A Fault Tolerant Structured Peer-to-Peer Distributed Web Crawler

Pratik Satish, Aniket Gattani, Aaron Thomas Cherian
Dept. of Computer Science
Rice University
{ps74, aag9, atc8}@rice.edu

***Abstract*—A crawler traverses the digital data that can be represented in tree-like structure. Traditional web crawlers are used by search engines for the purpose of indexing and returning relevant results. Websites contain lots of URL references that results in exponential burst of the crawl space as the search depth increases. This intensive workload cannot be done on a single host and requires the need of a distributed/parallel system.**

Distributed systems face many problems such as data races, poor distribution of workload between nodes, time consuming message passing between nodes, etc. Modern web crawlers use various architectures and configurations (peer-to-peer, master-slave, etc). A hybrid peer-to-peer and master-slave architecture offers the best features of both worlds. We believe our proposed architecture is a good fit for distributed crawling as it offers minimal communication overhead, correctness, better failure resilience and performance at scale.

I. Introduction

A crawler is a computer program that traverses the world wide web with the purpose of indexing web pages. Search engines typically use such programs for ranking/indexing pages. By ranking and indexing the web-pages on the world wide web, the search results provided by search engines become more relevant and faster. Crawling is not a concept restricted to web-pages. Crawling can be done on data that can be represented in a web/graph/tree fashion. TwitterEcho [1] is employs the crawl operation on Twitter messages(tweets) to gather insights about a demographic. Crawling is a suitable algorithm in this scenario since tweets and users in the Twitter domain can be represented as a tree data-structure.

Over the years the size of the web has exploded to 4.2 billion web pages and is ever increasing. It has become clear that the humongous task of crawling, indexing and ranking web pages can no longer be performed in a centralized fashion. This is because, on average, each URL at the root level will have hundreds and thousands of URL internally. The crawl space increases exponentially as the search depth increases. And the output of the crawling algorithm is immediately passed to the indexing/ranking module. The performance of such modules is proportional to the input space. A single node cannot cope the computational workload of a crawler system at large depths. Having a distributed web crawler allows the task of crawling to be parallelized so that it can scale linearly with the size of the search space.

Once we enter the space of distributed systems, many questions need to be answered along with it: what is the architecture of the system (peer-to-peer or master-slave), is the system fault tolerant, data-races and correctness of the results, does the performance scale as the number of nodes increase. etc.

Different distributed system solutions propose either master-slave [1] [2] or peer-to-peer architectures (and in some cases, variations and hybrids). These decisions are driven by the problem statements that these systems try to solve and the computation characteristics of their algorithm(s). For the purpose of web-crawling, we have gravitated towards a hybrid solution of peer-to-peer architecture with a notion of master peer. There are many factors that influenced this choice. Peer-to-peer systems use all nodes for computation whereas a master node in a master-slave configuration is typically reserved for co-ordination of work between the slave nodes. Since the crawling algorithm is computationally intensive, we believe wasting even a single node's compute bandwidth is an inefficient solution. A peer-to-peer system reduces the risk of a single point of failure.

A fault is when one or more components of a distributed system go down i.e crash/become unavailable or suffer network partition. If the system does not provide reliability guarantees, it may not be able to continue providing the intended service. An average compute server machine has a Mean Time To Failure(MTTF) of 3-5 years and a hard disk drive has an MTTF of 10-50 years. This means that in a compute cluster containing 10000 compute nodes and 10000 disks, about 5 nodes and 1 disk fail every day on expectation [3]. For internet scale companies, which use much larger clusters, node failures are business as usual. However, there is significant engineering efforts to make sure such faults do not hamper availability of service as a whole.

Although some master-slave systems [1] introduce backup nodes to reduce this failure, we believe these backup nodes waste precious compute bandwidth. In a typical structured peer-to-peer network, whenever a new node is discovered, multiple messages are broadcasted and leads to redundancy. This again wastes important bandwidth and scales exponentially with the number of nodes in the network. The master peer has been incorporated with the purpose of reducing the communication messages passed between the nodes in the distributed system. 14 A peer-to-peer network has better resilience since there is no single point of failure. But since we have a hybrid architecture with a master peer, there is still

a risk of the ”master” node dying. In the event that the master peer is dead, the other alive peers in the network should come to the conclusion that the master peer is dead. Once the peers come to this conclusion, they kick start an election process to appoint a new master peer. This is a well known problem in the research community (known by the multiple terms such as leader election, consensus algorithms, Byzantine Generals Problem) and some popular algorithms are RAFT [4], ripple consensus [5] and PAXOS [6]. An ideal implementation is non-trivial and fraught with bugs. Instead of diving into a full-fledged leader election algorithm, we implement a simple solution where we eliminate the search criteria of a ”good” leader. We randomly select one of the alive peers as a master based on some peer properties.

