Load-Aware Compressed Incremental Checkpointing for Primary-Backup Replication

by

Berkin Güler

A Thesis Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

Koç University

January 22, 2018

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Berkin Güler

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Öznur Özkasap (Advisor)

_____

Asst. Prof. Didem Unat

_____

Asst. Prof. Ayşe Yılmazer

Date: _____

To my family...

ABSTRACT

Several distributed services ranging from key-value stores to cloud storages require fault-tolerance and reliability features. For enabling fast recovery and seamless transition, primary-backup replication protocols are widely used in different application settings including distributed databases, web services and the Internet of Things. In this thesis, we address the communication cost of the primary-backup replication protocol, and propose utilizing the checkpointing concept for improving the efficiency and performance of primary-backup replication. We then develop a software framework by extending the open-source RocksDB key-value store of Facebook on the PlanetLab overlay network using a geographically replicated system setup, and evaluate various checkpointing algorithms including non-periodic, periodic, incremental and compressed checkpointing. Experimental scenarios utilize the well-known benchmarking tool YCSB to generate realistic query workloads. Using various metrics of interest including blocking time, checkpointing time, checkpoint size, compression ratio and throughput, and testing with realistic workloads, our findings indicate that incremental checkpointing combined with a periodic usage performs the best by providing better system throughput and decrease in average blocking times in comparison to the traditional primary-backup replication and other checkpointing algorithms. Based on our findings, we propose load-aware compressed incremental checkpointing method (LACPB) for large scale primary-backup replication. Large-scale and comparative experimental results indicate that LACPB maintains drastically higher system throughput as well as reduced and stable client blocking times even in the dynamic workload scenarios.

ÖZETÇE

Anahtar-değer veritabanlarından diğer bulut servislerine kadar olan bir çok dağıtık sistem uygulaması hata dayanıklılığı ve güvenilirlik özelliklerini sağlamak zorundadır. Hızlı kurtarma ve kesintisiz geçiş gibi özellikleri sayesinde, birincil-yedek replikasyon protokolü dağıtık veritabanları, web servisleri ve nesnelerin interneti gibi farklı alanlarda sıklıkla kullanılmaktadır. Bu tez kapsamında, birincil-yedek replikasyon protokolünün yüksek iletişim maliyeti göz önünde bulundurularak, etkinliği ve performansı arttırmak amaçlı denetim noktası alma yöntemine odaklanmaktayız. Bunu takiben, Facebook tarafından açık-kaynak olarak geliştirilmekte olan RocksDB anahtar-değer veritabanınını kullanarak özgün bir sistem önermekteyiz. Geliştirilen bu sistemi PlanetLab test ağında coğrafi olarak replike ederek, periyodik olmayan, periyodik, artımlı ve sıkıştırılmış denetim noktası alma gibi farklı yöntemleri test etmekteyiz. Farklı test senaryolarını, bu iş için sıklıkla tercih edilen YCSB test aracını kullanarak gerçekçi iş yükleri dahilinde değerlendirmekteyiz. Bekletme süresi, denetim noktası alma süresi, denetim noktası boyutu, sıkıştırma oranı ve sistem başarımı gibi farklı ölçüm noktalarını ve gerçekçi iş yüklerini kullanarak yapılan çalışmalarda periyodik artımlı denetim noktası yöntemini kullanmanın geleneksel birincil-yedek replikasyon protokolüne göre ve diğer denetim noktası alma yöntemlerine göre en iyi sistem başarımı ve en düşük bekletme süresi yakalayabildiği gösterilmiştir. Bulgularımıza dayanarak, geniş ölçekli birincil-yedek replikasyon uygulamaları için yükbilinçli sıkıştırılmış artımlı (LACPB) denetim noktasını önermekteyiz. Geniş ölçekli ve karşılaştırmalı test sonuçları, LACPB yönteminin oldukça yüksek sistem başarımı, ayrıca düşük ve istikrarlı bekletme sürelerini hareketli iş yükleri için de sağladığı gösterilmiştir.

ACKNOWLEDGMENTS

Obtaining a MSc degree is inevitably a very long and stiff journey which would be impossible to finish with success without the help and support of many people to whom the author would like to express his gratitude and appreciation. First and foremost, author would like to convey his sincere gratitude to his supervisor, Assoc. Prof. Öznur Özkasap who was generously helpful with all the assistance she offered and the guidance she showed.

Next, the author would like to express his gratitude to thesis committee members, Asst. Prof. Didem Unat and Asst. Prof. Ayşe Yılmazer, for their help, support and participation politeness.

The author also would love to appreciate the friendship, hospitality and every other valuable discussions offered by his fellow researcher mates in the DISNET Group and would like to explicitly thank to Seyhan Uçar for his invaluable friendship.

Assuredly, the author would like to thank his family for supporting and encouraging him since the beginning of and throughout his MSc journey.

Last but not the least, the author would like to praise Tuğçe Akman for her eight years of companion, unconditional love and life-changing experiences flourished together.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

As distributed services scale out geographically the number of nodes involved increase as well, and as the execution times of such services rise, service stability gets imperiled. Not to mention the availability promises are now need to be higher than ever as any outages that could last a few milliseconds of rise in response times may result in surprisingly high income losses [1]. Moreover, the possibility of facing with failures in these systems is inevitable due to exhaustive usage of software and hardware components with the long running applications that exceed the mean time between failures of the components [2].

Fault tolerance mechanisms enable reliability and dependability of a distributed service by ensuring the service to be robust in the event of failures. A recovery method is actually needed for supporting fault-tolerance, reliable and recovery oriented distributed systems. The most important and effective approach to deal with crash failures, which is mostly the main reason behind system failures, is replication. It is widely used as a fault-tolerance mechanism and finding optimal replication protocols is an active research area.

There are two main types of replication protocols namely active and passive. In the active replication which is also known as the state-machine replication, every incoming request is processed by every replica in the system resulting in multiple results to be collected. Once collected, they are reduced into a single result value using various algorithms and the client is notified accordingly. In the passive replication, which is also known as primary-backup replication, there exists a single primary replica and a group of backup replicas. Each request is executed only in the primary replica, the

result is then copied to backup replicas and the client is notified.

Another common technique used frequently is the checkpointing approach, which refers to saving the system state to a stable storage after critical executions. Although it is only used in the HPC applications, it can also be used in many other application areas as well thanks to its application independent concept and ease of implementation. In the event of any failures faced during the execution, the previously saved checkpoint can be restored as a failure-free system state enabling the execution continue over. This approach also facilitates a quick rollback feature even against unforeseen failures as well as immensely decreases the workload needed to revitalize a replica from zero state since with a single rollback the system state will be caught up with the latest failure-free state [3].

1.1 Contributions

In this study, we address combining checkpointing and replication mechanisms to further improve efficiency of replication in comparison to the traditional primary-backup replication in terms of lower client blocking time and higher overall system throughput. The contributions of this work are as follows:

- We address the communication cost of the primary-backup replication protocol, and propose utilizing the checkpointing concept for improving the efficiency and performance of primary-backup replication. We then propose an advanced primary-backup replication protocol that can work with checkpointing techniques.

- We develop a software framework by extending the open-source RocksDB key-value store being developed by Facebook. Using the framework in geographically distributed setting of the PlanetLab overlay network, we evaluate various checkpointing algorithms including non-periodic, periodic, incremental and compressed checkpointing in a replicated key-value store configuration.

- We conduct a thorough analysis of various checkpointing algorithms integrated with primary-backup replication. For this purpose, we consider full, incremental, differential, periodic-full, periodic-incremental, periodic-differential, compressed-periodic-incremental with different compression algorithms including GZIP, Snappy and Zstd. We apply realistic static workload scenarios through the Yahoo! Cloud Service Benchmarking (YCSB) tool and track various metrics including blocking time, checkpointing time, checkpoint size, system throughput and three more metrics for compressed checkpointing techniques, namely compression ratio, compression time and decompression time.

- Our findings indicate that incremental checkpointing combined with a periodic usage performs the best by providing better system throughput and decrease in average blocking times in comparison to the traditional primary-backup replication and other checkpointing algorithms [4]. Furthermore, Zstd found to be the most competent compression method when applied to periodic incremental checkpointing thanks to its high compression ratio with short compression times [5].

- Based on our findings, we propose load-aware compressed incremental checkpointing method (LACPB) for large scale primary-backup replication. Large-scale and comparative experimental results indicate that LACPB maintains drastically higher system throughput as well as reduced and stable client blocking times even in the dynamic workload scenarios.

1.2 Organization

The rest of this thesis is organized as follows. Chapter 2 gives a literature review on replication and checkpointing concepts. Chapter 3 introduces various checkpointing techniques that can be used in the primary-backup replication protocol. Chapter 4 presents the experimental setup used throughput the whole thesis. Chapter 5 demonstrates the experimental analysis and performance results of various checkpointing

algorithms. In Chapter 6 we propose the load-aware checkpointing technique that efficiently works even in dynamic workload scenarios rather than static ones and we show the experimental results accordingly. Finally, Chapter 7 concludes the thesis with final remarks and possible future work directions.

Chapter 2

BACKGROUND AND RELATED WORK

In this chapter, we present background and discussion of related works on replication, key-value stores, and checkpointing mechanisms.

2.1 Replication and Key-Value Stores

Replication in distributed systems is used to enhance reliability as well as performance. Replication protocols are divided into two broad categories as primary-based and replicated-write. The latter one does not have a single primary server and instead every request is replicated and executed in each of the replicas. The primary-backup replication [6] is a long-established protocol defined, discussed in the literature and is an example of the primary-based replication protocols. However, it is still an active research topic and especially a prominent initial point in designing current replication protocols in contemporary databases and key-value stores. The primary-backup replication protocol defines one exclusive node that is named primary and the rest of the nodes are defined as backup replicas. When a client issues an update request, it is processed solely by the primary node and the results are disseminated to the backup nodes through update messages. As aforementioned, several modern key-value stores follow the same replication protocol in some degree, as discussed next.

Cassandra [7] is an open-source key-value store developed in Java programming language. The distributed architecture of the database was based on Amazon's DynamoDB [8] and the underlying data structure was based on Google's BigTable [9]. Initially it was developed by Facebook but later on it was passed on to Apache in 2009. It is now a widely used key-value store in the industry and its users include well-known companies such as Facebook, Twitter, Cisco and Netflix.

Figure 2.1a depicts how the replication in Cassandra takes place for a given update request. The diagram simply shows how data is replicated once processed by its coordinator node [10]. Due to many parameters involved in the replication and being configured freely, Cassandra is an AP system according to the CAP theorem [11] meaning it prioritizes availability over consistency. Coordinator replica knows how many nodes and which nodes should receive a copy of the processed data, and transfers it to them. In this point of view, the Coordinator acts like a Primary replica and other replicas receiving the copy resemble Backup replicas.

Although the MongoDB [12] is classified as a document store rather than a key-value store, it can be considered as a store allowing nested key-value objects. It was initially being developed by the 10gen company in 2007 and open-sourced in 2009. Every record in MongoDB is actually a document and they are stored in the BSON format which is the Binary JSON format. BSON documents are actually objects containing the list of stored key-value pairs in that document. Well-known companies like Google, Bosch, EA and SAP are just a few of them using MongoDB actively.

As shown in Figure 2.1b, the replication protocol utilized in MongoDB is based on the primary-backup replication protocol with slight changes. Backup nodes are referred as Secondary and rest of the replication is very similar to primary-backup replication. Once the Primary node receives an Update request, it replicates the required values to the Secondary replicas while controlling possible crash failures using the well-known heartbeat protocol. Due to failure cases, the Primary server may go down and work like a Secondary replica whereas one of the Secondary replicas takes on the responsibility as a Primary server. By default settings, MongoDB is a CP system prioritizing consistency over availability as reads and writes go through only the primary server which ensures strong consistency; however, through changing a few parameters enabling reads from secondary replicas, it can alternatively act as an AP system.

