RIA crawling. Section 4 introduces the data recovery mechanisms. Section 5 evaluates the performance of the data recovery mechanisms and their impact on the crawling performance. A conclusion is provided in the end of the paper with some future directions for improvements.

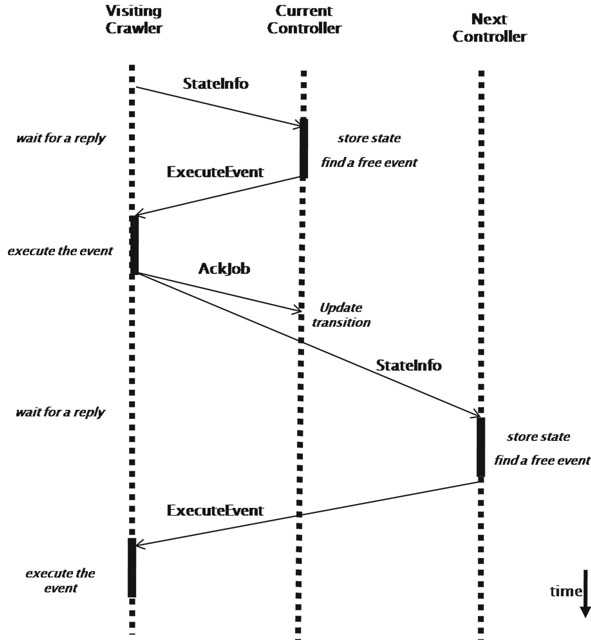

Fig. 1. The P2P RIA crawling introduced by Ben Hafaiedh et al. [19] during the exploration phase.

2 Related Work

In traditional web crawling, increasing the crawling throughput has been 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 may be achieved either through a central coordination process [9] that is responsible for coordinating the crawling task, or through a structured peer-to-peer network in order to assign different subsets of URLs to different crawlers [11]. Various decentralized architectures using DHTs have been proposed over different structured topologies in traditional web crawling such as Chord [5], CAN [2], Tapestry [12] and Pastry [3], which 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. This requires some resilience mechanisms on top of each of these architectures.

In RIA crawling, a distributed centralized crawling scheme [17] with the greedy strategy has been introduced, allowing each crawler to explore only a subset of a RIA simultaneously. In this system, all states are maintained by a single entity, called a controller, which is responsible for storing information about the new discovered states including the available events on each state. 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. The crawling is completed when all transitions have been explored. Maintaining the RIA states within a single unit in a faulty environment may be problematic since a failure occurring within the single controller will result in the loss of the entire graph under exploration.

A P2P RIA crawling system [18] 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. Crawlers are required to broadcast every newly executed transition to all other crawlers to find the shortest path from their current state to the next transition to be explored. Although this approach is appealing due to its simplicity, it is not fault tolerant. Moreover, it may introduce a high message overhead due to the sharing of transitions in case the number of crawlers is high.

A scalable P2P crawling system [19] using Chord [5] has been recently introduced to avoid the single point of failure. In this system, the P2P structure is composed of multiple controllers which are dispersed over a P2P network as shown in Fig. 2. In this system, 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 are independent processes running on different computers.

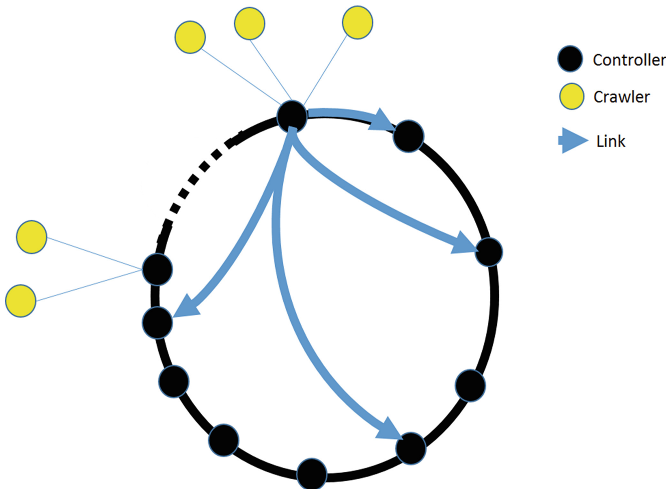

Fig. 2. Distribution of states and crawlers among controllers: each state is associated with one controller, and each crawler gets access to all controllers through a single controller it is associated with.