Data races and concurrency bugs are a danger of parallel/distributed systems. Truly peer-to-peer systems will need to use complicated safeguards to the shared data-structures to prevent data-races. The advantage of our hybrid peer-to-peer system is that the master peer will synchronize the requests and responses between the peers of the network. This synchronization block will ensure that no data races occur between the peer computations.

In this paper, we present the design and implementation of a distributed, scalable and fault-tolerant web crawler and page ranking system. The language of choice for this system is Python. The primary contribution of this work is a hybrid peer-to-peer distributed system that performs crawling operation on traditional web-pages. This system has features such as fault tolerance, correctness and low communication overhead. We evaluate our system with various configurations of search depth and node counts and see how it fares against a single node crawler.

The rest of the paper is divided as follows: section II introduces background material for the paper. Section III describes the implementation of the crawling algorithm and architecture. Section IV describes evaluation results that illustrate the utility of the capabilities described in this paper. Section V provides an overview of related work on distributed crawling. Section VI summarizes our conclusions and outlines our plans to extend this work in the future.

II. Background

This section will describe some concepts and terminology related to crawling and web-pages that will be used in the remainder of the paper.

A. Crawling

A typical consumer of the internet often uses search to navigate the colossal information repository, that is the internet. Any modern search service performs several steps before presenting the results to the user query. A lot of these steps are performed as pre-processing. One of the important steps performed as pre-processing is to crawl the various pages in the web and rank them.

Crawling essentially means discovering web pages. The crawling step typically involves starting from a few seed pages and finding all pages which are reachable from these web pages. In graph theoretic terms, this means collecting all web pages to which there is a path of edges or links between web pages starting from the seed. Any simple Depth First Search(DFS) or Breadth First Search(BFS) will give us the required set of web pages. However, there has been work [7] to show that BFS gives a set of closely related pages in the first few iterations itself. Any crawling system, distributed or not, will get overwhelmed if allowed to run indefinitely. Thus we need to prune the search space by limiting the number of iterations of BFS or the number of web pages crawled.

The Internet is constantly changing and expanding. Because it is not possible to know how many total web pages there are on the Internet, the crawler will start from a seed, or a list of known URLs. The crawler will crawl the web pages at those URLs first using spiders (which is another fancy term for a physical machine crawling web pages). Each spider will have its own set of web pages that it will be crawling. As the spiders crawl those web-pages, they will find hyperlinks to other URLs, and they add those to the list of pages to crawl next. Given the vast number of web-pages on the Internet that could be indexed for search, this process could go on almost indefinitely. However, the web crawler will search only for a given depth in this chain of links.

B. PageRank

After collecting the web pages, the next task is to rank them. The well known algorithm for this task is Google’s PageRank algorithm [8]. This algorithm essentially assigns a higher rank to pages which are linked by many pages which are themselves ranked high. We will point out however, that the main benefit of using the PageRank algorithm is that it is easily amenable to parallelization. Essentially, it can be converted into a data parallel program where each worker works on a part of the graph and only considers linkage within this sub-part of the node. More details in section III

The PageRank score for every web page which will help us sort the highest ranked web page among a given chain of web pages. Each peer will be computing the PageRank of its set of web pages. Since PageRank is a converging algorithm, we will start with a value and perform multiple iterations until it is about to converge.