Redis [13] is one of the well-known and widely used open-source key-value store. Redis is mostly compared with Memcached [14] and it aims to deliver fast responses

like Memcached. However, in contrast to Memcached, not only simple get/put operations but also complicated atomic operations are supported. On the other hand, Memcached only uses system memory to operate and hence does not store data permanently, but Redis can periodically flushes its data on a permanent storage. Systems such as Twitter, GitHub, SnapChat and StackOverflow utilize Redis for numerous tasks.

Figure 2.1c shows a diagram depicting how replication works on the Redis, where clients interact with the master Redis replica and send update requests to it. The master Redis replica replicates the data to all slave replicas and the slave Redis replicas manage a persistent storage to save data. As Redis is designed originally as a single node system and still works according to that principle, although replication is possible as shown, it is a CP system and prioritizes consistency over availability according to the CAP theorem. The client can use slave Redis replicas to issue Read requests. Overall, this protocol also resembles the primary-backup replication protocol where the master Redis replica corresponds to the primary and the slave Redis replicas act like backups.

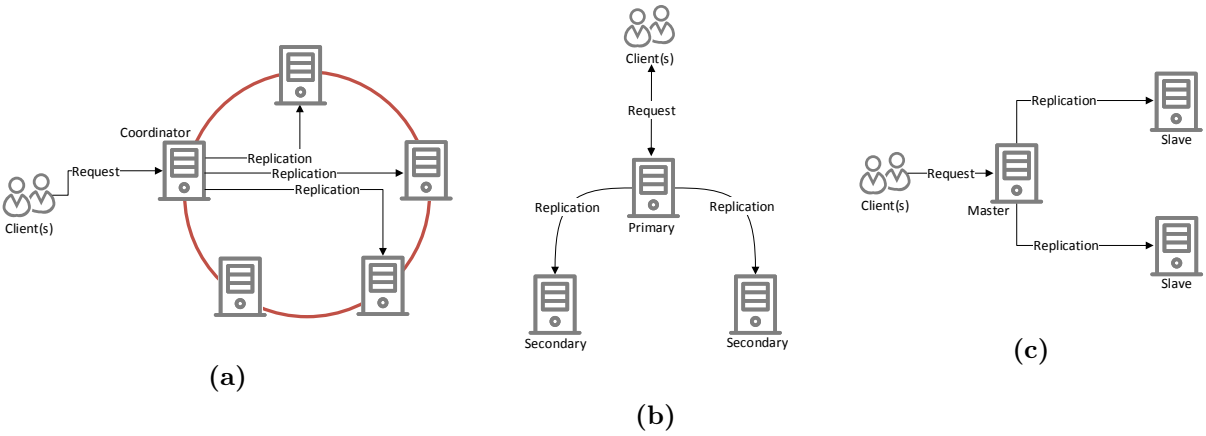

Figure 2.1: Replication Diagrams of (a) Cassandra, (b) MongoDB and (c) Redis

Furthermore, there are other notable works that aim to improve the performance of the primary-backup replication [15] or define advanced replication algorithms [16, 17]

that operate similar to the primary-backup replication, but present different features and guarantees specifically in terms of consistency. A tremendously important drawback that they all face, including our work, is that any intervention that aims to achieve better performance in primary-backup replication violates the strong consistency property and weakens the consistency level that the protocol can guarantee. Therefore, many of them either settle with causal and eventual consistency or propose novel consistency levels such as causal+ [18].

2.2 Checkpointing in Distributed Systems

Checkpointing is a frequently used technique to provide fault tolerance especially in parallel and HPC systems. Under normal operating conditions without any failures, the state of the system is saved to a stable storage so that if any failures occur in the further execution, the system would be able to return to an error-free state by restoring the latest checkpoint from the stable storage. That would also handle any kinds of unforeseen transient failures.

The checkpointing techniques discussed in the literature can be classified by considering the coordination between concurrent processes as well as considering whether there exist process blocking the execution or not. Therefore, the common way of classifying checkpointing techniques in the literature are as follows [19]. **Coordinated** techniques require all nodes involved in the execution to act collectively by collaborating each other during the checkpointing process which results in a global checkpoint that can be restored at any time without any problems. However, most of the algorithms labeled as coordinated block the execution of the whole system which means the execution pauses during checkpointing. On the other hand, **uncoordinated** techniques do not impose any coordination among the nodes, therefore it is more relaxed compared to the coordinated techniques. However, since the execution is not stopped at the same time in all nodes, it is much harder to find a global checkpoint for the system and it might result in the so-called domino effect problem which means there is no suitable global checkpoint for every node and the system has to go to its ini-

tial state [20]. The latest type is the **communication-induced** checkpointing which requires the nodes to piggyback some special information among them without any need to transfer explicit coordination messages. Moreover, this is said to prevent any domino effect that might occur while finding a global checkpoint [21].

Nevertheless, none of these checkpointing techniques are suitable to be used in a replication protocol and in a distributed setting. In fact, the problems with these techniques are that they require and consider the parallel nodes taking checkpoints to communicate with each other, however in the primary-backup replication context there are no multiple primary replicas to communicate with each other. Addressing these problems, we consider checkpointing techniques as a means to enable primary-backup replication to be more efficient in terms of fault-tolerance and to increase its overall throughput while also decreasing the response latency. Therefore, we propose integrating the checkpointing technique as a way to update backup replicas from the primary replica. Furthermore, we consider different checkpointing techniques suitable for replication, have conducted initial experimental analysis and presented the results in our prior work [4].

2.3 Compressing the Checkpoints

Table 2.1: Checkpoint Compression Mechanisms

Work	**Employed Compression Technique(s)**	**Assessed Metric(s)**
Giga-scale Checkpoint/Restore [22]	Hardware based LZW compression	Checkpoint Size, Compression Ratio, Data Prediction Rate, Address Prediction Rate, Buffer Size, Relative Performance
SIREN [23]	Delta Compression	System throughput
Viability of Compressed Checkpointing[24]	zip, 7zip, bzip2, pbzip2, rzip	Compression Factor, Compression Speed, Decompression Speed
MCRENGINE [25]	Parallel-Gzip, FPC, fpzip, QuickLZ	Compression Ratio, Checkpoint Storing Overhead, Restart Overhead, Average IO Time
Adaptive Incremental Checkpointing [26]	Delta Compression	Normalized Expected Turnaround Time

Compressing the checkpoints has been discussed in the literature as a method for decreasing the checkpoint overhead that leads poor response times to the clients. Table 1 provides a summary of the related work on checkpoint compression methods. Important mechanisms on decreasing the checkpointing overhead are diskless checkpointing [27], buffering [28], applying copy-on-write techniques [29] and also letting the compiler to determine the checkpoint locations [30]. A significant issue to note, nearly the same for all checkpointing literature, is that the compressed checkpointing studies are mainly discussed under the high performance computing (HPC) scope. To the best of our knowledge, using compression on a checkpoint algorithm that gets employed on a distributed environment and in particular replication protocol has not been studied.

CATCH [30] is the first work that discusses compressing the checkpoint data to reduce its size. They implemented a routine for compressing checkpoints using the LZW compression algorithm. However, they choose not to further analyse it as the compression process was computationally too costly; thus creates a huge overhead that is nearly impossible to compensate. Just a few years later, Libckpt [31] discusses compressing the checkpoint as a non-promising and not-implemented method since it is still computationally expensive and can only be beneficial in parallel systems that exhibit I/O contention.

In another work [22], compressing the checkpoint data using specially constructed compressors chips is discussed; however using extra hardware for a simple compression task degrades the viability of the work. Another study that discusses compressing checkpoint is SIREN [23], which is in fact a checkpointing algorithm. Interestingly, they choose to compress the checkpoint image by not using any compression algorithm but by following an incremental checkpointing technique. They accomplish it by looking at transaction logs and they keep only changed values in the checkpoint image which actually means reducing a full checkpoint image to an incremental checkpoint image.

The most extensive study on checkpoint compression [24] explores the compres-

sion ratio and the overhead of using different compression algorithms such as LZ77, LZMA, and bzip2. The test environment and the study itself aim at assessing the checkpoint performance of HPC applications. A few HPC applications, miniature scale ones, are from the Mantevo Project and the large scale application they chose to text is LAAMPS. Based on their evaluations, they discuss that it is feasible to employ compression algorithms on checkpoint data as an optimization for the runtime performance of a large scale scientific application.

MCRENGINE [25] is yet another work that utilizes compression for checkpoint data in HPC systems. The engine gathers checkpoints from different application processes and then compresses them using different compression techniques and reduces checkpoint and restart overheads. They use ALE3D, Cactus and Enzo scientific HPC applications as a test bed and parallel-gzip, FPC, fpzip and QuickLZ algorithms for compressing the data. With their work, they discuss that giving more computational time for data-aware compression produces better compression ratios and as a result lowers the IO traffic for the checkpoint data. Furthermore, they show that it is possible to reduce the IO bottleneck by 87% and reduce the restart time of a scientific application by 62%.

Compressing checkpoints is continued to be discussed in the HPC literature and in [26] they propose a new checkpointing algorithm for single threaded networked multicore systems (NMS) that uses data-aware compression on checkpoint data by moving compression process to another idle core of the system. However, instead of applying a compression algorithm like deflate or bzip2, they chose to compress the checkpoints using an adaptive incremental algorithm which calculates an estimation for the checkpointing process latency and decides if the checkpoint has to be taken or not. They show that using their own implementation of checkpointing it is possible to reduce the expected turnaround time of a scientific application by 47%.

Chapter 3

PRIMARY BACKUP REPLICATION ALGORITHMS AND CHECKPOINTING METHODS

As aforementioned in the previous chapters, both checkpointing and replication are well-known and commonly discussed techniques. Although they both have very wide application areas, to the best of our knowledge they have never been discussed together. In this chapter, we introduce how to use checkpointing and replication together to make replication more efficient in terms of system throughput and client blocking time. Furthermore, we explain our system model and present the results of performance evaluation conducted on the PlanetLab overlay.

3.1 System Model

The system is composed of geographically distributed nodes as part of the primary-backup replication protocol and clients. The clients are defined with the set of $\mathcal{CL} = \{cl_1, cl_2, \ldots, cl_n\}$ and we also define the replica nodes of the system with a super-set of $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ and two discrete sub-sets, $\mathcal{P}$ and $\mathcal{B}$ where $\mathcal{P}$ denotes the primary nodes and $\mathcal{B}$ denotes the backup nodes. Any active server within a non-failed state $s_i \in \mathcal{S}$ can either be $s_i \in \mathcal{P}$ or $s_i \in \mathcal{B}$ but not the both at the same time meaning any given replica in the primary-backup replication at any given time can be either in the primary replica mode or backup replica mode. However, it is valid for any $s_i \in \mathcal{S}$ to be $s_i \in \mathcal{P}$ at time τ_t and $s_i \in \mathcal{B}$ at time τ_{t+1} meaning it is possible for a given s_i to change its working mode from primary to backup or vise versa. Another limitation is, if $|\mathcal{S}| = n$, then $|\mathcal{P}| = 1$ and $|\mathcal{B}| = n - 1$ at any given time τ_t meaning at any time in the system there can be only one primary replica and rest of the replicas are backups. Finally, any primary-backup protocol following this approach can tolerate

$n-1$ failed $s_i \in \mathcal{S}$ given $|\mathcal{S}| = n$ meaning a primary-backup protocol consisting of n replicas can tolerate a failure in $n-1$ nodes and continue its execution without causing any further outages in the overall system or any other systems depending on it.

