ZORRO: ZERO-COST REACTIVE FAILURE RECOVERY IN DISTRIBUTED GRAPH PROCESSING

BY

MAYANK PUNDIR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisers:

Professor Roy H. Campbell
Associate Professor Indranil Gupta

ABSTRACT

Distributed graph processing frameworks have become increasingly popular for processing large graphs. However, existing frameworks either lack the ability to recovery from failures or support proactive recovery methods. Proactive recovery methods like checkpointing incur high overheads during failure-free execution making failure recovery an expensive operation.

Our hypothesis is that *reactive recovery of failures in graph processing that provides a zero-overhead alternative to expensive proactive failure recovery mechanisms is feasible, novel and useful*. We support the hypothesis with Zorro, a recovery protocol that reactively recovers from machine failures. Zorro utilizes vertex replication inherent in existing graph processing frameworks to collectively rebuild the state of failed servers. Surviving servers transfer the states of inherently replicated vertices back to replacement servers, which rebuild their state using the received values. This fast recovery mechanism prioritizes high degree vertices ensuring high accuracy of graph processing applications. We have implemented our approach in two existing distributed graph processing frameworks: LFGraph and PowerGraph. Experiments using graph applications on real-world graphs show that Zorro is able to recover between 87-92% graph state when half the cluster fails and maintains at least 97% accuracy in all experimental failure scenarios.

To my parents and brother, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisers Professor Roy H. Campbell and Associate Professor Indranil Gupta for their invaluable support and advice during my Masters degree. They helped ensure that I had all the resources and guidance I needed to complete my projects. I would like to thank Luke Leslie who collaborated with me on this project. I would like to acknowledge the guidance provided by Systems Research Group and Distributed Protocols Research Group students like Cristina Abad, Faraz Faghri, Muntasir Rahman and Mainak Ghosh. Finally, I would like to thank my family and friends for their love and support. My research assistantship during my Masters degree was funded by an NSF grant.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Large graphs derived from online social networks [31], search engines [10, 9] and biological networks [11] have made centralized methods of graph processing inadequate. Today, graphs consisting of trillions of edges are common in the Industry [16]. As graphs continue to grow, distributed graph processing is employed to extract valuable information. Google's Pregel [38] was one of the first distributed computation paradigms for processing large graphs. Subsequently, frameworks such as GraphLab [33], PowerGraph [23], LFGraph [26], GraphX [24], Pegasus [30] and GPS [46] have progressed the field of distributed graph processing.

To process large graphs, these distributed graph processing frameworks often run on clusters consisting of hundreds or even thousands of servers [38]. As the scale of distributed graph processing grows, failure recovery is increasingly needed. The most common failure recovery mechanisms in graph processing are proactive in nature i.e., they prepare for failures during failure-free execution. Pregel [38], Piccolo [42], Graph Processing System (GPS) [46], Distributed GraphLab [33] and PowerGraph [23] all use proactive checkpoint-based failure recovery mechanisms. Periodically, the framework saves a global snapshot of the system on reliable storage. After failures, the most recent snapshot is used to rebuild the last persisted graph state, from which processing resumes. Checkpoint-based recovery mechanisms incur a high overhead in planning for future failures during failure-free execution. For example, determining and saving a snapshot of application state on real world graphs partitioned across 16 servers increases the execution time of one iteration of PageRank application by $8-31$ times as shown in Figure 1.1

More recently, a replication based proactive failure recovery mechanism [55] has been proposed wherein $K+1$ replicas of each vertex are maintained to tolerate a K server failure. Even though this forced replication mechanism uses inherently created replicas, it incurs a constant overhead of updating the additional replicas in each iteration.

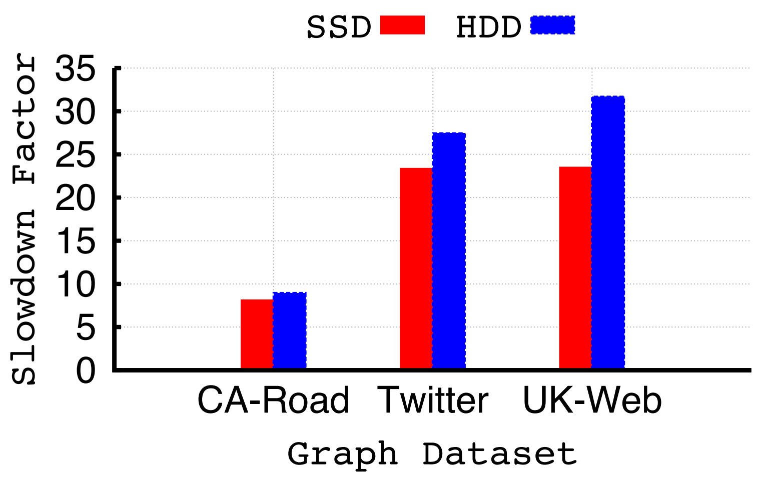


Figure 1.1: Overhead of determining and saving PageRank application state checkpoint on 16 servers (each with an SSD), using the graph datasets in Table 1.1.

Dataset	Vertex Count	Edge Count
Road Network (CA) [6]	1.96 M	2.76 M
Twitter [31]	41.65 M	1.47 B
UK Web Graph [10] [9]	105.9 M	3.74 B

Table 1.1: Graph datasets.

In this thesis, we argue that failure recovery mechanisms in graph processing that incur an overhead during failure-free execution are unnecessary. To this end, we introduce Zorro - a *Zero-COst Reactive RecOvery* protocol for distributed graph processing frameworks. Zorro does not prepare for server failures and hence, incurs no overhead during failure-free execution. When failures occur, it exploits vertex replication inherent in distributed graph processing frameworks to collectively rebuild the state of failed servers. Surviving servers transfer the states of inherently replicated vertices back to replacement servers which rebuild their state using the received states. This fast recovery mechanism prioritizes high degree vertices ensuring high accuracy of graph processing applications.

1.1 Contributions

Our hypothesis is that *reactive recovery of failures in graph processing that provides a zero-overhead alternative to expensive proactive failure recovery mecha-*

nisms is novel, feasible and useful.

We support the hypothesis using the following:

- Reactive recovery is novel: Existing graph processing frameworks either lack failure recovery or utilize expensive proactive recovery mechanisms like checkpointing and replication. To the best of our knowledge, we are the first to explore reactive failure recovery in distributed graph processing frameworks. We discuss related work in Chapter 2.

- Reactive recovery is feasible: We show that reactive recovery is feasible by implementing Zorro in two popular distributed graph processing frameworks: LFGraph [26] and PowerGraph [23]. We further discuss the motivation and feasibility of Zorro in Chapter 4.

- Reactive recovery is useful: We show that Zorro reactive recovery mechanism provides a fast, cheap and accurate alternative to expensive proactive recovery mechanisms. We show that recovery with Zorro is fast and takes less than the time taken by an iteration of graph processing applications. We show that Zorro results in accurate recovery with popular graph processing applications exhibiting more than 97% accuracy even with half of the cluster servers failing. We discuss detailed experimental evaluation in Chapter 7.

The rest of the thesis is organized as follows: In Chapter 2 we discuss related work and compare Zorro with existing literature. In Chapter 3, we present background of distributed graph processing. In Chapter 4, we motivate reactive recovery in graph processing. In Chapter 5, we discuss the design of Zorro reactive recovery protocol. In Chapter 6, we classify distributed graph processing frameworks intro two categories and discuss the implementation of Zorro in each category. In Chapter 7, we experimentally evaluate the implementation of Zorro in distributed graph processing frameworks using real-world graph and applications. We discuss directions for future work in Chapter 8. Finally, we conclude the thesis in Chapter 9.

Chapter 2

RELATED WORK

In this thesis, we have studied failure recovery in distributed graph processing frameworks. In this chapter, we discuss existing graph processing frameworks and their recovery mechanisms.

2.1 Distributed Graph Processing Frameworks

Distributed processing of large graphs began with the advent of data-parallel paradigms such as MapReduce [19]. Such paradigms enabled processing large datasets without the users being aware of the distributed nature of computation. The MapReduce paradigm and its open-source implementation Hadoop [3] utilize data partitioned across a distributed file system [50] [22] using a series of Map, Shuffle Reduce task. The MapReduce paradigm enabled the execution of graph processing applications such as PageRank [12], single-source shortest paths, connected components, label propagation, etc. For example, [43] discusses scalable methods of performing graph processing with the MapReduce paradigm.

However, the research community realized the inflexibility of the MapReduce paradigm for graph processing [36] which led to paradigms customized for graph processing. To this end, Pregel [38], a vertex-centric abstraction suitable for distributed processing of large graphs was proposed by Google.

Pregel and its open source implementations Piccolo [42] and Giraph [1] provide a vertex-centric abstraction for graph processing where a user-defined program is executed at vertices in parallel. The vertex-centric program follows the Gather-Scatter-Apply model whereby it executes the stages of Gather, Apply and Scatter. In the Gather phase, a vertex aggregates values from incoming neighbors. In the Apply phase, it processes the aggregated value to update its own value and in the Scatter phase, the result of processing is transferred to outgoing neighbors.

Graph Processing System [46] is an extension of the Pregel model with optimizations such as dynamic re-partitioning of vertices to reduce communication costs during the Scatter phase. To further reduce the communication costs in the Pregel model, GraphLab [34] and Distributed GraphLab‘[33] were introduced. GraphLab reduces the communication cost by replicating vertices on servers that need them for the Gather phase. These replicas are called Ghost vertices and are updated after each iteration by the master replica. PowerGraph [23], GraphLab's successor reduces the computation cost by distributing the computation load across these replicas. This is particularly useful for power-law graphs [7] where vertices may have arbitrarily large degrees. After each iteration, the mirrored replicas send the values of their partial computation to the master replica which aggregates them and transfers the aggregate value back to the mirrored replicas. LFGraph [26] further reduces the computation and communication cost for the Gather-Apply-Scatter model of graph processing. PowerLyra [15] provides an extension to PowerGraph for handling low-degree and high-degree vertices separately to better handle graphs with power-law distribution. In LFGraph [26], vertices are mirrored only at their outgoing neighbors on remote servers reducing communication cost to update the mirrors.

More recently, a Resilient Distributed Datasets (RDDs) abstraction of data-parallel processing was proposed in Spark [58] [59] [5] which is used by the graph processing framework GraphX [24] [2]. An RDD is an in-memory, read-only dataset which can be built either from data stored on persistent storage such as disks or from other RDDs. Each RDD maintains the sequence of operations required to build it in the form of a lineage graph. Different combinations of operations provided by the RDD abstraction (such as map, join and filter) to transform one RDD to another enable distributed graph processing in GraphX. In addition to graph processing, GraphX also provides the operations performed by a general data-flow system such as graph loading and graph analytics.

2.2 Failure Recovery in Graph Processing

To the best of our knowledge, we are the first to explore reactive failure recovery in distributed graph processing frameworks.

In the MapReduce paradigm, tasks being executed at workers are monitored by a master. The master receives regular progress reports from workers in the form of heartbeats. In the absence of heartbeats from a task, the master considers the task failed and initiates a new copy of the task on a different server. Additionally, the master measures the progress of Map tasks and spawns backup copies of slow or straggler tasks. This process of mitigating straggler tasks is called speculative execution [19]. Fault tolerance using re-execution of tasks and speculative execution are possible in MapReduce because of the independence among different Map tasks. Re-execution of tasks is not possible in specialized graph processing frameworks because of dependence among tasks being executed across servers. The Gather-Apply-Scatter phases of graph processing need to be performed simultaneously at all servers. Even with asynchronous computation models defined in GraphLab [33] and PowerGraph [23], tasks being executed by a server have a strong dependency on the in-memory graph state.

In specialized graph processing frameworks, two proactive recovery techniques are prominent: (i) Replication-based recovery and (ii) Checkpointing-based recovery.

Imitator [55] has a similar goal as Zorro of reconstructing graph state from surviving servers. However, it utilizes vertex replication mechanism to ensure that each vertex has at least $K+1$ replicas to tolerate a K server failure. The vertex replication mechanism is optimized by using the replicas that are already created on remote servers by the computation model in Hama [4]. This forced replication mechanism incurs a constant overhead of updating the additional replicas in each iteration. Additionally, the replication mechanism requires setting the value of K which may be hard as failures are unpredictable.