In this system, 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 overlay network is affected and some controllers become unreachable from other controllers. Since a P2P network is a continuously evolving system, it is required to continuously repair the overlay to ensure that the P2P structure remains connected and supports efficient lookups. The maintenance of the P2P network consists of maintaining its topology as controllers join and leave the network and repairing the overlay network when failures occur among controllers independently of the RIA crawling.

There are mainly two different approaches for maintaining a structured P2P network when failures occur: The active and the passive approaches. In the active approach, a node may choose to detect failures only when it actually needs to contact a neighbor. A node n_x may perform actively the repair operation upon detecting the disappearance of another node n_y in the network, i.e. the node n_x trying to reach n_y becomes aware that n_y is not responsive. Node n_x then runs a failure recovery protocol immediately to recover from the failure of n_y using $ID(n_y)$. One drawback of the active approach is that only the routing table of some neighboring nodes are updated when a node n_y fails. The passive approach solves theses inaccuracies by running periodically a repair protocol by all nodes to maintain their routing tables up-to-date, called the *idealization protocol* [8]. The *idealization protocol* runs periodically by every single controller in the network where each controller attempts to update its routing information. Liben-Nowell et al. [8] suggests to use the passive approach for detecting failures to avoid the risk that all of a node neighbors fail before it notices any of the failures. In this paper, we use the passive approach for maintaining the structured overlay network.

The structured P2P overlay network allows for partial resilience only, i.e. avoiding the single point of failure allows the non-faulty crawlers and controllers to resume the crawling task in case of a node failure, after the reestablishment of the overlay network, rather than failing completely. However, this system is not fully resilient since lost states and transitions are not recovered after the network recovery.

3 Fault Tolerant RIA Crawling

In the fault tolerant P2P RIA crawling system we propose, crawlers and controllers must achieve two goals in parallel: Maintaining the P2P network and performing the Fault Tolerant RIA crawling using a data recovery mechanism.

3.1 Assumptions

- The unreliable P2P 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.

– 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.

3.2 Protocol Description

A major problem we address in this section is to make the proposed P2P RIA crawling system introduced by Ben Hafaiedh et al. [19] resilient to node failures, i.e. to allow the system to achieve the RIA crawling when controllers and crawlers 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 overlay network. 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 of 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 Fig. 1, an acknowledgment for an assigned transition was sent by a crawler informing the controller responsible for the transition about the destination state that follows from the transition execution. 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 known by 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 acknowledged to the previous controller. Therefore, we introduce a change to the P2P crawling described in Fig. 1 to make it fault tolerant, as shown in Fig. 3: 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

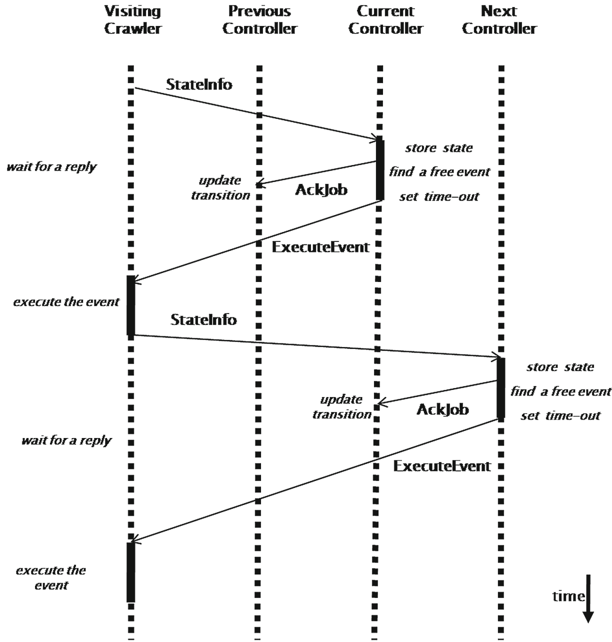

Fig. 3. The fault tolerant P2P RIA crawling during the exploration phase.