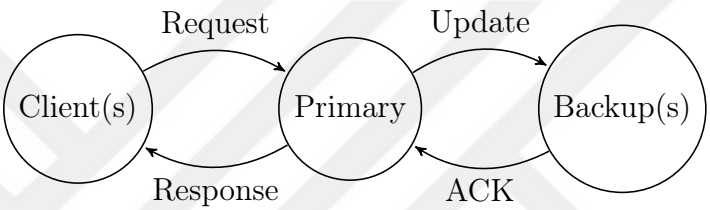

Figure 3.1: Primary-Backup Replication

A diagram of the system following the traditional primary-backup protocol is shown in Figure 3.2 and a more detailed interaction among every component of the system is given in Figure 3.1. The clients interact with the primary replica and send either update or read requests to it. When the primary receives a read request, it responses to the client immediately after issuing the read request locally. However, if the request involves an update operation, then after issuing the update locally, primary replica sends an update message to all backup replicas indicating the system state change it observed. Once backup replicas receive this update message, they acknowledge the primary accordingly and in the meantime the primary replica wait for all backup replicas to acknowledge. This behaviour enables the system to have the strong consistency property while ensuring the linearizability. Nonetheless, this strong consistency feature comes with the high latency cost observed by the client since the client has to wait the background tasks happening between the primary and backup replicas to finish before getting a response to its update request from the primary replica.

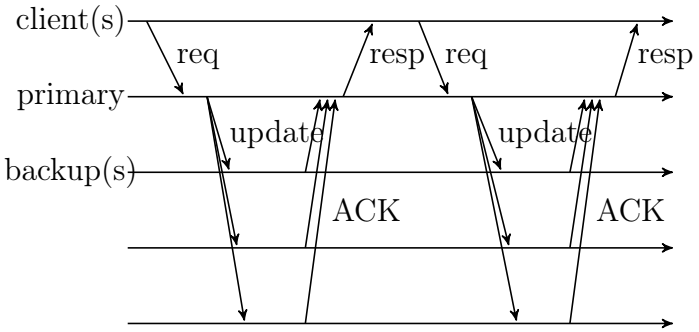

Figure 3.2: Interaction amongst the clients, primary and the backups

3.2 Replication Algorithms

Considering the traditional primary-backup replication protocol, we propose modifications in both the primary replica and backup replica to achieve better throughput, lower latency and faster recovery times.

3.2.1 Primary Replica

The primary replica algorithm given in Algorithm 1 executes a request by first checking the request type. If a READ request is received, the primary does not need to communicate with backup replicas and directly sends the value of the requested key by issuing the internal get command, and hence the response latency would be relatively low. However, if an UPDATE request is received, the latency perceived by the client would be much higher since the primary has to communicate with every backup node, wait for their acknowledgements after first internally updating the value of the given key. The main contribution in our proposed approach is, instead of defining a vague update message to be sent by the primary to backup replicas, we define a checkpoint data to be sent. Then, by considering various checkpointing methods, we investigate the efficiency of them in comparsion to the traditional update message approach. The checkpointing period and the action counter used in the algorithm are discussed in Section 3.3.4.

Algorithm 1: Primary Replica Algorithm

Input: $\mathcal{C}_T$: checkpoint type
$\mathcal{C}_{P0}$: initial checkpoint period
$\mathcal{B}_{list}$: backup replica list

$\mathcal{U}_{counter} \leftarrow 0$
$\mathcal{C}_P \leftarrow \mathcal{C}_{P0}$
upon event *new request forwarded from coordinator* **do**
  if $request.Type == READ$ **then**
    $value \leftarrow get(request.Key)$
  end
  else if $request.Type == UPDATE$ **then**
    synchronized
      $value \leftarrow put(request.Key, request.Value)$
      increment $\mathcal{U}_{counter}$
      if $\mathcal{U}_{counter} \bmod \mathcal{C}_P == 0$ **then**
        $\mathcal{CHK} \leftarrow create\_checkpoint(\mathcal{C}_T)$
        $send\_checkpoint(\mathcal{CHK}, \mathcal{B}_{list})$
        $wait\_for\_ACKs(\mathcal{B}_{list})$
      end
    end
  end
  $respond(value)$
end

3.2.2 Backup Replica

Each backup replica of the system follows the algorithm steps given in Algorithm 2. The backup replicas receive connection from the primary replica in order to get updated and not to go out of sync during the execution, therefore if a backup replica receives a connection from the primary including a checkpoint data, the backup directly applies this checkpoint to its local system state which means it has the same system state with the primary replica and in any failure event happening at primary replica, the backup is ready to take over without any extra checkpoint restoring time. This is an improvement we propose different than previous works [4, 5] which are based on the assumption that a list of checkpoints available is available in the backup replica and if it has to take over, it needs to first compose an up-to-date system state using that checkpoint list which take a few milliseconds according to our previous measurements. Another case for a backup replica is that it might receive a connection coming from the coordinator thereby from the clients, including either a READ or an UPDATE request. This means, the coordinator service has detected the failure of the primary replica, therefore the backup is selected as the new primary replica and has to go under a mode switching operation from backup to the primary. After that mode update, this replica would continue its execution by following the primary replica algorithm provided in Algorithm 1.

Algorithm 2: Backup Replica Algorithm

```
   Input: C_t: checkpointing type
1  foreach incoming data do
2  |  if data.Type = CHECKPOINT then
3  |  |  restore(data)
4  |  |  respond(ACK)
5  |  end
6  |  else if data.Type = REQUEST then
7  |  |  modeSwitch(data, C_t)
8  |  end
9  end
```

3.3 Checkpointing Methods

A conceptual discussion of checkpointing and its usage in parallel and high performance computing literature were provided in Section 2. The well-known way of classifying checkpointing methods is not suitable for a distributed application following the primary-backup replication protocol, since the primary replica is the only node executing requests and there exist no other node for creating a collaborative checkpoint. Inspired by the semantic checkpointing types defined in [32], we utilize them in our proposed framework for checkpoint-assisted primary-backup replication.

3.3.1 Full Checkpointing

In this type of checkpointing, the system creates a checkpoint data that contains the exact copy of its current state. As demonstrated in Figure 3.3, every U_i denotes an update request processed by the primary replica that results in a change in the key-value store. Therefore, after every U_i, a checkpoint C_i is created which contains the entire state of the store. For instance, after processing U_1, the system creates checkpoint C_1 which involves the initial state and the latest update U_1. However, with every U_i, the C_i contains more and more data which increases the checkpoint data size as a drawback.

The advantage of full checkpointing method is that any replica receiving a checkpoint without having an initial or partial state can directly load the checkpoint as a current state and be in synchronization with the primary replica sending the full checkpoint. However, since the entire system state is saved as a checkpoint, the full checkpoint would result in a very big data size which would take much longer time to transfer the checkpoint data from one one to the other in the network.

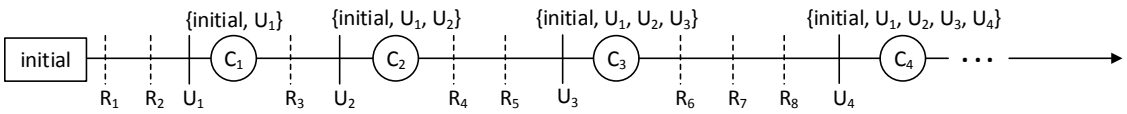

Figure 3.3: Full Checkpointing (R_i: Read requests U_i: Update requests C_i: Checkpoint requests)

3.3.2 Incremental Checkpointing

In this method, the node creating the incremental checkpoint has to either save one-step-previous state of its own or track down changes happened in the system state since the last time it created an incremental checkpoint, as depicted in Figure 3.4. Every U_i denotes an update request processed by the primary replica that resulted in a change in the key-value store. However, in contrast to the full checkpointing, only the updated data are included in every C_i. Hence, the incremental checkpointing, promises lower checkpoint data size with a few minor requirements compared to the full checkpointing. Furthermore, the node that receives the incremental checkpoint needs to have some initial or one-step-behind system state of the checkpoint owner to be able to apply the received incremental checkpoint data and be in synchronization with the sender of the checkpoint data.

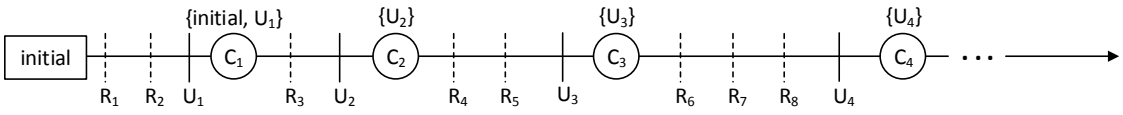

Figure 3.4: Incremental Checkpointing (R_i: Read requests U_i: Update requests C_i: Checkpoint requests)

3.3.3 Differential Checkpointing

In this checkpointing method, the node creating the differential checkpoint would compare its initial system state with the current system state (instead of the one-step-previous system state) or track down every change that has happened in the

system state since the initial state. Once it founds the difference of its current state with its initial state, the node can create the checkpoint data. The node receiving the checkpoint needs the following to construct an in-sync system state with the sender; the initial state of the sender and the checkpoint data.

The differential checkpoint can be considered as a special case of the incremental checkpoint. It is advantageous, if only a few parts of the system state is being altered regularly, however, if big changes occur compared to the initial state, the resulting checkpoint data would be large and take longer transfer times in the network.

An example working scenario for differential checkpointing is demonstrated in Figure 3.5. Every U_i stands for an update request processed by the primary replica that result in a change in the key-value store and a checkpoint C_i is created afterwards accordingly. While creating a C_i, the node compares its current state with its initial state. Therefore, in the given scenario, every C_i contains very similar data to the full checkpoint case given in Figure 3.3 with an exception that the initial state is not included.

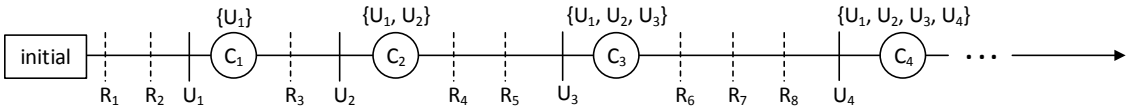

Figure 3.5: Differential Checkpointing (R_i: Read requests U_i: Update requests C_i: Checkpoint requests)

3.3.4 Periodic Checkpointing Variants

All semantic checkpointing methods can be applied periodic variants as illustrated in Figure 3.6 where the node keeps track of a period denoted by p before creating the next checkpoint. The period $p > 1$ dictates that p state changes has to occur before creating a new checkpoint. For instance, a checkpoint C_i is created after every p requests. Therefore, the primary replica would take a checkpoint and send it to the backup replicas after every p state changes. Although, the definition is easy to follow,

there exist important issues to consider in periodic checkpointing methods.

In periodic full checkpointing, the checkpoint data size would not be affected by the period length. However, in the case of periodic incremental checkpointing, the checkpoint data size would increase as the period length gets larger. For the periodic differential checkpointing case, there is no definite way of assuming whether the checkpoint data size would increase or not depending on the period length.

The periodic variants of semantic checkpointing methods would result in a degrade of the consistency level.Therefore, the periodic checkpointing types utilized in the primary-backup replication would break the strong consistency property. The reason behind is that there would be a delay between the state changes in one node and their reflection in the other. Hence, if the node ahead fails before delivering these changes to the other as the checkpoint period was not completed, the other would be left behind and experience inconsistency in some data updates.

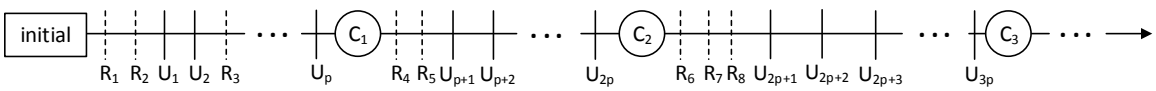

Figure 3.6: Periodic Checkpointing (R_i: Read requests U_i: Update requests C_i: Checkpoint requests)

3.3.5 Compressed Checkpointing

The main idea behind compressed checkpointing methods is to compress the resulting checkpoint data, so that it would take less time to transfer it through the network which would likely lower the latency and improve throughput. Thus, an efficient compression method applied to the checkpointing mechanism has the potential for performance improvements.

However, it is noteworthy to state that there exists a trade-off that compressing a checkpoint data plus transferring the compressed data over the network should not take longer than transferring the uncompressed checkpoint data. Otherwise, the compression would not help but worsen the performance. This also depends on the

compression method being used since most of the compression libraries promising very high compression ratios may have longer compression times, so a balanced compression method needs to be chosen to take advantage in terms of reduced network latency.

