DEGREE PROJECT IN COMMUNICATION SYSTEMS, SECOND LEVEL
STOCKHOLM, SWEDEN 2015

A Multi-leader Approach to Byzantine Fault Tolerance

Achieving Higher Throughput Using Concurrent Consensus

MUHAMMAD ZEESHAN ABID

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

A Multi-leader Approach to Byzantine Fault Tolerance

Achieving Higher Throughput Using Concurrent Consensus

Muhammad Zeeshan Abid

2015-07-01

Master's Thesis

Examiner and Academic adviser
Prof. Dejan Kostić

Advisers at Technische Universität Braunschweig
Prof. Rüdiger Kapitza (Supervisor) and Bijun Li

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

A Multi-leader Approach to Byzantine Fault Tolerance

Achieving Higher Throughput Using Concurrent Consensus

Muhammad Zeeshan Abid

Master of Science Thesis

Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

July 01, 2015

Advisers at TU Braunschweig: Prof. Rüdiger Kapitza, & Bijun Li
Examiner and Academic adviser: Prof. Dejan Kostić

Abstract

Byzantine Fault Tolerant protocols are complicated and hard to implement. Today's software industry is reluctant to adopt these protocols because of the high overhead of message exchange in the *agreement* phase and the high resource consumption necessary to tolerate faults (as $3f + 1$ replicas are required to tolerate f faults). Moreover, total ordering of messages is needed by most classical protocols to provide *strong consistency* in both *agreement* and *execution* phases. Research has improved *throughput* of the execution phase by introducing concurrency using modern multicore infrastructures in recent years. However, improvements to the *agreement* phase remains an open area.

Byzantine Fault Tolerant systems use State Machine Replication to tolerate a wide range of faults. The approach uses leader based consensus algorithms for the deterministic execution of service on all replicas to make sure all correct replicas reach same state. For this purpose, several algorithms have been proposed to provide total ordering of messages through an elected leader. Usually, a single leader is considered to be a bottleneck as it cannot provide the desired throughput for real-time software services. In order to achieve a higher throughput there is a need for a solution which can execute multiple consensus rounds concurrently.

We present a solution that enables multiple consensus rounds in parallel by choosing multiple leaders. By enabling concurrent consensus, our approach can execute several requests in parallel. In our approach we incorporate application specific knowledge to split the total order of events into multiple partial orders which are causally consistent in order to ensure safety. Furthermore, a dependency check is required for every client request before it is assigned to a particular leader for agreement. This methodology relies on *optimistic* prediction of dependencies to provide higher throughput. We also propose a solution to correct the course of execution without rollbacking if dependencies were wrongly predicted .

Our evaluation shows that in normal cases this approach can achieve up to 100% higher throughput than conventional approaches for large numbers of clients. We also show that this approach has the potential to perform better in complex scenarios.

Keywords: Byzantine Failures, Fault Tolerance, Performance, Reliability

Sammanfattning

Byzantine Fault Tolerant protokoll är komplicerade och samtidigt svåra att implementera. Dagens mjukvaruindustri är motvillig till att anta dessa protokoll pågrund av den höga resursanvändningen vid meddelandeutbyte i avtalsfasen samt den höga resurskonsumtionen som behövs för att tolerera okontrollerbara fel (fault) (eftersom $3f$ +1 replikas måste tolerera f antal fel (faults)). En enhetlig meddelandeordning behövs av de flesta klassiska protokoll för att tillhandahålla en stark enhetlighet både när det gäller avtalfasen och genomförandefasen. Forskning har förbättrat throughput i genomförandefasen genom att införa parallellism med hjälp av dagens moderna flerkärniga (multicore) infrastruktur. Trots detta finns det en hel del att göra när det gäller förbättringar i avtalet.

Byzantine Fault Tolerant systemet använder tillståndsmaskinsreplikering (State Machine Replication) för att kunna tolerera en lång rad av fel (faults). Ledningsbaserade konsensusalgoritmer är nödvändiga för det deterministiska tjänstegenomförandet påalla replikas för att säkerställa att alla fungerande replikas når samma skick. Flera algoritmer har föreslagits för att tillhandahålla en enhetlig meddelandeordning med hjälp av en vald ledare. Vanligtvis anses en enda ledare vara en flaskhals eftersom den inte kan tillhandahålla den önskade throughputen för en mjukvarutjänst med realtidskrav. För att uppnåen högre throughput såbehövs en lösning som kan utföra flera konsensusomgångar samtidigt.

Vi presenterar en lösning som möjliggör flera konsensusomgångar som utförs parallellt genom att designera flera ledare bland alla replikas. I vårt tillvägagångssätt inkorporerar vi applikationsspecifik kunskap och delar upp hela ordningen av händelser i flera delordningar som är kausalt konsistenta för att säkerställa safety. En beroendekontroll är nödvändig för varje klientförfrågan innan den utlämnas till en specific ledare för avtal. Samtidig konsensus leder till parallell genomföring av förfrågningar, vilket förbättrar throughput. Denna metodologi förlitar sig påoptimistiska förutsägelser av beroenden för att tillhandahålla en högre throughput. Vi föreslår ocksåen lösning för att rätta till sig själv om den var felaktigt förutspådd, utan att börja om.

Vår utvärdering visar att i normala fall såkan detta tillvägagångssätt uppnåupp till 100% högre throughput än konventionella tillvääagagångssätt och har en

potential att prestera ännu bättre i komplexa scenarion.

Acknowledgements

First of all, thanks to ALLAH, the most merciful and the most beneficent for giving me strength, courage and wisdom that allowed me to complete this thesis. Without His blessings it would not be possible to finish this thesis.

I wish to express my sincere gratitude to **Prof. Dejan Kostic** and **Prof. Gerald Q. Maguire Jr.** for their assistance & supervision. Their invaluable guidance, feedback and suggestions helped me in all the problems I faced during the progress of this thesis project. I would like to give a special thanks to my supervisors **Prof. Rüdiger Kapitza** & **Bijun Li** (at Technische Universität Braunschweig) for their continuous support and helping me to develop technical skills for this thesis.

Last but not least, I would like to acknowledge and thank my family, especially my parents and my wife, for believing in me. I am grateful for their never ending support throughout this thesis project.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms and Abbreviations

This document requires readers to be familiar with the following terms and concepts. For clarity we provide short description of these terms when first using them in the thesis.

BFT	Byzantine Fault Tolerant
CAP	Consistency, Availability and Partition-tolerance
C-Dep	Command Dependencies
CFT	Crash Fault Tolerant
C-G	Command-to-Group
CPU	Central Processing Unit
CRC	Cyclic Redundancy Ceck
FIFO	First-in-First-out
GB	Gigabytes
GHz	Gigahertz
IP	Internet Protocol
KB	Kilobytes
LAN	Local Area Network
MB	Megabytes
Mbps	Megabits per second
MLBFT	Multi Leader Byzantine Fault Tolerance
MTU	Maximum Transmission Unit

ODRC	On-Demand Replica Consistency
PBFT	Practical Byzantine Fault Tolerance
RAF	Request Analysis Function
R_c^{rn}	Request by client c with request-number rn
REFIT	Resource-efficient Fault and Intrusion Tolerance
rn	Request number
SMR	State Machine Replication
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network

Chapter 1

Introduction

The *client-server* architecture has been one of the most important technologies developed to deliver software services. It has served well for a long time, but it cannot meet all the needs of today's software industry. With increasingly large numbers of customers using online software services (e.g., e-commerce, banking, and social networks), the client-sever architecture is unable to handle large numbers of requests from clients located all over the world. Today, clients experience problems such as slow responses and service failures. As a result there was a need for a paradigm which better handles common faults, while enabling the service to be fast and reliable for all the clients, even when geographically distributed.