C. Peer

Each peer is a physical machine with a unique id crawling a specific set of web pages. Since we are building a decentralized structured P2P network, each peer will have its own data-set for computation and will store or update the data in the central database. This database will be connected to all the peers. The distribution of this data-set will be completely dynamic and will depend on the number of peers. Each peer will receive almost an equal portion of the data-set. A peer can receive two kinds of requests: Crawling the web page. Computing the PageRank for the web pages. Peers can come and go within a network and on any such event, the data-set would be redistributed between all the peers. Which is why, all the peers

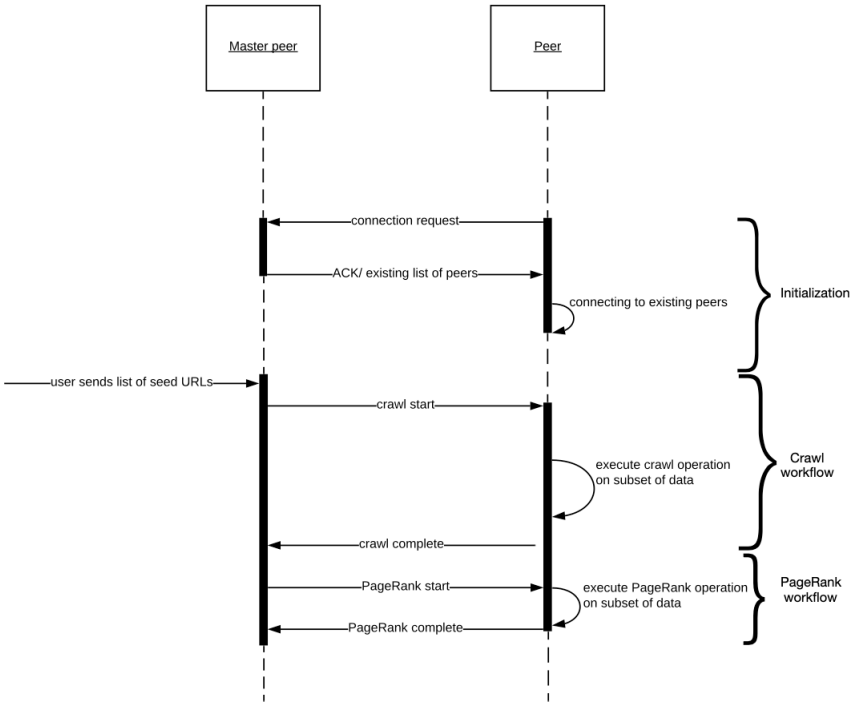

Fig. 1. Sequence diagram representation of the entire system

will periodically send a heartbeat message to the central peer to indicate that they are alive and working on their individual tasks. If any peer fails to send a heartbeat to the central peer, the central peer will relay a message to all peers that the peer has died and everyone would mutually agree the same and release their connections. This is essential because we are striving for a structured P2P network in which every peer has to be connected to every other peer.

D. Central Peer

There will be a central peer to receive requests for crawling or computing PageRank. Having a central peer saves the cost of broadcasting messages from every peer. It helps us to have a single source of truth and avoids logical bugs in the network. The responsibility of a central peer is:

- Accept the request from the user regarding the crawling of a set of websites. Each request will have its own job id.
- Add the list of web pages input by the user in the central database.
- Broadcast the message to all peers regarding the new request from the user.
- Send a request to all peers for computation of page rank after every user request.
- Compute its own responsibility for the request.
- Send acknowledgement for every heartbeat.
- Broadcast the message to all the peers that a new peer has joined and add an entry in the DB for the peer. Each peer will have its own id.
- Broadcast message to all the peers that a peer is dead and the peers should distribute its corresponding work between themselves.

E. Leader Election

We will be performing a leader election algorithm and will be choosing the leader in case the master peer crashes. This makes our system fault tolerant. (We will be implementing a naive version of the leader election in which the peer with the least id will be elected)

F. Central Database

A data-store is useful for sharing data between multiple nodes in a distributed network. Since nodes could dynamically add up or die down, we need to dynamically distribute the work and also save peer information in the DB. In case of any such event, the database will help the nodes to dynamically compute a hash value. The database also helps every peer to know what exactly the other peer is performing and can help speedup the process of work distributions and save on network bandwidth. If not for a central DB, every peer would need to constantly relay messages to the entire network to fetch the work. This would also help us save network bandwidth. The hash value of a node would be a function of the number of peers and the peer IP address and the job id. We would also require to store the page ranks computed by the peers in the DB. On every request by the user, the peers will input the info in the data-store. (We haven't decided yet on the database to be used so far).