Chapter 4

EXPERIMENTAL SETUP

Nodes of the system are geographically distributed and the system functions as a geographically replicated fault-tolerant key-value store. From the clients point of view, the system is observed as one piece and they do not interact with different replicas but with a single node. This enables a fault-tolerant structure and moving any backup replica into the primary replica mode without interrupting the communication with clients. However, due to the nature of TCP connections, this is not possible and hence we utilize an external service called coordinator in our experimental setup, that is placed in between the clients and the primary-backup replicas.

The coordinator node is considered as a trusted and stable node that acts as a TCP forwarder in this configuration as shown in Figure 4.1. The real purpose of this service is to keep track of the primary and backup replicas so that the clients perceive the system as a single box and do not need to keep track of distributed nodes included in the replication and their current operating modes.

Furthermore, it is noteworthy that coordinator services are widely used in distributed settings to handle the communication between the clients and the system and there exist different options to have a fault-tolerant coordinator service as well, such as enabling floating IP address services and having multiple coordinator nodes or using DNS fail-overs with very low TTLs, yet we do not focus on any kind of failures of the coordinator node.

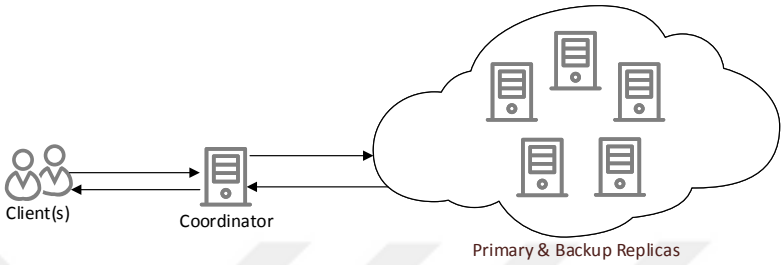

Figure 4.1: Experimental Setup

4.1 PlanetLab Testbed

The experimental testbed consists of geographically distributed replica servers on the PlanetLab research network [33]. Table 4.1 provides information regarding the nodes used where each node has a different initial role in the configuration. There exist initially five backup replica nodes, one primary replica node and one coordinator node. Note that, the entire PlanetLab overlay network is shared with many researchers running different workloads simultaneously, and subject to the large amount of background traffic and processing.

Table 4.1: Utilized replica nodes on the PlanetLab network

Hostname	Initial Type	CPU
planetlab1.cs.okstate.edu	Primary	Intel Xeon X3430
planetlab2.cs.okstate.edu	Backup	Intel Xeon X3430
planetlab03.cs.washington.edu	Backup	Intel Pentium D
earth.cs.brown.edu	Backup	Intel Xeon 3060
planetlab5.williams.edu	Backup	Intel Xeon X3330
planetlab2.telenet.unc.edu	Backup	Intel Pentium D
planetlab-04.cs.princeton.edu	Coordinator	Intel Xeon X5650

4.2 Key-Value Store

The database or the key-value store choice is another factor that has a significant affect on the system throughput and the latency. Choosing any key-value store with a default support for networking tasks, replication or any other feature would be also limiting our experiments since we had to either use the supported variations or try to implement something that would work for that store which would mean some more extra unneeded effort. Therefore, we use the RocksDB [34] as a key-value store backend in our system which is an open-source key-value store being developed by Facebook and used internally by them as well as with companies such as Yahoo! and LinkedIn for various projects.

RocksDB was initially built on the LevelDB which is developed by Google [35]. The most important feature that LevelDB was lacking is the Java support since we build our system using Java, we needed a key-value store providing a Java API. Many statistics also show that RocksDB perform better than the LevelDB thanks to custom optimizations and improvements carried out by Facebook.

4.3 Coordinator

The coordinator has an important task in our system configuration. The clients interacting with the system only know the IP address of the coordinator so that they do not need to keep track of which replica is in the primary position and should be contacted. Therefore, the coordinator forwards incoming requests to the current primary replica. Accordingly, the primary servers sends its response to the coordinator and it forwards this response to the client.

We use the HAProxy [36] service which is a reliable and high performance TCP / HTTP load balancer tool. It is widely used in many web services such as Twitter, Tumblr, Instagram and StackOverflow. In our configuration, we use the Layer 4 load balancing by enabling health checks on our replicas so that if the primary replica is down, a backup replica is automatically selected as the new primary by the HAProxy.

The health checks are performed via checking if the replica listens on the given TCP port.

4.4 Compression Algorithms

As briefly discussed previously, compressing a checkpoint might result in better performance if appropriate compression algorithms are used. In our previous work [5], we have analyzed Deflate [37], LZMA2 [38] and Zstd [39] compression algorithms and show that Zstd outperforms other libraries. We extend this work and analyze additional contemporary compression libraries along with the Zstd.

Snappy [40]:

The aim of Snappy is having a blazingly fast compression time while having a moderate compression ratio. Snappy is developed by Google and being internally used for BigData and MapReduce operations.

GZIP [41]:

It is the ZIP implementation made by GNU, promising better compression stats compared to Deflate algorithm [37].

Zstd [39]:

It was open-sourced by Facebook in mid-2016 and the aim is to achieve high compression ratio for small data sized in near real time.

4.5 Benchmarking Tool and Metrics

We use the Yahoo! Cloud Service Benchmarking (YCSB) tool for benchmarking and measuring various metrics of our system and proposed algorithms [42]. It provides realistic workload scenarios to test various key-value stores and cloud systems and even allow custom implementations through extending the provided interfaces. In our experiments, we measure various performance metrics described in Table 4.2 and use the following workloads with a total of 5000 requests originated from 64 client threads and 5000 initial database records:

- W-medium: 50% read - 50% update requests
- W-writeHeavy: 25% read - 75% update requests
- W-readHeavy: 75% read - 25% update requests

Table 4.2: List of Metrics

Name	**Definition**	**Unit**
Blocking Time	Latency perceived by a client after sending a request	ms
Checkpointing Time	Time spent on the primary replica for creating a checkpoint, sending it to backup replicas and receiving acknowledgements	ms
Checkpoint Size	Size of the checkpoint data	bytes
Failover Time	Duration for which there is no primary replica in the system	ms
System Throughput	Maximum number of requests which primary replica can process in a second	requests/sec
Metrics below are considered only for algorithms using compression		
Compression Ratio	Amount of compression that can be applied on the raw checkpoint data	-
Compression Time	Duration for compressing a checkpoint data	ms

Chapter 5

EXPERIMENTAL ANALYSIS AND PERFORMANCE RESULTS

This chapter presents details of experimental analysis and performance results. First, the results of non-periodic checkpointing techniques are described while comparing them with the traditional primary-backup replication. Next, the results for periodic checkpointing techniques are demonstrated. Finally, results for the periodic-incremental checkpointing technique combined with different compression libraries are provided, followed by overall discussion of analysis results.

5.1 Non-Periodic Checkpointing Techniques

Figure 5.1a shows the comparison of the average blocking time which is the latency perceived by the YCSB tool after every request. As expected, when the workload involves more update request, the average blocking time also increases for every checkpointing technique including the traditional primary-backup replication. More update requests means more changes occurring in the system state and the backups need to be updated more frequently and it in return increases the blocking time.

This discussion can also be backed up by analyzing the Figure 5.1b as well. It shows that, the average checkpointing time is very similar for every workload type for a given checkpointing technique. Therefore, more checkpointing operation results in a higher blocking time. It can also be discussed that the traditional primary-backup replication internally operates very similar to the incremental checkpointing that we have defined. The worst performing checkpointing technique is the full checkpointing, since it has to transfer the whole system state, which in our case means the whole key-value store, from primary to backup replicas and the best performing one is the

incremental checkpointing technique. The differential checkpointing technique is not much suitable in this type of system as the system state involves more and more changes with every new request compared to the initial state.

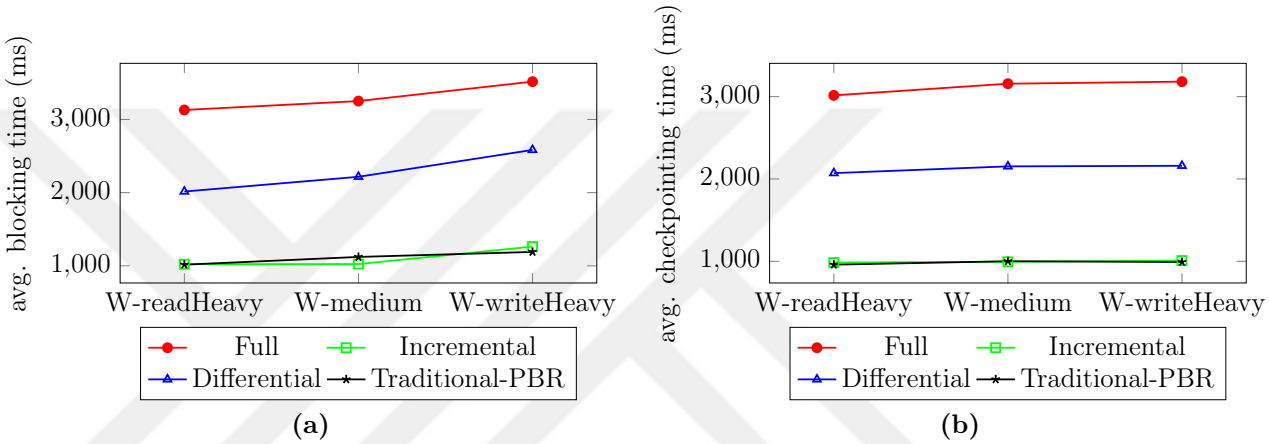

Figure 5.1: Blocking and Checkpointing Time Measurements

Figure 5.2a shows the size of the checkpoint data that is transferred from the primary replica to backup replicas during the checkpointing operation. As the whole database if transferred in the full checkpointing, the checkpoint size is around 5.5-6 megabytes and this also explains the behaviour shown in Figure 5.1a and Figure 5.1b. Transferring a few megabytes of data from one place to another after every request results in performance degradation. The incremental checkpointing and the traditional primary-backup replication transfer only the change occurred in the system state so they result in just a few bytes of checkpoint data.

The last measurement to be reported is the metric showing the overall system performance and is given in Figure 5.2b. Naturally these values are inversely proportional to the results given in Figure 5.1a. If the client is blocked for a very large time after a single request, the throughput observed would also be very low. The full checkpointing type results in a throughput less than 1 and hence it is not applicable in practical system. The incremental and the traditional primary-backup replication achieve a constant throughput around 5 and depending on the workload

type, can reach around 6 requests processed per second. However, it is noteworthy to state that still these are very low throughput values due to the strict enforcing of the strong consistency.

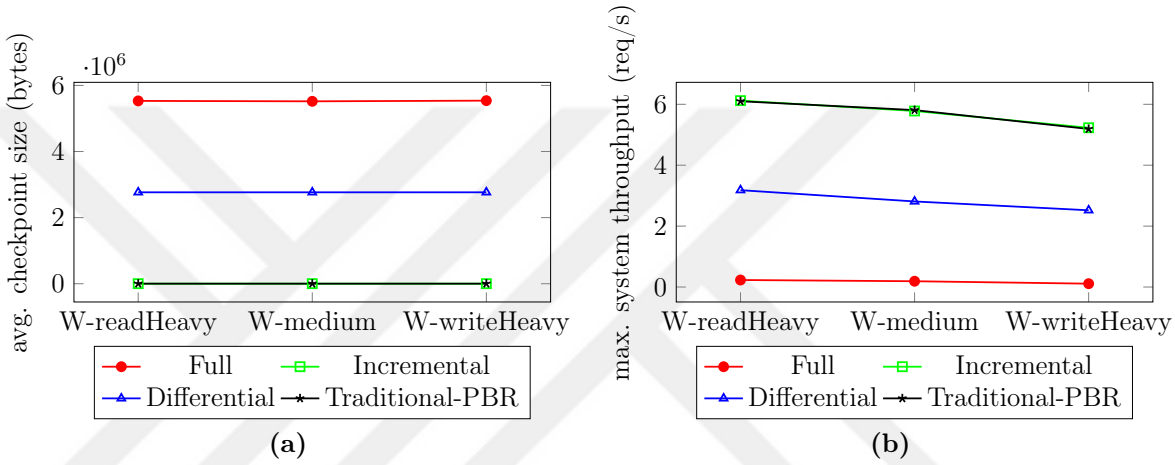

Figure 5.2: Checkpoint Size and Overall System Throughput Measurements

5.2 Periodic Checkpointing Techniques

Figure 5.3a shows the average blocking time measurements for the periodic checkpointing algorithms with different workloads. Naturally, the average blocking time increases as the workload includes more update requests causing a more frequent checkpointing fashion which in return causes prolonging blocking in which the write-heavy workload case having the highest average blocking time readings. The values given in Figure 5.3b indicate an interesting point which is the case of clients getting blocked for more than 2 seconds if a checkpointing process is going on which is an unacceptable latency in practice. However, this behaviour can only be overcome by allowing the checkpoint process to take place asynchronously which would even decrease the consistency level of the system. Nonetheless, the periodic incremental checkpointing achieves the lowest checkpointing time values since it tracks down fewer data compared to the periodic full checkpointing which still tries to transfer the whole

database periodically leading to tremendously long checkpointing times.