1.1 Replication

Replication is widely used to improve the availability and reliability of services by mirroring the data from a server on multiple machines. If the primary server goes down the data is not lost because it is available on one of the other machines. A question arises as to how many mirror copies should be made as backups? The answer depends upon the nature of the service and how many faults the service is willing to tolerate. Each extra machine requires resources (e.g., money and bandwidth) and adding these additional resources is not always feasible. Furthermore, a faulty primary can either be fixed or can be replaced with backup server by replication system. However, whether a replication system must decide to act upon a failure by automatic or manual measures is outside the focus of this thesis and will not be discussed further. Figure 1.1 shows a high-level architecture of a service in which replication is transparent to clients. The vertical line in the figure represents a proxy which maps requests from clients to one of the replicas.

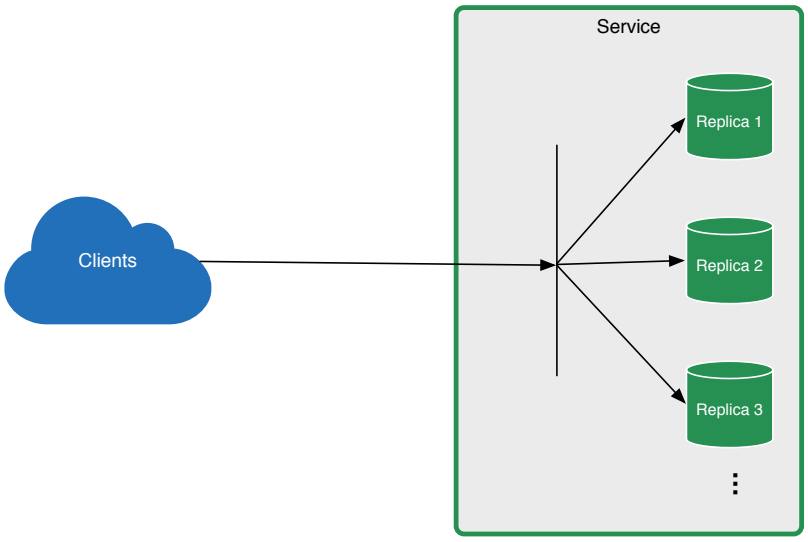

Figure 1.1: A high-level service replication architecture

1.2 Faults

In a replicated service a "*fault*" can be anything which might lead to a system failure. Faults are bound to happen naturally and cannot be avoided. A replicated system can be designed in a way that even if certain types of fault occurs, these faults will not affect the ability of the system to function correctly. However, in such a system these faults must first be detected and then measures taken to correct them. Generally, two types of faults are addressed in a replicated service. These faults are:

Omission faults
: Omission faults occur when a node does not send a message which would have been sent by a correctly operating node. Omission faults are common and most systems today address these kind of faults.

Commission faults
: Commission faults occur when a node sends a message which would not have been sent by a correctly operating node. Commission faults are generally not addressed by systems today because they are less common and quite difficult to resolve.

1.3 Byzantine faults

Byzantine faults are arbitrary faults that can occur in a system and make the system either un-reliable or un-responsive to any client requests. As a result the service can behave arbitrarily, i.e., it can lie, delay messages, send erroneous messages, or not respond at all. In general, Byzantine faults can happen anywhere and anytime in a distributed service, thus making them difficult to resolve. These faults were first addressed by *Lamport* in his paper *"The Byzantine Generals Problem"* [1] hence the name *Byzantine faults*.

Over the last decade Byzantine faults have become the focus of researchers as more and more services have moved to web platforms. New protocols have been developed over the last three decades to enable services to remain up and running *despite* the presence of arbitrary faults in the system, while still providing good performance. Unfortunately, these algorithms are still not being used in practice in today's distributed systems [2, 3, 4].

1.4 Purpose of this Thesis

One of the reasons Byzantine Fault Tolerant protocols have not yet been adopted by industry is that they lack the performance of Crash Fault Tolerant (CFT) systems [3, 4]. This is mainly because of the high overhead of the *agreement* phase to order the requests in Byzantine Fault Tolerant protocols. In Byzantine Fault Tolerant protocols all replicas must agree on a total ordering of the incoming requests in order to provide *consistency*. A lot of work has been done to improve the performance of the *execution* phase [5, 6, 7, 8]. However, little work has been done to improve the *agreement* phase.

This thesis will introduce concurrency in both the agreement and execution phase to improve the throughput of a protocol, such that it is comparable to the throughput of CFT, while still tolerating arbitrary faults occurring within the system. The hope is that Byzantine Fault Tolerant systems can be used in services which cannot tolerate even a single fault without compromising the throughput.

Our approach uses application specific knowledge to enable the total order of the messages to be achieved by dividing the messages into multiple partially ordered set of messages. We maintain the casualty of these partial orders so that two or more events which are casually related will always be executed in order. The application specific knowledge is used to predict dependencies between the messages. These multiple partially ordered set of messages are executed in parallel boosting performance by utilizing the full power of modern multicore servers. Figure 1.2 shows how a replica divides requests into multiple partially ordered queues and then executes these requests in parallel. However,

there is a possibility that the prediction is wrong and we find out there was a dependency missing in the prediction; hence when this occurs we stop executing the request and *re-predict* the dependencies given the new information. Now that this dependency is included in the prediction we resume execution. This cycle is repeated until the original request is completed.

Our hypothesis is that this approach can enable a Byzantine Fault Tolerant system to deliver **better throughput**. We also hypothesize that our approach can handle **simple** as well as **cross-border** (see Section 3.1.1.2) requests in an effective manner.

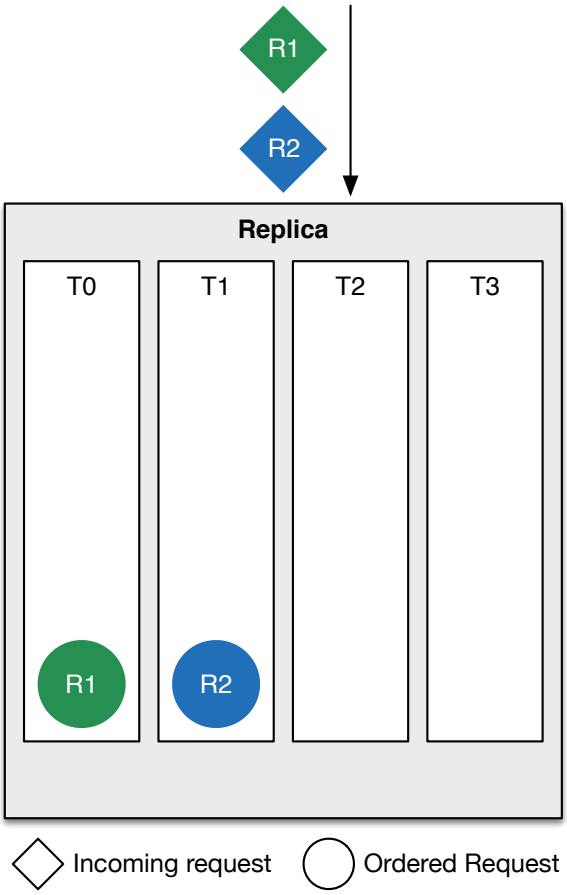

Figure 1.2: A Byzantine Fault Tolerant replica ordering requests in parallel

1.5 Suggested Approach

In Section 2.2 we explain the classical single-leader Practical Byzantine Fault Tolerance (PBFT) protocol [9]. In this protocol requests (by clients) are totally ordered by one leader and then they are passed to the execution stage. If a large number of requests are received, then they must wait for their turn to be ordered by the leader -making it a point of congestion. In practice, not all the requests are dependent upon each other and only dependent requests should be ordered. By exploiting this idea, we propose to address this problem through splitting total-order into multiple partial-orders using multiple leaders in a Byzantine Fault Tolerant service. The idea is to split the *independent* requests in multiple agreement rounds. If the requests are dependent on each other then they will be ordered by same leader to maintain the consistency of the service. Ideally every replica should have two identities: leader of a particular ordering stage and follower of the other ordering stages.