G. HTML and DOM tree

HTML stands for Hypertext Markup Language. It is the language used to build web-sites. The web browser parses the HTML file and creates a DOM (Document Object Model) tree. The resulting DOM tree is rendered onto the window and is the end-result that users see on screen.

III. Implementation

In this section, we will focus on the implementation details of the modules within our crawler system. We will have the initialization phase of the crawler system in subsection III-A. Then we will explore the crawl workflow in subsection III-B and the underlying crawler algorithm in subsection III-C. In subsections III-D and III-E, the PageRank workflow our system and the PageRank algorithm is described in detail. Failure contention modules are discussed in subsections III-F and III-H.

A. Initialization

Figure 1 represents the sequence diagram of the entire system. The initial three messages in this diagram represents the initialization phase. The master peer is the entry point into the system. The script for the master peer is executed first which brings up the master peer. After the master peer is up, any peer can join into the network at any point in time. There is a separate script used by peer nodes to enter the distributed network. The script arguments include the master peer's IP address and port and a numeric identifier for the peer node. These values can be fetched from the database. Once the peer requests connection to master peer, master peer acknowledges the connection and send the list of all peers currently present in the network to the newly joined peer. The newly joined peer on receiving this list will establish connection with every other peer in accordance with a structured network. Every new connection leads to an exchange of the corresponding ids. Thus every node is aware of the existence of other nodes.

Each peer connected to the network will keep on waiting on its socket for any incoming request from the master peer. On receiving any such request, the peer will compute its work and send acknowledgments back to the master peer. When the user (who is in control of the master peer) will send the list of seed URLs as input for the crawl operation, that point marks the end of the initialization phase.

B. Crawl Workflow

The crawling workflow requires a user to trigger the workflow by inputting a list of URLs. We can easily extend the interaction with an API trigger to automate this workflow.

Figure 2 illustrates the entire crawl workflow in detail. The workflow begins when the leader/master peer receives a initial set of web pages to crawl from the user. The master peer will then create a job id and store it in the database. The master peer will then start broadcasting the message to all peers that a new request has arrived along with the job id. All peers (including master peer) are responsible for finding out their chuck/subset of crawl computation from the entire workload.

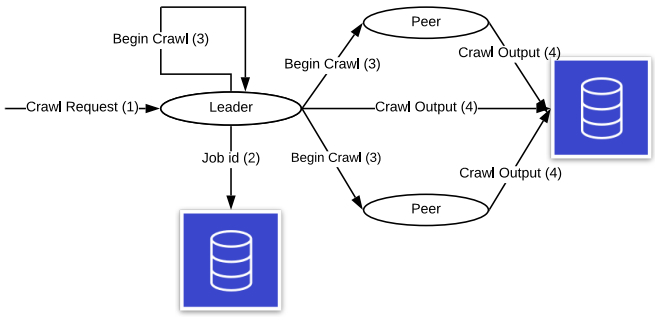

Fig. 2. Steps of the Crawl workflow

They do so by a hash function, the inputs of the hash function being the peer's id and the number of peers in the network. The hash function outputs a pair of integers that signify the start and end of the subset rows in the database. The subset that the peer derives from the hash function will be a sequential pair of rows in the database (which will be efficient when performing range queries).

After each peer determines its crawl workload, they get started with the crawler computation. We will look at how each peer performs the crawler computation in the next subsection. After the computation is complete, the peer will push its respective crawl output to the central database. It will also push an entry into a database table which serves as a checkpoint that the peer has completed its job. The peers after completion of their tasks will send an acknowledgement back to the master peer to notify that their task is completed. The master peer will wait until it receives acknowledgement from all peers before starting the PageRank workflow. If a new node joins or an existing node leaves in middle of the crawling workflow, the master peer broadcasts the info about the new peer to the network and all peers redistribute the work.

The results of the crawl operation are pushed to the database in the 'graph' table. Subsequent operations such as PageRank pull the results from this table. This design removes the need for message passing of results between the nodes.

C. Crawler Algorithm

The crawler algorithm is a modular in that it is unaware of a distributed network. The crawler module is represented as a class (called Crawler) which takes a list of URLs as input in its constructor. The input URLs represent the URLs at depth zero of the operation. Each peer creates its own instance of the Crawler class and passes its subset of crawl computation URLs to the constructor. The constructor sets up the data-structures pertaining to the crawl operation. The peer then calls the only public function within the Crawler class, crawl().