Checkpoint-based recovery is the most common failure recovery mechanism in distributed graph processing frameworks. Pregel [38], Piccolo [42], Graph Processing System [46], Distributed GraphLab [33] and PowerGraph [23] all provide failure recovery using checkpoint-based mechanisms.

In Pregel [38], workers checkpoint the state of vertex and edge values as well as the received messages from incoming neighbors. Additionally, the master checkpoints the state of global aggregators. Pregel uses a heartbeating mechanism for worker membership maintenance wherein worker failures by the absence of heartbeats. The master process reassigns partitions that failed servers were processing to either remaining servers or replacement servers. The authors in [38] also propose a *confined* recovery mechanism wherein workers checkpoint the state of outgoing messages in addition to vertex and edge values and received messages. Checkpointing of outgoing messages reduces the recovery cost by ensuring that only the replacement servers need to repeat the lost iterations. However, the authors do not evaluate the overhead involved in checkpointing or during failure recovery.

Piccolo [42], an open-source implementation of Pregel uses checkpointing along with user-assistance to recover from failures. At periodic intervals, it uses the Chandy-Lamport snapshot algorithm [14] to determine a global snapshot of the system which is then checkpointed. The user needs to assist the process by checkpointing the program information. In addition to original Chandy-Lamport snapshot algorithm based checkpointing, Distributed GraphLab [33] also proposes an asynchronous variant of the algorithm suitable for GraphLab's asynchronous computation mode. The graph processing application may proceed along with the snapshot algorithm which masks the cost of checkpointing. However, after failures, some iterations may need to be repeated which increases the recovery cost significantly.

The authors in [49] propose a partition-based recovery (PBR) mechanism that achieves faster recovery than traditional checkpoint-based mechanisms. It does so by generating a recovery plan to parallelize and partition the recovery task among surviving servers with the aim of greedily optimizing the computation and communication cost of rebuilding the lost state. The authors claim that the proposed recovery mechanism is the first in distributed graph processing frameworks to handle cascading failures. It does so by initiating a new recovery plan considering the most recent cluster state.

All proactive recovery mechanisms incur a constant overhead even during failure-free execution of graph processing frameworks. The authors of GraphX [24] state that checkpointing in distributed graph processing and data-flow systems is so expensive that users never turn it on. The authors of Distributed GraphLab [33] state that checkpoint-based failure recovery makes users explicitly balance failure recovery costs against restarting computation. Zorro adopts the stance that it is unnecessary to incur this overhead by showing that it is possible to recovery a large percentage of the graph ($87-92\%$) even with half of the servers failing in a cluster without incurring any overhead during failure-free execution. Additionally, Zorro recovers from arbitrary number of independent and cascading server failures with over 97% accuracy for graph processing applications. To summarize, Zorro makes recovery from failures reactive, cheap and simple.

To reduce dependence on checkpointing, GraphX [58] uses lineage graph-based recovery provided by the Resilient Distributed Datasets (RDDs) abstraction provided by Spark [58]. An RDD stores the sequence of transformations that were used to create it and replays the transformations on the input data which present on persisted storage. However, even with the fast reconstruction of RDDs the execution time with one server failure (out of 16) incurs an overhead of 36% [24] while performing PageRank using the UK web graph [9] [10]. Additionally, lineage-based recovery may require checkpoints to reduce the size of lineage graphs.

2.3 Failure Recovery in Related Systems

The authors in [47] propose an optimistic failure recover mechanism for iterative data-flow systems such as Stratosphere [21]. They claim that the processing state in such systems can reach a consistent state even after failures using "algorithmic compensation" functions. The system allows users to specify a "compensate" function appropriate for the distributed application. They show that such functions exist for three categories of applications involving link exploration (e.g. centrality computation), path exploration (e.g. shortest paths computation) and matrix factorization.

RAMCloud [41], an in-memory storage framework distributes data replicas across servers. After failures, RAMCloud proposes fast recovery by enabling surviving servers to participate in the state reconstruction of the failed servers in parallel.

There has been a lot of research in the field of failure recovery [20, 13]. Optimistic failure recovery has also been explored in distributed systems [53, 29, 51, 8] and large networks [35]. However, most of the optimistic failure recovery mechanisms use checkpointing. Even though checkpoint-based mechanisms have been very effective in storage systems [22] [44] and virtualization systems [17] [40], they incur high overheads in distributed graph processing.

Chapter 3

BACKGROUND

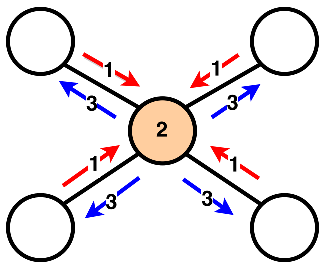
Distributed processing becomes a necessity for graphs consisting of billions of vertices and trillions of edges [16]. The MapReduce [19] distributed data processing paradigm and its open source implementation [3] enabled processing of large datasets without users worrying about the distributed nature of computation. MapReduce performs a specified data processing job using map, shuffle and reduce tasks. Map tasks process the input data in parallel to create intermediate key-value pairs, while reduce tasks process values aggregated on keys by shuffle tasks. Even though the MapReduce paradigm proved very successful for batch processing of large datasets [18], it often proved too rigid and inefficient for large scale graph processing [38] [36]. In this chapter, we provide background of distributed graph processing frameworks and failure recovery mechanisms employed by them.

3.1 Gather-Apply-Scatter Model

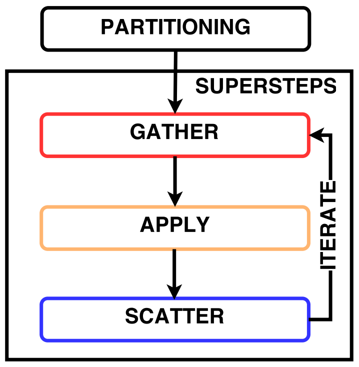
To eliminate the inefficiencies of graph processing using general data processing paradigms like MapReduce, Google proposed Pregel [38], one of the first distributed processing frameworks built for efficiently processing large graphs. Pregel utilizes a vertex-centric graph processing paradigm within the Gather-Apply -Scatter decomposition, wherein graph processing is divided into iterations of a user-defined program (called vertex-program) executed in parallel at vertices. Within each iteration, a vertex program performs three phases shown in Figure 3.1:

- Gather: In the gather phase, a vertex collects values form its incoming neighbors (step 1 in Figure 3.1(a)).

- Apply: In the apply phase, a vertex processes the values collected in the



(a) Gather (Step 1), Apply (Step 2) and Scatter (Step 3) on an example graph.



(b) Phases in synchronous Gather-Apply-Scatter (GAS) graph processing model.

Figure 3.1: Gather-Apply-Scatter model of distributed graph processing.

gather phase and modifies its own value based on the processing (step 2 in Figure 3.1(a)). The actual processing depends on the user-defined vertex program.

- Scatter: In the scatter phase, a vertex transfers the result of processing to its outgoing neighbors (step 3 in Figure 3.1(a)).

Consider an example using the single-source shortest paths (SSSP) graph processing application. SSSP computes the distance to every vertex in the graph from a user-specified source vertex. In the first iteration, vertices do not gather any values and instead perform an initialization step. The initialization assigns 0 to the source vertex and infinity to the remaining (non-source) vertices. Vertex values are then transferred to outgoing neighbors, combined with the weight of connecting edges if edges are associated with weights. In subsequent iterations, each vertex collects values from incoming neighbors in the gather phase, and then updates its current value as the minimum value from the collected messages in the apply phase. The updated value therefore represents the vertex's current minimum distance from the source vertex. If the value of a vertex is changed in this iteration, it performs the scatter phase by transferring its own value (plus the edge weight of its outgoing edges) to the local outgoing neighbors. This iterative process converges when no vertex undergoes an update. Any vertices that still have infinity as their associated value are not reachable from the source.

Figure 3.1 shows the stages involved in a synchronous GAS model. The graph is first partitioned across the cluster servers. The graph processing application then proceeds in iterations. In each iteration, vertices gather values from incoming neighbors, process and apply the values, and then scatter the results to their outgoing neighbors. More recently, an edge-centric graph processing paradigm [45] has been proposed. In edge centric iterations [45], edges gather values from source vertices and scatter to target vertices. In the Gather-Apply-Scatter model, the values associated with vertices and/or edges collectively form the graph state. In the rest of this thesis, we will use the terms state and value interchangeably.

In the synchronous GAS decomposition, the gather, apply and scatter stages in each iteration are synchronized using barriers to ensure that all servers execute these stages simultaneously. GraphLab [34] [33] and PowerGraph [23] also provide an asynchronous computation model which may benefit iterative machine learning applications such as Alternating Least Squares (ALS) [27] and Gradient Descent [56].

3.2 Distributed Graph Processing Frameworks

Graph Processing System [22], GraphLab [34] [33], PowerGraph [23] and LF-Graph [26] have all extended the original distributed graph processing paradigm introduced by Pregel [38]. In this section, we discuss the computation and communication models used in popular distributed graph processing frameworks.

In Pregel's communication model, a server transfers all messages sent by local vertices to neighboring vertices on remote servers. This action may require transferring multiple copies of a message to a remote server if multiple neighbors of a vertex reside on that server. To reduce the communication overhead by removing such redundancy, GraphLab [33] introduced the concept of *ghost* vertices. After partitioning vertices across servers, GraphLab creates these ghost vertices on each server for neighbors of its local vertices. These ghost vertices are updated after each iteration.

The authors of PowerGraph [23] observed that performing computation on a vertex requires collecting values from neighbors, which may be very expensive for high-degree vertices prominent in power-law graphs [52]. To this end, PowerGraph proposes partitioning edges across servers where each edge is assigned to exactly one server. For vertices having edges on multiple servers, one of the copies is labeled as the master while others as mirrors. Mirrors help reduce the computation load on the master replica. In each iteration, the master and the mirrors of each vertex perform partial computation based on the neighbors available at their respective servers. After each iteration, all mirrors transfer their values to the master which combines the values and transfers the combined result back to the mirrors.

LFGraph [26] reduces the computation and communication load in distributed graph processing even further. LFGraph utilizes cheap hash-based partitioning of vertices across servers to reduce total execution time. Each vertex maintains its set of incoming neighbors while each server maintains the updated values of incoming neighbors of its local vertices. In each iteration, each vertex uses values of its incoming neighbor vertices stored locally to update its current state. Each server then transfers the updated values of its local vertices to servers containing their outgoing neighbors. Before iterations start, each server builds a subscribe list for every other server in the cluster. The subscribe list for a remote server contains vertices whose updated values are required from the remote server. Based on these subscriptions, each server builds a publish list for every other server. This publish list is used to transfer updated vertex values to outgoing neighbors on remote servers.

Apart from these frameworks, graph processing frameworks like X-Stream [45], GraphChi [32] and LLAMA [37] are centralized and out of the scope of this thesis. GraphX [24] is a dataflow framework which performs graph processing using Resilient Distributed Datasets (RDDs) abstraction provided by Spark [58]. An RDD is an in-memory, read-only dataset which can be built either from data stored on persistent storage such as disks or solid state drives or from other RDDs. The RDD abstraction provides a set of operations such as map, join, filter and so on to transform one RDD to another. Different combinations of these operations are used in GraphX to enable distributed graph processing. In addition to graph processing, GraphX also provides the operations performed by a general data-flow system such as graph loading and graph analytics.

Chapter 4

MOTIVATION

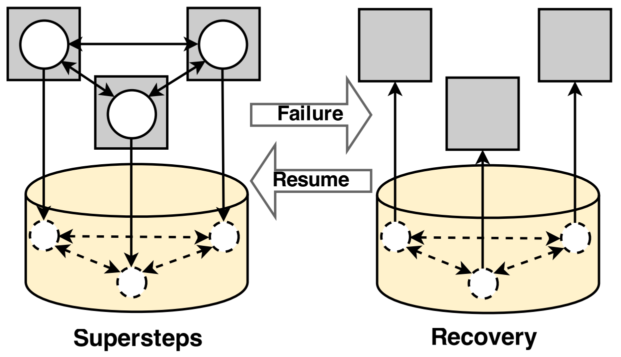
We define *failure recovery* in distributed graph processing systems as the recovery of the graph state from the iteration on which failures occurred. In an ideal scenario, failure recovery mechanisms in distributed graph processing frameworks should exhibit the following three characteristics:

- **Complete failure recovery**: A failure recovery mechanism should be able to recover the entire state of graph processing application after failures and maintain complete accuracy of results.
- **Zero-overhead failure recovery**: A failure recovery mechanism should not incur any overhead preparing for failures during failure-free execution.
- **Fast failure recovery**: A failure recovery mechanism should be able to recover from any number of failures quickly.