Introducing multiple Byzantine Fault Tolerant ordering instances brings up the challenge of separating *dependent* requests from *independent* requests. To address this issue we need to understand the application service in detail. As a solution a prediction stage is added to the replica before forwarding the request to the ordering stage. The prediction stage utilizes application knowledge and forecasts the set of objects relevant for each request. These objects are divided into partitions. We rely on prediction knowledge to pick the relevant partitions for a given request. Furthermore, each partition operate its own ordering instance with a dedicated queue for each instance. These predicted partitions are then translated to their respective ordering instances. In case of multiple partitions, all of the ordering instances must order this request to ensure partial-order. This ordered request is then added to the tail of respective queue in the ordering instance. In the execution stage each queue is operated by a dedicated worker thread. This enables multiple requests to be executed at the same time, hence improving throughput. Moreover, all the queues are shared between worker threads and care must be taken while modifying these queues. If a request has been ordered in more than one queue, then it must be synchronized with all the relevant worker threads operating on those queues. In this case a worker thread is deterministically picked to execute the request and other relevant threads will wait until the request has finished its execution. We discuss this approach in detail in Chapter 3.

Deploying multiple leaders will utilize more computing power of each machine, as the replica has more than one identity now. Each identity will be represented by at least one thread on a physical machine. The overhead of thread scheduling depends on the specifications of the machine and the software (operating system and the application service). Multiple threads perform better on a multi-core machine due to the decreased overhead of scheduling. On the

other hand, if the machine is not multi-core (or the cores are less than replica identities) then the overhead might increase leading to lower throughput. Given this reasoning, this approach is ideal for applications with high workload and multi-core replicas.

In this approach it is very difficult for a client to know where to send its request, as any replica could be the leader for this particular request. There are two solutions to resolve this issue. Either client sends its request to all the replicas (thus increasing the number of messages) or a proxy on the client sends the request to only the relevant replicas. The latter approach is an optimization that will reduce the number of messages sent between clients and replicas.

1.6 Structure of this Thesis

Chapter 2 introduces the background knowledge needed to understand this thesis. The chapter starts by explaining the building blocks of a Byzantine Fault Tolerant service. Then it explains the classical PBFT protocol. In addition, the chapter discusses related work that has been done to improve throughput.

Chapter 3 presents Multi Leader Byzantine Fault Tolerance (MLBFT), a multi-leader approach to Byzantine Fault Tolerant system. It explains the system model, proposed approach, assumptions, and implementation strategies of the protocol.

Chapter 4 provides an analysis of the proposed protocol. In addition, it presents some benchmarking results and a comparison with conventional approaches.

Chapter 5 concludes the thesis by giving some insights and challenges faced in the proposed solution. Furthermore, it suggests some future work that may be of interest in order to continue this work.

Chapter 2

Background

Some background knowledge is required to understand the research that was conducted. This chapter introduces state machine replication, then briefly explains the Practical Byzantine Fault Tolerance (PBFT) protocol. Finally, related research is discussed in Section 2.3.

2.1 State Machine Replication

State Machine Replication (SMR) [10, 11] is a widely used means to implement fault tolerant systems, because it is effective and works. Almost every modern service uses this technique to tolerate faults. Although, state machines can be *non-deterministic* (see Section 2.1.5 for non-deterministic applications of SMR), this thesis will only target *deterministic* state machines. That implies, that given any initial state and sequence of requests, all service processes will produce the same response and reach the state. All messages should be received in the same order by every state machine so that a common response can be produced. This property is crucial for replicated services and Byzantine Fault Tolerant (BFT) systems highly rely on determinism. In order to realize this, all messages are broadcasted atomically [12] to all replicas and all correctly executing nodes will deliver their messages in same order. In order to tolerate arbitrary faults, a BFT system implements replication through SMR. From now on the term BFT means Byzantine Fault Tolerant State Machine Replication protocol in this thesis, unless stated otherwise.

2.1.1 Design and Performance

The concept of SMR was proposed to improve the *availability* of the service, not its *performance*. It is possible in some cases that a single-server implementation

of a service might outperform a replicated service. This is because a replicated service must order the requests sequentially in order to provide *linearizability* while a single-server can benefit from other improvements (e.g., concurrently executing requests).

The architecture of a BFT service which uses SMR can be divided into two stages: *agreement* and *execution* [13]. Figure 2.1 shows basic architecture of replicated BFT service with separate agreement and execution stages. This separation reduces the replication cost and provides a *privacy firewall* [13] architecture to preserve confidentiality through replication. However, details of a privacy firewall architecture are outside the scope of this research and will not be discussed further in this thesis.

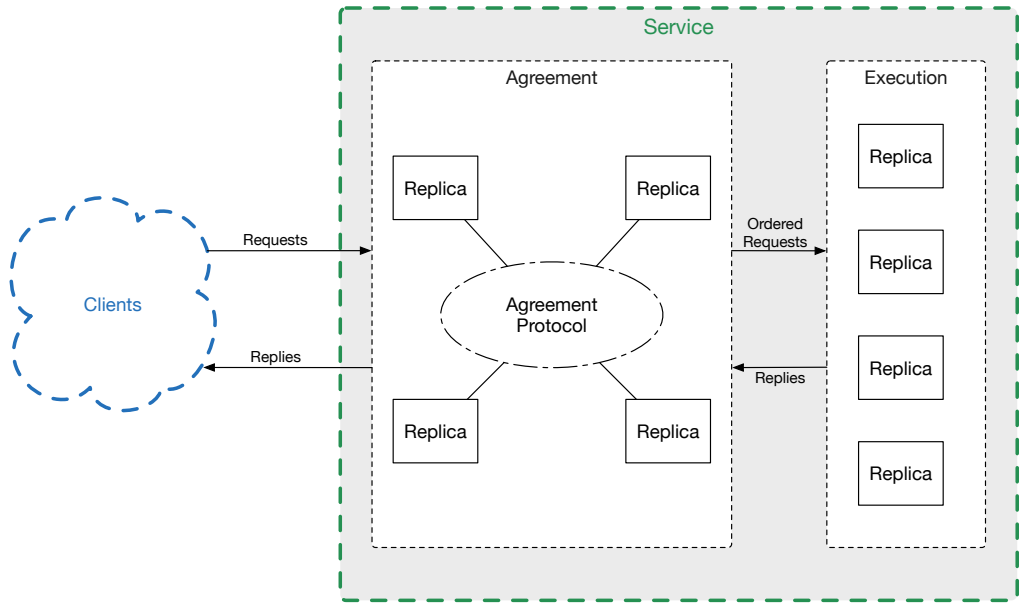

Figure 2.1: Basic architecture of SMR based BFT service

2.1.2 Replica Coordination

As mentioned in the previous section, all correctly operating replicas in SMR must execute the same sequence of requests in order to ensure determinism. Replicas coordinate with each other by sending asynchronous messages (in the case of the BFT model) to each other. This can expressed in the following two requirements of the state machine [10]:

Agreement

Agreement is sometimes also known as *consensus*. An agreement

requirement is satisfied by running the *agreement protocol* on every replica. This protocol ensures two properties:

- All non-faulty replicas agree on the same value.
- If the sender is non-faulty, then all non-faulty replicas use its proposed value as the agreed value.

Order

The order requirement satisfies the property that all non-faulty replicas process the request it receives in the same relative order. *Schneider* [10] propose "total-ordering" of requests using logical clocks [11]. Total-ordering is also known as "linearizability".

Together these two requirements will be referred as the "agreement" phase in this thesis. Figure 2.1 shows the agreement stage which runs an agreement protocol to order the requests. Although the classical BFT protocol [9] and some of its decedents require linearizability, we will relax this requirement in this thesis. Section 3.2.4 explains the agreement stage of our approach in detail and explains how total-order can be split into partial orders.

