# Zorro: Zero-Cost Reactive Failure Recovery in Distributed Graph Processing

Mayank Pundir, Luke M. Leslie, Indranil Gupta and Roy H. Campbell
University of Illinois at Urbana-Champaign
{pundir2, lmlesli2, indy, rhc}@illinois.edu

## Abstract

Distributed graph processing systems largely rely on proactive techniques for failure recovery. Unfortunately, these approaches (such as checkpointing) entail a significant overhead. In this paper, we argue that distributed graph processing systems should instead use a reactive approach to failure recovery. The reactive approach trades off completeness of the result (generating a slightly inaccurate result) while reducing the overhead during failure-free execution to zero. We build a system called Zorro that imbues this reactive approach, and integrate Zorro into two graph processing systems – PowerGraph and LFGraph. When a failure occurs, Zorro opportunistically exploits vertex replication inherent in today's graph processing systems to quickly rebuild the state of failed servers. Experiments using real-world graphs demonstrate that Zorro is able to recover over 99% of the graph state when 6-12% of the servers fail, and between 87-95% when half the cluster fails. Furthermore, using various graph processing algorithms, Zorro incurs little to no accuracy loss in all experimental failure scenarios, and achieves a worst-case accuracy of 97%.

***Categories and Subject Descriptors*** C.2.4 [*Computer Systems Organization*]: Distributed Systems

## 1. Introduction

Distributed graph processing systems are widely employed to process large graphs, including online social networks [22], web graphs [5, 6], and biological networks [7]. Such graphs may comprise billions of vertices and trillions of edges [10]. Consequently, distributed graph processing systems are often run on large clusters [27]. Examples of distributed graph processing systems include Pregel [27], Giraph [1], PowerGraph [15], LFGraph [19] and GPS [35].

However, despite their widespread and increasing popularity, existing systems for distributed graph processing offer recovery after server failures only through the use of expensive proactive mechanisms such as checkpointing [15, 24, 27, 31, 35]. These checkpointing-based failure recovery mechanisms periodically and synchronously save the global *graph state*, consisting of application-specific values associated with vertices and/or edges. For instance, PowerGraph, Pregel and Giraph offer the option for each server to periodically save a synchronous snapshot of its graph partition to reliable storage such as HDFS. After failure, the most recent snapshot is used to rebuild the last persisted graph state.

Despite the success of checkpoint-based failure recovery mechanisms in storage [14, 33] and virtualization systems [11, 29], we find that proactive recovery mechanisms incur unnecessary and expensive overhead during common-case failure-free processing. Figure 1 illustrates that turning on checkpointing in PowerGraph [15] (running PageRank) incurs an $8-31\times$ increase in per-iteration time – the larger the graph, the higher was the overhead. This overhead is largely because checkpointing incurs periodic and excessive I/O, and is particularly prohibitive given the large mean time between failures of a machine in modern clusters (e.g., 360 days [24]). As a result, many users prefer to disable failure recovery mechanisms and simply restart computation [16].

In this paper, we argue that distributed graph processing systems should instead adopt a reactive approach to failure recovery. Achieving this in practice requires several challenges to be met. First, a reactive approach does not prepare for failures, and hence can only use information available *after* failure has occurred. Second, failures should be allowed to occur at any time during computation without resulting in inconsistencies during recovery. Third, failures should be allowed to also occur during recovery itself (*cascading failures*) without interfering with ongoing recovery. Finally, the recovery mechanism must be fast and use few resources.

The reactive approach trades off completeness of the result (generating slight inaccuracy) while eliminating overhead during failure-free execution. We build a failure recov-


*SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA,* .

http://dx.doi.org/10.1145/2806777.2806934

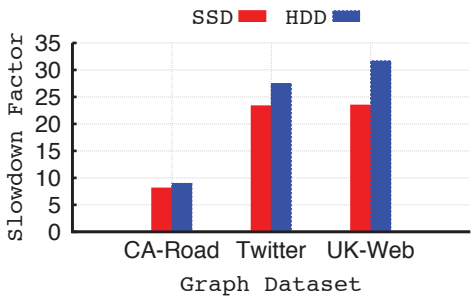



**Figure 1.** Per-iteration checkpointing slowdown with 16 servers (using SSDs and HDDs), for the graphs in Table 1.

ery mechanism called Zorro that realizes this reactive philosophy, and we integrate Zorro into two graph processing systems – PowerGraph [15] and LFGraph [19]. Zorro does not prepare for server or network failures and incurs essentially zero additional cost during failure-free execution (at most 0.8% added execution time). When failure occurs, Zorro opportunistically exploits vertex replication inherent in today's graph processing systems to quickly and consistently rebuild the state of failed servers. Our primary finding is that this existing level of replication (a function of the system and graph structure) is sufficient to achieve high accuracy after failure.

Experiments using real-world graphs containing billions of edges demonstrate that Zorro is able to quickly recover over 99% of the graph state when 6-12% of the servers fail, and between 87-95% when half the cluster fails. We further evaluate Zorro's performance with various graph processing algorithms, including PageRank, single-source shortest paths, connected components, and k-core decomposition. Zorro incurs little to no accuracy loss in all experimental failure scenarios, and achieves a worst-case accuracy of 97%.[1]

During the design of Zorro, we explored a few alternative (and simpler) options for reactive failure recovery. We describe why these are untenable, and why Zorro's design is essential the way it is. One option after failure was to restart the entire computation from scratch. However, this incurs the overhead of reloading the graph state, wastes work, and prolongs completion time – the effect is particularly bad if the failure occurs later in the computation. Further, this overhead is incurred on every failure occurrence. The second option we considered was to let failed servers be replaced, and merely let the computation continue. This version is not available in today's graph processing systems. So, for comparison purposes, we implemented a working version into PowerGraph and LFGraph – unfortunately, this option gave much lower accuracy than Zorro. In particular, the inaccuracy was 25% for PageRank on PowerGraph and 51% on LFGraph with this option, vs. 0% inaccuracy using Zorro. Section 6.4 expands on the trade-offs between various failure recovery mechanisms (including restarting computation).

[1] We evaluate Zorro's performance with additional algorithms and graphs in an extended technical report [32]. Results are similar in all cases.

## 2. Background and Motivation

In this section, we first give an overview of distributed graph processing systems, and then discuss challenges and limitations of existing failure recovery mechanisms.

### 2.1 Distributed Graph Processing Systems

A distributed graph processing system performs computation on a graph partitioned among a set of servers.

***Partitioning:*** Distributed graph partitioning can take the form of either *vertex* or *edge partitions*, where vertices or edges are uniquely assigned to servers, forming a *local subgraph* at each. Although existing frameworks offer intelligent mechanisms to partition graphs (with the aim of reducing communication overhead), recent studies [19] have demonstrated that such mechanisms can occupy up to 80% of processing time, and therefore should be abandoned in favor of cheap hash-based partitioning.

***Computation:*** The synchronous, vertex-centric Gather-Apply-Scatter (GAS) decomposition is the most common graph computation model, and is supported by most popular systems (e.g., [1, 2, 15, 19]). Computation in GAS occurs in iterations (also called supersteps), wherein vertices *Gather* values from neighbors, aggregate and *Apply* the values, and then *Scatter* the results to neighbors. Depending on the system and algorithm, computation within an iteration may be restricted to only active vertices. We define the *vertex state* to be a vertex's most recent applied value.

***Communication:*** Partitioning the graph across servers requires vertex states to be propagated over the network to neighbors at remote servers. Different systems implement separate ways of performing this communication but, as we will discuss in Section 2.4, all approaches introduce a level of vertex state replication.

***Failure Recovery:*** We define *failure recovery* in distributed graph processing systems as the recovery of all vertex states to the iteration from just before failure occurrence. We define *state loss* as all vertex states that must be recomputed.