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.

4 Data Recovery Mechanisms

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 known by a controller so that crawlers can resume crawling from where a failed controller has stopped. We introduce three data recovery mechanisms to achieve the RIA crawling task properly despite node failures, as follows:

4.1 Retry Strategy

The Retry strategy [10] 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 from 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 is sent to all other controllers and allows them 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 un-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.

4.2 Redundancy Strategy

The Redundancy strategy is a strategy based on Redundant Storage [10] 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. 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 overlay network 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 acknowledged to the previous controller.

4.3 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 controllers may become overloaded. 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 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, 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.

5 Evaluation

We compare the efficiency of the Retry, the Redundancy and the Combined strategies in terms of the overhead they introduce during the exploration phase as controllers fail. We use the following notation: t_t is the average time required for executing a new transition, T is the total crawling time with normal operation, c is the average communication delay of a direct message between two nodes, n is the number of controllers and λ_f is the average failure rate of a node in the P2P overlay network, which is of the order of 1 failure per hour per node. Moreover, since the recovery of the overlay network is performed in parallel and is independent of the RIA crawling, we ignore the delay introduced by running the *idealization protocol* and we assume that queries are resolved with the ideal number of 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. Notice that this simplified model may be extended to allow simultaneous failures among controllers, with the condition that r back-up copies must be maintained by each controller to allow r simultaneous correlated failures, where $r < n$.

We performed a simulation study on experimental data-sets in a real execution environment, and measurements from the simulation results are used as

parameters in the following analytical evaluation. One of the tested real large-scale applications we consider in this study is the Bebop[1] RIA. It consists of 5,082 states and 468,971 transitions with a reset cost that is equivalent to 3 transition executions. The average communication delay c is 1 ms. For a crawling system composed of 100 controllers and 1000 crawlers, the average transition execution delay t_t is 0.3 ms. The delay introduced by each data recovery mechanism, when a controller fails, is described in the following.

5.1 Retry Strategy

When a controller fails, all states associated with the controller are lost and all transitions from these states have to be re-executed. Since states are randomly distributed among controllers, the fraction of transitions to be re-executed when a controller fails is of the order of $1/n$. 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. Therefore, the overhead of the Retry strategy is equivalent to $(\lambda_f.T)/(2.n)$.

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.

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 $(2.c)/(t_t)$.

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

5.3 Comparison of the Retry and the Redundancy Strategies When Controllers Are Not Overloaded

Preliminary analysis of experimental results [17] 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.

Figure 4 compares the the overhead of the Retry and the Redundancy Strategies with respect to the P2P node failure failure λ_f when controllers are not overloaded. Figure 4 shows that the Redundancy strategy significantly outperforms the Retry strategy as the number of failures increases. However, the Redundancy strategy may not remain efficient compared to the Retry strategy when controllers are overloaded, due to the repetitive back-up update of every executed transition required for redundancy.

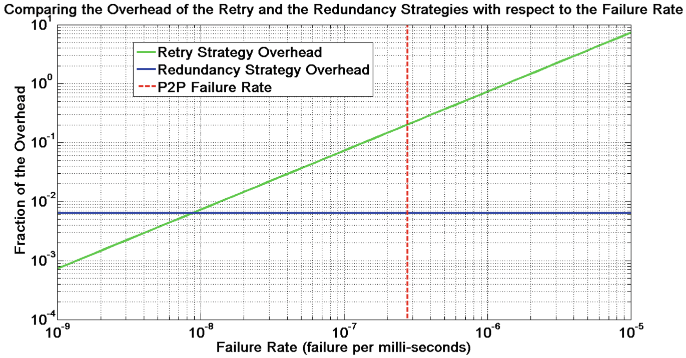

Fig. 4. Comparing the overhead of the Retry and the Redundancy Strategies with respect to the failure rate, assuming that controllers are not overloaded.