2.1.3 Execution

The request is handed over to *execution* phase after it has been ordered by agreement phase (see Figure 2.1). The execution of the client request takes place in this stage. The application state objects are accessed (*read*) or modified (*write*) and a result is returned to the client. The execution phase can utilize features of the programming model, the system model, and the application knowledge to serve the client request. In fact, some of the BFT protocols [5, 6, 7, 8] are specifically designed to improve throughput in execution phase.

2.1.4 Checkpoints

Checkpoint stores the state objects of a state machine to ensure that a replica has executed all of the requests up to a certain point in the state machine. SMR must keep a log of executed requests for future use in case of faults. If this log is left unbounded it will grow until it exhausts all available storage. Checkpoints are created as a means to limit the log entries by forgetting old requests that have contributed to the checkpoint. A checkpoint usually contains all the state objects of the replica. Furthermore, a checkpoint is also useful when a new replica wants to join the system, as this replica can ask for the latest checkpoint from an existing replica and then start to participate $after$ processing all of the messages sent that did not contribute to the checkpoint.

2.1.5 Non-determinism

If the service using SMR is non-deterministic, then all the replicas will diverge and there will be no way to find out which one is the correct replica. For example, a service might execute the $seed()$ function to initiate a generator for *(pseudo-)random* numbers. Or a service might request the $time\_of\_the\_day$ which may be different for each replica. Fortunately, there are ways to handle non-deterministic parts of a service which must produce the same outcome on all replicas. One method to do this is to have the client pass all the non-deterministic data as a part of the request. In this case all replicas receive the request and accompanying data, hence there will not be any divergence. Another method is to have all replicas agree upon same non-deterministic values. In this approach a replica ($leader$) decides upon a value and sends it to other replicas. The other replicas accept this value after voting. Given that there are these two methods to address non-determinism, non-determinism will not be addressed further in this thesis.

2.2 Practical Byzantine Fault Tolerance

Section 1.3 introduced Byzantine faults and why they are important to tolerate. This section describes the classical PBFT protocol which was first proposed in 1999 by *M. Castro and B. Liskov* [9, 14]. This protocol is considered to be a baseline for BFT protocols because many subsequent protocols are either an extension to or some how based on PBFT.

2.2.1 Motivation

As mentioned earlier a lot of research over the last few years has been done to implement and deploy services which can tolerate these faults [2]. Unfortunately, industry is still reluctant to adopt these protocols. Byzantine faults are the most general kind of problems that a service can have. This makes them very interesting and at the same time very difficult to solve. A service which implements BFT can withstand any arbitrary error, such as a small software bug (which occurs $independently$ on different replicas), physical failure of a machine, even the service being compromised by an external attacker.

A question arises as to why these protocols are not being implemented in practice if they can handle any arbitrary fault? The main reason for this is that BFT protocols are very resource hungry protocols with $3f + 1$ nodes required to tolerate f faults [15]. This scaling seems to be impractical when f is large. Additionally, throughput of BFT protocols are not comparable to their CFT counterpart as BFT

protocols require a lot of messages to communicate internally (three rounds of message exchange) each of which encounters network delays, hence slowing the system's responses to the clients. A lot of work [5, 6, 7, 8, 16, 17, 18, 19, 20, 21] has been done to highlight different aspects of BFT protocols for practical deployment.

It is observed that the agreement phase is a bottleneck, as it requires all of the requests to be totally ordered by a single leader. Additionally, PBFT provides *strong consistency* by using $linearizability$ through the agreement phase. However, little has been done in this area to improve the agreement phase in BFT. This thesis project attempts to improve the throughput of BFT by parallelising the agreement phase.

2.2.2 Protocol

This section briefly describes the PBFT protocol. As stated earlier $3f + 1$ replicas are required to tolerate f faults in a BFT system [15]. Therefore, a minimum of four replicas are required to tolerate a single fault. Moreover, three additional replicas are required to increase the system's fault-tolerance by one. Furthermore, increasing the number of replicas also increases the number of messages, hence slowing down the system.

In order to tolerate byzantine faults, PBFT protocol implements replication through SMR. A replica (called a leader) with some special responsibilities is selected among the replicas. The leaders responsibilities include ordering the client requests and sending these requests to all of other replicas. The remaindered of the replicas act as followers and execute the request in same the order as determined by the leader. In the normal case a client sends the request to a leader, then followers execute this request and send their response directly to the client. Figure 2.2 shows the message sequence in the normal case. The leader (replica 0) receives a request from client **C** and starts the ordering phase. The remaining replicas are followers, hence they receive requests from the leader and execute them. Messages shown in red will not affect the result. A *view-change* protocol is initiated and a new leader is chosen if the current leader is detected to be faulty.

The *agreement* phase starts when a request is received by the leader. There can be multiple requests in the consensus, but we assume that there is only one request per consensus. The leader starts the process to reach an agreement by sending a *pre-prepare* message to all replicas. Upon receiving a *pre-prepare* message, each replica sends a *prepare* message to all other replicas. The predicate $prepared()$ is true if and only if the replica has inserted the *request*, the matching *pre-prepare* message, and $2f$ $prepare$ messages from different replicas into its log. The leader does not have to send a *pre-prepare* message to itself and advances itself to the $prepare$ stage directly.

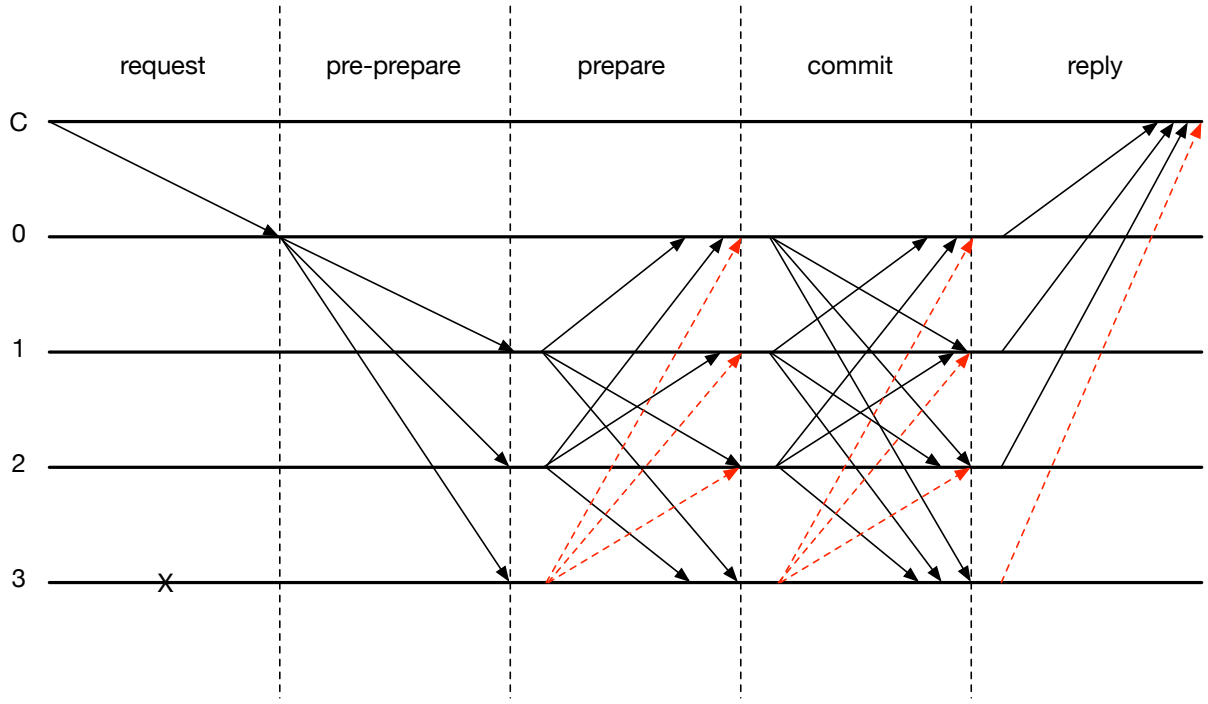

Figure 2.2: A normal case PBFT operation

When *prepared*() is true, each replica sends a *commit* message to all other replicas. Upon receiving $2f + 1$ *commit* messages (possibly including its own) the actual execution of the requested message is started. After successful execution of the requested operation the replica sends its response directly to the client. The client waits for $f + 1$ similar responses from different replicas. It is proven [14] that at least $f + 1$ non-fault replicas will send their response to the client. If the client does not receive $f + 1$ similar responses from different replicas within a certain amount of time, it is assumed that *leader* is faulty and the client re-transmits the request to all replicas. A *view-change* protocol is triggered when replicas receive a request from a client which they have not seen previously suspecting that the leader is faulty.