The most common mechanisms for failure recovery are checkpoint-based [15, 24, 27, 31]. These approaches offer the ability to recover the entire *distributed graph state* after failure by periodically saving each node's *local subgraph state* to reliable storage such as HDFS [39]. Failed nodes are replaced and all nodes load their subgraph state from the most recent checkpoint. Lost progress from iterations after checkpoint creation must be recomputed.

### 2.2 Desirable Properties for Failure Recovery

***Scope:*** We assume graph processing follows the synchronous GAS decomposition. We consider only fail-stop failures. One or more servers may fail simultaneously or in a cascading manner. Network failures such as rack outages are special cases of this failure model. We do not consider high message loss rates or Byzantine failures.

***Desirable Properties:*** Under failure-prone executions, there are three desirable properties of failure recovery mechanisms for distributed graph processing systems:

ZO (**Zero Overhead**): No overhead is incurred during failure-free execution.

CR (**Complete Recovery**): Results in the face of failures are fully accurate.

FR (**Fast Recovery**): Recovery after failure is quick and does not require additional iterations.

We note that we consider only the results of a graph processing application, rather than full recovery of all vertex states.

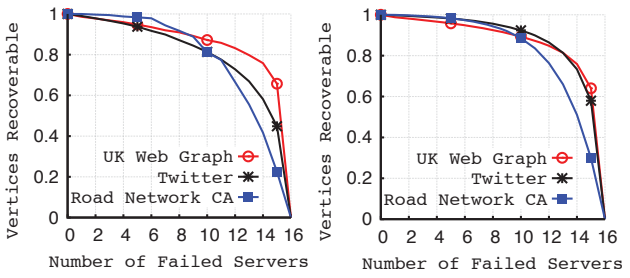
It is difficult to fully satisfy all three properties in a distributed graph processing failure recovery mechanism. To see why, consider a system that does not checkpoint dynamic graph state. When machine failures occur, the in-memory state of the graph application will be incomplete, potentially violating CR (in our experiments, this option gave 25-51% inaccuracy). On the other hand, without *a priori* knowledge of failure occurrence, the only option is to proactively checkpoint the in-memory graph state – however, this incurs high overhead (Figure 1) and violates ZO. Existing mechanisms for failure recovery strive to achieve completeness (CR) over zero-overhead (ZO). In this paper, we demonstrate that it is possible to achieve FR and ZO without sacrificing application accuracy by too much (i.e., achieving *almost*-CR).

### 2.3 Limitations of Checkpointing

Despite the success of checkpoint-based failure recovery mechanisms in storage [14, 33] and virtualization systems [11, 29], two serious issues arise with checkpoint-based recovery in distributed graph processing. First, the process of synchronous snapshot determination and checkpointing can incur high runtime overhead, resulting in significant execution delays for relatively short-lived graph computations. As demonstrated in Figure 1, per-iteration checkpointing slowdown in our experiments ranges from $8-31\times$. Second, recovery from a checkpoint requires all progress from the interval between checkpointing and failure to be recomputed (potentially comprising many repeated iterations).

The two issues above jointly imply another drawback: significant user involvement. If the checkpointing interval for an application is low, most checkpoints will be unused due to comparably high mean time between failures (MTBF). On the other hand, high checkpointing intervals make it likely that either no checkpoint will exist after failure, or that it will be stale and result in lengthy recomputation. For example, with a single-server MTBF of 360 days [24], the MTBF of a cluster of 16 servers is 22 days.

Young's model [42] provides an approximation for the optimal checkpointing interval as $t^* = \sqrt{2t_c \cdot t_{MTBF}}$, where $t_c$ is the checkpointing time and $t_{MTBF}$ is the per-server MTBF [24, 27]. In our experiments, PowerGraph takes on average $t_c = 493.81$ seconds to create a *single* checkpoint of




(a) Giraph/LFGraph/Hama (b) PowerGraph

**Figure 2.** Fraction of recovered state as a function of the number of failed servers (out of 16) for the graphs in Table 1.

| Dataset | Edges | Vertices |
|---|---|---|
| (E) CA Road Network [3] | 2,766,607 | 1,965,206 |
| (P) Twitter [22] | 1,468,365,182 | 41,652,230 |
| (P) UK Web Graph [5, 6] | 3,738,733,648 | 105,896,555 |

**Table 1.** Graph datasets. (E) represents an exponential graph, and (P) a powerlaw graph.

PageRank running on the UK Web graph [5, 6] partitioned across 16 servers with SSDs. Using Young's model, the optimal checkpointing interval in the scenario above turns out to be 12 hours, which can far exceed typical application execution times ($\sim$ 22 seconds per iteration in this scenario).

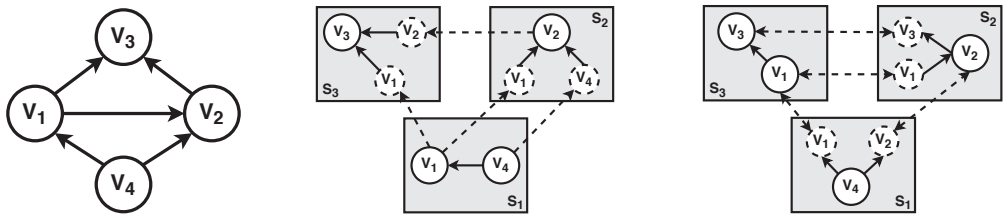
Recent work is consistent with our concerns. For example, the authors of GraphX [16] mention that most users of distributed graph processing systems leave checkpointing disabled due to performance overheads. Even the authors of Distributed GraphLab [24] state users must explicitly balance failure recovery costs against restarting computation.

### 2.4 Replication in Existing Systems

In order to realize zero overhead during the common case of failure-free executions, we must adopt a reactive approach for when failures do occur. However, since we do not *a priori* replicate vertex state, we are forced to rely opportunistically on replication that the underlying system already provides.

In this section, we argue that existing graph processing systems provide sufficient replication of vertices for real-world graphs, thus making our reactive approach feasible. Below, we classify popular systems into two classes based on their replication methodology. We refer to a vertex state replica as a copy of the vertex state created on a remote server by the communication model of the system.

1. **Out-Neighbor Replication**: In this class of systems, vertex states are replicated at out-neighbors on remote servers (see Figure 3(b)). Concretely, replication of a vertex $v$'s state exists at servers containing $v$'s out-neighbors. These systems can be further divided into two subclasses as a function of how these replicas are maintained:




(a) Example graph. (b) Vertex replication in LFGraph. (c) Vertex replication in PowerGraph.

**Figure 3.** Vertex replication in the two system classes, for a graph partitioned across three servers using consistent hashing.

(a) *Message-Based*: E.g., Pregel [27], Giraph [1] and Hama [2]. Each vertex is assigned to one server and maintains its out-edges. Replication of a vertex $v$'s state exists as buffered messages received from $v$ during the previous iteration, at $v$'s out-neighbors.

(b) *Value-Based*: E.g., LFGraph [19]. Each vertex is hashed to one server and maintains its in-neighbors and their states. Each server maintains updated neighbor values of local vertices. Updated vertex states are sent to servers containing out-neighbors of the vertex.

2. **All-Neighbor Replication**: E.g., PowerGraph [15] and its predecessor, Distributed GraphLab [24]. In PowerGraph, each edge is assigned to one server (see Figure 3(c)) – thus, each vertex is present on all servers which store adjacent edges. For vertices having edges on multiple servers, one replica is labeled as the master while others are labeled as mirrors. Replication may thus occur at all remote neighbors. In each iteration, mirrors send the local results of Gather to the master, which combines them and synchronizes the result with mirrors.

We exclude centralized graph processing systems (e.g., [23, 26, 34]). GraphX [16] provides failure recovery using lineages from the RDD abstraction, with optional support for checkpointing in the case of long lineage chains [43]. An extension of Zorro to GraphX is left as future work.