5.4 Combined Strategy

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 the number of lost transitions that have not been copied have to be re-executed again. 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:

$$T_p = N_t.t_t \text{ units of time} \quad (1)$$

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.

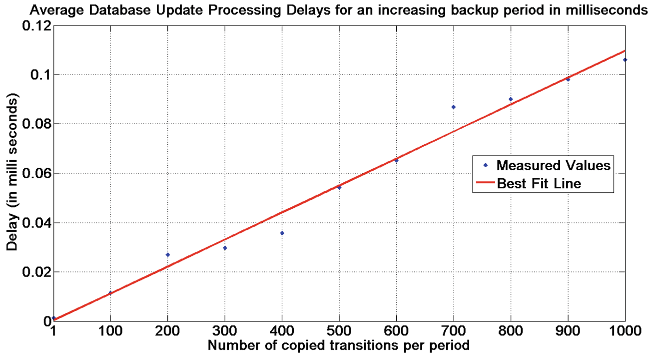

Fig. 5. Measurements of the processing delay p for updating the database for an increasing number of copied transitions.

Redundancy Management Delay: We measure by simulation the processing time required for updating the database with back-up transitions and we plot the average delay required for processing the back-up updates with an increasing number of transitions with a crawling system composed of 100 controllers and 1000 crawlers.

Based on the processing time measurements of Fig. 5, we obtain the linear equation $Overhead_{Redundancy}$ as a function of the number of copied transitions per update period N_t, as follows:

$$Overhead_{Redundancy} = 0.0001094.N_t + 0.00030433 \tag{2}$$

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. 5. Moreover, there is an additional communication delay required for sending the backup copy and receiving the acknowledgment back from the back-up controller of $2.c$ time units. Therefore, the total delay introduced by the redundancy management operation at the end of each period, called T_{bp}, is given by:

$$T_{bp} = \frac{T_p.p}{t_t} + 2.c \tag{3}$$

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. Assuming that failures occur on average in the middle of an update period, the retry processing delay is given by:

$$T_{rp} = \frac{\lambda_f.T_p^2}{2} \tag{4}$$

Total Overhead Introduced by the Combined Strategy: The overhead introduced by the Combined strategy is given by:

$$Overhead_{CombinedStrategy} = \frac{T_{bp} + T_{rp}}{T_p} = \frac{\lambda_f.T_p}{2} + \frac{2.c}{T_p} + \frac{p}{t_t} \quad (5)$$

The minimum overhead is obtained when $d(Overhead)/d(T_p) = 0$. We have:

$$T_p = 2\sqrt{\frac{c}{\lambda_f}} \quad (6)$$

The value of T_p with the minimum Combined strategy overhead is shown in Fig. 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.

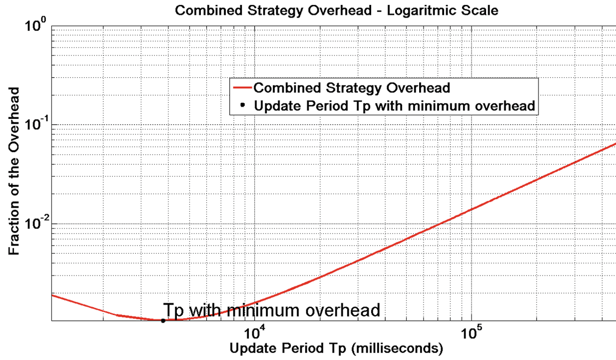

Fig. 6. Minimum overhead of the combined strategy.

Comparison of the Data Recovery Mechanisms: Analytical results show a high delay related to the Retry strategy compared to the Redundancy strategy when controllers are underloaded. Moreover, the Combined strategy outperforms the Redundancy strategy when controllers are overloaded 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 compared to the Redundancy strategy.