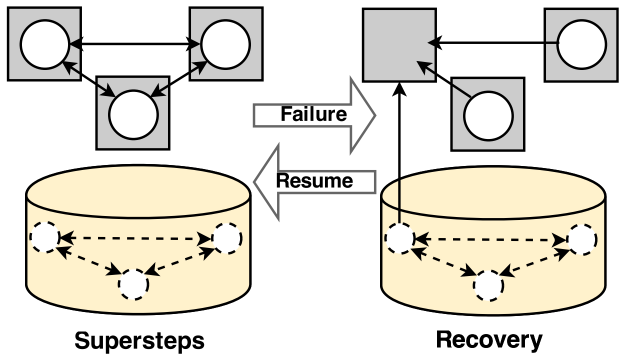
It is difficult for a failure recovery mechanism to exhibit all three characteristics; it must choose one out of costless failure recovery and complete failure recovery. This is because, after server failures, the in-memory state of the graph application will be lost if not persisted. On the other hand, if the in-memory state of the graph application is persisted, it will incur an overhead during failure-free execution.

4.1 Limitations of Proactive Approaches

Existing failure recovery mechanisms in distributed graph processing are proactive in nature and can be classified into the following two categories: (i) Checkpoint -based recovery mechanisms, and (ii) Replication-based recovery mechanisms. Both categories of proactive failure recovery mechanisms prefer complete failure recovery over costless failure recovery.



(a) Proactive checkpointing-based failure recovery in distributed graph processing.



(b) Reactive replication-based failure recovery in distributed graph processing.

Figure 4.1: Proactive vs. reactive failure recovery in distributed graph processing.

4.1.1 Checkpoint-based recovery mechanisms

Checkpoint-based failure recovery mechanisms are most common in distributed graph processing frameworks. Pregel [38], Piccolo [42], Graph Processing System [46], Distributed GraphLab [33] and PowerGraph [23] all use checkpoint-based failure recovery mechanisms. An example checkpoint-based failure recovery mechanism is shown in Figure 4.1(a). These mechanisms periodically checkpoint a snapshot of the graph state. After failures, computation resumes by loading the previously saved checkpoint. Even though these mechanisms achieve complete recovery, the processes of snapshot determination and checkpointing incur high overhead during failure-free execution, violating the costless failure recovery characteristic of an ideal mechanism.

Checkpoint-based failure recovery mechanisms have been successful in storage systems [22] [44] and virtualization systems [17] [40]. However, we argue that they incur unnecessary overhead for distributed graph processing applications because of a trade-off that is difficult to address. If the checkpoint interval is low, most of the checkpoints will not be used because of low Mean Time Between Failures (MTBF). For example, in a cluster of 16 servers, assuming that the MTBF of a single server is 360 days, the MTBF of any server may be as high as 22 days. In fact, even in a cluster of 1024 servers, the MTBF of any server may be as high as 8 hours. On the other hand, if the checkpoint interval is high, it is likely that a checkpoint is not available for the failed graph application execution. For example, it takes 493.81 seconds for PageRank application in PowerGraph to create a checkpoint of the UK Web graph [9] [10] on 16 servers each containing a solid state drive.

The optimum checkpoint interval [57], assuming the MTBF for a single sever to be 360 days, in a cluster of 16 servers should be 12 hours. However, the execution time of a graph processing application is usually much lower (average iteration time of PageRank application is 22 seconds in the same scenario). In fact, the users of distributed graph processing frameworks usually turn off checkpointing due to the associated overhead during failure-free execution [24].

4.1.2 Replication-based recovery mechanisms

A replication-based recovery mechanism [55] has been recently proposed. It creates $K+1$ replicas of the vertex graph state and maintains consistency between all replicas to tolerate a failure of up to K servers. The vertex replication mechanism is optimized by using the replicas that are already created on remote servers by the computation model in Hama [4]. Like checkpoint-based recovery, this replication-based recovery mechanism gives preference to complete recovery over costless recovery. The forced replication incurs an overhead in updating the replicas in each iteration to maintain consistency between all replicas. The overhead is dependent on the value of K which may need to be increased with the increase in the cluster size due an increased probability of failures. Additionally, the replication mechanism requires setting the value of K which may be difficult as failures are unpredictable.

4.2 Reactive Failure Recovery

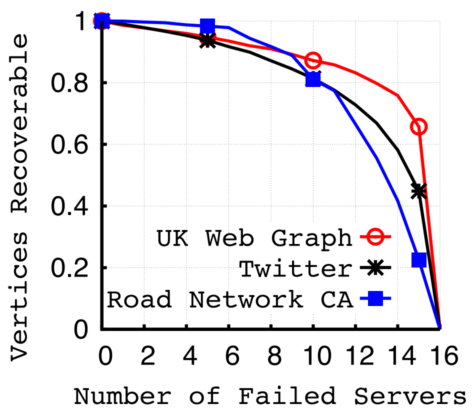
Zorro, a reactive failure recovery mechanism, gives preference to zero-overhead recovery over complete recovery. An example reactive failure recovery mechanism is shown in Figure 4.1(b). Zorro does not prepare for failures and incurs zero overhead during failure-free execution. When failures occur, it utilizes graph state inherently replicated in distributed graph processing frameworks at surviving servers to rebuild the state of failed servers. Graph state is replicated in distributed graph processing frameworks in the following forms:

- **Messages**: Servers maintain messages received from incoming neighbors present on remote servers from which the state of incoming neighbors can be derived. For example, Pregel [38] and its open source implementations Giraph [1] and Piccolo [42].
- **Shadows**: Servers maintain shadow copies of incoming neighbors present on remote servers from which the state of the incoming neighbors can be derived. For example, LFGraph [26].
- **Mirrors**: Servers maintain mirrored copies of both incoming and outgoing neighbors present on remote servers from which their sate can be derived. For example, distributed GraphLab [33] and PowerGraph [23].

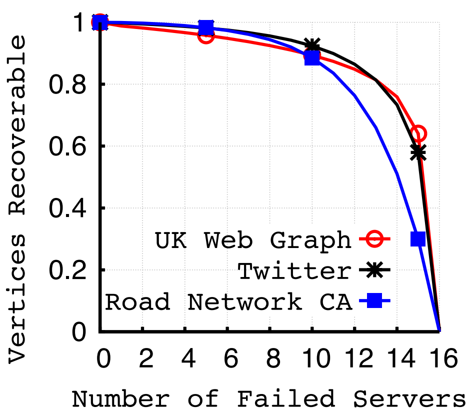
As Zorro prefers zero-overhead recovery over complete recovery, a fraction of the most recent graph state is lost in recovering from failures. The fraction of the graph state that is recovered by Zorro depends on the following:

- Structure of the graph used for executing graph processing applications.
- Partitioning function used for initial graph partitioning across cluster servers.
- Graph computation and communication model used by the framework which determines the graph state that is replicated across servers.

Figure 4.2 shows the percentage of recoverable state with Zorro in existing distributed graph processing frameworks assuming hash-based graph partitioning. As shown in the figure, Zorro is able to recover more than 95% of the graph state even with a quarter of the servers failing and more than 87% of the graph state even with half of the servers failing. Additionally, the Figure shows that Zorro



(a) LFGraph/Giraph/Hama



(b) PowerGraph

Figure 4.2: Percentage of recovered state in distributed graph processing frameworks as a function of the number of failed servers for graph datasets in Table 1.1.

recovers a high percentage of graph state for both power-law graphs (UK Web Graph [9] [10] and Twitter [31]) and exponential graphs (Road network CA [6]).

By performing reactive recovery using this inherently replicated state, Zorro exhibits the following characteristics:

- **Zero-overhead recovery**: Zorro does not incur any cost during failure-free execution.

- **Fast recovery**: Zorro transfers state replicated at surviving servers to rebuild the state of failed servers. This transfer occurs concurrently with graph loading at replacement servers, leading to extremely fast (often sub-second) recovery. We discuss more implementation details in Chapter 6.

- **Highly accurate recovery**: Zorro incurs a slight loss of the graph state after failures. The probability of recovering the state of a vertex after failures increases exponentially as its neighbor count increases or as the number of failed servers decreases. This allows Zorro to recover more than 95% of the graph state even with a quarter of the servers failing and more than 87% of the graph state even with half of the servers failing in a cluster of 16 servers. In Chapter 7, we show that graph applications maintain at least 97% accuracy even with a half of the servers failing.

- **Scalable recovery**: The fraction of the graph state recovered by Zorro depends on the number of failed servers and not on the total number of servers.

This allows Zorro to scale independent of the total number of servers in the cluster.

Chapter 5

ZORRO PROTOCOL DESIGN

In this Chapter, we present the design of Zorro reactive recovery protocol. We state the systems assumptions and discuss the workflow of graph processing frameworks equipped with Zorro reactive failure recovery.

5.1 Systems Assumptions

In this section, we state Zorro's systems assumptions before discussing Zorro failure recovery protocol.

5.1.1 Synchronous GAS Model

Zorro assumes a *synchronous Gather-Apply-Scatter* (GAS) graph processing model. Most popular frameworks such as PowerGraph [23], GraphLab [34], LFGraph [26] and Giraph [1] support this processing model. In Chapter 3, we discussed the synchronous Gather-Apply-Scatter model. Here, we give a brief overview.

Figure 3.1 shows the phases involved in a synchronous GAS model. The graph is first partitioned across the cluster servers. The graph processing application, then proceeds in iterations. In each iteration, vertices *gather* values from incoming neighbors, process and *apply* the values and *scatter* the results to their outgoing neighbors. In this model, graph state constitutes the values associated with the vertices of the graph. We use the terms graph state and vertex values interchangeably. In the synchronous GAS model, the gather, apply and scatter stages in each iteration are synchronized using a barrier to ensure that all servers execute these stages simultaneously. More recently an asynchronous graph processing model [23] has also gained popularity. We leave extending Zorro reactive recovery for asynchronous models as future work.

5.1.2 Failures and Replacements

Zorro assumes crash-stop failures with replacements where any server can crash at any time, even during recovering from other failures. Zorro assumes that each crashed server is replaced for computation to proceed. The assumption of each failed server getting replaced can be avoided by supporting elasticity techniques in graph processing where graph partitions are modified on the fly to compensate for failed servers. We leave extending Zorro for such a scenario as future work.

5.2 Framework Classification

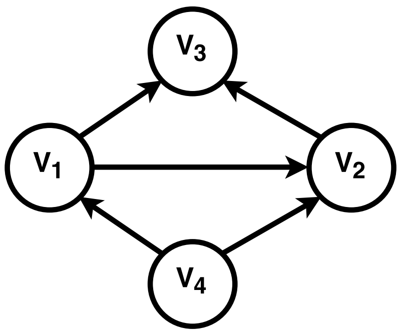
In this Section, we classify popular distributed graph processing frameworks into two categories based on the inherent vertex replication in their computation and communication model. In subsequent sections, we discuss Zorro recovery protocol in these categories of frameworks.

5.2.1 Out-neighbor Replication

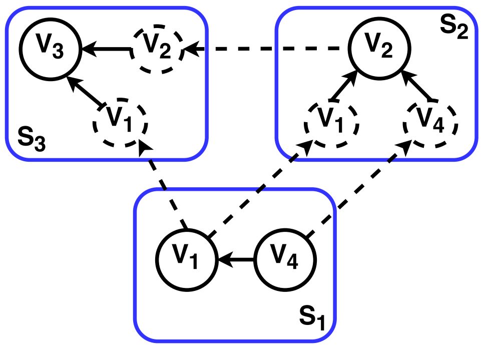
In frameworks belonging to out-neighbor replication category, the state of vertices local to a server is replicated at remote servers hosting their outgoing neighbors. Graph processing frameworks LFGraph [26], Giraph [1] and Hama [4] belong to this category. For example, consider the case shown in Figure 5.1(b) wherein the example graph shown in Figure 5.1(a) is partitioned across servers using consistent hashing of vertices. Original vertices are shown with solid circles while replicated vertices with dashed circles. In the Figure, the state of vertex V_2 hosted on server S_2 is maintained at server S_3. Similarly, the state of vertex V_1 hosted on server S_1 is maintained at servers S_2 and S_3, and the state of vertex V_4 also hosted on server S_1 is maintained at server S_2.