Figures 2(a) and 2(b) illustrate the fraction of recoverable vertex states in out- and all-neighbor replication frameworks, respectively. The replication models of both classes allow recovery of a large fraction of the vertex states – half the servers failing still results 87-95% of vertices recovered. As we will demonstrate in this paper, Zorro is able to achieve little to no inaccuracy in popular graph algorithms using this recovered state, even in the face of high numbers of failures.

## 3. Zorro Design

In this section, we present Zorro, a general protocol for zero overhead reactive failure recovery in distributed graph processing systems. Zorro gives preference to the zero overhead (ZO) characteristic of an ideal failure recovery mechanism and, as such, does not add overhead during failure-free execution. Rather, after a failure occurs, Zorro reactively kicks in and executes the following stages:

R1 (**Replace**): After failure, each failed server is replaced by a new server – we call these *replacement servers*. Replacement servers start with zero state.

R2 (**Rebuild**): Each replacement server collects relevant state information from all surviving servers and rebuilds its local state.

R3 (**Resume**): After all replacement servers have finished rebuilding their local states, all servers resume computation from the start of the iteration in which failure occurred.

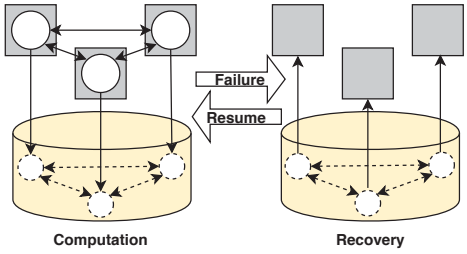
As we will discuss in Section 3.4, stage R1 may be nested inside R2 in order to handle failures during recovery. Figure 4 contrasts proactive and reactive recovery mechanisms. Proactive failure recovery mechanisms (Figure 4(a)) periodically save the graph state to persistent storage during computation. After failure, servers initialize from the checkpoint. In contrast, reactive failure recovery mechanisms (Figure 4(b)) do not persist state during computation. Rather, after server failures, replacements initialize their local subgraph from persistent storage and receive states from survivors.

### 3.1 Replacing Failed Servers

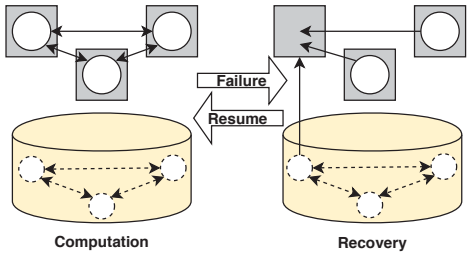
After failure is detected, survivors suspend computation, retain the local subgraph state in memory, and wait for replacements to rejoin. We assume the presence of a membership service that detects failures and informs the surviving servers. Such mechanisms are already running inside today's graph processing systems, e.g., ZooKeeper is supported by PowerGraph [15] and LFGraph [19], heartbeating mechanisms are used in Pregel [27], Giraph [1] and Hama [2], etc. In particular, the synchronous nature of the execution (per iteration) requires the use of barriers, which rely on a membership service by design. Thus, Zorro's use of a membership service does not add extra overhead.

### 3.2 Rebuilding Local Subgraph States

We refer to the *local subgraph state* of distributed graph processing systems as the vertex values of locally-hosted vertices, as well as the replicated values of remote neighbors.




(a) Proactive checkpointing-based recovery.




(b) Reactive replication-based recovery.

**Figure 4.** Proactive vs. reactive failure recovery.

In the most general case, Zorro has survivors send all local vertex states which are required to rebuild the replacement's local subgraph state. Replacement servers receive vertex state data in parallel with initialization (e.g., loading graph partitions from persistent storage), and apply the received values afterward. As a result, the Rebuild stage of each replacement is independent and concurrent across survivors; this facilitates recovery in the case of scenarios such as cascading failures (discussed in Section 3.4). Furthermore, GAS helps ensure consistency of state information due to the enforcement of synchronicity via barriers. We later prove in Theorem 1 (Section 4.1) that, under the assumption of hash-based partitioning, the number of vertex states recovered during Rebuild is independent of the cluster size, and depends instead on the fraction of servers that fail.

As an example of general Rebuild, consider the failure of server $s_3$ in Figure 3(c). The values of vertex $v_1$ (replicated at servers $s_1$ and $s_2$) and $v_3$ (replicated at server $s_2$) are sent to the replacement server, and are thus both recovered.

We quantify the overhead during Rebuild in Section 6.2.2. In our experiments, state transfer during Rebuild is masked by concurrent graph initialization, (the resulting overhead is a fraction of the cost of a single iteration). We note that in systems such as PowerGraph, edges must be distributed among servers during graph loading, and recovery may thus introduce extra overhead. To help mitigate this, we discuss system-specific optimizations to eliminate redundant value transfer and balance network overhead in Section 5.

### 3.3 Resuming Computation

Recall that the Scatter stage in GAS involves notifying neighbors about updates. After failure recovery, the messages available from the previous Scatter will be unavailable at replacement servers. Therefore, for execution correctness, Zorro performs a *partial Scatter* stage after recovering from failures. In systems such as PowerGraph, this stage of Zorro is entirely local and merely involves transferring the value of a vertex to local neighbors. In other systems such as LFGraph and Giraph, the additional communication overhead is incurred only among replacement servers, and is less than that of a normal Scatter stage during computation.

After the partial Scatter, Zorro resumes computation from the start of the iteration during which failure occurred. Zorro either sends the iteration number through networking channels, or stores it on the membership service after failure detection. Synchronicity ensures that only a single value will be available for each vertex at the end of Resume.

### 3.4 Cascading Failures

We define *cascading failures* as failures that occur while the system is recovering from a previous failure. Handling cascading failures can be a difficult task due to the interleaving of recovery stages and failures. However, Zorro utilizes the independence guarantees of its generalized Rebuild to significantly alleviate issues typically associated with recovery subject to cascading failures.

Zorro treats cascading failures during either the Replace or Resume stages in the same fashion as failures during execution. When failures occur during Rebuild, Zorro performs nested Replace stages alongside vertex state transfer to existing replacements. To prevent a scenario where all servers fail during recovery and no progress can be made, existing replacements also assist in Rebuild by sending back any previously-received vertex states to new replacements.

For example, in Figure 3(c) consider the case where the system 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 during recovery, the Rebuild of $s_3$ from server $s_2$ is unaffected and $s_3$ assists in the rebuild of $s_1$ by sending back $v_1$'s state.

### 3.5 Recovery Flow

An example run of a graph processing system during failure recovery using Zorro is illustrated in Figure 5. Events are numbered in order of our discussion below.

For this example, we assume Zorro manages membership lists through a membership service (MS in the figure) such as ZooKeeper. Upon detecting failures (1), the MS issues a callback (`leave_cb`) to surviving servers (2), after which the surviving servers suspend computation and wait for the replacement servers to reload their graph partitions. Graph reloading after failure rehashes the graph as during initialization, but rebuilds only the partitions on replacement servers.