6 Conclusion

We have presented a resilient P2P RIA crawling system for crawling large-scale RIAs by partitioning the search space among several controllers that share the information about the explored RIA, which allows for fault tolerance, when both crawlers and controllers are vulnerable to crash failures. We defined three different data recovery mechanisms for crawling RIAs in a faulty environment: The Retry, the Redundancy and the Combined strategies. The Redundancy strategy outperformed the Retry strategy when controllers are not overloaded since it allows for reestablishing the situation that was before the failure, while the Retry strategy results in a high delay due to the repetitive execution of lost transitions. However, the Combined strategy outperforms the Redundancy strategy when controllers are overloaded by reducing the number of updates among backup controllers. This makes the Combined strategy the best choice for crawling RIAs in a faulty environment when controllers are overloaded. However, there is still some room for improvement: We plan to evaluate the impact of the data recovery strategies on the crawling performance when controllers are overloaded through simulation studies.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
2. Ratnasamy, S., et al.: A scalable content-addressable network. In: Proceedings of ACM SIGCOMM (2001)
3. Rowstron, A., Druschel, P.: Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
4. Schollmeier, R.: A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: Proceedings of IEEE International Conference on Peer-to-Peer Computing, Linkping, Sweden (2001)
5. Stoica, I., et al.: Chord: a scalable peer-to-peer look-up service for internet applications. In: Proceedings of ACM SIGCOMM, San Diego, California, USA (2001)
6. Cho, J., Garcia-Molina, H.: Parallel crawlers. In: Proceedings of the 11th International Conference on World Wide Web, WWW, vol. 2 (2002)
7. Fiat, A., Saia, J.: Censorship resistant peer-to-peer content addressable networks. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, Pennsylvania, USA, pp. 94–103 (2002)
8. Liben-Nowell, D., Balakrishnan, H., Karger, D.: Analysis of the evolution of peer-to-peer systems. In: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, pp. 233–242 (2002)
9. Shkapenyuk, V., Suel, T.: Design and implementation of a high performance distributed Web crawler. In: Proceedings of the 18th International Conference on Data Engineering (2002)
10. Hwang, S., Kesselman, C.: A flexible framework for fault tolerance in the grid. J. Grid Comput. **1**, 251–272 (2003)
11. Boldi, P., et al.: UbiCrawler: a scalable fully distributed Web crawler. Softw. Pract. Exp. **34**, 711–726 (2004)

12. Zhao, Y., et al.: Tapestry: a resilient global-scale overlay for service deployment. In: IEEE J. Sel. Areas Commun. (2004)
13. Paulson, L.D.: Building rich web applications with Ajax. Computer **38**, 14–17. IEEE Computer Society (2005)
14. Li, X., Misra, J., Plaxton, C.G.: Concurrent maintenance of rings. In: proceedings of the 23rd ACM Symposium on Principles of Distributed Computing, pp. 126–148 (2006)
15. 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, Markham, Ontario, Canada, pp. 146–160 (2012)
16. Peng, Z., et al.: Graph-based AJAX crawl: mining data from rich internet applications. In: Proceedings of the International Conference on Computer Science and Electronic Engineering, pp. 590–594 (2012)
17. Mirtaheri, S.M., Von Bochmann, G., Jourdan, G.V., Onut, I.V.: GDist-RIA crawler: a greedy distributed crawler for rich internet applications. In: Noubir, G., Raynal, M. (eds.) NETYS 2014. LNCS, vol. 8593, pp. 200–214. Springer, Heidelberg (2014)
18. Mirtaheri, S.M., Bochmann, G.V., Jourdan, G.-V., Onut, I.V.: PDist-RIA crawler: a peer-to-peer distributed crawler for rich internet applications. In: Benatallah, B., Bestavros, A., Manolopoulos, Y., Vakali, A., Zhang, Y. (eds.) WISE 2014, Part II. LNCS, vol. 8787, pp. 365–380. Springer, Heidelberg (2014)
19. Ben Hafaiedh, K., Von Bochmann, G., Jourdan, G.V., Onut, I.V.: A scalable peer-to-peer RIA crawling system with partial knowledge. In: Noubir, G., Raynal, M. (eds.) NETYS 2014. LNCS, vol. 8593, pp. 185–199. Springer, Heidelberg (2014)