These frameworks can be further classified into two categories based on the form in which the state of vertices is replicated at servers hosting outgoing neighbors:

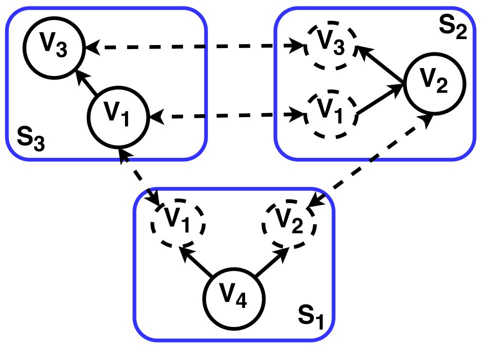
Message-based out-neighbor replication frameworks: Graph processing frameworks Giraph [1] and Hama [4] belong to this category. In these frameworks, vertices are partitioned across servers where each vertex is assigned to exactly one server. Each vertex maintains its outgoing edges and buffered messages re-



(a) Example graph.



(b) Out-neighbor vertex replication in LF-Graph and Hama.



(c) All-neighbor vertex replication in Power-Graph.

Figure 5.1: Vertex replication for a graph partitioned across three servers using hash-based partitioning.

ceived in the previous iteration. In each iteration, a vertex receives messages from incoming neighbors, processes the messages to update its state, and finally sends updates to its outgoing neighbors. Servers are responsible for passing received messages to the appropriate vertices. In these frameworks, the state of vertices local to a server is replicated at remote servers hosting their outgoing neighbors in the form of messages delivered to them for the correct execution of the apply phase in the GAS model. On failures, graph state derived from buffered messages is transferred to replacement servers.

Shadow-based out-neighbor replication frameworks: Graph processing framework LFGraph [26] belongs to this category. Like message-based frameworks, vertices are partitioned across servers where each vertex is assigned to exactly one server. Each vertex maintains its incoming neighbors while each server maintains shadows of incoming neighbors of its local vertices. In each iteration, a vertex utilizes state of its incoming neighbor vertices to update its current state.

Each server then transfers the updated states of its local vertices to servers containing their outgoing neighbors. In these frameworks, the state of vertices local to a server is replicated at remote servers hosting their outgoing neighbors in the form of shadows created on them for the correct execution of the apply phase in the GAS model. On failures, graph state derived from these shadows is transferred to replacement servers.

5.2.2 All-neighbor Replication

The frameworks in this category maintain the state of both incoming and outgoing neighbors either in the form of ghosts in GraphLab [33] or mirrors in PowerGraph [23]. We only discuss the case in PowerGraph because it is GraphLab's successor.

In PowerGraph, edges are partitioned across servers where each edge is assigned to exactly one server. For vertices having edges on multiple servers, one of the copies is labeled as the master while others as mirrors. In each iteration, the mirrors and the master of each vertex perform partial computation based on the neighbors available at their respective servers. After each iteration, all mirrors transfer their values to the master which combines the values and transfers the combined value back to the mirrors. In these frameworks, graph state is replicated at servers in the form of vertex mirrors of both incoming and outgoing neighbors on remote servers. On failures, graph state derived from these mirrors is transferred to replacement servers.

For example, consider the case shown in Figure 5.1(c) wherein the example graph shown in Figure 5.1(a) is partitioned across servers using consistent hashing of edges. Master vertices are shown with solid lines while mirrored vertices with dashed lines. Server S_1 maintains replicas of outgoing neighbors V_1 and V_2 on remote servers of its local vertex V_4. Similarly, server S_2 maintains replicas of incoming neighbor V_1 and outgoing neighbor V_3 on remote servers of its local vertex V_2.

Apart from these categories, graph processing frameworks such as X-Stream [45], GraphChi [32] and LLAMA [37] are centralized and do not benefit from Zorro reactive recovery. We also exclude GraphX [24] which utilizes the lineage-based failure recovery mechanism of the underlying system (Spark [58]).

5.3 Zorro Failure Recovery Protocol

Zorro, a reactive failure recovery mechanism, gives preference to costless recovery over complete recovery. Zorro does not prepare for failures and incurs zero overhead during failure-free execution. When failures occur, it utilizes vertex state inherently replicated in distributed graph processing frameworks at surviving servers to rebuild the state of failed servers.

The execution flow of distributed graph processing during failure recovery using Zorro is shown in Figure 5.2. In the discussion below, we number the events according to their labels in the figure. Zorro performs failure recovery in three phases:

- **Replace Phase**: Zorro recovery protocol utilizes a membership maintenance service (indicated by MS in Figure 5.2) such as Zookeeper [28] to maintain the list of member servers. During the replace phase, the failure of failed servers is identified (1) and the surviving servers are informed about the failed servers using a leave callback (`leave_cb`) (2). On receiving the callback, surviving servers suspend their ongoing graph processing and wait for the failed servers to get replaced and perform pre-processing steps such as graph loading. After the failed server gets replaced and joins the cluster (3), all servers are informed about the newly joined servers by the MS using a join callback (`join_cb`) (4).

  Relying on a membership maintenance service is not considered an overhead during failure-free execution as existing frameworks already include a membership maintenance service such as ZooKeeper or a heartbeating mechanism.

- **Rebuild Phase**: In the rebuild phase, replacement servers rebuild their local state using the state replicated at and transferred by surviving servers. In Figure 5.2), on receiving the join callback, surviving servers transfer (using `send_state`) replicated state of vertices hosted on replacement servers (5). The replacement servers acknowledge the completion of state transfer using a send callback (`send_cb`) (8).

- **Resume Phase**: In the resume phase, all servers resume the graph processing application from the iteration at which failures occurred. Zorro utilizes

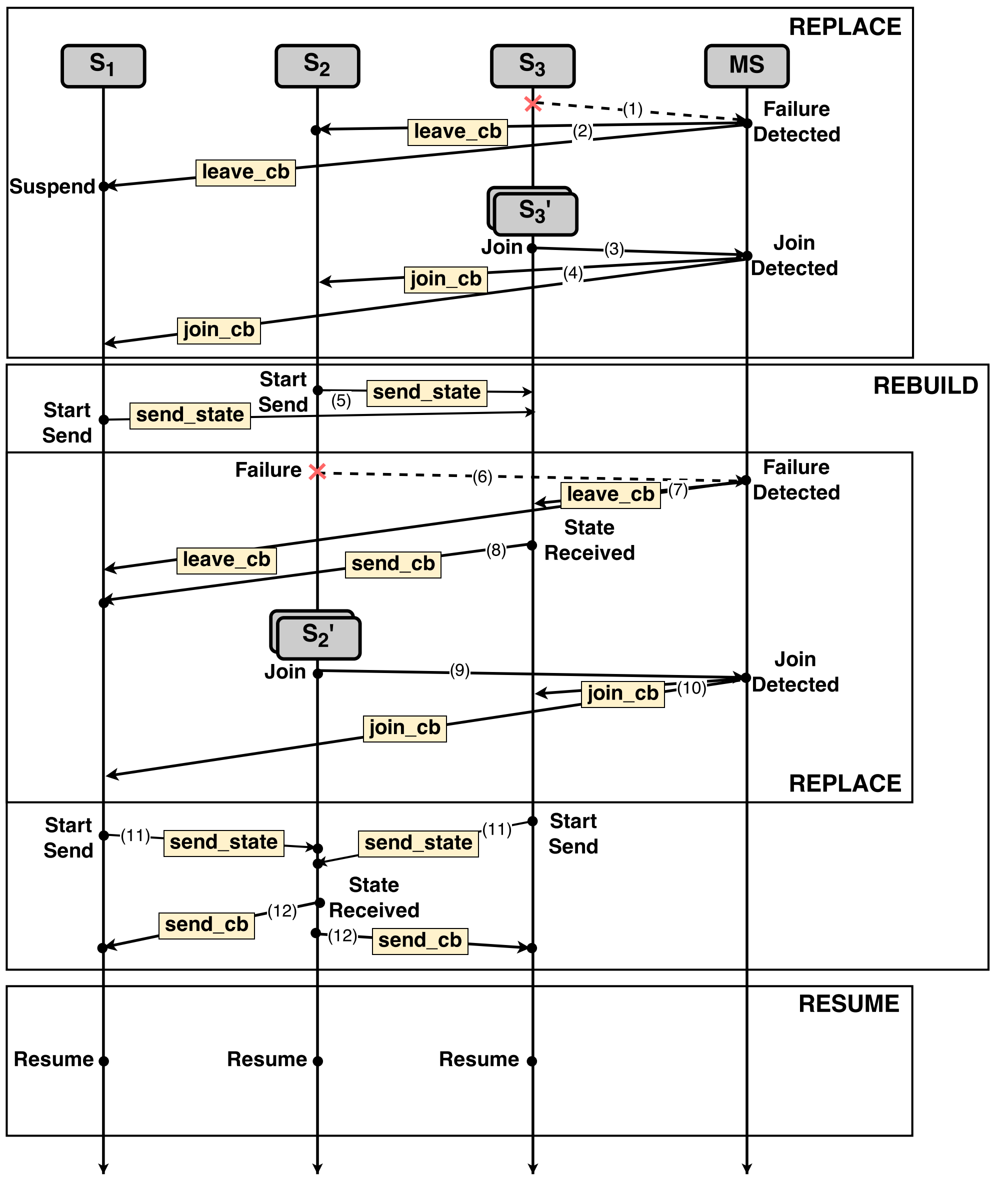


Figure 5.2: Zorro reactive recovery protocol timeline.

the membership maintenance service (MS) to store the most recent iteration number.

In LFGraph (Figure 5.1(b)), after the failure of server S_1, the state of vertex V_1 replicated at servers S_2 and S_3 and that of V_4 replicated at server S_2 are transferred to the replacement server. In this case, the replacement server recovers the values of both vertices V_1 and V_4 assigned to it. In PowerGraph(Figure 5.1(c)), after the failure of server S_2, the state of vertices V_2 replicated at server S_1 and that of vertices V_1 and V_3 present at server S_3 are transferred to the replacement server. In this case, the replacement server recovers the value of its master vertex V_2 as well as its mirrored vertices V_1 and V_3.

5.3.1 Rebuild Phase

The replacement servers build the state of the graph partitions assigned to them using the received state from surviving servers. Additionally, the replacement servers build the state that requires replication on them for the framework's computation and communication model. This replicated state can then assist in the recovery of future failures. For example, in LFGraph, surviving servers transfer the state replicated on them in the form of shadows as well as their own state that will be replicated at the replacement servers. In PowerGraph, surviving servers transfer the state of vertices that either have the master or a mirror on the replacement servers. We discuss application specific optimizations during the rebuild phase in Chapter 6.

Zorro may not be able to recover the state of some vertices on the replacement servers. The state of vertices having no replicated state on any of the surviving servers are lost. For the case shown in Figure 5.1(b), Zorro cannot recover the state of vertex V_3 after the failure of server S_3. Similarly, in Figure 5.1(c), Zorro cannot recover the state of vertex V_2 is both servers S_1 and S_2 fail. For the vertices whose state is not recovered, Zorro performs application specific initialization. For example, in PageRank application, the value associated with the vertices whose values could not be recovered by Zorro are initialized to the value of 1. Users may want to re-initialize vertices whose state could not be recovered using their own functions. We leave extending Zorro to allow a user-defined re-initialization function as future work.

Some graph processing applications in PowerGraph maintain graph state at edges, in addition to vertex state. The edge states often represent static edge weights which are rebuilt during graph loading. In the case of dynamic edge states, the state of an edge can be obtained from the state of its source and/or target vertices.

5.3.2 Cascading Failures

Cascading failures are those failures that occur while the system is recovering from a previous failure. The strength of Zorro is that recovery of a replacement server is independent of the recovery of other replacement servers and can be performed in parallel for all replacement servers. This enables Zorro to handle cascading failures trivially. In addition, existing replacement servers assist in the recovery of newly failed servers by sending back any state that they may have received during recovery.