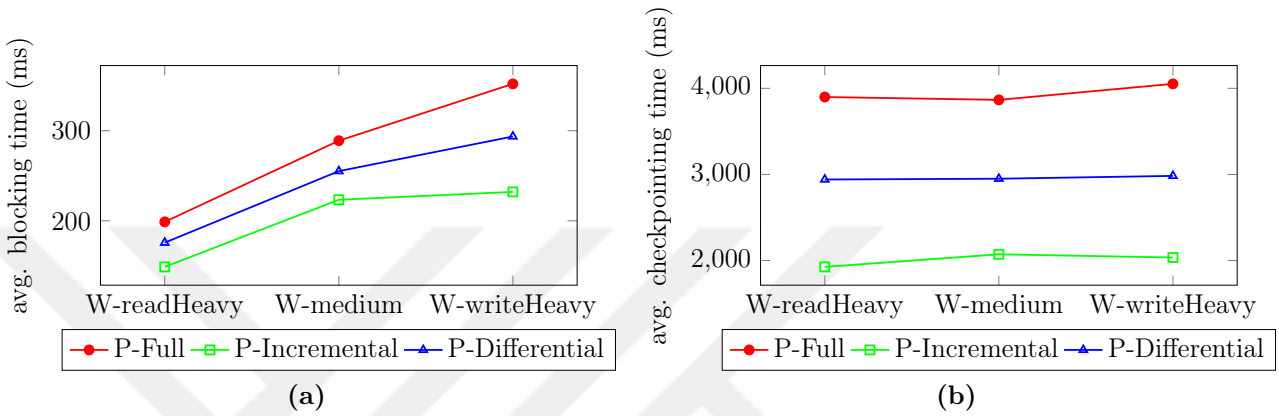

Figure 5.3: Blocking and Checkpointing Time Measurements (period=500 update requests)

Figure 5.4a shows that checkpoint size is not related to workload type but only the checkpointing type itself. Although compared to non-periodic incremental checkpointing, periodic checkpointing in our experiments, creates up to 60 times bigger checkpoint data, it is still a few kilobytes whereas the periodic full checkpoint technique has the same data size with the non-periodic version.

The throughput measurements are given in Figure 5.4b. Just like the blocking time readings given in Figure 5.3a but in the opposite manner, as the proportion of the update requests enclosed in the workload increase, the amount of requests that the system can process in a second drops mildly. However, comparing these throughput results to the non-periodic checkpointing types given in Figure 5.2b indicate that periodic techniques promise up to nearly 70 times better performance as the periodic-incremental checkpointing peaks with more than 400 req/s.

5.3 Periodic Incremental Checkpointing with Compression

Blocking time measurements of the periodic-incremental checkpointing coupled with different compression libraries are given in Figure 5.5a. The values show significant improvement over Figure 5.3a and Figure 5.1a. Average blocking time that

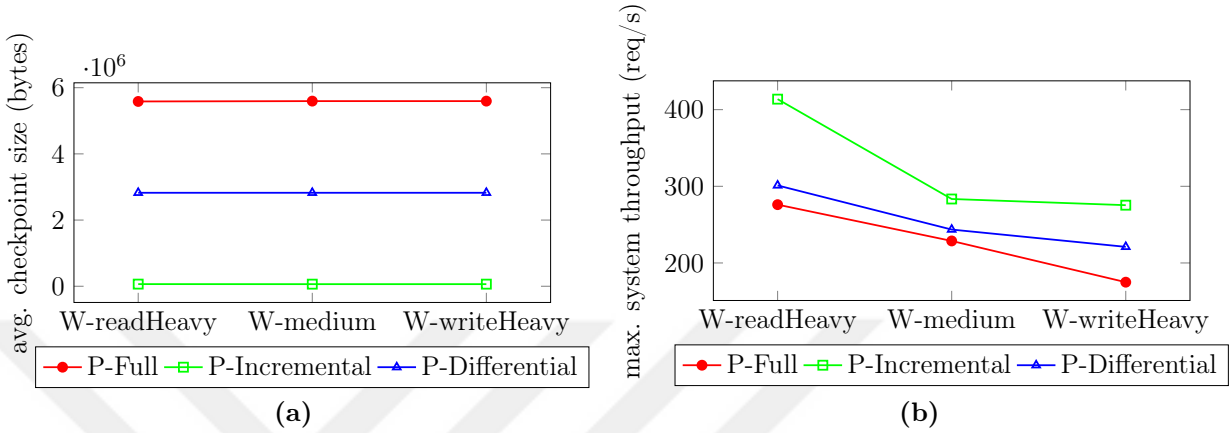

Figure 5.4: Checkpoint Size and Overall System Throughput Measurements (period=500 update requests)

the periodic-incremental checkpointing with Snappy compression reaches is nearly 10 times lower than the traditional primary-backup replication. Zstd compression library also attains very close results with the Snappy compression library yet it is obvious that in each workload case, Snappy library can provide the lowest timings. Figure 5.5b demonstrates very close results for each library in terms of checkpointing times as some of them either compresses blazingly fast and brings moderate compressed checkpoint size or compresses in a fair time but attains the smallest compressed checkpoint size. As can be seen, each library has few milliseconds of difference between them in terms of checkpointing time.

Figure 5.6 shows the compressed checkpoint data sizes for each different cases. Checkpoint size is similar for each workload type as the update request proportion does not change the amount of data altered in the fixed period. The most important factor in the checkpoint size values is the maximum compression ratio that can be attained by the compression library and it is given in Table 5.1. The Zstd compression algorithm achieves around 35% compression ratio followed by the GZIP around 33% thus resulting in the lowest checkpoint size readings whereas the Snappy library can barely reach 13% compression ratio thereby having the highest checkpoint data size values in Figure 5.6. Table 5.1 also s interesting compression and decompress-

ing timings for each compression library. As promised by Snappy, although it cannot achieve high compression ratio, the compression and decompression times are tremendously low compared to other compression libraries despite all having considerably fast readings.

The overall throughput of the system reported by the YCSB tool is given in Figure 5.7 and shows improved results especially for medium and write heavy workloads compared to periodic-incremental checkpointing results given in Figure 5.4b. Thanks to the compression with the Snappy library, the overall throughput does not drop below the 300 req/s even in the write-heavy workload case and climbs up to 450 req/s in the read heavy scenarios. The Zstd compression library is also on the back of the Snappy compression library with having just a bit lower throughput reading.

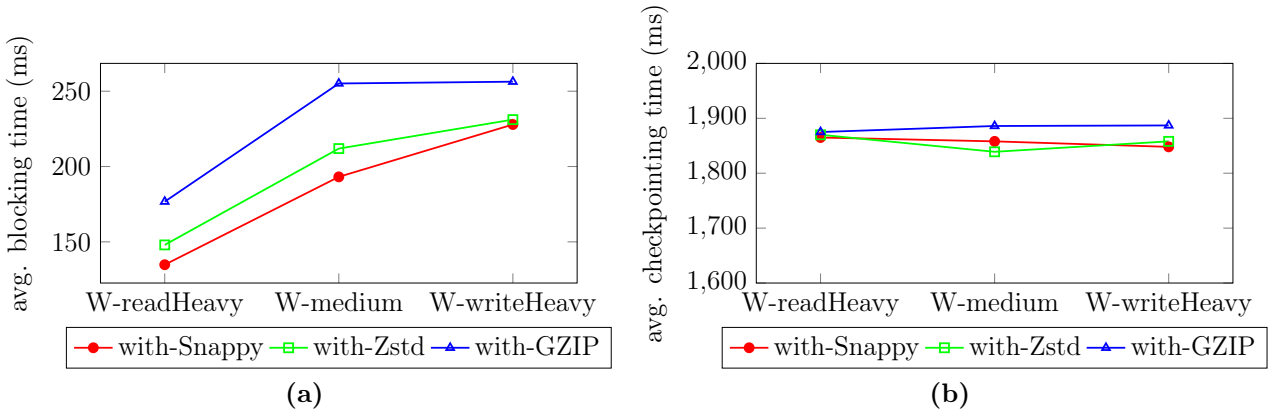

Figure 5.5: Blocking and Checkpointing Time Measurements

Table 5.1: Compression Related Metrics

Compression Library	Avg. Compression Ratio(%)	Avg. Compression Time(ms)	Avg. Decompression Time(ms)
Snappy	13%	0.25	0.05
Zstd	35%	1.06	0.25
GZIP	33%	1.78	0.65

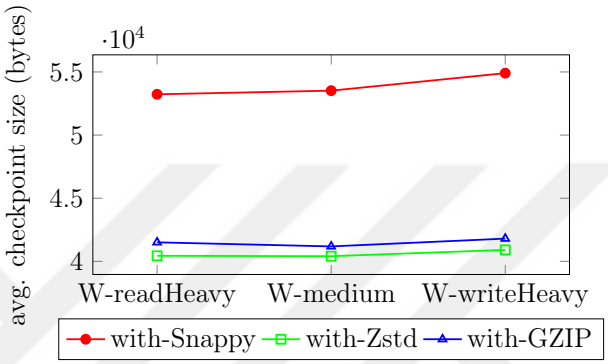

Figure 5.6: Checkpoint Size Measurements

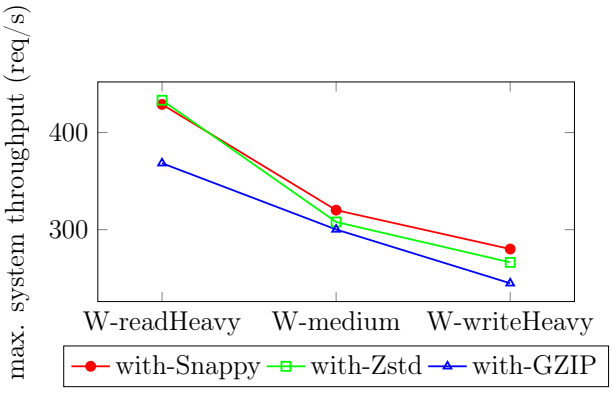

Figure 5.7: Overall System Throughput Measurements

5.4 Discussion of Results

In this section, we compare the best performing technique for each set of results along with the traditional primary-backup replication. Figure 5.8a demonstrates the comparison in terms of the blocking time metric. As mentioned in the previous sections, periodic techniques obtain tremendously low average blocking time compared to non-periodic techniques as the non-periodic algorithms have a drawback which requires them to checkpoint after every update request which in return results in unacceptable blocking times. Enabling compression with the Snappy compression algorithm further improves the blocking time especially in read-heavy and medium workloads compared to the periodic-incremental checkpointing that does not employ any compression.

Fig 5.8b presents the overall throughput comparison. As aforementioned, the traditional primary-backup replication and incremental checkpointing attain very similar results which are around a few requests per second due to very high blocking time occurring after every update request which is not desirable in a modern system. However, considering the physical conditions and the geographically distributed setup, periodic techniques reach reasonably high throughput measurements. Furthermore, enabling the Snappy compression, especially results in an improvement in the medium workload.