The replacement server joins the cluster by notifying the MS (3), which then issues a callback (`join_cb`) to survivors (4). After receiving a join callback, surviving servers send

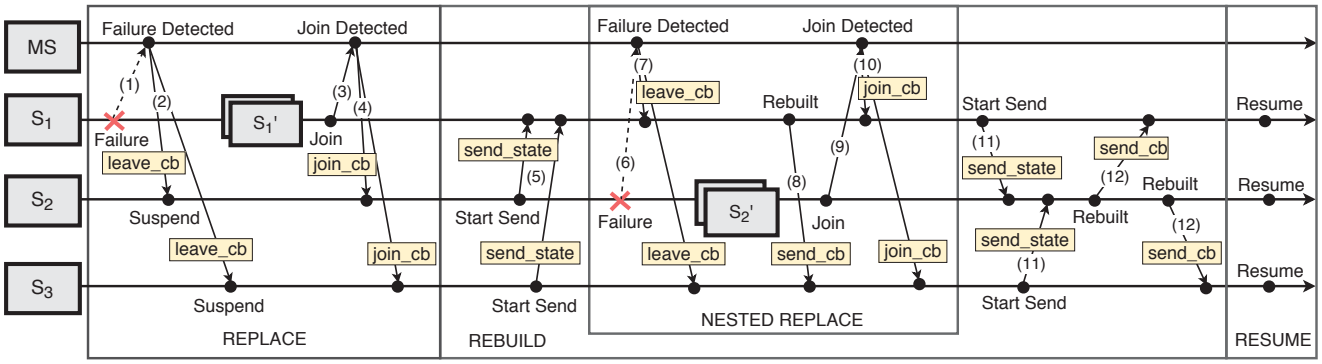



**Figure 5.** Zorro reactive recovery protocol under a cascading failure.

the most recent state of vertices (`send_state`) hosted on replacement servers if replicas are locally available (5). Now, as the cluster is recovering from the failure of server $s_1$, another server ($s_2$) fails. This failure of server $s_2$ is detected by MS (6) and the surviving servers are informed about the failure using a callback (`leave_cb`) (7). We note that the failure of server $s_2$ does not interfere with the recovery of server $s_1$. After the transfer of replicated state from server $s_3$ to server $s_1$ completes, the replacement server sends an acknowledgment (8). Since the failure of server $s_2$ is a cascading failure, replacement server of $s_1$ also participates in its recovery (11) by transferring state that it might have received from $s_2$ in (5). Finally, $s_2$'s replacement server acknowledges completion of state transfer from server $s_1$ and $s_3$ (12).

### 3.6 Handling Lost Vertex States

After failure, Zorro recovers only an approximation of the global state from before failure. So, after Rebuild, the resulting graph may contain vertices without recovered state information. For example, the value of vertex $v_3$ in Figure 3(b) cannot be recovered from a survivor if $s_3$ fails. When a vertex state is unrecoverable, Zorro reinitializes the vertex with the default value used for initialization by the application (e.g., for single-source shortest path, default initialization is zero for the source vertex and infinity for others).

Zorro minimizes the impact that lost values have on the global graph state through an implicit prioritization of high-degree vertices; for most partitioning functions, the probability of vertex recovery monotonically increases with the degree. As a result, vertices with lost states are generally confined to those whose neighbors do not span partitions. In our experiments, we find that those vertices with lost states are still able to quickly reconverge.

***Edge States:*** Some applications in PowerGraph maintain edge states in addition to vertex states. However, these values can be obtained from their source and/or target vertex values and do not need to be maintained separately. Alternatively, edge states assume static values (e.g., edge weights) for some applications, and can therefore be restored from graph partitions during Rebuild.

## 4. Recovery Analysis

In this section, we analyze the number of vertex states recovered by Zorro, as well as the overhead of Rebuild. Proofs below can be skipped without loss of continuity.

For a graph $G = (V,E)$, we define the set of *recovery neighbors* $\Gamma_r(v), \forall v \in V$, as the set of vertices that enable remote replication of $v$ (e.g., in Figures 3(b) and 3(c), $\Gamma_r(v_1) = \{v_2, v_3\}$ and $\{v_2, v_4\}$). Let $\Gamma_{in}(v)$ and $\Gamma_{out}(v)$ be the set of in- and out-neighbors, respectively. For the two classes described in Section 2.4, $\Gamma_r(v)$ exhibits the following property:

- *Out-Neighbor*: $|\Gamma_r(v)| = |\Gamma_{out}(v)|$
- *All-Neighbor*: $|\Gamma_r(v)| = |\Gamma_{out}(v) \cup \Gamma_{in}(v)|$

We note that PowerGraph has $|\Gamma_r(v)| = |\Gamma_{out}(v) \cup \Gamma_{in}(v)| - 1$ due to neighbor collocation from edge partitioning.

Suppose graph processing is performed on a set $S$ of $m$ servers and some set of $f \leq m$ servers fail. Let $V_S$ be the set of vertices that were primarily hosted at surviving servers, $V_F = V \setminus V_S$ be the set of vertices whose state must be recovered from failed servers, and $V' \subseteq V$ ($|V'| = n'$) be the true set of vertex states recovered after failure.

### 4.1 State Recovery

We wish to quantify $n_r = n' - |V_S|$, the number of vertex states recovered by Zorro from $V_F$. Under the assumption that vertices/edges are assigned to servers using a consistent hashing function, we have the following results.

**Theorem 1.** *The expected number of vertex states recoverable from $V_F$ is given by:*

$$\mathbb{E}[n_r] = \sum_{v \in V_F} \left(1 - \left(\frac{f}{m}\right)^{|\Gamma_r(v)|}\right) \tag{1}$$

*Proof.* $\forall v \in V_F$, the probability that $v$ is recoverable ($v \in V'$) is equal to the probability that $\Gamma_r(v) \cap V_S \neq \emptyset$, i.e., that $v$ has at least one surviving recovery neighbor. Let $r(v)$ be the binary recovery event for $v$. We therefore have:

$$\mathbb{E}[r(v)] = 1 - \prod_{v' \in \Gamma_r(v)} \frac{f}{m} = 1 - \left(\frac{f}{m}\right)^{|\Gamma_r(v)|} \tag{2}$$

By linearity of expectation, $\mathbb{E}[n_r] = \sum_{v \in V_F} \mathbb{E}[r(v)]$. □

Theorem 1 says that the expected number of recovered vertices and the probability of recovery are dependent on the *fraction* of servers that fail, rather than the actual number of failures. Moreover, the probability of recovery exhibits rapid convergence to 1 as the number of recovery neighbors increases, or as the fraction of servers that fail decreases.

**Theorem 2.** *Letting $V_F$ equal the total set of vertices primarily hosted on servers that fail before and during recovery, $\mathbb{E}[n_r]$ presents a lower bound on the expected number of vertices recovered after cascading failures.*

*Proof.* Failures during state transfer from survivors to replacements will occur after some subset $V'_F \subseteq V_F$ of vertex states have been received at a replacement. Therefore, the expected number of vertex states recovered is equal to:

$$|V'_F| + \sum_{v \in V_F \setminus V'_F} \left(1 - \left(\frac{f}{m}\right)^{|\Gamma_r(v)|}\right) \geq \mathbb{E}[n_r] \tag{3}$$

□

**Theorem 3.** *If the number of recovery neighbors of vertices in G follows a power-law distribution, $\mathbb{E}[n_r]$ can be expressed directly in terms of the power-law constant $\gamma$:*

$$\mathbb{E}[n_r] = |V_F| \left(1 - \frac{1}{\sum_{d=1}^{|V_F|-1} d^{-\gamma}} \sum_{d=1}^{|V_F|-1} \left(\frac{f}{m}\right)^d d^{-\gamma}\right) \tag{4}$$

*Proof.* Let $|\Gamma_r(v)|$ be a Zipf random variable with constant $\gamma$. The expectation in Equation 1 is equivalent to:

$$\mathbb{E}\left[\left(\frac{f}{m}\right)^{|\Gamma_r(v)|}\right] = \frac{1}{\sum_{d=1}^{|V_F|-1} d^{-\gamma}} \sum_{d=1}^{|V_F|-1} \left(\frac{f}{m}\right)^d d^{-\gamma} \tag{5}$$

Substituting back in gives:

$$\mathbb{E}[n_r] = \sum_{v \in V_F} \left(1 - \frac{1}{\sum_{d=1}^{|V_F|-1} d^{-\gamma}} \sum_{d=1}^{|V_F|-1} \left(\frac{f}{m}\right)^d d^{-\gamma}\right) \tag{6}$$

which reduces to our result. □

As demonstrated in [13] and [15], the power-law constant of most natural graphs is typically around $\gamma \approx 2$; for example, the Twitter graph has in-degree $\gamma = 1.7$, out-degree $\gamma = 2$, and recoverability illustrated in Figure 2.