Once the crawl function is called by the peer, we sequentially create a HTTP request for each URL at $depth = 0$ (input URLs). The response of each request will contain the HTML content of the corresponding URL. We use a python library called Beautiful soup [9] to parse and extract all URLs present in the HTML content. We filter out the subset of URLs from

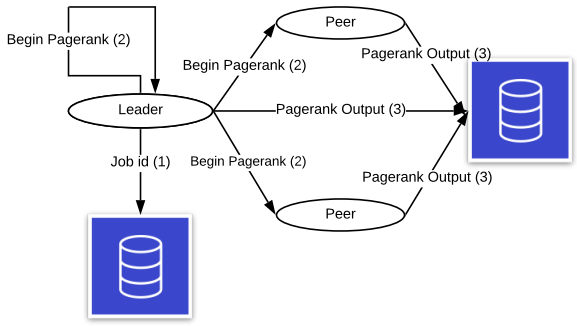

Fig. 3. Steps of the PageRank workflow

this set that have been encountered previously. A graph data-structure maintains all the extracted URLs as a child node for vertex corresponding to the input URL.

Once HTTP requests for all input URLs is complete, we have extracted a new set of URLs ($depth = 1$). Similarly, the steps of HTTP request, URL parsing and validation are repeated for URLs at $depth = 1$ to get new URLs at $depth = 2$. We keep repeating these steps in a loop till the maximum crawl depth (specified in the Crawler class) is reached. Once these steps are complete and maximum tree depth is attained, the crawl function returns the graph data-structure containing the crawled results to the peer node.

D. PageRank Workflow

This is the second workflow in the system and commences after the crawl outputs are published in the repository. As shown in Fig 3, the leader/master peer will commence the PageRank workflow after receiving acknowledgments from all the peers with respect to the crawl workflow.

The structure of message flow between the leader and other peers for the PageRank workflow is similar to the crawl workflow. The PageRank workflow starts with the master peer creating a job id and broadcasting it to the peers. The peers on receiving this broadcast message will initiate the PageRank process by computing the subset of PageRank workload to be computed by the peer. This subset is calculated using the same hash function that the peers used inside the crawl workflow. The peers begin the PageRank computation once they determine their respective computation subset. Once the peers are done with the computation, they push their results to the database. Each peer checkpoints its completion to the database and also sends an acknowledgement back to the master peer. The master on receiving all acknowledgments will finally end the PageRank workflow.

E. PageRank Algorithm

Our implementation of PageRank runs in multiple rounds. Each round itself is an iterative process which supplements the current PageRank with the linkage information of the sub-graph.

Let G be the complete crawled part of web graph. Let S $\subset$ G which is assigned to a single peer in a round. We run the

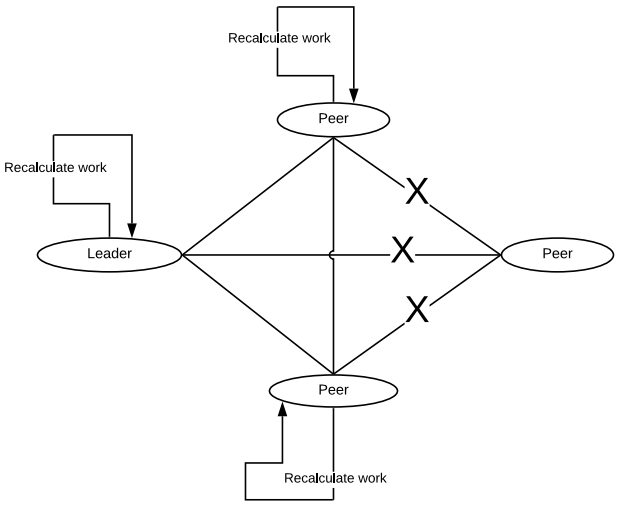

Fig. 4. Fault tolerance

regular iterative PageRank procedure on S until convergence. The page rank update formula is

$$PR(u) = \sum_{v \in B_u} \frac{P(v)}{L(v)}$$

where B_u is the set of pages which have a link to u, and L(V) is the cardinality of page v within the same sub-graph.