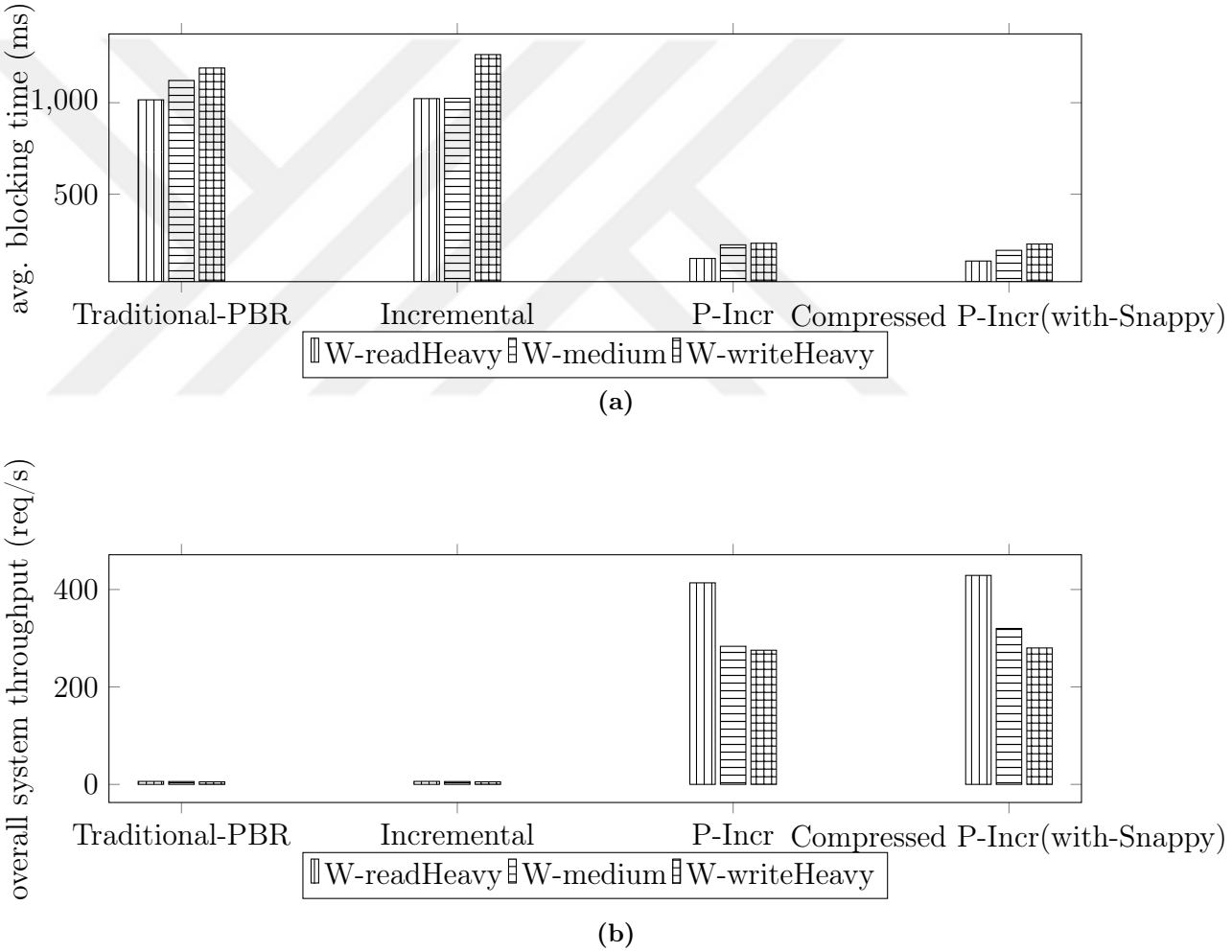

Figure 5.8: Average Blocking Time and Overall System Comparisons

Chapter 6

LACPB: LOAD-AWARE INCREMENTAL CHECKPOINTING

Based on our experimental analysis and findings, we propose load-aware compressed incremental checkpointing method (LACPB) for large scale primary-backup replication, and present the accordingly modified primary-backup replication protocol algorithms. After that, we discuss the performance analysis of LACPB and comparative results.

6.1 Effect of Checkpointing Period

As demonstrated in the previous chapter, employing a periodic approach along with the compression provides exceptionally higher system throughput and decreased blocking time albeit lowering the consistency level than strong consistency. However, an important issue is how to set the correct period for the previously proposed periodic checkpointing methods. Furthermore, selecting a static period p might be suitable for some workload characteristics.

Consider a large system period p value and system load as x req/s. If x value is very low compared to the p, this means that the period would be completed in a large amount of time and it would be inevitable to loose intermediate un-checkpointed data. On the contrary, if x value is very high, the system would be overloaded by not only the incoming requests but also trying to create checkpoints and send them to backup replicas several times in a second. Addressing these issues, we propose load-aware incremental checkpointing technique namely LACPB which aims to set the period value dynamically on the run-time depending on the current system load.

Figure 6.1 shows the effect of different period values under static system load.

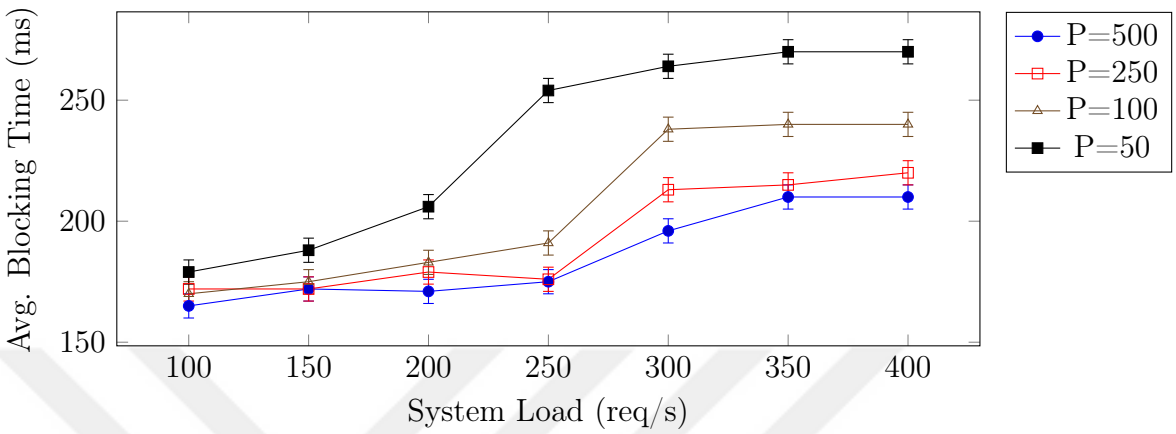

Figure 6.1: Avg. Blocking Time measurements vs. Different Checkpointing Periods

The experimental setup is identical to the one described in the previous chapters, and the W-medium workload defined in Section 4.5 is used with the YCSB tool. It is evident that, having higher checkpointing period results in lower average blocking time readings in higher system loads, but the blocking time difference is not big for lower system loads.

6.2 Load-Aware Checkpointing and Modified Primary-Backup Replication Algorithms

This section introduces our proposed LACPB algorithms along with the coordinator algorithm in order to present a thorough protocol definition. However, the coordinator algorithm gives a brief characteristic overview of the real implementation and the details of the coordinator service were already mentioned in Section 4.3.

6.2.1 Coordinator Service Algorithm

Algorithm 3 shows how the coordinator service algorithm works briefly. It has one input set and the algorithm automatically explodes this set into two subsets of the primary node and backup nodes. Afterwards, the coordinator starts listening for incoming requests from the clients. As soon as it receives a new request, the

coordinator forwards this request to the current primary node and expects a result. If a **TIMEOUT_ERROR** occurs during the execution, coordinator service picks a new primary node out of the backup nodes and eliminates the old primary node that caused the failure. It again tries forwarding the request until successfully forwarding it and retrieving a result. Once it gathers the result, it responds to the client.

Algorithm 3: Coordinator Service Algorithm

```
   Input: R_list: list of replica nodes
 1 P_node ← R_list.primary_node // primary node
 2 B_list ← R_list.backup_nodes // list of backup nodes
 3 upon event new request from a client do
 4 |   result ← forward_request(P_node)
 5 |   while result == TIMEOUT_ERROR do
 6 |   |   P_node ← pick_primary(B_list)
 7 |   |   result ← forward_request(P_node)
 8 |   end
 9 |   respond(result)
10 end
```

6.2.2 Primary Replica Algorithm

The modified primary replica algorithm is defined in Algorithm 4 that has two parallel events to track. One is upon receiving a forwarded request from the coordinator, executing the request accordingly and if needed creating a new checkpoint, sending that to backup nodes, collecting ACKs and responding to the coordinator service. The other event that is tracked is periodic task automatically timed out every $\mathcal{T}$ seconds which is an input to the algorithm. Once it is triggered, a new checkpoint period $\mathcal{C}_p$ is calculated and set accordingly. The calculation is done via the linear function shown in the line 21. $\mathcal{T}$ and α input values are the most important key points in this algorithm. $\mathcal{T}$ defines a time-window for the checkpointing period to be re-set and the α value is used during the re-calculation of the checkpointing period.

Algorithm 4: Primary Replica Algorithm

Input: $\mathcal{C}_T$: checkpoint type
$\mathcal{C}_{P0}$: initial checkpoint period
$\mathcal{T}$: window-size
α: predefined coefficient
$\mathcal{B}_{list}$: backup replica list

$\mathcal{U}_{counter} \leftarrow 0$
$\mathcal{C}_P \leftarrow \mathcal{C}_{P0}$
upon event *new request forwarded from coordinator* **do**
  if $request.Type == READ$ **then**
    $value \leftarrow get(request.Key)$
  end
  else if $request.Type == UPDATE$ **then**
    synchronized
      $value \leftarrow put(request.Key, request.Value)$
      increment $\mathcal{U}_{counter}$
      if $\mathcal{U}_{counter} \bmod \mathcal{C}_P == 0$ **then**
        $\mathcal{CHK} \leftarrow create\_checkpoint(\mathcal{C}_T)$
        $send\_checkpoint(\mathcal{CHK}, \mathcal{B}_{list})$
        $wait\_for\_ACKs(\mathcal{B}_{list})$
      end
    end
  end
  $respond(value)$
end
upon event *every* $\mathcal{T}$ *seconds* **do**
  $\mathcal{C}_P \leftarrow system\_load() * \alpha$
end

6.2.3 Backup Replica Algorithm

Algorithm 5 shows how the backup replica operates. The backup replica constantly waits for incoming data and when new data arrives, it checks the type of the data. Under normal operating conditions, if there is no failure in the primary replica, the backup replica should never receive a request from the coordinator and always needs to receive only checkpoint data from the primary node. Once it receives a new checkpoint data, backup replica restores the checkpoint and sends an ACK back to the primary replica. However, if the primary replica fails, then the coordinator forwards a request originally coming from a client. Upon receiving a new request, the backup replica goes under a mode switching operation which results in transforming this backup replica into a primary replica. Thereafter, this replica moves on to run Algorithm 4.

Algorithm 5: Backup Replica Algorithm

Input: $\mathcal{C}_t$: checkpointing type

```
1 foreach incoming data do
2     if data.Type = CHECKPOINT then
3         restore(data)
4         respond(ACK)
5     end
6     else if data.Type = REQUEST then
7         modeSwitch(data, C_t)
8     end
9 end
```

6.3 Choosing $\mathcal{T}$ and α Values

The load-aware checkpointing system depends on two crucial values, $\mathcal{T}$ and α, to operate as seen in Algorithm 4. This section discusses the effects of different $\mathcal{T}$ and α values on the client blocking time. The experimental setup is the same with the one given in Section 4 and we use the YCSB tool to calculate the blocking time. The workload is the W-medium defined in Section 4.5.