***Expected Recovery:*** The expected number of vertex states lost by Zorro is therefore equal to $|V_F| - \mathbb{E}[n_r]$, and the expected number recovered is given by:

$$\mathbb{E}[n'] = |V_S| + \mathbb{E}[n_r] \tag{7}$$

This value represents a general metric on the approximation given by Zorro, specified as the total number of recovered vertex states after failure. However, application specific metrics may present more useful results to the user and will be discussed in Section 6.

### 4.2 Rebuild Overhead

We define the *rebuild time* of Zorro after failures as the total time to rebuild the local graph states at replacement servers. Let $S_R, S_S \subseteq S$ be the set of replacement and surviving servers, respectively, and let $V(s)$ be the set of vertices primarily hosted on $s \in S$. Using the approach in Section 3.2, the expected upper bound on rebuild network overhead between two servers $s \in S_R$ and $s' \in S_S$ can be calculated as:

$$c_r(s,s') = \sum_{v \in V_F(s)} |\Gamma_r(v) \cap V_S(s')| \cdot \eta(v) + \sum_{v \in V_S(s')} |\Gamma_r(v) \cap V_F(s)| \cdot \eta(v) \tag{8}$$

where $\eta(v)$ is the size of $v$'s vertex state message. Hence, the expected communication cost is given by:

$$\begin{aligned}\mathbb{E}[c_r(s,s')] = &\sum_{v \in V_F(s)} |\Gamma_r(v)| \cdot \left(1 - \frac{1}{m-f}\right) \cdot \eta(v) \\ &+ \sum_{v \in V_S(s')} |\Gamma_r(v)| \cdot \left(1 - \frac{1}{f}\right) \cdot \eta(v)\end{aligned} \tag{9}$$

***Expected Rebuild Time:*** Let $\varphi(s,s')$ be the symmetric bandwidth between any two servers $s, s' \in S$, $s \neq s'$. The expected upper bound on total rebuild time can be approximated as the maximum per-server rebuild time using [18]:

$$\mathbb{E}[t_r] = \max_{s \in S_R} \sum_{s' \in S_S} \frac{\mathbb{E}[c_r(s,s')]}{\varphi(s,s')} \tag{10}$$

An optimization for rebuild when neighbors are known is provided in Section 5.2. Furthermore, the upper bound on network overhead is less than that of a single operation during a normal GAS iteration, as vertex state values are transferred to only a subset of servers, rather than to all (Section 6.2 further demonstrates this result).

## 5. Implementation

To demonstrate the generality and effectiveness of Zorro across the two classes of systems discussed in Section 2.4, we discuss implementation on one out-neighbor replication system, LFGraph, and one all-neighbor replication system, PowerGraph (v2.2). In both systems, *Replace* is handled using ZooKeeper to identify failures. Hence, we focus on the *Rebuild* and *Resume* stages in our descriptions. We then discuss how Zorro maintains state consistency after failure.

### 5.1 LFGraph

Our implementation of Zorro in LFGraph modifies the `computation_worker` and `communication_worker` classes within the JobServer, which implements GAS.

***Rebuild:*** LFGraph maintains, for each vertex, the vertex state, a copy (for lock-free read/write) in the *local value store*, and vertex state replicas of remote in-neighbors in the *remote value store*. After failure occurs, survivors send replacements all vertex states previously hosted at that server

in both the remote and local value store. Replacements receive vertex state data concurrently with initialization (i.e., graph loading) and apply the received values afterward.

***Resume:*** As discussed in Section 3.3, Zorro performs a partial Scatter before computation resumes. In LFGraph, Zorro performs this operation only among replacement servers. Each replacement performs a Scatter (over the network) to other replacements, rebuilding the vertex states of incoming neighbors on these servers.

***Maintaining State Consistency:*** In the Scatter stage, servers send the states of updated vertices to servers hosting outgoing neighbors. As a consequence, failures during Scatter may result in states being received at only a subset of survivors, leading to possible inconsistency during subsequent computation. To enforce consistency, Zorro ensures that servers receive 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. Zorro also merges vertex state copies after Scatter, rather than between Apply and Scatter as in vanilla LFGraph. The resulting average per-iteration overhead incurred by ensuring vertex state consistency is just 0.8%. This is the only instance of overhead we ever found in Zorro.

### 5.2 PowerGraph

Our implementation of Zorro in PowerGraph modifies both the `synchronous_engine` class, which implements GAS, and the `local_graph` class, which encapsulates the local subgraph at each server.

***Rebuild:*** As discussed in Section 2.4, PowerGraph maintains, for each vertex, a master and a set of mirrors. After failure occurs, survivors retain the local subgraph state in memory and send replacements all vertex states (either masters or mirrors) previously hosted at that server. Replacements receive vertex state data concurrently with initialization (i.e., graph loading and ingress) and then update local states.

PowerGraph stores the IDs of servers containing each vertex, and thus allows an optimization to reduce network overhead during the rebuild stage. Per replacement server, $s \in S_F$, and relevant vertex, $v \in V_F(s)$, only a single survivor sends the associated state information. Relevant survivors evaluate candidacy using the following function:

$$s_r(s,v) = \underset{s' \in S_S(v)}{\operatorname{argmin}} \left| s'.id - ((v.id - s.id) \ \% \ m) \right| \qquad (11)$$

where $S_S(v)$ is the set of surviving servers that contain recovery neighbor(s) of $v$. This approach ensures balanced state transfer and low network overhead by (1) minimizing the transfer of redundant state information and (2) removing skew associated with high-degree vertices. After cascading failures, servers reiterate over vertices and send any values for which they *newly* satisfy Equation 11. We present the performance improvement of this approach in Section 6.2.2.

***Resume:*** As discussed in Section 3.3, Zorro performs a partial Scatter before computation resumes. In PowerGraph, this operation is entirely local, performed only at replacements, and rebuilds local message buffers.

***Maintaining State Consistency:*** In the Apply stage, the master aggregates partial accumulators (the results of performing local Gather at mirrors) and synchronizes the results back to the mirrors. Failures during Apply may result in vertex state inconsistency at survivors. For example, a failed server may synchronize a mirror at one survivor but not at another. Zorro ensures that servers receive updates for all mirrors before Applying the updated values, aborting if failures occur during transfer. This modification required changing only two lines of code and, interestingly, *reduced* the average per-iteration time of PowerGraph by $\sim 26\%$ in all experiments. We attribute this reduction to the overhead incurred by synchronously interleaving send and apply in the original PowerGraph.

## 6. Evaluation

In this section, we describe the experimental setup and evaluation of Zorro using our exemplar systems and the graphs from Table 1. Our evaluation measures: (1) the accuracy of graph algorithms after various numbers of server failures and after failures in different iterations, (2) the recovery overhead with Zorro, and (3) the effects of using different partitioning strategies. Failures encompass random servers and results are averaged across three trials. For consistency of results between PowerGraph and LFGraph, in addition to the reasons outlined in Section 2.1, we evaluate (1) and (2) using hash-based partitioning. We evaluate Zorro using PowerGraph's intelligent partitioning strategies in (3) and [32].

***Cluster Setup:*** All experiments were conducted on a 16-machine cluster. Each machine has 2 $\times$ 4-core Intel Xeon E5620 processors with hyperthreading enabled (16 virtual cores), 64 GB of RAM, a 500 GB SSD and 2 TB HDD. The connectivity between any two machines is 1 Gbps.