PBFT proposes three optimizations to reduce the communication between replicas and clients. The first optimization uses single a replica to send the actual result, other replicas send the digest of the result to be verified by client. The second optimization reduces the number of messages by executing the request as soon as *prepared*() becomes true for the request. The third optimization multicasts the read-only request to all replicas and the request is executed immediately.

Replicas and clients communicate with each other through message passing.

The algorithm uses User Datagram Protocol (UDP) [22] for point-to-point communication between replicas. That implies message communication is unreliable, thus messages may be duplicated, lost, or delivered out of order.

The PBFT algorithm has been proven by Castro and Liskov [14] to provide safety and liveness when no more than one third of the replicas are faulty. The algorithm does not depend on *synchrony* (i.e., that an upper bound on message delivery time is known) to ensure safety, but must depend on *synchrony* to ensure liveness [23].

2.3 Related work

As mentioned earlier BFT systems are expensive to deploy in real applications. The classical PBFT protocol requires a single *leader* and *all-to-all* message exchanges for total ordering in both the *agreement* and *execution* stages. This introduces a high latency in the response time for any particular request by client. Moreover, the requirement for total ordering of messages in both phases makes the leader a bottleneck in the protocol. A lot of work has targeted the *execution* stage and introduced concurrency in order to improve the throughput of the protocol. This section categorises related studies into two groups. The first group of related work concerns the agreement phase and the second group of related work concerns the execution phase.

2.3.1 Agreement

Today agreement in BFT systems remains a bottleneck. As of this time, no version of the protocol has completely eliminated the single leader problem in the protocol. However, we discuss some attempts to improve the performance of the agreement phase in the following paragraphs.

2.3.1.1 Parallel state-machine replication

Marandi, Bezerra, and Pedone [24, 25] have contributed some work that is closest to eliminating the single leader. However, their work only applies to the CFT model. In this thesis project we will use some of their ideas by adapting them to BFT systems.

This protocol assumes a crash failure model and that the application state can be divided into state-objects. These state objects are divided into groups and each group is operated on by a dedicated thread. The number of groups is a system parameter and this parameter can be modified depending on the number of processing units in the server. A client must multicast its request to

all servers. The client and server uses proxies based on command dependencies (*C-Dep*), to indicate which commands depend on each other. Each command must indicate the group or multiple groups, in the case of dependent commands, that this command will affect during its execution. C-Dep is used to compute a function called *Command-to-Group (C-G)*. Each worker thread uses the C-G function to determine whether it should run in *parallel* or *synchronous* mode. If the command accesses only a single group, then the command runs in parallel mode, otherwise the command must be synchronized with other threads.

The amount of concurrency depends directly on C-Dep. A highly application aware C-Dep function can utilize its knowledge to maximize concurrency. Their system model relies on *atomic multicast* [12, 24, 26]. Threads deliver messages in deterministic order across all replicas. The ordered delivery of messages is sufficient to make it a deadlock free algorithm. Being deadlock free is very useful because the server does not have to run any deadlock detection or recovery algorithm.

2.3.1.2 Spin One's Wheels?

BFT with a spinning primary, introduced by *Veronese, et al* in "Spin One's Wheels?". [27], is another attempt to modify the agreement phase. Because the performance of BFT systems degrades in the presence of faults, a spinning primary tries to improve the performance of the agreement phase even when faults occur. The idea is that after every round of pending requests the primary automatically changes in a round robin fashion. They claim this offers two main advantages over other BFT systems. First, performance attacks by a faulty primary can be avoided by always changing the primary after each round. Secondly, the throughput is improved in the absence of faults by load balancing the requests over all of the correct replicas. View changes are expensive in the classical PBFT protocol, but this does not happen in *Spinning* primary, rather they introduce the concept of *view-merge*. Whenever the next in line view is faulty they merge the view and put the replica in a blacklist to avoid it being selected again as the primary.

It has been observed from their experimental evaluations that the throughput performance does improve, especially in the case of faulty replicas. However, they do not completely eliminate the primary replica which still does more work than the other replicas, thus it can be a bottleneck when the number of clients and requests increases.

2.3.1.3 Mencius

Mencius [28] is very similar to Spin One's Wheels [27]. Mencius is based on CFT systems and requires a perfect failure detector. Just as for Spin One's Wheels?, leaders are rotated in a round robin fashion to improve throughput. Each client is connected to a leader within a Local Area Network (LAN) to minimize latency on all sites. The main idea behind this algorithm is to minimize latency and improve throughput over multiple sites connected via a Wide Area Network (WAN).

This work is worth mentioning because the protocol is based on classical Paxos consensus [29]. Although we cannot use this protocol in a BFT system, it provides high throughput for real-time services under high client load and low latency under a low client load. As Mencius is designed for WANs, it provides low latency even when the network is changing (due to network or node failures).

2.3.1.4 Scalable BFT for Multi-Cores

Scalable BFT for Multi-Cores proposed by *Behl, et al.* [30] is a recent contribution for multi-core systems. The main idea is to introduce parallelism for complete instances of BFT protocols instead of only for certain tasks. They propose the concept by binding messages and tasks within a specific agreement round to a particular processor core, hence lowering the data dependencies between consensus rounds. Agreement rounds are still initiated by single leader, but now leader can initiate multiple rounds binding each one to its particular processor core. Leaders for agreement rounds are chosen in a round robin fashion. Early evaluation shows over 200% increased throughput than PBFT on a twelve cores machine which is a desirable improvement.

2.3.2 Execution

Modern (multicore) servers can execute multiple requests in parallel. The resources of these servers are under-utilized if we run a traditional SMR services on them. This suggests that we should parallelize the execution phase. Both BFT and CFT systems have been exploring this area for a long time in order to introduce concurrency in SMR. There are a lot of options and ideas to explore with regard to parallel execution. Most of the time the amount of parallelization depends on application specific knowledge and the programming model used by the application. In this section we will only discuss parallel execution in the case of BFT systems.

2.3.2.1 All about Eve

All about Eve [8] is a very non-traditional protocol when compared with other SMR systems. Instead of a traditional *agree-execute* architecture All about Eve uses an *execute-verify* model. All about Eve uses application knowledge to partition requests in to different groups, each of these groups can be executed concurrently. All replicas *execute* the groups and then they $verify$ whether they can reach an agreement. Morover, divergence is minimized by using application knowledge. In case of divergence, if there is a stage where no agreement can be achieved, then the execution is rollbacked and the request is executed sequentially. This means the programming model used for the applications must either support transactional memory or it must provide a rollback function for every request. This requirement limits the set of applications and programming models which can implement this protocol.