6.3.1 Effect of Parameter $\mathcal{T}$ on the Blocking Time

The $\mathcal{T}$ value decides, how frequent should the checkpointing period $\mathcal{C}_p$ be updated with respect to the current system load and its unit is second. The experimental analysis demonstrates the effect of different $\mathcal{T}$ values ranging from 1 to 50 with various step sizes and the goal is to find a $\mathcal{T}$ value that gives the lowest blocking time readings.

Figure 6.2 shows that the lowest average blocking time reading can be obtained with setting $\mathcal{T} = 5s$. Although the difference with lower $\mathcal{T}$ values does not seem to be quite much, the higher values result in evidently elevated blocking times, therefore $\mathcal{T} = 5s$ is taken as the default value. Higher $\mathcal{T}$ values means that the system would follow the dynamic system load from a more distant point so would not be able to adapt itself in a rapid fashion which results in poor blocking times whereas very small values of $\mathcal{T}$ would force the system to adapt to the current system load faster than the required which would create overloading effects on the system.

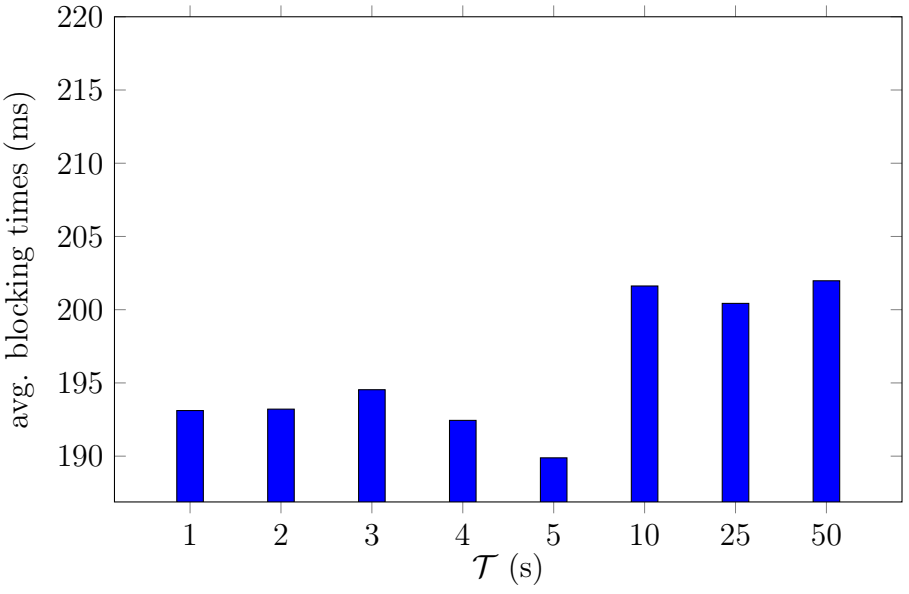

Figure 6.2: Avg. Blocking Time vs $\mathcal{T}$ ($\alpha = 1.5$)

6.3.2 Effect of Parameter α on the Blocking Time

The α value is coefficient used in the linear formula while calculating the new checkpointing period $\mathcal{C}_p$ as shown in Algorithm 4. The following evaluation demonstrates the effect of various α values, ranging from 0.5 to 2.0 with a step size of 0.25, on the average blocking time.

Figure 6.3 shows that the lowest average blocking time can be obtained by setting $\alpha = 1.5$. Basically, setting a value of $\alpha < 1$, would indicate that the system would create more frequent checkpoints in contrast to setting it to a value of $\alpha > 1$. Creating too frequent checkpoints would overload the system and have a negative effect on the overall system performance and creating too less checkpoints would result in very big checkpoint data in terms of data size that needs to be transferred over the network which in turn would decrease the system performance because of high IO waits on the network.

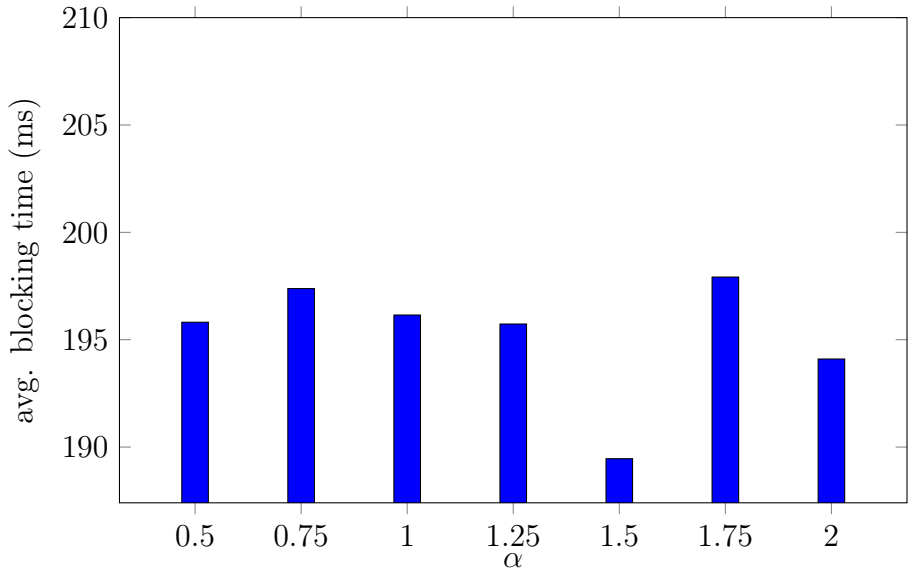

Figure 6.3: Avg. Blocking Time vs α ($\mathcal{T} = 5$s)

6.4 Experimental Analysis and Results

This section demonstrates the analysis, results and discussion of results of the proposed load-aware checkpointing method. The experimental setup is given in Section 4 and we use the YCSB to evaluate different scenarios using the load-aware checkpointing. However, since YCSB only provides static workloads, we use a modified YCSB in order to obtain the following dynamic workload pattern given in Figure 6.4. The core workloads are the same workloads defined in Section 4.5 so only the request rate pattern is dynamic depending on the time.

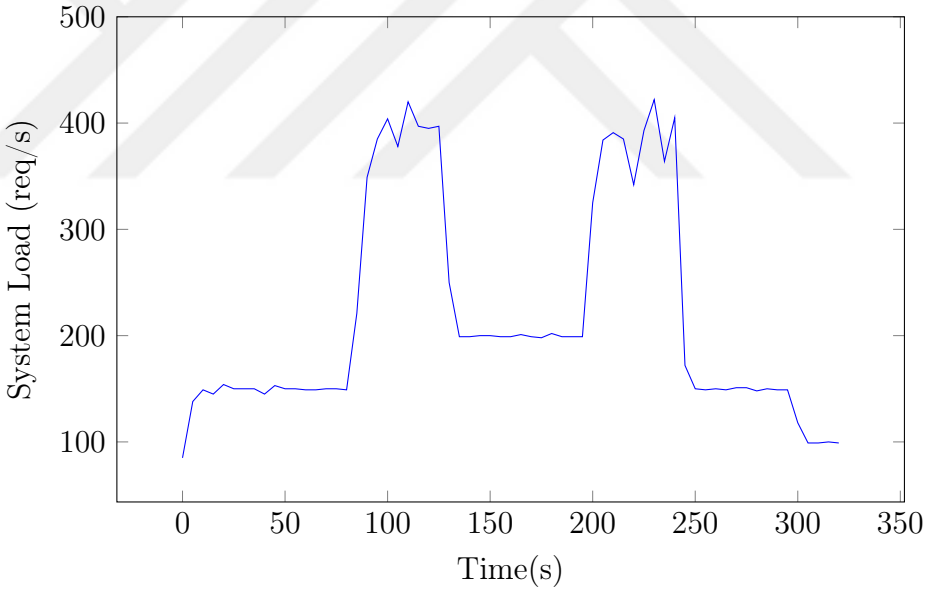

Figure 6.4: Dynamic Workload Pattern vs Time

The following results and discussions compare the previously proposed the most efficient checkpointing technique compressed periodic incremental checkpointing (CP-Incr) with the load-aware compressed periodic incremental checkpointing (LACPB), both of which uses the Snappy library to compress checkpoint data. CP-Incr uses the static period of 500 actions.

Figure 6.5 shows the average time spent for checkpointing which basically includes creating the checkpoint data, compressing it using the Snappy library, sending it to all

backup nodes and collecting an ACK from all of them. Although the time difference might seem rather high at a first glance, actual readings differ around less than 100ms. Therefore, it is evident that using LACPB has no clear overhead and instead can help reducing the checkpointing time. This is because the average request rate of the system is around 200-250 req/s and the average period length set by the LACPB is probably around 300 which is less than the static period of CP-Incr. Therefore, the shorter period is resulting in lower checkpoint size which shortens the checkpointing time.

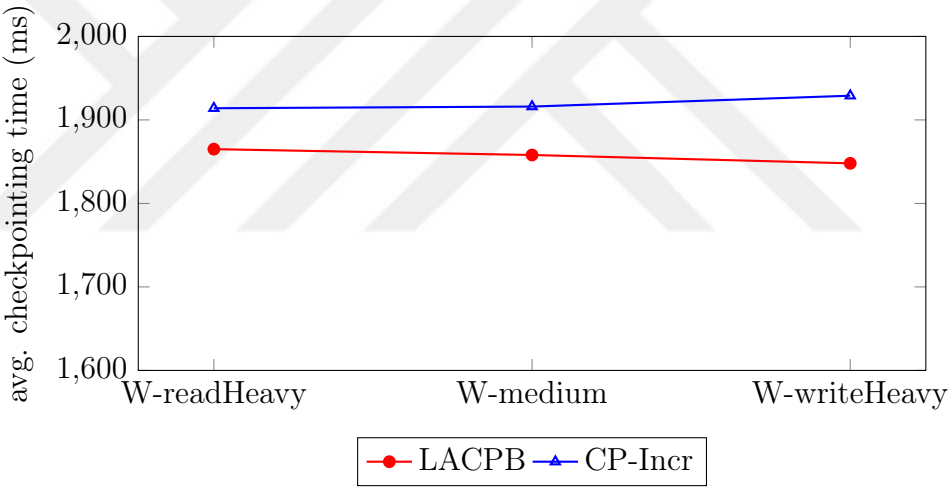

Figure 6.5: Checkpointing Time Measurements

As briefly mentioned above and it can also be seen in Figure 6.6 that the CP-Incr has higher checkpoint size since it runs on a higher static period (500 actions) resulting in to contain more changes occurred between that period starting and end points. On the contrary, LACPB sets the period length on the run time depending on the current workload of the system and this results in an average period of roughly 300 actions. Therefore every checkpoint includes less data changes compared to the CP-Incr and ultimately the created checkpoint data has smaller checkpoint size. The advantage of having smaller data size is it requires shorter amount of time to transfer it over the network.

The most important metric measurements are given in Figure 6.7. We present the

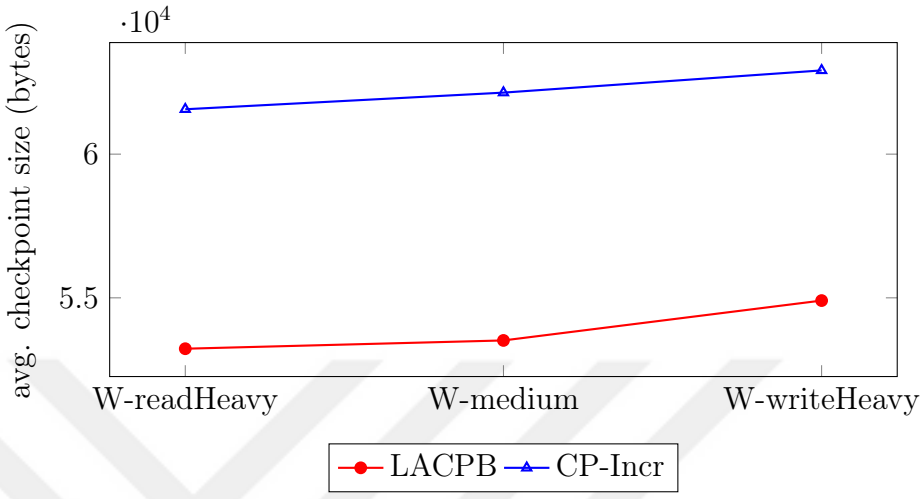

Figure 6.6: Checkpoint Size Measurements

blocking time comparison of CP-Incr and LACPB in Figure 6.7a and although the difference is not big, it is shown that the LACPB helps reducing the average blocking time observed by a client which is a really important metric considering which makes users to perceive a system as working slow or fast.