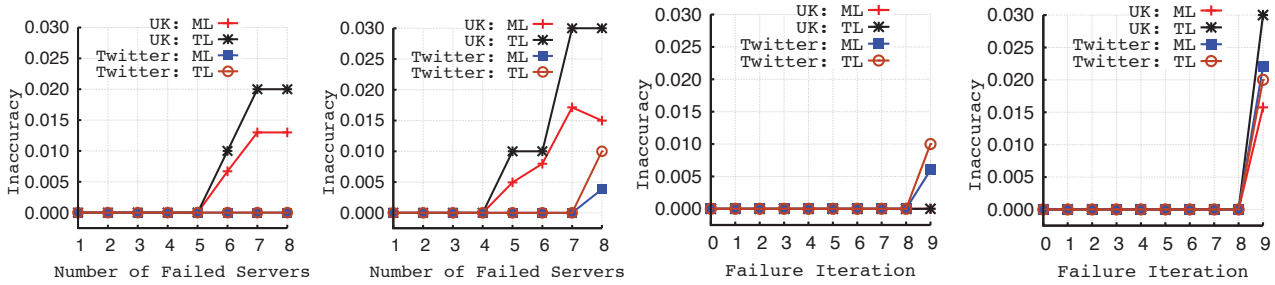
### 6.1 Algorithm Accuracy

We evaluate Zorro's accuracy using four graph analytics algorithms: (1) PageRank, (2) single-source shortest paths, (3) connected components, and (4) k-core decomposition. Results using additional applications are presented in [32].

#### 6.1.1 PageRank

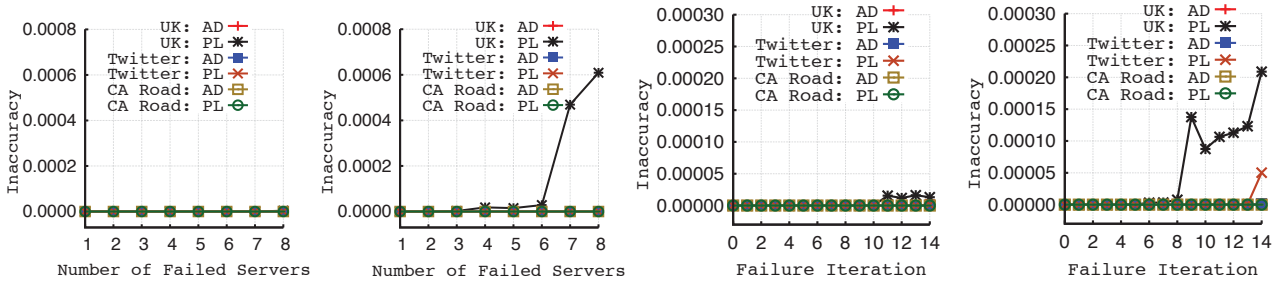
We first evaluate Zorro's accuracy loss after failures while running PageRank with 10 iterations on the Twitter and UK Web graphs.[2] Let $P_n$ be the set of top-$k$ PageRank vertices after failure recovery, and $P_t$ be the true top-$k$ PageRank vertices from execution without failure. We use the following metrics from [28] to evaluate Zorro's accuracy:

[2] CA-Road was excluded due to most values remaining 1 after 10 iterations, thus yielding no inaccuracy but also offering no information about Zorro.




(a) PowerGraph: number of failures. (b) LFGraph: number of failures. (c) PowerGraph: failed iteration. (d) LFGraph: failed iteration.

**Figure 6.** PageRank inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 10 iterations.




(a) PowerGraph: number of failures. (b) LFGraph: number of failures. (c) PowerGraph: failed iteration. (d) LFGraph: failed iteration.

**Figure 7.** SSSP inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.

- **Top-$k$ Lost (TL):** The fraction of lost top-$k$ ranked vertices: $|P_t \setminus P_n|/|P_t|$.
- **Mass Lost (ML):** The fraction of total top-$k$ PageRank mass lost: $\sum_{v \in P_t \setminus P_n} p(v) / \sum_{v \in P_t} p(v)$, where $p(v)$ is the PageRank score of $v$.

The metrics above provide an evaluation both of how many of the top PageRank vertices are lost (TL), as well as their relative importance in the rankings (ML). For our experiments, we set $k = 100$.

As demonstrated in Figure 6, Zorro on both frameworks achieves *no* accuracy loss in a majority of failure scenarios. In fact, even when half the servers fail (8 out of 16), Zorro results in an inaccuracy of only 2% top-$k$ lost (i.e., two of the top-100 PageRank vertices are not present in the new result), and even lower mass lost (i.e., the two lost vertices were low in the ranking). Even with a lower replication model (out-neighbor only), Zorro on LFGraph still manages to achieve a maximum inaccuracy of only 3% for failures at the last iteration, or with half the servers failing. In both systems, lost mass is always less than top-$k$ lost, implying that lost vertices rank low in the original top-$k$ result.

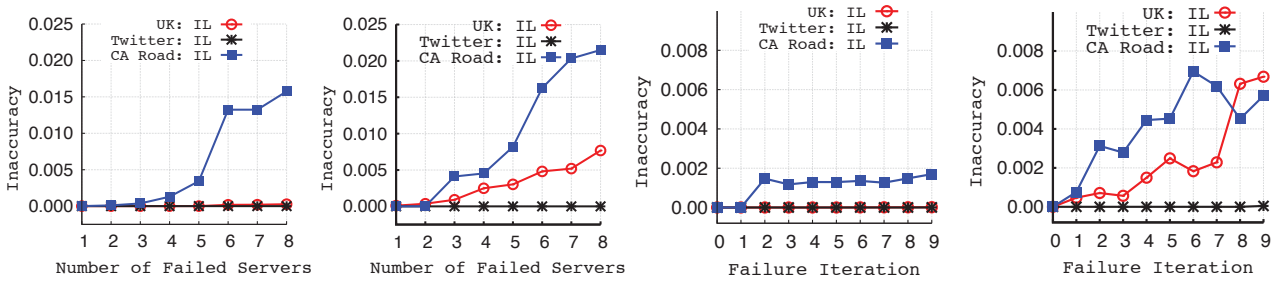
From Figures 6(c) and 6(d), we note that, even with 4 servers failing, Zorro incurred inaccuracy only for failures in the last iteration. This is due to a high likelihood of subsequent reconvergence to the correct value in later iterations.

### 6.1.2 Single-Source Shortest Paths (SSSP)

We evaluate Zorro's accuracy after failures while running SSSP on all three graphs. The SSSP algorithm computes the distance from a given source vertex to all other vertices in the graph. PowerGraph's default setting was used for source selection in both systems. The number of iterations is increased to 15 to reach a larger set of vertices in graphs with large directed diameter (UK Web and CA Road Network). We use the following metrics to evaluate Zorro's accuracy:

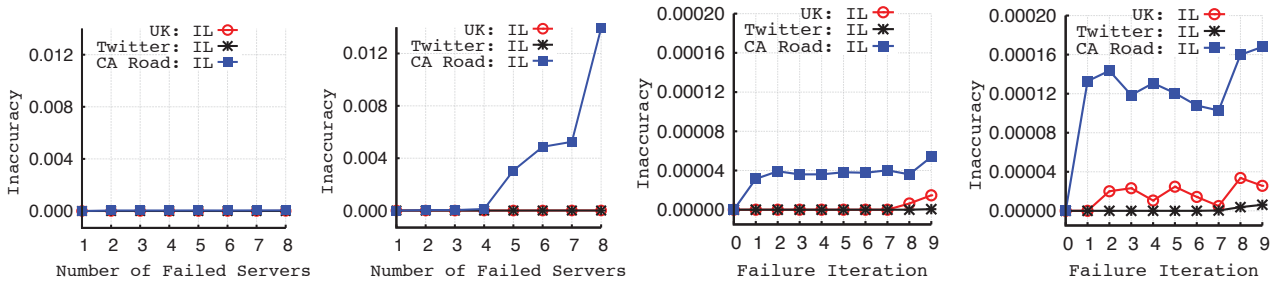
- **Paths Lost (PL):** The fraction of reachable vertices with lost paths after failure.
- **Average Difference (AD):** The average normalized difference of shortest paths [17]: $\frac{1}{|V'|} \sum_{v \in V'} (l_t(v) - l_n(v))/l_t(v)$, where $V'$ is the set of reachable vertices and $l_t(v)$ and $l_n(v)$ are the original and resulting shortest path length, respectively, from the source to vertex $v$.