We show the execution flow of graph processing frameworks during recovery from cascading failures in Figure 5.2. As the cluster is recovering from the failure of server S_3, server S_2 fails. The failure of server S_2 is detected by the membership maintenance service (MS) (6) and the surviving servers are informed about the failure using a callback (`leave_cb`) (7). The failure of server S_2 does not interfere with the recovery of server S_3. The replacement server of S_3 participates in the recovery by transferring any state it may have received from server S_2 when it was alive back to the replacement server of S_2 (11). After the transfer of replicated state from server S_1 to server S_2 and from server S_3 to S_2 gets completed, the replacement server acknowledges transfer completion (12).

For example, in Figure 5.1(b) we consider the case where the framework is recovering from the failure of server S_1 and server S_2 crashes. The recovery of server S_1 involves sending the state of vertex V_1 from servers S_3 and S_2 and vertex V_4 from server S_2. If server S_2 fails, the recovery of S_1 from server S_2 is unaffected. Similarly, in Figure 5.1(c) we consider the case where the framework is recovering from the failure of server S_3 and server S_1 crashes. The recovery of server S_3 involves sending the state of vertex V_1 from servers S_2 and S_3 and vertex V_3 from server S_2. If server S_1 fails, the recovery of S_3 from server S_2 is unaffected. This independence of recovery for each failed server allows Zorro to recover from cascading failures efficiently.

5.4 Analyzing Reactive Recovery

In this section, we theoretically analyze the probability of recovering a vertex with Zorro reactive recovery.

We define *recovery neighbors* as the neighbors of a vertex that enable replication of the vertex and hence, enable its recovery. Let us denote the set of recovery neighbors for a vertex v as $N(v)$. For the two classes of frameworks, the set of recovery neighbors is defined as follows:

- Out-neighbor Replication Frameworks: In these frameworks, the state of vertices local to a server is replicated at remote servers hosting their outgoing neighbors. Hence, for a vertex v, its state is replicated at remote servers hosting its out-neighbors. Hence, the number of recovery neighbors for this class of frameworks is equal to the number of outgoing-neighbors i.e., $N(v) = O(v)$, where $O(v)$ is the outgoing neighbors of vertex v.

- All-neighbor Replication Frameworks: In these frameworks, servers maintain the state of both incoming and outgoing neighbors present on remote serves. Hence, for a vertex v, its state is replicated at remote servers hosting its in or out-neighbors. Hence, the number of recovery neighbors for this class of frameworks is equal to the number of incoming and outgoing-neighbors i.e., $N(v) = I(v) + O(v)$, where $I(v)$ is the incoming neighbors and $O(v)$ is the outgoing neighbors of vertex v. PowerGraph is a special case of all-neighbor replication frameworks where due to edge partitioning, $N(v) = I(v) + O(v) - 1$. This is because at least one neighbor of vertex v belongs on the same server as v.

Next, we analyze the probability of recovering a vertex by virtue of its recovery neighbors being present on at least one of the surviving servers. We consider a scenario where f number of servers fail in a cluster of size m. For the simplicity of the analysis, we assume that the vertices are partitioned across servers using consistent hashing. The probability, $P(v_i)$ of recovering a vertex v_i after failures is given by:

$$P(v_i) = \left(1 - \left(\frac{f}{m}\right)^{|N(v_i)|}\right) \tag{5.1}$$

We can also quantify the expected number of vertices that can be recovered by Zorro using Equation 5.1. Let us denote the set of vertices that were present on the set of failed servers by V_f. Let V_r be the set of vertices that can be recovered by Zorro, then the expected value of the set V_r is given by:

$$\mathbb{E}\left[V_r\right] = \sum_{v_i \in V_f} \left(1 - \left(\frac{f}{m}\right)^{|N(v_i)|}\right) \tag{5.2}$$

Equation 5.2 shows that the probability of recovering the state of a vertex after failures increases exponentially as its neighbor count increases or as the number of failed servers decreases.

Chapter 6

IMPLEMENTATION DETAILS

We implement and evaluate Zorro reactive recovery protocol in exemplar frameworks from the two categories: LFGraph [26] from Out-neighbor replication category and PowerGraph [23] from All-neighbor replication category.

6.1 LFGraph

LFGraph is an example of shadow-based out-neighbor replication frameworks. Servers in LFGraph maintain shadows of incoming neighbors on remote servers.

Before iterations start, each server builds a subscribe list for each remote server containing vertices whose state is required from the remote server for the gather phase. Shadow copies of the vertices present in the server's subscription lists are created. Based on these subscriptions, each server builds a publish list for every remote server. The publish list is used to transfer vertex states to outgoing neighbors on remote servers and update the shadow copies.

These values are stored on remote servers as shadows. For the case shown in Figure 5.1(b), server S_1 has vertex V_1 in the publish list for servers S_2 and S_3 and vertex V_4 in the publish list for server S_2. Similarly, server S_2 has vertex V_2 in the publish list for server S_3 while its publish list for server S_1 is empty. Finally, the publish lists of server S_3 are empty for both servers S_1 and S_2.

In the gather phase, each vertex iterates over its incoming neighbor state replicated locally as shadows, processes them and modifies its own state based on the processing. This state is then transferred to outgoing neighbors using the publish lists in the scatter phase to update remote shadows. The scatter phase at the end of each iteration ensures that the shadows at each server reflect the latest changes. In LFGraph, each server maintains two copies of its local vertices: an original copy and a backup copy. During the apply phase, updates are made to the original copy which are then merged into the backup copy at the end of apply phase.

We implement Zorro in LFGraph by modifying the `computation_worker` and `communication_worker` classes within the JobServer. The three stages of Zorro reactive recovery in LFGraph are summarized below:

1. **Replace:** LFGraph uses ZooKeeper [28] to identify server failures. On failures, Zookeeper issues a leave callback to all surviving servers. On receiving the leave callback, the surviving servers suspend their iteration, save the iteration number in Zookeeper and wait at a barrier for failed servers to get replaced. As replacement servers join the cluster after loading their respective graph partitions, Zookeeper issues a join callback to all surviving servers.

2. **Rebuild:** The graph state maintained at each server as the shadows of incoming neighbors is used to re-build the state of failed servers. On receiving a join callback, each surviving server initiates the rebuild process for the replacement server. The rebuild process transfers two sets of vertices to the replacement server:

    - Shadow vertices on surviving server having their master copies on the failed server. These vertices constitute the incoming neighbors of the surviving server's local vertices present on the failed server i.e., vertices present in the surviving server's subscribe list for the failed server. These vertices build the publish list at the replacement server for the sender surviving server.
    - Master vertices on surviving server having their shadows on the failed server. These vertices constitute the incoming neighbors of the failed server's local vertices present on the surviving server i.e., vertices present in the surviving server's publish list for the failed server. These vertices build the subscribe list at the replacement server for the sender surviving server.

    The transfer of graph state from surviving to replacement servers ensures that the publish and subscribe lists at replacement servers get re-built for all surviving servers. As such, the publish-subscribe phase needs to be repeated only among replacement servers. LFGraph stores original and backup vertex copies separately. This allows sending the two sets of vertices in parallel to replacement servers.

3. **Resume:** Servers resume computation from the start of the last iteration before failure. This iteration number is saved by surviving servers with ZooKeeper. For correctness of computation after recovering from failures, each server operates on the most recent states of the incoming neighbors of its local vertex set. A server receives the states of the incoming neighbors of its local vertex set as part of the replicated state during rebuild phase. Additionally, Zorro performs an additional partial scatter among replacement servers to ensure that each vertex works on the most recent states on its incoming neighbors.

6.1.1 Correctness

We discuss failures during phases of the Gather-Apply-Scatter model in LFGraph and the correctness of our approach in handling these failures.

In the gather phase, vertices iterate over incoming neighbors present on the local server as shadows. Zorro handles failures during the gather phase as servers resume the iteration from beginning after recovery.

In the apply phase, vertices process the states of incoming neighbors collected in the gather phase to update their own state. As mentioned before, each server maintains two copies of its local vertices: an original copy and a backup copy. The updated state after apply phase is written to the original copy by default. The updates to the original copy are merged with the backup copy after all vertices have finished the apply phase. Zorro handles failures during the apply phase by transferring vertex states from backup copies during recovery which are updated only if all servers have finished the apply phase. This ensures that the backup copies do not reflect any changes of the apply phase if failures occur during the apply phase. Essentially, Zorro merges vertex state copies after Scatter, rather than between Apply and Scatter as in vanilla LFGraph.

In the scatter phase, updated states of vertices are transferred to remote servers hosting outgoing neighbors using publish lists. Zorro handles failures during the scatter phase by receiving all updated values from incoming neighbors before updating the remote value store by creating a copy of the remote value store in the background during the Apply phase. This results in an average per-iteration overhead of just 0.8%. This is the only instance of overhead incurred by Zorro.

6.2 PowerGraph

PowerGraph is an example of all-neighbor replication frameworks. Servers in PowerGraph maintain mirrors of both incoming and outgoing neighbors on remote servers.

In PowerGraph, edges are partitioned across servers where each edge is assigned to exactly one server. For vertices having edges on multiple servers, one of the copies is labeled as the master while others as mirrors. For the case shown in Figure 5.1(c), the edges (V_4, V_1) and (V_4, V_2) are assigned to server S_1, (V_1, V_2) and V_2, V_3 to server S_2 and (V_1, V_3) to server S_3. Vertices V_1, V_2 and V_2 span multiple servers and hence, have mirrors. We represent the master vertex copies with solid circles and mirror vertex copies with dashed circles. Vertex V_2 has its master at server S_2 while a mirror at server S_1, vertex V_3 has master at S_3 and mirror at S_2 while V_1 has master at S_3 and mirrors at S_1 and S_2.

The mirrors on remote servers provide the replication required by Zorro to recover from failures. In each iteration, the mirrors and the master of each vertex perform partial computation based on the neighbors available at their respective servers. After each iteration, all mirrors transfer their values to the master which combines the values and transfers the combined value back to the mirrors. Such a design distributed the computation load at a vertex due to a large degree across the master and mirrors. This optimization is aimed at power-law graphs wherein vertices may have arbitrarily large degrees.

We implement Zorro in PowerGraph by modifying the `synchronous_engine` class which implements synchronous Gather-Apply-Scatter model. We also modify the `local_graph` class which provides data structures for the representation of the local graph on each server.

1. **Replace:** As with LFGraph, PowerGraph identifies failures using ZooKeeper. After failures, survivors retain the local graph state while terminating the synchronous engine.

2. **Rebuild:** The graph state maintained at each server as the mirrors and masters of vertices present on the failed servers is used to re-build their state. After failures, all servers load their respective graph partitions and transfer the edges previously held by failed servers to the replacement servers. The servers join the cluster after loading graph partitions by contacting Zookeeper which issues join callbacks to broadcast the join operation.

On receiving a join callback, each surviving server initiates the rebuild process for the replacement server. Surviving servers iterate over local masters and mirrors and transfer the state of vertices that have either a master or a mirror on the replacement servers. This is possible as each vertex copy maintains the set of servers where it is present either as the master or mirrors. The replacement servers update their graph state with the received vertex states.

Each vertex copy maintaining its set of hosting servers allows an optimization to reduce the network costs during recovery. Only one of the surviving servers need to transfer the state of a vertex held by the replacement servers either as the master or mirrors. Zorro achieves this using the following check for each vertex:

$$procid == \operatorname*{argmin}_{p \in P_S(v)} |p.procid - (v.id \ \% \ |S|)| \tag{6.1}$$

,where procid is the process id of the server, $P_S(v)$ is the set of surviving servers that hold vertex v either as the master or a mirror and S is the set of all servers in the cluster. This optimization ensures that only a single surviving server is responsible for transferring the state of a vertex required by the replacement servers.

The above mentioned optimization requires handling cascading failures differently in PowerGraph. Cascading failures require that the recovery of failed servers is not affected by more servers failing during the recovery. This may require surviving servers to make multiple iterations over their local vertices as they may become responsible for transferring the state of vertices due to failures during the recovery process.

3. **Resume:** Servers resume computation from the start of the last iteration before failure. This iteration number is saved by surviving servers with ZooKeeper. For correctness of computation after recovering from failures, each server requires knowledge of the vertices active in the current iteration and the messages that they should work on from their incoming neighbors. To ensure this, Zorro performs an additional partial scatter where mirror and master vertices scatter to their local neighbors. This scatter is local and incurs no network overhead as vertices only need to *signal* their local neighbors in PowerGraph.