This update process is run repeatedly on all the nodes of the sub-graph until there is a reasonable convergence. In the next round, the sub-graph partition is changed. This means that a different uniformly distributed sub-graph is chosen to run on a peer. This helps to account for different cross links across rounds. We find that this process of partitioning and taking rounds leads to a page rank allocation much similar to the original algorithm, when evaluated on simple yet large graphs having a logical structure, where page rank can be easily verified by basic rules of symmetry.

The advantage of this iterative process in rounds is that we can cleanly divide G into mutually exclusive sub-graphs of near equal size and assign to peers. Also, this process makes crawling much more faster.

F. Fault Tolerance

The system ensures that any peer when moving in or out of the system leads to a consistent state. This is done by maintaining three invariants:

1) All peers will only commit the transactions when they are done.
2) If a peer moves out of the system, the master peer should detect it. The master peer will then send a communication to everyone regarding the peer activity. The peers will recalculate their hash and redo the uncommitted work by the failed node. To accomplish this, every peer will themselves store a list of URLs they have already computed upon.

3) If a peer moves in the system, the peers will again receive a message from the master regarding the same and they will discard any updates corresponding to the new peer.

G. Work redistribution

Our redistribution logic tries to allocate work whenever a new node joins or leaves the network. The logic is robust to avoid as much as possible redundant work and only considers the uncommitted work done by every node by performing a lookup in the database.

Whenever a new node joins while a crawl operation is in progress, the master node sends a message to the new node to let it know that it has joined in the middle of an crawl operation. The master node also sends message to the rest of the peers about the new node's entry. The new node checks all the uncommitted work by every other node in the network by looking up in the database and tries to find the optimal arrangement by diving the work almost equally.

All peers only commit 1 URL at a time. Hence before committing their ongoing results to the database or before picking a new URL to crawl, the peers perform similar calculations and check if the URL needs to be skipped.

In case of a node leaving the network, the peers perform a different computation. All of the peers find the work left over by the leaving node. The peers then divide the overall work almost equal chunks and take up the chunk according to their rank(id) in the network.

The database serves as a primary source of truth for all peers to identify their portion of work while also providing transparency of what the rest of the peers are computing. The final status can be viewed in crawl_status which shows for each job id, what are the URLs computed and by whom. crawl_done table lets us know what is the status of every node for every job id. If they have finished, the status should be 1. The master peer keeps polling the status of all nodes using this table and waits until all have completed.

H. Leader Election

Our system assumes the master peer to always exist. Hence, we construct a fault tolerant network where even the leader can be expected to get terminated and the system should resist effectively against such an occurrence. Once the connection with the master peer is broken for any node, the leader election algorithm commences.

If the master peer goes down, other peers will notice this since it is part of a structured network. Once the peer notices this, it will check if it has the least id in the network. If so, it will send a message to every peer stating that the old master is down and it is the new master. All other peers receiving this message will check if the existing master is indeed down. If it is, then the peers will update their existing entry and will send an acknowledgment back to the new peer (which wants to be the new leader) or else will send a reject. If the new master receives acknowledgment from more than half the peers, it will broadcast again that it is the new master and create an entry in the database. All the peers will actively terminate their connection with the earlier master and update the information about the new master locally.

A database table leader_node keeps tab of the current leader node. In the event of leader election, the newly elected node updates the leader node entry to its own IP address.

IV. Results and Evaluation

In this section, we evaluate the performance of the modules/algorithms used within the distributed crawler system. There are two primary modules in our system: crawler and PageRank. We pass two input URLs for the analysis: https://www.rice.edu and https://www.stackoverflow.com.

A. Crawl Workflow

Table I describes the execution time spent by the crawl workflow. We observed that the system execution time increases exponentially then the search depth is increased. This implies that a single node approach for larger depths is inefficient and time-consuming.

B. PageRank Workflow

Table I also highlights the performance of the PageRank implementation. Although its too early to gather insights from our preliminary analysis, we can see that PageRank workflow fares better than the crawl workflow for most cases. The one scenario where PageRank workflow is relatively slower is the single node configuration at $search\ depth = 3$. This is because as explained in the implementation of PageRank section, the multi round, multi-node pagerank algorithm greatly reduces the time complexity of dense graphs by partitioning the graph into subgraphs and only considering linkages inside the subgraph. This reduces the time complexity from $O(N^2)$ to $O((\frac{r}{M^2}) * N^2)$ where N is the size of entire graph, M is the number of nodes and r is the number of rounds

C. Correctness

Ideally we would define a distributed solution as correct if the crawl results of a single node match with that of a distributed crawler. It is not possible to meet this strict correctness criteria for a variety of reasons.