As in PageRank, Figure 7 demonstrates that Zorro achieves zero (or near-zero) inaccuracy for most failure scenarios. Even with 8 failures, the resulting maximum inaccuracy with LFGraph was only 0.06% PL (i.e., only 0.06% of the original reachable vertices were unreachable after failure), and no AD. Zorro's accuracy on PowerGraph was consistently higher than on LFGraph due to PowerGraph's replication model – zero loss was achieved in all but a few scenarios.




(a) PowerGraph: number of failures. (b) LFGraph: number of failures. (c) PowerGraph: failed iteration. (d) LFGraph: failed iteration.

**Figure 8.** CC inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.




(a) PowerGraph: number of failures. (b) LFGraph: number of failures. (c) PowerGraph: failed iteration. (d) LFGraph: failed iteration.

**Figure 9.** K-core inaccuracy vs. (1) the number of failures (at the middle iteration), and (2) the iteration at which a quarter of the servers fail. There are a total of 16 servers and 15 iterations.

#### 6.1.3 Connected Components (CC)

We evaluate Zorro's inaccuracy after failures while running CC with 10 iterations on all three graphs. We use the weak connected components algorithm popular in distributed graph processing systems [27], and evaluate Zorro's inaccuracy using the following metric:

- **Incorrect Labels (IL):** The fraction of vertices with a different label (i.e., component) than the original result.

Figure 8 illustrates the result. The CA Road network resulted in the highest inaccuracy, with a maximum of 2.2% of vertices incorrectly labeled (i.e., assigned to the wrong component) in LFGraph, even with half of the servers failing. Zorro on PowerGraph resulted in 1.6% of vertices incorrectly labeled in the same scenario. Zorro produced no inaccuracy under all scenarios using the Twitter graph. Failures in later iterations increased inaccuracy due to a lower likelihood of reconvergence in later iterations.

#### 6.1.4 K-Core Decomposition

K-core decomposition [37] of a graph identifies induced subgraphs such that included vertices have at least $k$ neighbors. We evaluate Zorro's accuracy after failures while running k-core decomposition with 10 iterations. Here, $k = 10$ for Twitter/UK Web, and $k = 3$ for the more sparse CA Road. We use the same metric as in Connected Components. In this context, the label is a binary value corresponding to a vertex's inclusion in the induced k-core subgraph.

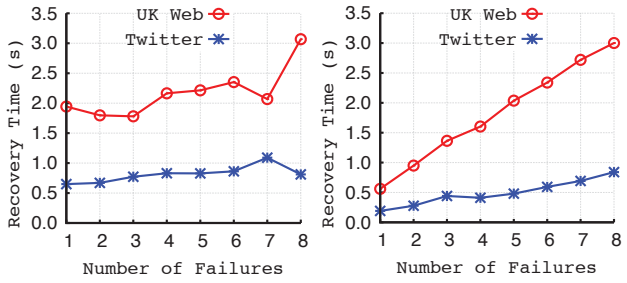
The results are illustrated in Figure 9. Zorro's inaccuracy is again low, achieving a maximum of 1.4% of vertices incorrectly labeled with half of the servers failing using LFGraph. As expected, both larger numbers of failures and failures in later iterations increased inaccuracy.

### 6.2 Overhead during Recovery

In this section, we evaluate the overhead Zorro incurs during failure recovery in terms of (1) added recovery time beyond initialization, and (2) network communication cost.

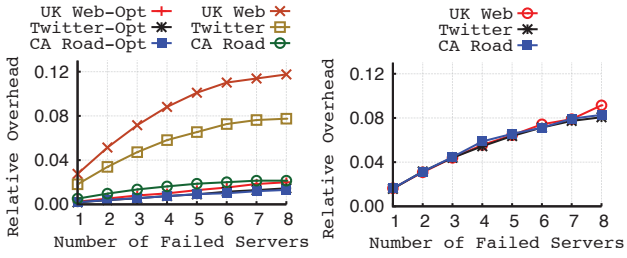
#### 6.2.1 Recovery Time

Figure 10 shows the total recovery time excluding initialization (i..e, subgraph loading) for simultaneous failures in PowerGraph and LFGraph with the UK Web and Twitter graphs (CA Road is excluded due to negligible recovery time). Zorro allows replacement servers to rebuild their graph state while loading their respective graph partitions, resulting in quick recovery. Using PowerGraph, the recovery time for both graphs exhibits insignificant variation with increasing numbers of failed servers, and increases for larger graph sizes. Using LFGraph, the recovery time increases linearly with the number of failed server due to the non-local partial scatter discussed in Section 5.1. PowerGraph's all-




(a) PowerGraph: number of failures. (b) LFGraph: number of failures.

**Figure 10.** Additional recovery time beyond initialization as a function the number of failures (in the middle iteration).




(a) PowerGraph: number of failures. (b) LFGraph: number of failures.

**Figure 11.** Network communication overhead of recovery with Zorro relative to total failure-free PageRank overhead.

neighbor replication model produces slightly higher average recovery time than LFGraph. Most importantly, the recovery time is a small fraction of the average iteration time in both PowerGraph (11.7 seconds and 22 seconds for PageRank with Twitter graph and UK Web graph, respectively) and LFGraph (2 seconds and 5.6, respectively).

#### 6.2.2 Rebuild Network Overhead

Figure 11 illustrates the network overhead incurred during recovery for PowerGraph and LFGraph, relative to the total overhead of 10 PageRank iterations. For PowerGraph (Figure 11(a)), we illustrate the overhead both with and without our optimization from Section 5.2 – overhead with the optimization is approximately 10% of that without (attaining a maximum of around 2%). For LFGraph (Figure 11(b)), recovery overhead is less than the average overhead of a single iteration ($\sim$ 8% vs. 10%). We also note that relative overhead scales with graph size for unoptimized Rebuild in PowerGraph, but remains static in LFGraph – this result can be explained by the difference in replication models between the two frameworks.

### 6.3 Different Partitioning Methods

To examine the effect of different partitioning methods (applied to the graph during initialization), we perform experiments comparing our previous results with two further approaches available in PowerGraph [15] (LFGraph utilizes only cheap hash-based partitioning):

- **Oblivious:** Servers in the cluster greedily and independently partition the graph segment locally read during initialization.
- **Grid:** Randomly places edges using a grid constraint. We note that this approach only works if the cluster comprises a perfect-square number of machines.

For brevity, we only present results under the worst-case scenarios (i.e., half the cluster fails or a quarter fails on the last iteration) with PageRank and SSSP on the Twitter graph. As illustrated in Figure 12, intelligent partitioning methods result in *at most* a 1% and 1.2% increase in inaccuracy with PageRank and SSSP, respectively. In some cases, changing the partitioning function resulted in no (or negligible) increase in inaccuracy (e.g., Figures 12(b) and 12(c)).