6.2.1 Correctness

We discuss failures during phases of the Gather-Apply-Scatter model in PowerGraph and the correctness of our approach in handling these failures.

In the gather phase, the mirrors and the master perform partial computation using the neighbors available locally on the their respective servers. The partial computation results are stored in accumulators. Zorro handles failures during the gather phase trivially as all servers terminate their ongoing computation and the state of accumulators does not affect vertex states.

In the apply phase, partial results from mirrors are transferred to the master which aggregates them and transfers the result back to mirrors to synchronize their values. Zorro handles failures during the apply phase by ensuring that all masters receive all partial results from their respective mirrors before updating their states.

In the scatter phase, vertices signal their neighbors is their state gets updated. Zorro handles failures during the scatter phase by performing a partial scatter after recovery.

Chapter 7

EVALUATION

In this Chapter, we experimentally evaluate Zorro reactive recovery protocol. We have implemented Zorro in LFGraph [26], an in-neighbor replication framework and PowerGraph [23], an all-neighbor replication framework. Our evaluation goals for each framework are as follows:

- Evaluate the accuracy of graph applications with varying number of failed servers
- Evaluate the accuracy of graph applications with varying iteration number at which failures occur.
- Evaluate the time take by Zorro to recover from failures with varying number of failed servers.
- Evaluate the network overhead incurred by Zorro during recovery.

7.1 Experimental Setup

We perform our experiments on a cluster consisting of 16 servers. Each server contains 16 Intel Xeon E5620 processors and 64 GB RAM. The servers are connected to each other using a 1 Gbps network.

We use the following three graph datasets for our experiments:

- Road Network (CA): The graph representing California road network [6]. It is an exponential graph containing 1.96 M vertices and 2.76 M edges.
- Twitter: The graph representing follower-followee relationship between Twitter users [31]. It is a power-law graph containing 41.65 M vertices and 1.47 B edges.

- UK Web: The graph representing UK webpages and links between them. It is a power-law graph containing 105.9 M vertices and 3.74 B edges.

We introduce failures at random servers by terminating the graph processing framework processes within a given iteration. In our experiments, we run graph applications for a fixed number of iterations and not wait for convergence after failures. For example, if PageRank application fails at iteration 5 when being executed for 10 iterations, the application, after recovery, resumes from iteration 5 and terminates at iteration 10 irrespective of convergence. This allows us to evaluate the accuracy with Zorro reactive recovery protocol correctly. For each experiment, we report the average over three trials.

7.2 Applications

We evaluate Zorro reactive recovery using two popular graph processing applications: PageRank [12], Single-Source Shortest Paths (SSSP) [38], Connected Components (CC) [46] and K-core [48].

7.2.1 PageRank

PageRank [12] computes the rank of each page iteratively based on the ranks of each incoming neighbors. Pregel [38] introduced a vertex-centric PageRank which has been adopted in all distributed graph processing frameworks [23] [26] [1] [4] [42].

Metrics

Let P_o^k be the original top-*k* pages based on their PageRank values. Let P_n^k be the top-*k* pages after recovery from failures using Zorro. We evaluate the accuracy of PageRank application using the top-*k* pages based on their PageRank values using the following metrics used in [39]:

- Top-*k* Lost (TL): This metric measures the fraction of the top-*k* vertices lost due to failures. Mathematically, PageRank top-*k* lost is represented as $\frac{|P_o \setminus P_n|}{|P_o|}$.

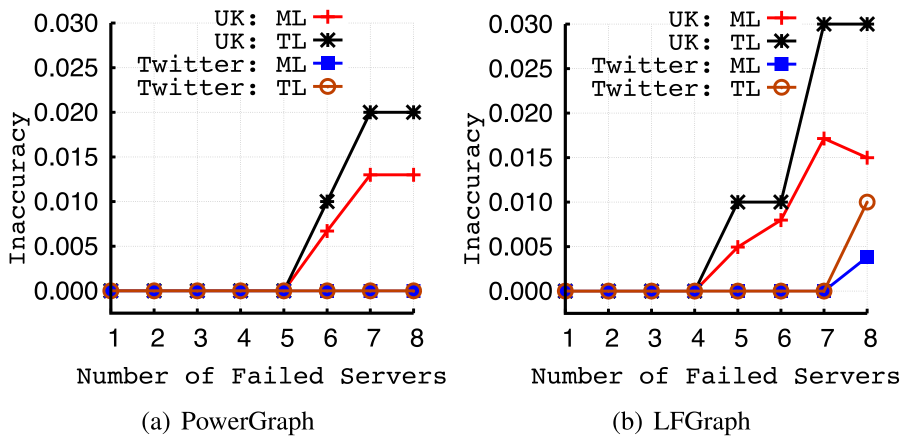


Figure 7.1: PageRank inaccuracy as a function of the number of failures. Servers are failed randomly at the fifth (out of 10) iteration.

- Mass Lost: This metric measures the fraction of PageRank scores lost by the application after recovering from failures. Mathematically, PageRank mass lost is represented as $\sum_{e \in P_o \setminus P_n} Rank(e) / \sum_{e \in P_o} Rank(e)$

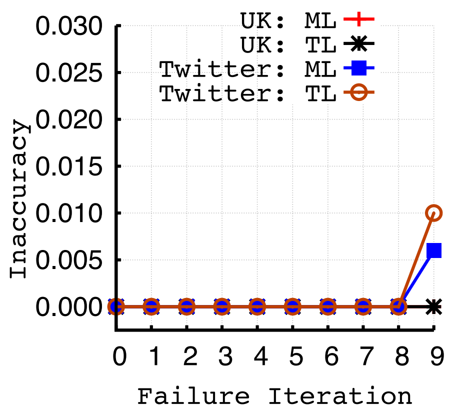
Results

We evaluate the accuracy obtained with Zorro reactive recovery by varying the number of failed servers and by varying the iteration at which failure occurs.

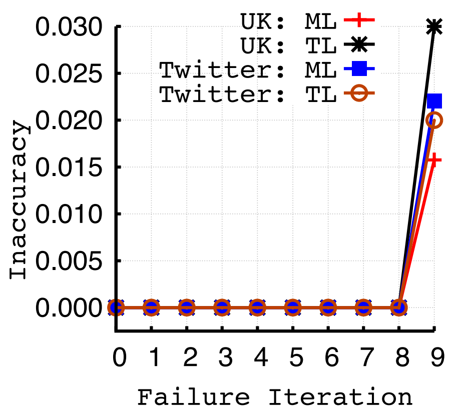
Figure 7.1(a) shows the accuracy of PageRank application on PowerGraph with varying number of failures recovered using Zorro. PowerGraph incurs *no* accuracy loss on the Twitter graph, and no accuracy loss on the UK Web graph for fewer than 6 server failures out of 16. Even with half of the servers failing, the accuracy loss in PowerGraph with the UK Web graph is only 2% of the intersection (i.e., two of the top-100 ranked vertices are not present in the results after recovering from failures), and less than 1.5% mass retained.

Figure 7.1(b) shows the accuracy of PageRank application on LFGraoh with varying number of failures recovered using Zorro. LFGraph, even with a less favorable replication model, exhibits no accuracy loss with the Twitter graph for fewer than 8 server failures out of 16. Even with half of the servers failing, the accuracy loss is at most 1% with Twitter graph and 3% with UK Web graph.

Since LFGraph exhibits out-neighbor replication, the accuracy loss with LFGraph is higher than with PowerGraph. In out-neighbor replication, only the out-neighbors of vertices are replicated at remote servers whereas in all-neighbor



(a) PowerGraph



(b) LFGraph

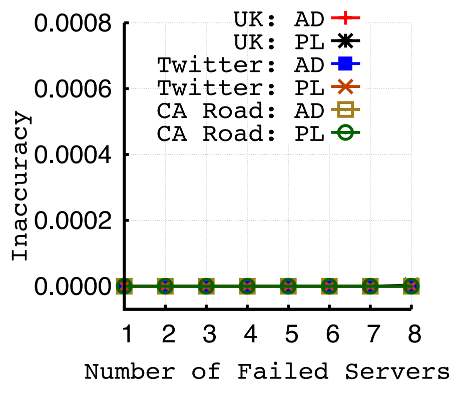
Figure 7.2: PageRank inaccuracy as a function of the iteration at which a quarter of the servers (4) fail.

replication (exhibited by PowerGraph), both in- and out-neighbors are replicated at remote servers. In LFGraph, the vertices which are lost from top-100 results are only those which have no out-neighbors and, hence, no replicas on remote servers. We expect very few vertices to have zero out-degrees contributing to high accuracy results even in LFGraph.

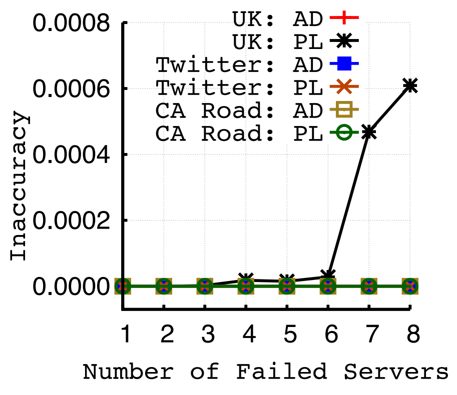
Next, we present accuracy results by varying the failure iteration (out of 10 iterations) and fixing the number of failures at 4 out of 16 servers. Figure 7.2(a) shows the accuracy of PageRank application on PowerGraph with varying the iteration at which failures occur. PowerGraph incurs no accuracy loss on the Twitter graph for failures occurring before iteration 8, and no accuracy loss on the UK Web graph. Even with a quarter of the servers failing on the last iteration, PowerGraph incurs 1% accuracy loss.

Figure 7.2(b) shows the accuracy of PageRank application on LFGraph with varying the iteration at which failures occur. LFGraph incurs no accuracy loss with both Twitter and UK Web graphs for failures occurring before iteration 9. Even with a quarter of the servers failing at the last iteration, the accuracy loss in LFGraph is at most 3% with the UK Web graph and less than 2.5% with the Twitter graph..

As failures occur at later iterations, loss of accuracy increases because those vertices whose value could not be recovered after failures are not able to reconverge within the remaining iterations. Note that, in our experiments, we fix the number of iterations and do not increase the number of iterations even after failures.



(a) PowerGraph



(b) LFGraph

Figure 7.3: SSSP inaccuracy as a function of the number of failures. Servers are failed randomly at the fifth (out of 10) iteration.

7.2.2 Single-Source Shortest Paths (SSSP)

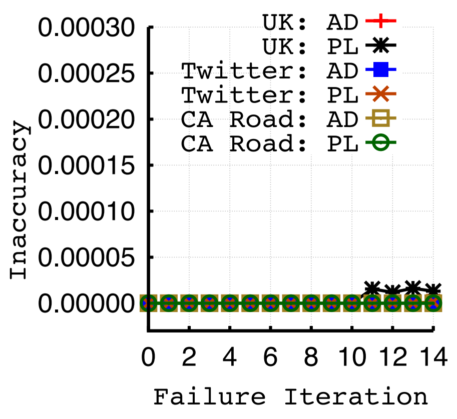
Single-source shortest paths application computes the lengths of shortest paths from a user-defined source vertex to every vertex in the graph. For our experiments, we use the vertex with the maximum degree as the source.

Metrics

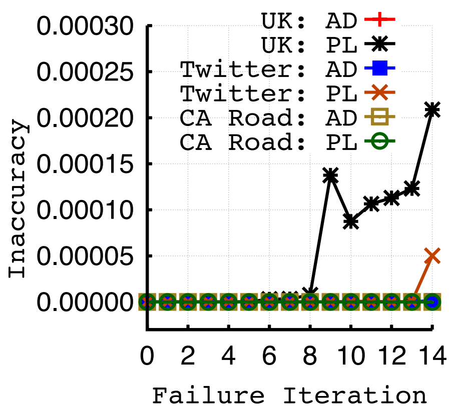
We evaluate the accuracy of SSSP application using the following metrics:

- Average Difference (AD): This metric measures the average normalized difference in the shortest path lengths of vertices [25]. Mathematically, it is represented as $\frac{1}{|V|}\sum_{v \in V}(sssp_v^n - sssp_v^o)/sssp_v^o$, where V is the set of vertices in the graph, $sssp_v^o$ is the actual shortest path length from source to vertex v, and $sssp_v^n$ is the shortest path length after recovering from failures. Average difference does not include vertices for which SSSP application could not determine a path after failures but a path actually exists. These vertices are measured using the fraction of paths lost metric.