First, the crawler algorithm fires HTTP requests for URLs (refer section III-C). It is possible that a HTTP request gets an error code as response due to reasons such as server downtime, too many requests to the domain from a single IP address, etc. Thus it is difficult for even multiple runs of the same configuration (either single node or multi-node) to get the same results. Theoretically, single node crawlers can get status code: 429 (Too Many Requests) from servers since it is likely that they breach the server limit for connections from a single IP address. Thus we cannot guarantee that single node vs multi-node results will be same, although it will be very similar if run for enough number of rounds. Since the results of our PageRank algorithm depends on the ways of partitioning the graph into subgraphs across rounds, we may get a different results across different runs for the same input, depending on the partition/number of nodes.

TABLE I
EXECUTION RESULTS OF DISTRIBUTED WEB CRAWLER

No. of input URLs	Search depth	No. of nodes	Crawler (sec)	PageRank (sec)
2	2	1	49.82	7.52
2	2	2	30.51	5.7
2	3	1	733.52	1598.47
2	3	2	249.48	189.26

Since we use a master peer that synchronizes requests between the peers of the network, there are no data-races in performing the crawl and PageRank computations. The hash function ensures that there is no overlap of work between all peers in the network. The invariants maintained by the system (refer section III-F) ensure that failed peers don't leave the system in an inconsistent state.

Although we haven't addressed any DOS attacks in our implementation, we can configure the master peer to throttle incoming requests to ensure reliability. Our current assumption is that the workflow will not exceed for more than 15 minutes assuming a request for about 10 webpages crawled with a $search\ depth = 3$.

V. RELATED WORK

Typical distributed-crawlers like Scrappy [10] and Google Crawler [11] have a Controller/Agent configuration. Controller nodes delegate the work to the agent nodes. As long as the controller is alive, the pending work of dead agent nodes can be redistributed to other active agent nodes. But if the controller node dies, the entire workload is in jeopardy. The agents become stateless and there is no coordination between them.

The UbiCrawler [12] implements a distributed completely decentralized p2p crawler. However, in order to make peers exactly identical in responsibility, there is a huge network overhead for peer to peer synchronization due to all-to-all broadcast messages for peer discovery. In order to get around this, they assume that all the peers are known apriori. Also, while their system handles peer node failure, but the nodes are not allowed to rejoin the distributed system upon recovery.

TwitterEcho [1] is a crawling infrastructure developed for Twitter application. This system extracts demographic properties by crawling through tweets of a particular region and language. This work underscores a major benefit of using distributed systems for crawling. Crawling operation can be performed upon different types of data (e.g. web pages, tweets). The servers that hold this data often have limits on how much a single client/API call can access data. If a single node is used for crawling, we cannot run the crawling to scale because of the data access limitations. When multiple clients are used for crawling, each can crawl through data within the limits imposed by the server, but the cumulative result can be substantial. TwitterEcho sets priorities to different tweets (based on user properties). This priority affects the crawling data-plane. Focusing on promising root nodes for the crawl operation is an interesting proposition. It should be explored if we can incorporate such a priority scheme inside our system. TwitterEcho has a master-slave architecture. There are a few backup master nodes in place to replace the current master node in the event of a failure. Even though the backup nodes provide more stability, a disadvantage of this modified master-slave configuration is that the backup node's compute power isn't leveraged. Also, there is still a rare possibility that all master nodes go down.

Dist-RIA [2] is a distributed crawler for internet applications. It handles crawling of web-apps where the rendered HTML content changes primarily due to user interactions. Such web-apps are dynamic in nature and require mocking of the entire JavaScript event space for each DOM element to encounter all web page scenarios and subsequently extract all URLs. We believe our solution to be better in terms of performance and simplicity since it scrapes URLs straight from the HTML file's text content rather than extracting the same from DOM's content. Working with the HTML ensures that all the text and URLs are extracted by the crawler infrastructure. Thus we make do with the default HTML file that is downloaded on a URL hit and don't deal with complicated JavaScript events. Dist-RIA scales its crawl operation by using multi-core setup and also using multi-threading within a core/node. Adding more parallelism often doesn't give linear speedup and there is also the added burden of fixing data races. Dist-RIA employs star topology aka master-slave architecture. As discussed before, such architectures are not robust and fault-tolerant.