Same for the overall system throughput, it can be seen in Figure 6.7b that the LACPB obtains higher throughput values in all workload types compared to the CP-Incr. LACPB outperforms the CP-Incr in all comparisons since it can adapt its own checkpoint period $\mathcal{C}_p$ by observing the system load. Therefore, using LACPB should be not only improving the efficiency of the system in terms of blocking time and system throughput but also should be increasing the consistency of the system but the further analysis of this fashion is yet to be analyzed.

6.5 Fault Tolerance Behaviour of LACPB

In this section, we provide additional fault-tolerance analyze for the LACPB method we proposed and present a live snapshot of the response time of the system captured by the YCSB tool.

We use the W-writeHeavy workload type defined in Section 4.5 on the YCSB tool and the experimental setup is the same with the one given in Section 4. However,

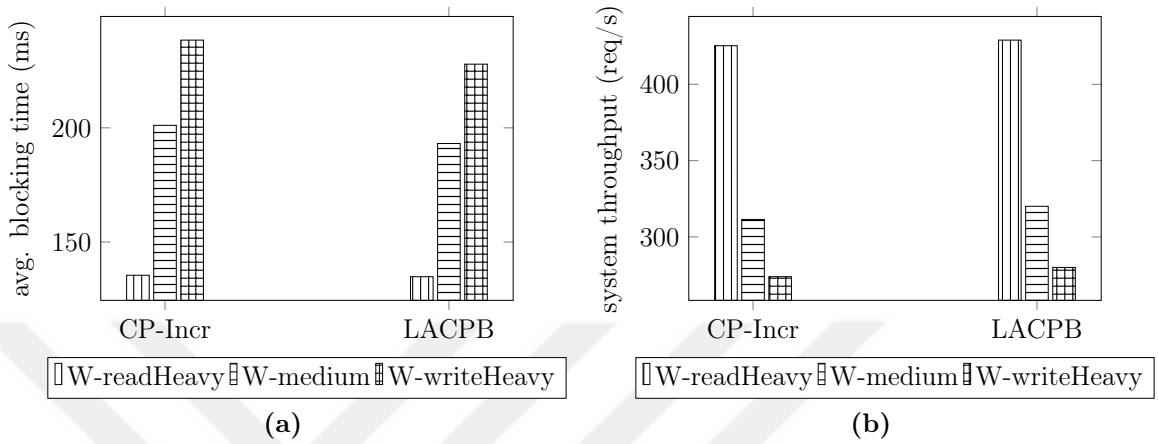

Figure 6.7: Average Blocking Time and Overall System Comparisons

instead of simulating 64 clients with YCSB, this time we force YCSB to use single client mode in order to obtain better and smoothed blocking time readings. We evaluate the system for five minutes and every minute we manually crash the primary replica that is in charge at that time.

The sudden spikes that can be seen in the Figure 6.8 corresponds to the times we manually crash the primary replicas. Except those failure times, the blocking time is considerably stable around 220ms and even after the failure times, the system goes back to its stable state immediately. This is because, the coordinator service selects a new primary replica immediately and the newly selected primary replica which was actually a backup replica has always been ready to switch to primary replica mode with updating its current state to the most recent one as given in Algorithm 5.

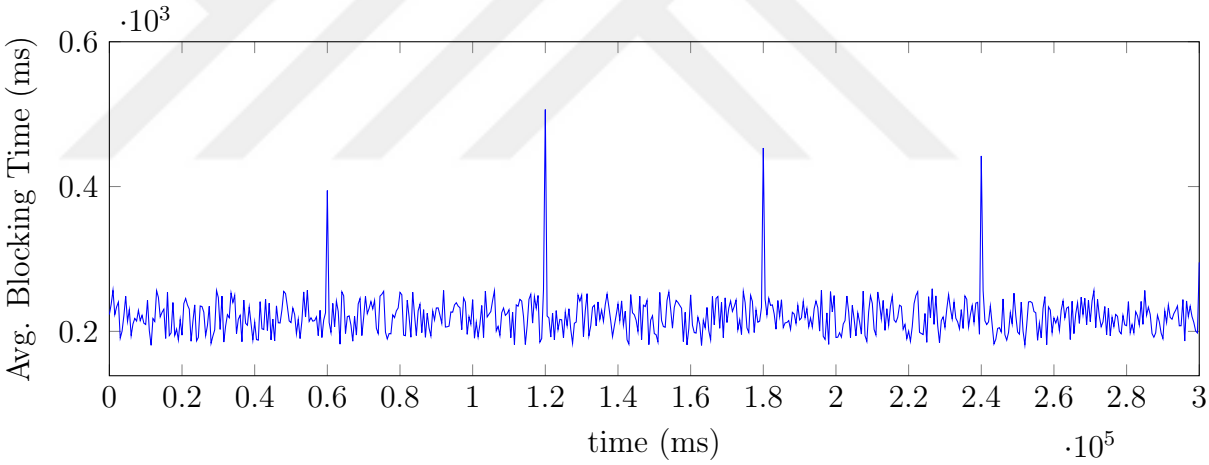

Figure 6.8: Avgerage blocking time during manual crash test of primary replicas every 60 seconds

Chapter 7

CONCLUSIONS AND FUTURE WORK

In this thesis, we defined and analyzed various checkpointing algorithms that can be integrated into the primary-backup replication protocol to increase its efficiency specifically in terms of lower blocking time and higher throughput considering a several key-value stores use a similar replication protocol to the primary-backup replication. We implemented our proposed checkpointing techniques by extending the open-source RocksDB key-value store which is being developed by the Facebook and configured our own geographically replicated key-value store on the PlanetLab testbed using geographically distributed nodes. We conducted extensive benchmarking tests using the YCSB tool taking realistic workload scenarios into account and presented results for numerous metrics including blocking time, checkpointing time, checkpoint size, failover time and system throughput. Furthermore, we have discussed further improvements on the checkpointing algorithms using compression and demonstrated the increased efficiency along with additional metrics like compression ratio, compression time and decompression time.

In terms of non-periodic checkpointing algorithms, the incremental checkpointing performs almost identical to the traditional primary-backup replication protocol. However, we show that by introducing periodic checkpointing techniques, it is possible to reach 5 times lower blocking time and 70 times higher system throughput using the periodic-incremental checkpointing compared to the traditional primary-backup replication. Enabling the compression on that technique also reduces blocking time and improves system throughput further.

However, the periodic checkpointing approaches violate the strong consistency property, nevertheless by employing a dynamic period, we aim at higher checkpoint-

ing rate without decreasing the system performance yet still partially violating the strong consistency at some point. We proposed load-aware compressed incremental checkpointing method (LACPB) for large scale primary-backup replication. Large-scale and comparative experimental results indicate that LACPB maintains drastically higher system throughput as well as reduced and stable client blocking times even in the dynamic workload scenarios.

As future works, we have designated two different viable studies. One is to further analyze and evaluate the new consistency level of the LACPB which should be lower than the strong consistency level. Other future work is to propose an intelligent system for setting the checkpointing period $\mathcal{C}_p$ such as using a machine learning approach which could use more sophisticated and robust ways to calculate the period rather than a simple linear equation.

BIBLIOGRAPHY

[1] A. Singla, B. Chandrasekaran, P. Godfrey, and B. Maggs, "The internet at the speed of light," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 1.

[2] G. M. D. Vieira and L. E. Buzato, "Distributed checkpointing: Analysis and benchmarks," *SBRC '06*, May 2006.

[3] A. V. Vathsala and H. Mohanty, "A survey on checkpointing web services," in *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, ser. PESOS. New York, NY, USA: ACM, 2014, pp. 11–17.

[4] B. Guler and O. Ozkasap, "Analysis of checkpointing algorithms for Primary-Backup replication," in *PEDISWESA Workshop, 2017 IEEE Symposium on Computers and Communication (ISCC)*, Jul 2017.

[5] B. Guler and O. Ozkasap, "Compressed incremental checkpointing for efficient replicated key-value stores," in *PEDISWESA Workshop, 2017 IEEE Symposium on Computers and Communication (ISCC)*, Jul 2017.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[7] J. Han, E. Haihong, G. Le, and J. Du, "Survey on nosql database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE, 2011, pp. 363–366.

[8] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 729–730.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[10] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[11] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[12] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc.", 2013.

[13] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[14] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[15] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *OSDI*, vol. 4, 2004, pp. 91–104.

[16] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 363–376.

[17] D. Didona, K. Spirovska, and W. Zwaenepoel, "Okapi: Causally consistent geo-replication made faster, cheaper and more available," *arXiv preprint arXiv:1702.04263*, 2017.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 401–416.

[19] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[20] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 989–1000.

[21] I. C. Garcia, G. Vieira, and L. E. Buzato, "A rollback in the history of communication-induced checkpointing," *arXiv preprint arXiv:1702.06167*, 2017.

[22] A. Moshovos and A. Kostopoulos, "Cost-effective, highperformance giga-scale checkpoint/restore," *University of Toronto, Tech. Rep*, 2004.

[23] A.-P. Liedes and A. Wolski, "Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases," in *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 99–99.

[24] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE, 2012, pp. 148–157.

[25] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. De Supinski, and R. Eigenmann, "Mcrengine: A scalable checkpointing system using data-aware aggrega-

tion and compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 17.

[26] I. Jangjaimon and N.-F. Tzeng, "Adaptive incremental checkpointing via delta compression for networked multicore systems," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 7–18.

[27] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[28] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 874–879, 1994.

[29] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*. IEEE, 1992, pp. 39–47.

[30] C.-C. Li and W. K. Fuchs, "Catch-compiler-assisted techniques for checkpointing," in *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*. IEEE, 1990, pp. 74–81.

[31] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[32] K. Perumalla, *Introduction to Reversible Computing*, ser. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2013.

[33] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[34] Facebook, "Rocksdb: A persistent key-value store for flash and ram storage," *URL: https://github.com/facebook/rocksdb/*, 2016.

[35] S. Ghemawat and J. Dean, "Leveldb," *URL: https://github.com/google/leveldb*, 2011.

[36] V. Kaushal and A. Bala, "Autonomic fault tolerance using haproxy in cloud environment," *Int. J. of Advanced Engineering Sciences and Technologies*, vol. 7, no. 2, pp. 54–59, 2011.

[37] L. P. Deutsch, "Deflate compressed data format specification version 1.3," *URL: https://tools.ietf.org/html/rfc1951*, 1996.

[38] I. Pavlov, "Lzma sdk (software development kit)," *URL: http://www.7-zip.org/sdk.html*, 2017.

[39] Facebook, "Zstandart - real-time data compression algorithm," *URL: http://facebook.github.io/zstd/*, 2016.

[40] Google, "Snappy," *URL: http://google.github.io/snappy/*, 2011.

[41] L. P. Deutsch, "Gzip file format specification version 4.3," *URL: http://www.zlib.org/rfc-gzip.html*, 1996.

[42] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

VITA

BERKİN GÜLER was born in Bursa, Turkey on 26th August. He completed his high school education in Şükrü Şankaya Anatolian High School, Bursa and then received his BSc degree in Computer Engineering from Izmir Institute of Technology, Izmir in 2015. He joined the MSc program in Computer Science and Engineering at Koç University in the same year later, 2015 and has been worked as a research and teaching assistant. During his studies, he focused on distributed systems, fault-tolerant systems, cloud networks and key-value databases. He has co-authored 3 IEEE conference papers, 1 IFIP poster paper, 1 ACM poster paper and 1 TÜBİTAK conference paper. He also has a journal paper under submission.