- Fraction of Paths Lost (PL): This metric measures the fraction of vertices which are actually reachable from the given source vertex but SSSP application could not find a path to them after recovering from failures.



(a) PowerGraph



(b) LFGraph

Figure 7.4: SSSP inaccuracy as a function of the iteration at which a quarter of the servers (4) fail.

Results

We evaluate the accuracy obtained with Zorro reactive recovery by varying the number of failed servers and by varying the iteration at which failure occurs.

Figure 7.3(a) shows the accuracy of SSSP application on PowerGraph with varying number of failures recovered using Zorro. PowerGraph incurs *no* accuracy loss on the Twitter and the graph of California road network (CA Road), and no accuracy loss on the UK Web graph for fewer than 8 (out of 16) server failures. Even with half of the servers failing with the UK Web Graph, PowerGraph incurs no average difference and negligible paths lost ($3.6 \times 10^{-4}\%$ of the total).

Figure 7.3(b) shows the accuracy of SSSP application on LFGraph with varying number of failures recovered using Zorro. LFGraph exhibits no accuracy loss with the Twitter graph and the CA road graph. For the UK Web graph, LFGraph incurs no average difference and only 0.06% paths lost even with half of the servers failing.

The accuracy loss with PowerGraph is lower than that with LFGraph due to the different replication models.

Next, we present accuracy results by varying the iteration at which failures occur (out of 15 total) and fixing the number of failures at 4 out of 16 servers. Figure 7.4(a) shows the accuracy of SSSP application on PowerGraph with varying the iteration at which failures occur. PowerGraph again incurs no accuracy loss on the CA road graph. With Twitter graph, it exhibits no accuracy loss for failures occurring before iteration 14 and even with a quarter of the servers failing on the last iteration, only 0.24% paths are lost. With UK Web graph, it incurs no accuracy loss for failures occurring before iteration 11 and even with a quarter of the servers failing on the last iteration, accuracy loss is negligible (0.0013% paths lost and 0.0025%).

Figure 7.4(b) shows the accuracy of SSSP application on LFGraph with varying the iteration at which failures occur. LFGraph exhibits no accuracy loss with the CA road graph. With the Twitter graph, it incurs no accuracy loss for failures occurring before iteration 14 and even with a quarter of the servers failing on the last iteration, accuracy loss is negligible (0.005% paths lost). With the UK Web graph, it incurs at most 0.02% paths lost.

As failures occur at later iterations, inaccuracy increases as vertices whose values could not be recovered after failures are not able to re-converge within the remaining iterations.

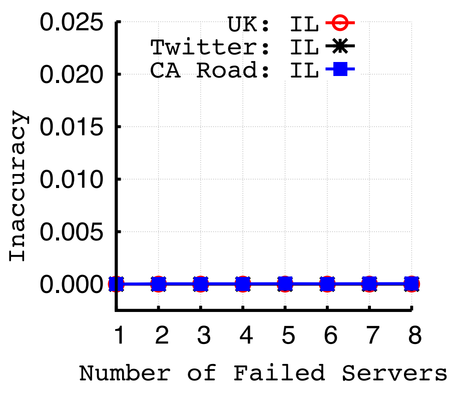
7.2.3 Connected Components (CC)

Connected Components application implements the label propagation algorithm wherein vertices propagate their component labels. We use weak connected components algorithm popular in distributed graph processing systems [38] where the label of a component is the minimum vertex ID among its member vertices. We evaluate Zorro's inaccuracy after failures while running CC with 10 iterations on all three graphs. For LFGraph, we use undirected versions of Twitter and UK Web graphs for this set of experiments.

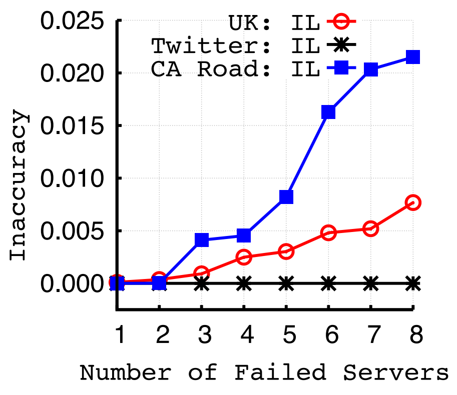
Metrics

We evaluate the accuracy of CC application using the following metric:

- Incorrect Labels (IL): The fraction of vertices which have a different label (i.e., component) than the original result. This metric evaluates the inaccuracy incurred by the algorithm in determining connected components after failures.

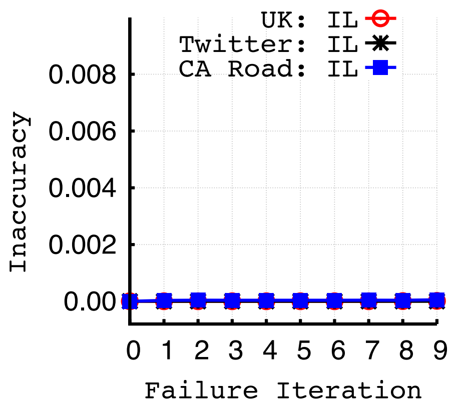


(a) PowerGraph

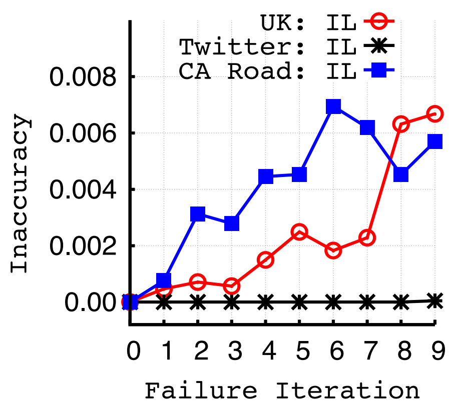


(b) LFGraph

Figure 7.5: CC inaccuracy as a function of the number of failures. Servers are failed randomly at the fifth (out of 10) iteration.



(a) PowerGraph



(b) LFGraph

Figure 7.6: CC inaccuracy as a function of the iteration at which a quarter of the servers (4) fail.

Results

We evaluate the accuracy obtained with Zorro reactive recovery by varying the number of failed servers and by varying the iteration at which failure occurs.

Figure 7.5(a) shows the accuracy of CC application on PowerGraph with varying the number of failures. PowerGraph incurs negligible accuracy loss for the UK Web and Twitter graphs even with half of the servers failing (at most 0.02% and 2.4×10^{-6}% with UK Web and Twitter respectively). With CA road graph, it incurs an accuracy loss of 1.57% with 8 servers failing while only 0.13% with 4 servers failing. The accuracy loss is higher in CA road network because of its high diameter where vertices take much longer to converge.

Figure 7.5(b) shows the accuracy of CC application on LFGraph with varying the number of failures. LFGraph incurs negligible accuracy loss ($2.4 \times 10^{-5}\%$) for the Twitter graph even with half of the servers failing. With 8 servers failing, the accuracy loss with UK Web graph is 0.77% and with CA road graph, it is 2.15%.

Next, we present accuracy results by varying the iteration at which failures occur (out of 10 total) and fixing the number of failures at 4 out of 16 servers. Figure 7.6(a) shows the accuracy of CC application on PowerGraph with varying the iteration at which failures occur. PowerGraph incurs negligible accuracy loss for the UK Web and Twitter graphs even with failures occurring at the last iteration (at most 0.0015% and $2.86 \times 10^{-4}\%$ with UK Web and Twitter respectively). With CA road graph, it incurs an accuracy loss of 0.17% with 8 servers failing while only 0.13% with 4 servers failing.

Figure 7.6(b) shows the accuracy of CC application on LFGraph with varying the iteration at which failures occur. LFGraph incurs negligible accuracy loss (0.005%) for the Twitter graph even with half of the servers failing. With 8 servers failing, the accuracy loss with UK Web graph is 0.67% and with CA road graph, it is 0.57%.

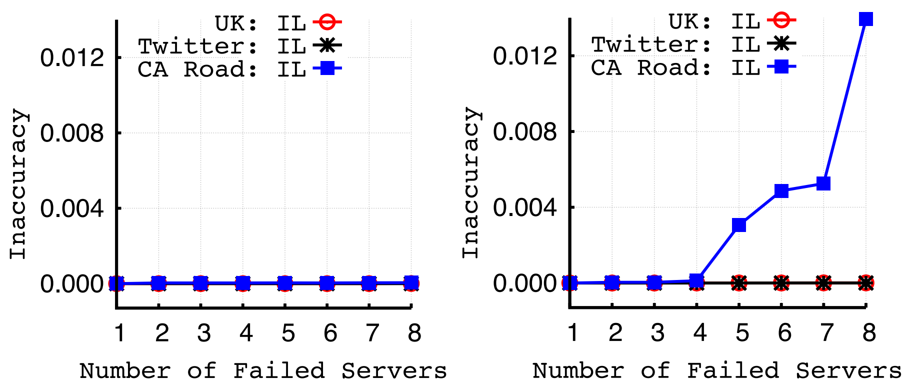
7.2.4 K-core

K-core application [48] identifies sub-graphs within a given graph such that vertices in the induced sub-graphs have at least *k* neighbors. We evaluate Zorro's inaccuracy after failures while running k-core decomposition with 10 iterations on all three graphs. For LFGraph, we use undirected versions of Twitter and UK Web graphs for this set of experiments.

Metrics

We evaluate the accuracy of k-core application using the following metric:

- Incorrect Labels (IL): The fraction of vertices which have a different label (i.e., k-core membership) than the original result. This metric evaluates the inaccuracy incurred by the algorithm in determining k-core membership after failures.



(a) PowerGraph (b) LFGraph

Figure 7.7: K-Core inaccuracy vs. the number of failures. Servers are failed randomly at the fifth (out of 10) iteration.

Results

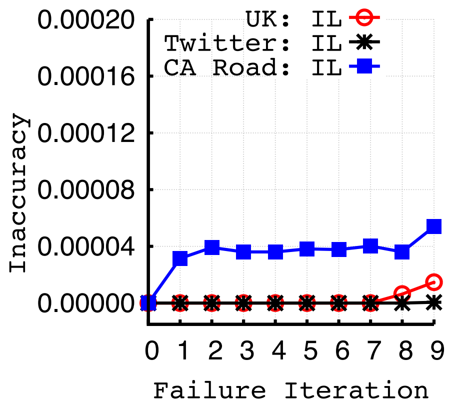
We evaluate the accuracy obtained with Zorro reactive recovery by varying the number of failed servers and by varying the iteration at which failure occurs.

First, we present results by varying the number of failed servers. Figure 7.7(a) shows the accuracy of k-core application on PowerGraph with varying the number of failures. PowerGraph incurs no accuracy loss for the Twitter graph. Even with 8 servers failing, it incurs negligible accuracy loss with the UK Web ($6.03 \times 10^{-4}\%$) and CA road graph (0.0047%).

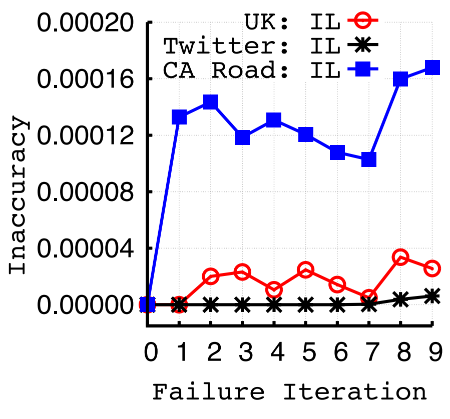
Figure 7.7(b) shows the accuracy of k-core application on LFGraph with varying the number of failures. LFGraph incurs $5.58 \times 10^{-4}\%$ accuracy loss with the UK Web graph, $3.05 \times 10^{-4}\%$ with the Twitter graph and 1.39% with the CA road graph with half of the servers failing.

Next, we present results by varying the iteration at which failures occur. Figure 7.8(a) shows the accuracy of k-core application on PowerGraph with varying the iteration at which failures occur. PowerGraph incurs negligible accuracy loss for all three graphs even with failures occurring on the last iteration (0.0015% for UK Web, $5.52 \times 10^{-5}\%$ for Twitter and 0.0053% for CA road).