2.3.2.2 On-Demand Replica Consistency

On-Demand Replica Consistency (ODRC) [7] improves performance by executing a request on only a subset of replicas. It uses application knowledge to split the state objects between the replicas. These objects are called *maintained objects*. A Request Analysis Function (RAF) analyzes incoming requests and outputs a set of objects to every replica. If any of these objects is maintained by some replica, then the request will be executed on that replica. Otherwise the request will not be executed and will simply be stored for later use. This causes the replicas to diverge, hence they will have an inconsistent set of objects. If a request is received that requires a consistent value of an object which is not maintained on some replica, then this replica needs to bring the relevant state objects up to date. This is done by updating the objects from the latest stable checkpoint and then selecting those requests from stored requests that affects the objects accessed by new request. Other objects remain unmaintained and inconsistent until a request is received that requires access to those objects. In a normal case scenario only $f + 1$ replicas execute the request, rather than all $3f + 1$ replicas. In the case of faults additional replicas are required to execute the request until $f + 1$ replies are received by the client.

2.3.2.3 Storyboard

Deterministic execution is crucial for SMR, but today's concurrent execution of programs can produce non-deterministic results. Storyboard [5] is an attempt to deterministically execute multi-threaded programs. Storyboard uses application knowledge to forecast which execution path is most likely to be followed. An ordered sequence of locks is generated and all threads are monitored to ensure

that they follow the generated sequence. If the forecast is correct, then threads can execute in parallel without any problems. Otherwise, the execution path is corrected when the forecast is detected to be incorrect.

Storyboard is applicable to both CFT and BFT models. In BFT the forecast is detected to be incorrect when at least $f + 1$ replicas indicate the forecast was incorrect. As the protocol is very expensive, due to a large number of message exchanges, each forecast should be as precise as possible. We use some of the ideas from Storyboard to predict likely execution path of requests in our own work.

Chapter 3

MLBFT: A Multi-leader Approach

BFT services usually have a performance penalty. Therefore, a service which can tolerate arbitrary faults performs worse than its un-replicated non-fault tolerant counterpart [9]. This performance degradation occurs because a BFT service must order the requests before execution to achieve determinism (see Section 2.1) and consistency. Both determinism and consistency are essential requirements to provide a reliable BFT service. Increasing the performance of BFT systems has been widely researched in recent years. Traditionally a leader is chosen among all replicas whose responsibility is to produce a total-order of the incoming requests. Usually, the leader has no knowledge about the requests and it considers all the requests dependent of each other. In practice, not all of the requests are dependent on each other. Furthermore, BFT services are not equipped to scale on modern multi-core systems. In some scenarios BFT systems can utilize concurrency only in the execution stage (see Section 2.3.2).

To address this problem we propose a new scheme to deploy multiple leaders in the system. This gives two benefits. First, multiple consensus can be started concurrently, hence eliminating the single leader as a bottleneck. Second, service can utilize the power of multi-core systems by executing multiple requests in parallel. Preliminary evaluation shows (see Chapter 4) that multi-leader approach can achieve higher throughput than classical PBFT algorithm by more than a factor of two for large number of clients.

Our approach uses application knowledge to distinguish between dependent and independent requests. Furthermore, independent requests are ordered concurrently by different leaders and executed in parallel by worker threads on each replica. This approach provides concurrency in both agreement and execution phases, hence considerably improving the throughput. Additionally, it should be noticed that this approach does not decrease the total number of protocol messages. Figure 3.1 shows the high-level architecture of the proposed multi-leader approach.

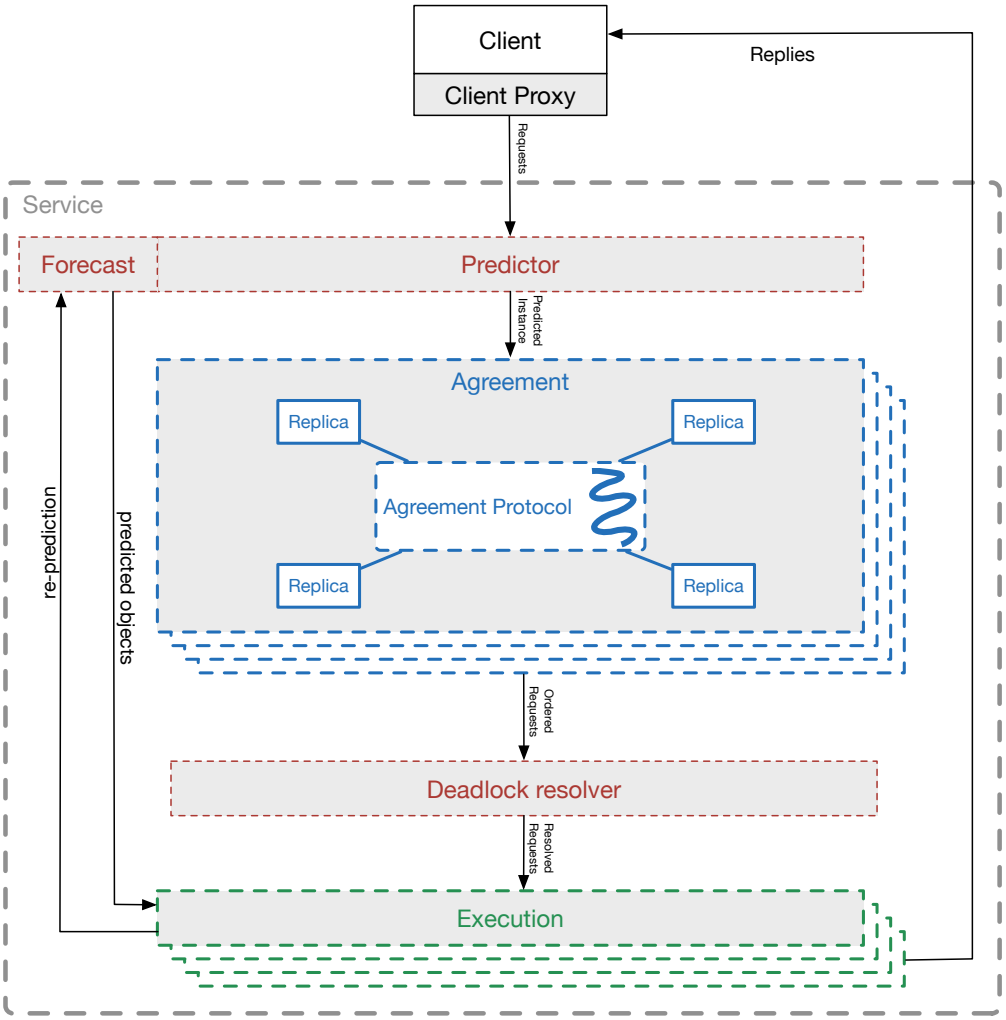

Figure 3.1: High-level architecture of Multi-leader BFT service replica

3.1 System Model

This section explains the basic architecture in which the proposed multi-leader approach can be applied. We assume this architecture in the context of this thesis.

Our system model comprises a distributed *client-server* architecture. We assume the standard system model used for BFT SMR [6, 9, 13, 31]. There may be an arbitrary number of clients and a fixed number of servers. The server side involves a group of replicas which are also referred to as *nodes*. Replicas maybe located geographically distant, may operate at different speeds, and usually run on different physical machines. Clients and replicas interact with each other

through an overlay network. They communicate solely by message passing and reliable transmission of the messages is not guaranteed. Therefore, messages can be corrupted, delayed, delivered out of order, or may not be delivered at all.

3.1.1 Client

The client side component which directly interacts with server side is referred to as a "*client*". This client can be implemented as a library which interacts with the server side on behalf of a user. We assume that the client is aware of the replication and can send messages to any replica. Implementing the client as a library is usually a good design decision because the application does not have to be replication-aware. On the other hand, this approach increases the size of the client library and creates coupling with the replicas. Each client is identified by a system-wide unique id. In addition to this client-id, each client maintains local counter called *request number* (*rn*) that is incremented each time a request is issued. The client-id and request-number, together, are called a *request-id* and must be included in every issued request. The request-id can be used to uniquely identify a request by system components and to ensure that the request is processed only once (in the case of faults). The client also maintains information about the leaders of all BFT instances and updates the state information whenever a new leader is elected [9, 14].