VI. CONCLUSIONS AND FUTURE WORK

Crawlers are essential for modern search engines to improve user experience and perform efficiently. The world wide web in growing at an exponential rate and thus crawlers too face a computational workload having exponential upper bound in terms of runtime complexity. Single node crawlers will take hours/days to meaningfully scan a small corpus of the world wide web. This is a great opportunity for using a distributed systems solution.

Distributed systems come in two primary configurations: master-slave and peer-to-peer. Each configuration has advantages/disadvantages over each other. We propose a hybrid solution so as to get most of the benefits of both architectures. Our architecture is primarily a peer-to-peer network with the exception that one of the peers in the network has more responsibilities than others. This is the leader/master peer.

The master peer reduces the communication overhead that traditional peer-to-peer networks suffer from. The master peer synchronizes the work distribution amongst all the peers in the network to ensure that there concurrency bugs due to race condition amongst the peers. The master peer takes an equal chunk of the overall computation, which is definitely beneficial for programs with high runtime complexity. Unlike traditional master-slave systems, our hybrid system is resilient to master peer failure and ensures that the computation reaches completion state whenever possible.

We will enhance the capability of the distributed crawler by improving the leader election mechanism. At present, we don't focus on the quality of a candidate peer when electing it as a leader. We assume that all peers in the network can be trusted. This assumption is incorrect and some checks should be in place to assure bad peers aren't elected as master peers. With some efforts, we should be able to implement a reliable consensus algorithm to cater this requirement.

We posit that our hybrid peer-to-peer network with a master peer is a good architectural choice for our system since the number of messages passed in the network is low. We have based this assumption on theoretical complexity analysis. It would be beneficial to compare performance of our system with other benchmarks/crawlers the gauge the differences amongst these different architectural choices.

Currently, due to limits on available compute resources, each peer in our network is a process within one physical machine i.e. all peers are processes running on the same machine. This is a great way to create a working prototype of the crawler system. The next step would be to run peers in independent devices and see the latency incurred for message passing between peers, database requests, etc. It would also be interesting to see the fault tolerance in action on a network where different nodes have different latency.

ACKNOWLEDGMENTS

We would like to thank Dr. Nathan Dautenhahn for reviewing at multiple conjunctures and providing valuable feedback. We would like to thank Sushovan Das for giving insights on developing distributed system architectures.

REFERENCES

[1] M. Bošnjak, E. Oliveira, J. Martins, E. Mendes Rodrigues, and L. Sarmento, "Twitterecho: A distributed focused crawler to support open research with twitter data," in *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, (New York, NY, USA), p. 1233–1240, Association for Computing Machinery, 2012.

[2] S. M. Mirtaheri, D. Zou, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut, "Dist-ria crawler: A distributed crawler for rich internet applications," in *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pp. 105–112, 2013.

[3] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," 2010.

[4] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 305–319, USENIX Association, June 2014.

[5] D. Schwartz, N. Youngs, A. Britto, *et al.*, "The ripple protocol consensus algorithm," *Ripple Labs Inc White Paper*, vol. 5, no. 8, p. 151, 2014.

[6] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[7] M. Najork and J. L. Wiener, "Breadth-first crawling yields high-quality pages," in *Proceedings of the 10th international conference on World Wide Web*, pp. 114–118, 2001.

[8] I. Rogers, "The google pagerank algorithm and how it works." https://www.cs.duke.edu/courses/cps049s/spring08/Slides/bryan.ppt, 2002.

[9] L. Richardson, "Beautiful soup documentation." https://www.crummy.com/software/BeautifulSoup/bs4/doc, 2007.

[10] Y. Fan, "Design and implementation of distributed crawler system based on scrapy," in *IOP Conference Series: Earth and Environmental Science*, vol. 108, p. 042086, IOP Publishing, 2018.

[11] M. Najork and A. Heydon, "High-performance web crawling," in *Handbook of massive data sets*, pp. 25–45, Springer, 2002.

[12] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.