Figure 7.8(b) shows the accuracy of k-core application on LFGraph with varying the iteration at which failures occur. LFGraph incurs negligible accuracy loss for all three graphs even with failures occurring on the last iteration (0.0026% for UK Web, $6.2 \times 10^{-4}\%$ for Twitter and 0.017% for CA road).

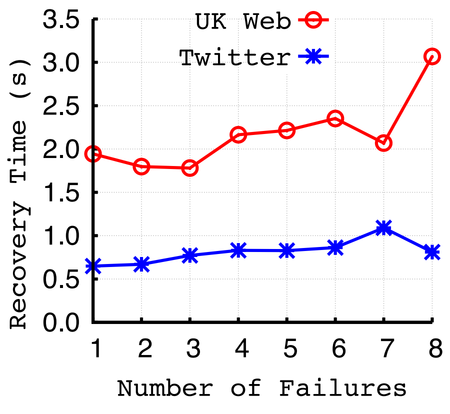


(a) PowerGraph

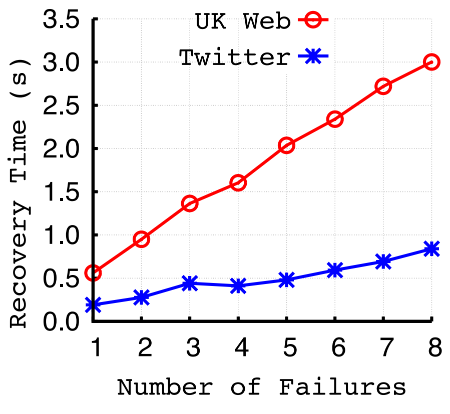


(b) LFGraph

Figure 7.8: K-Core inaccuracy as a function of the iteration at which a quarter of the servers (4) fail.



(a) PowerGraph.

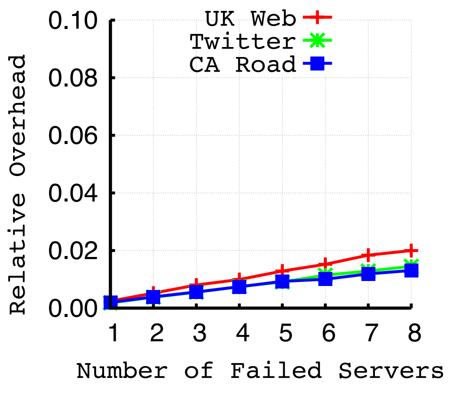


(b) LFGraph.

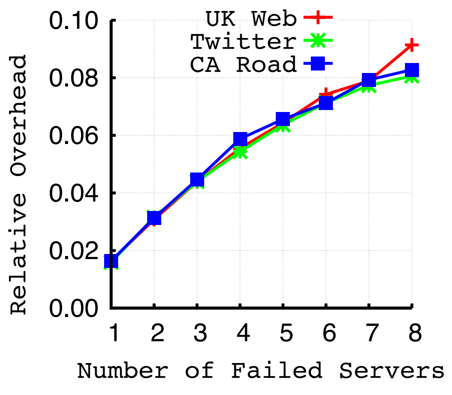
Figure 7.9: Recovery time as a function the number of failures.

7.3 Recovery Time

In this experiment, we evaluate the recovery time incurred by graph processing applications with Zorro reactive failure recovery. Figure 7.9(a) and Figure 7.9(b) shows recovery time as a function of the number of failed servers in PowerGraph and LFGraph. For PowerGraph, we observe that the recovery time does not vary significantly with the number of failed servers demonstrating the scalability of the protocol with the number of failed servers. For LFGraph, we observe a linear increase in recovery time as the number of failed servers increases. This is because of the partial scatter phase that needs to be performed after recovery among replacement servers. For both frameworks, the recovery time with the UK Web



(a) PowerGraph.



(b) LFGraph.

Figure 7.10: Ratio of bytes transferred over 10 iterations of SSSP to bytes transferred during failure recovery by Zorro as a function the number of failures.

graph is more than that with the Twitter graph because of its larger size. More graph state is transferred over the network with the UK Web graph than with the Twitter graph. Most importantly, the recovery time in both PowerGraph and LFGraph is less than the average iteration time. The average iteration time in PowerGraph is 11.7 seconds with the Twitter graph and 22 seconds with the UK Web graph while in LFGraph it is 2 seconds with the Twitter graph and 6 seconds with the UK Web graph.

The low recovery times with Zorro reactive recovery protocol are because of the ability of servers to load graph partitions while receiving replicated state from surviving servers in parallel. This requires to store the state transferred by surviving servers in a temporary datastore before merging it with the loaded graph. The low recovery times demonstrate the ability of Zorro reactive recovery protocol to enable quick recovery.

7.4 Communication Overhead

In this section, we evaluate the network communication overhead incurred by Zorro recovery protocol. Figure 7.10 shows the ratio of the bytes transferred during recovery with Zorro to the total bytes transferred during failure-free execution of 10 iterations of PageRank application. This relative network overhead in PowerGraph (Figure 7.10(a)) is at most 2% while it is at most 9.15% in LFGraph with half of the servers failing. The relative overhead is smaller in PowerGraph than LFGraph because of the rebuild optimization possible in PowerGraph model where in only a single surviving server is responsible for transferring the state of a vertex required by the replacement servers. This is achieved by using the following check for each vertex:

$$procid == \underset{p \in P_S(v)}{\operatorname{argmin}} \, |p.procid - (v.id \ \% \ |S|)\,| \tag{7.1}$$

,where procid is the process id of the server, $P_S(v)$ is the set of surviving servers that hold vertex v either as the master or a mirror and S is the set of all servers in the cluster.

This optimization is not possible in LFGraph as the locations of the replicas of a vertex are not available.

Chapter 8

FUTURE WORK

Distributed GraphLab [33] and PowerGraph [23] provide an asynchronous computation model. Asynchronous computation may benefit iterative machine learning applications like Alternating Least Squares (ALS) [27]. One immediate direction for future work is the application of Zorro reactive recovery protocol for asynchronous computation in PowerGraph. Another important direction is to study the feasibility of Zorro-based failure recovery in GraphX [24] to assist its default recovery using the lineage graph. Yet another direction of future work is to study the application of Zorro for applications that use delta-updates between iterations.

In the current design, Zorro assumes that all failed servers are replaced before graph processing continues. Studying elasticity techniques to enable the framework to scale-out/in graph computation on failures depending depending upon the availability of replacement servers is an important future direction. After failures, different applications may require different initialization functions for the vertices whose state could not be recovered. To this end, a graph processing framework could allow users to define a re-initialization function based on their needs. On a theoretical side, it is important to study the roles different vertices play in the convergence of graph applications. The studied properties can be used to enhance reactive recovery mechanisms in distributed graph processing frameworks.

Finally, reactive failure recovery mechanisms offer a cheap and effective alternative to expensive proactive recovery mechanisms. Studying their application in other distributed data processing frameworks like stream processing frameworks [54] [60] is an important future direction.

Chapter 9

CONCLUSION

In this thesis, we have shown that reactive failure recovery mechanisms provide a cheap, useful and accurate alternative to proactive recovery mechanisms. We presented Zorro, a zero cost reactive recovery protocol which recovers from any number of independent and cascading failures in distributed graph processing frameworks. By allowing replacement servers to rebuild their state using inherently replicated state in distributed graph processing, Zorro enables failure recovery which maintains very high levels of accuracy.

BIBLIOGRAPHY

[1] Apache Giraph. `http://giraph.apache.org/`.

[2] Apache GraphX. `https://spark.apache.org/graphx/`.

[3] Apache Hadoop. `https://hadoop.apache.org/`.

[4] Apache Hama. `https://hama.apache.org/`.

[5] Apache Spark. `https://spark.apache.org`.

[6] Stanford Network Analysis Project. `http://snap.stanford.edu/`.

[7] A.-L. Barabsi and R. Albert. Emergence of Scaling in Random Networks. In *Science*, 1999.

[8] B. Bhargava and S.-R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems-An Optimistic Approach.

[9] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the International Conference on World Wide Web (WWW)*. ACM, 2011.

[10] P. Boldi and S. Vigna. The WebGraph Framework I: Compression Techniques. In *Proceedings of the International World Wide Web Conference (WWW)*. ACM, 2004.

[11] M. Bota, H.-W. Dong, and L. W. Swanson. From gene networks to brain networks. In *Nature neuroscience*, 2003.

[12] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Computer networks and ISDN systems*, 1998.

[13] R. H. Campbell and B. Randell. Error Recovery in Asynchronous Systems. *Transactions on Software Engineering*, 1986.

[14] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. In *Transactions on Computer Systems (TOCS)*. ACM, 1985.

[15] R. Chen, J. Shi, Y. Chen, H. Guan, B. Zang, and H. Chen. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[16] A. Ching. Scaling apache giraph to a trillion edges. *Facebook Engineering blog*, 2013.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2005.

[18] J. Dean. Experiences with mapreduce, an abstraction for large-scale computation. In *PACT*, 2006.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Communications of the ACM*. ACM, 2008.

[20] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys (CSUR)*, 2002.

[21] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl. Iterative Parallel Data Processing with Stratosphere: An Inside Look. In *Proceedings of International Conference on Management of Data (SIGMOD)*. ACM, 2013.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SIGOPS operating systems review*. ACM, 2003.

[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2012.

[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2014.

[25] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and Accurate Estimation of Shortest Paths in Large Graphs. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. ACM, 2010.

[26] I. Hoque and I. Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *Proceedings of Conference on Timely Results In Operating Systems (TRIOS)*. ACM, 2013.

[27] Y. Hu, Y. Koren, and C. Volinsky. Collaborative Filtering for Implicit Feedback Datasets. In *Proceedings of the International Conference on Data Mining (ICDM)*. IEEE, 2008.

[28] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the Annual Technical Conference (ATC)*. USENIX, 2010.

[29] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems using Asynchronous Message Logging and Checkpointing. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*. ACM, 1988.

[30] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the International Conference on Data Mining (ICDM)*. IEEE, 2009.

[31] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of International Conference on World Wide Web (WWW)*. ACM, 2010.

[32] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*. ACM, 2012.

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of VLDB Endowment*, 2012.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[35] A. Lowry, J. R. Russell, and A. P. Goldberg. Optimistic Failure Recovery for Very Large Networks. In *Proceedings of the Symposium on Reliable Distributed Systems*. IEEE, 1991.

[36] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. In *Parallel Processing Letters*, 2007.

[37] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. 2015.

[38] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of International Conference on Management of Data (SIGMOD)*. ACM, 2010.

[39] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis. FrogWild!–Fast PageRank Approximations on Graph Engines. In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, 2014.

[40] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *Proceedings of the International conference on Supercomputing (SC)*. ACM, 2007.

[41] D. Ongaro, S. M. Rumble, R. Stutsman, and J. Ousterhout. Fast crash recovery in RAMCloud. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*. ACM, 2011.

[42] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[43] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin. Scalable big graph processing in mapreduce. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014.

[44] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems (TOCS)*. ACM, 1992.

[45] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

[46] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *Proceedings of International Conference on Scientific and Statistical Database Management*. ACM, 2013.

[47] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All Roads lead to Rome: Optimistic Recovery for Distributed Iterative Data Processing. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)*. ACM, 2013.

[48] S. B. Seidman. Network structure and minimum degree. *Social networks*, 1983.

[49] Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor. Fast Failure Recovery in Distributed Graph Processing Systems. In *Proceedings of the VLDB Endowment*, 2015.

[50] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[51] S. W. Smith, D. B. Johnson, and J. Tygar. Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. In *Proceedings of the International Symposium on Fault-Tolerant Computing*. IEEE, 1995.

[52] S. H. Strogatz. Exploring Complex Networks. *Nature*, 410, 2001.

[53] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, 1985.

[54] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm @ Twitter. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2014.

[55] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based Fault-tolerance for Large-scale Graph Processing. In *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2014.

[56] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic Gradient Boosted Distributed Decision Trees. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*. ACM, 2009.

[57] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. In *Communications of the ACM*. ACM, 1974.

[58] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of Conference on Networked Systems Design and Implementation(NSDI)*. USENIX, 2012.

[59] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of Conference on Hot topics in Cloud Computing*. USENIX, 2010.

[60] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.