Furthermore, we assume that client is synchronous. Therefore, a client can only send one request at a time and it must wait for the reply before sending another request. Applications that need to send asynchronous requests (more than one request at a time) can use multiple instances of the client.

Any number of clients can behave arbitrarily and may fail without informing the server. PBFT [14] handles faulty clients by adding them to a blacklist in order to prevent them from further using the service. Furthermore, Aardvark [21] proposes some important ideas to handle malicious clients. However, we do not address the problem of faulty clients in this thesis.

3.1.1.1 Client Proxy

The client proxy is a small component in a client library that intercepts all requests issued by a client and predicts which partitions are going to be accessed. The client proxy forwards the request only to the BFT instances that are responsible to order and execute the request (see Section 3.2.3). The approach is an optimization to reduce the number of messages between a client and replicas, otherwise the client must send its request to all BFT instances. Furthermore, we assume that a client proxy is implemented on the client side.

3.1.1.2 Client Requests

A client c can send one request at a time with request number rn uniquely identified by *request-id* (R_c^{rn}). The client waits for a reply before sending another request. Each client request may: read state objects, modify state objects, or produce a response. The request can access state objects divided into one or more *partitions* (see Section 3.1.2.1). We classify client requests into two categories:

Simple Requests
: A request that reads or modifies one or more state objects belonging to same the *partition*.

Cross-Border Requests
: A request that reads or modifies one or more state objects belonging to more than one *partition*.

Furthermore, there can be a *dependent* or *independent* relationship between any two requests. Two requests are said to be independent if they either read/write different state-objects or only read common state objects. Conversely, two requests are dependent if they access common state objects and at least one of the requests modifies them. In Section 3.2 we explain in detail how agreement and execution stages behave differently for different types of requests.

We assume that a simple request cannot be further split into smaller parts. This means that a simple request must be ordered in a serialized manner. Furthermore, in practice, it is possible to split a request into smaller parts (e.g., cryptography and message authentication) and execute those parts concurrently. However, parallelizing smaller parts of the request is outside the scope of this thesis and will not be discussed further.

3.1.2 Replica

Replicas are server side components of the service. These replicas are sometimes referred to as *servers* or *nodes*. The number of replicas (denoted by n) must be equal to $3f + 1$, as we are implementing a BFT protocol, where no more than f byzantine faults can occur at the same time. However, a BFT protocol can tolerate arbitrary number of faults after reconfiguration as long as no more than f faults occur simultaneously. Replicas are usually deployed on different physical machines. We assume that these machines are multi-core (i.e., multiple instructions can be executed in parallel). Ideally replicas should run on machine with o number of cores as the algorithm instantiates o number of BFT ordering instances and e number of worker threads for execution stage. We assume that o and e are the same and each ordering stage has a corresponding worker thread.

As mentioned earlier replicas interact with each other solely by message passing. Transmission Control Protocol (TCP) [32] can be used to avoid temporary network failures, provide ordered delivery, and provide congestion control. UDP [22], on the other hand, is a better choice when network failures and congestion control are not problem (e.g. if all replicas are connected with a LAN).

Yin, et al. [13] proposed a solution to separate agreement and execution stages in order to reduce the replication cost. Our system model does not force any restriction on this separation, but for simplicity we assumed that all replicas participate in both agreement and execution stages. Moreover, our approach can be implemented with agreement and execution stages on different replicas. Furthermore, we introduce another stage called "*prediction*" (see Section 3.2.3) before passing the request to the agreement stage. In addition to the prediction stage, we also introduce a component "*Deadlock resolver*" as a part of execution stage. Figure 3.1 shows these stages and the flow of messages between these them.

3.1.2.1 Service State

The service is defined by a *state machine* [10, 11] and consists of state objects [31] that encode the state-machine's state. Our system model requires that all non-faulty replicas maintain the same state. State can be modified by a set of client requests (see Section 3.1.1.2). Moreover, we divide the state S to *state objects* in each replica [31]. These objects may have different sizes, but altogether they cover ($\cup O_i = S$) the whole state S. It is also assumed that these objects are disjoint ($O_i \cap O_j = \emptyset$). However, overlapping state objects may be used for this approach, but we will not discuss it further in this thesis.

We further divide these state objects into disjoint *partitions* ($P_i \cap P_j = \emptyset$ and $\cup P_i = S$). These partitions should be balanced, such that all partitions serve equal number of client requests on average, for optimal performance; but this load may vary in practice. The number of partitions may depend on the implementation, but for simplicity we assume that number of partitions $|P|$ directly corresponds to the number of ordering instances o (see Section 3.2.4). As a result, for simplicity we assume that each partition corresponds to a dedicated ordering stage *and* a dedicated execution stage (see Section 3.2.4 and Section 3.2.5).

3.1.2.2 State Partitioning

As mentioned in previous section, the state objects are divided into partitions. MLBFT utilizes application knowledge to partition these state objects. This division of the state objects into partition is important to deliver a better performance of the service. Ideally, each partition should handle equal number

of the client requests. The performance of the service will be improved if the client requests are uniformly distributed to all partitions. On the other hand the performance will be negatively affected if the client requests are skewed towards a particular partition. This is because the BFT instance responsible for that particular partition will order more requests than other BFT instances.

In Chapter 4 we consider a Key-Value store as a case study. We assume that client requests are uniformly distributed over the key space. We use a simple hash function (modulo operation) to distribute the keys over 4 partitions. It is possible that a particular object is accessed very often by the clients for an application. In this case it is better to place the object, that is accessed very often, to one partition and rest of the objects to other partitions instead of evenly distributing the objects. This allows the service to order the client requests accessing that particular object by one BFT instance and order rest of the client requests by other BFT instances. This approach will attempt to distribute the client requests uniformly to all partitions which is the desired behaviour of MLBFT. However, this approach will fail to uniformly distribute the client requests if more than 25% (as there are 4 partitions) of the client requests access those popular objects.

3.1.3 Assumptions

In addition to the properties described above, this section explains additional assumptions about our system model.

3.1.3.1 Deadlocks in Application Service

We do not force any restriction on concurrent programming, hence an application service may or may not execute client requests concurrently. However, if an application service decides to use a concurrent programming model, then it is assumed that this application is deadlock free. It is also assumed that all the rules to prevent or avoid deadlocks will be implemented by the application service. This means that the state objects will be guarded by appropriate locks and only one thread modifies state objects. (Do not confuse application threads with worker threads in execution stage. See Section 3.2.5). Concurrent execution might lead to non-determinism which violates the essential requirement for SMR (see Section 2.1), hence it is assumed that service will be deterministic and replicas will not diverge when successfully executing client requests.

3.1.3.2 Programming Model

The multi-leader approach can be implemented using any general programming model. We assume that the programming model used provides constructs to

implement concurrent execution (e.g. threads, locks, and monitors). However, it is possible to use a programming model without these constructs, but such a programming model will not be optimal because application may or may not implement the agreement and execution stages as separate processes.

The suggested approach does not require any functionality to rollback partially executed client requests; therefore, any kind of memory (transactional or non-transactional) maybe used to store state objects.

Furthermore, N-version programming can be applied to provide diversity among system components of different replicas. N-version programming can improve the fault tolerance of individual components in practice [9]. However, N-version programming is outside the scope of this thesis and will not be discussed further.

3.1.3.3 Cryptography

Clients and replicas use public-key signatures, message authentication codes, and message digests to detect spoofing and corrupted messages. Replicas and clients are able to verify messages. It is assumed that all cryptographic techniques used to sign or authenticate messages cannot be broken [9, 21]. Furthermore, collision-resistant hashing must be used to produce message digests.