### 6.4 The Trade-off Space

In Table 2, we compare various failure recovery mechanisms with PowerGraph running PageRank on the Twitter Graph, based on the overhead incurred (both during normal execution and with failures), and the resulting inaccuracy.

| | Recovery Mechanism | | | |
|---|---|---|---|---|
| **Overhead** | **Checkpoint** | **Restart** | **Continue** | **Zorro** |
| **Normal** (s) | 261 per chkpt | 0 | 0 | 0 |
| **Re-init.** (s) | 60 | 117 | 117 | 117 |
| **Recover** (s) | $11.7 \times (i_f - i_c)$ | $11.7 \times i_f$ | 0 | $\leq 1$ |
| **Inaccuracy** | 0% | 0% | up to 25% | $\leq 1\%$ |

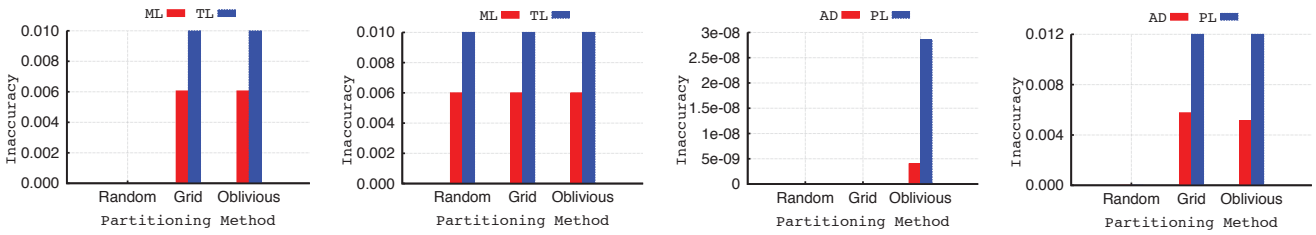
**Table 2.** A comparison of various failure recovery techniques for PowerGraph with the Twitter graph (16 servers, SSDs). $i_f$ corresponds to the iteration at which failure occurs and $i_c$ corresponds to the last checkpointed iteration.

Checkpointing mechanisms incur high overhead both during failure-free execution (261 seconds per checkpoint with SSDs, 321 with HDDs) and after failures, and must recompute lost iterations between the checkpoint and failure. A recovery mechanism that simply restarts computation incurs initialization overhead (117 seconds) and must recompute all previous iterations (such a mechanism also fails to make progress during cascading failures). A recovery mechanism that continues processing after failures suffers from very high inaccuracy. Zorro performs better than all alternatives, exhibiting no overhead during failure-free execution, requiring no recomputation, and achieving $\leq 1\%$ inaccuracy.

## 7. Related Work

Failure recovery has been widely researched [8, 12], including optimistic recovery in systems and networks [4, 20, 25, 40]. To the best of our knowledge, we are the first to explore reactive failure recovery in distributed graph processing.

***Proactive Checkpoint-Based Recovery:*** Failure recovery using checkpoints is most common in distributed graph processing systems (e.g., Pregel [27], Piccolo [31], GPS [35], Giraph [1] and PowerGraph [15]).




(a) PageRank: half failures. (b) PageRank: last iteration failure. (c) SSSP: half failures. (d) SSSP: last iteration failure.

**Figure 12.** PageRank and SSSP inaccuracy with various PowerGraph partitioning methods, for half (8 out of 16) servers failing in the middle iteration, and a quarter (4 out of 16) servers failing in the last iteration.

In Pregel [27], workers checkpoint the state of vertices, edge values and received messages. The Pregel master checkpoints the state of global aggregators and detects worker failures via heartbeating. [27] also proposes a *confined* recovery mechanism in which workers checkpoint outgoing messages, restricting recomputation to failed workers.

Piccolo [31], an open-source implementation of Pregel, and Distributed GraphLab [24], both use the Chandy-Lamport algorithm [9] to calculate either a synchronous or asynchronous global snapshot of the system. The asynchronous variant of Chandy-Lamport allows computation to proceed along side the snapshot algorithm to mask checkpointing cost. However, after failures, some iterations may need to be repeated, significantly increasing recovery cost.

The authors in [38] propose a partition-based recovery (PBR) mechanism that relies on checkpointing. PBR achieves faster recovery than traditional checkpointing by parallellizing recomputation of failed partitions among survivors. PBR handles cascading failures by initiating a new recovery plan considering the most recent cluster state. However, PBR additionally logs all outgoing messages to disk, increasing overhead during failure-free execution.

***Proactive Replication-Based Recovery:*** Imitator [41] proactively ensures that vertices have at least $(K+1)$ replicas to tolerate the failure of at most $K$ servers. However, the choice of $K$ needs to be implicitly linked to the cluster size (given MTBF), and thus the induced network overhead to update replicas would become infeasible with large clusters. Furthermore, such an approach: (1) requires estimating the number of failures an application must tolerate, (2) enforces an artificial lower bound on replication that prevents the use of graph partitioning heuristics, and (3) is unable to handle cascading failures without re-replicaton after failure.

Zorro, on the other hand, recovers a near-perfect approximation of the graph state without any upper bounds on the number of failures. Furthermore, the number of vertex states recovered is independent of the cluster size (depending instead on the fraction that fails), and Zorro can recover from arbitrary numbers of independent and cascading failures.

***Failure Recovery in Iterative Distributed Computation:*** To eliminate checkpointing, GraphX [16] uses the Resilient Distributed Datasets (RDD) abstraction provided by Spark [43]. Spark allows fast reconstruction of RDDs using their lineage graph. However, even with the fast reconstruction of RDDs, the execution time with one server failure incurs an overhead of 36% [16], and checkpointing is required in the case of long lineage chains.

The authors in [36] propose a reactive mechanism to recover from failures in iterative data-flow systems. In the proposed mechanism, the processing state can reach consistency even after failures using correct "algorithmic compensations". The mechanism allows users to specify the "compensate" function and discuss such functions for algorithms involving link/path exploration and matrix factorization.

In distributed storage systems, RAMCloud [30] distributes data replicas across the cluster servers. In case of failures, the surviving servers jointly reconstruct the state of failed servers in parallel for fast failure recovery. [21] performs opportunistic recovery for MapReduce with forced proactive replication.

## 8. Conclusions

In this paper, we proposed Zorro, a reactive recovery mechanism for distributed graph processing systems. We have implemented Zorro in both PowerGraph and LFGraph. Using real world graphs and popular algorithms, we demonstrated that Zorro recovers a highly accurate approximation of the graph state, even after a significant number of failures. Moreover, Zorro is able to achieve at least 97% accuracy when compared to the original output of graph algorithms, achieving perfect accuracy in many scenarios. We have argued that failure recovery in distributed graph processing systems is best done via reactive approaches like Zorro, rather than expensive proactive approaches that are the norm today. Furthermore, we believe that this approach opens up the avenue to explore reactive recovery in other computation systems.

***Acknowledgments:*** We thank our shepherd Foto Afrati and the anonymous SoCC reviewers for their valuable feedback. This work is supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, a VMware Graduate Fellowship and a Siebel Scholarship.

## References


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

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

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

[4] B. Bhargava and S. R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems-An Optimistic Approach. In *Proceedings of the Symposium on Reliable Distributed Systems*. IEEE, 1988.

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

[6] 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.

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

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

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

[10] A. Ching. Scaling Apache Giraph to a Trillion Edges. *Facebook Engineering Blog*, 2013.

[11] 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.

[12] 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.

[13] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SIGOPS Operating Systems Review*. ACM, 2003.

[15] 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.

[16] 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.

[17] 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*. ACM, 2010.

[18] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing*, 20(3):389–398, 1994.

[19] 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.

[20] 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.

[21] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 2015.

[27] 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.

[28] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis. FrogWild!–fast PageRank approximations on graph engines. In *Proceedings of VLDB Endowment*, 2015.

[29] 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.

[30] 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.

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

[32] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. *Technical Report, IDEALS*, 2015. URL `https://ideals.illinois.edu/handle/2142/75959`.

[33] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transac-*

*tions on Computer Systems (TOCS)*. ACM, 1992.

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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.