3.2 Protocol design

MLBFT is designed to perform multiple consensus rounds in parallel. We propose to realize this approach by deploying multiple BFT ordering instances on replicas. This allows MLBFT to run complete consensus instances concurrently. These instances run independent of each other sharing no intermediate state, requests, or data. Multiple instances enables the protocol to execute the ordered requests concurrently, hence improving throughput.

3.2.1 Basic Principle

The client proxy intercepts all the requests sent by the client (see Figure 3.1). Each request is analyzed by this client proxy and a set of partitions is predicted. This set contains all the partitions that will be accessed by the request. After prediction the request is forwarded to those BFT ordering instances responsible for the predicted partitions (see Section 3.2.4). The request is ordered by all the responsible BFT instances and then placed in the ordered queue. Each partition has a separate ordered queue, thus requests can be executed in parallel. A worker thread takes the first available request from the ordered queue, executes it, and

returns a response to the client who issued the request. If it is a simple request, then the request is executed without any synchronization (see Section 3.1.1.2). If the request is a cross-border request then the worker thread waits until *this* request is available at the head of ordered queues for all relevant partitions. The worker thread moves on to next available request after current request has been executed.

3.2.2 Request Execution

This section explains how a request is ordered and executed in MLBFT. Figure 3.2 shows this process by generating and executing a request on BFT-1 which is responsible of partition-1. A request is generated by a client to access the state objects residing in the partition-1. The client proxy, which is implemented as an optimization (see Section 3.1.1.1), intercepts this request from client and performs the prediction of partitions. The client proxy maintains the information about all BFT instances and their leaders. The prediction function discovers that BFT-1 is responsible for partition-1 and replica-1 is the leader of this BFT instance. The client proxy sends this request to replica-1 only as it is the responsible leader of BFT-1. All the replicas participates in all BFT instances. Each replica has multiple threads and each of these threads participates in a different BFT instance. For example, thread-1 at replica-1 participates only in BFT-1. Furthermore, thread-1 at replica-1 is the leader of BFT-1 and thread-1 is follower of BFT-1 on rest of the replicas. The prediction stage at replica-1 receives the incoming request from the client and performs the prediction. The server side prediction is performed to verify that the incoming request in fact belongs to *this* replica. The request is forwarded to the relevant thread which is responsible for the partition accessed by incoming request. The request is forwarded to thread-1 in this case. Thread-1 will start the consensus by initiating a PBFT protocol (see Section 2.2) as thread-1 is the leader of the BFT instance responsible for the request. BFT-1 is represented by red color in the Figure 3.2, hence all threads communicating over BFT-1 are represented in red. Furthermore, execution stages on all the relevant threads are represented by red gradient to show that the request was ordered by BFT-1. Only the thread-1 on all replicas will participate in this consensus round. Each replica learns about the request and inserts the request into the relevant queue after an agreement has been reached. The request is forwarded to the execution stage after the agreement round. The exec-1 (represented by red gradient) is the responsible execution stage on all replicas for this request. Each replica executes the given request in a worker thread and send the replies to the client. The client waits for $f + 1$ similar replies. As soon as the client collects $f + 1$ similar replies it becomes ready to send the next request (as client is synchronous, see Section 3.1.1).

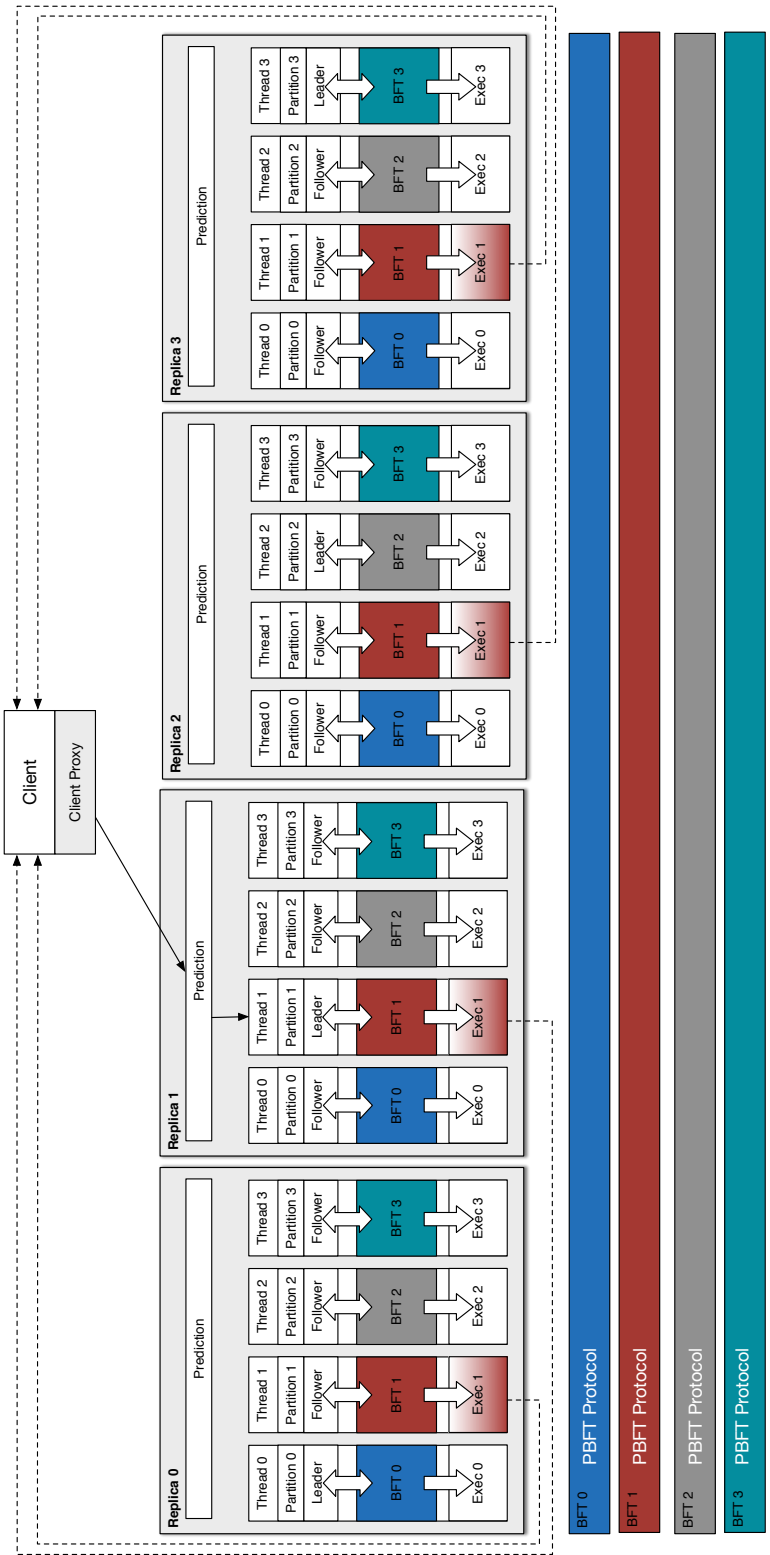

Figure 3.2: MLBFT approach ordering and executing a simple request on BFT-1

3.2.3 Prediction

All requests are processed by the prediction stage before they are forwarded to agreement stage (see Figure 3.1). The prediction stage implements a PREDICT() function to analyse the incoming request. PREDICT() takes client a request as input and returns a set of partitions:

$$Set < partitions > \text{PREDICT}(\ Request\ req\ )$$

This function uses application knowledge to inspect the client request and discovers which state objects might be accessed during the execution of the request. The PREDICT() function maps these state objects to their corresponding partitions and a set of partitions is returned. The PREDICT() function is executed on both the client (by the client proxy) and on the server side. A client proxy executes this function to forward the request to only relevant BFT ordering instances. A replica executes this function to verify that a correct BFT ordering instance has received the request. The replica starts the agreement round after successful verification of the request (see Section 3.2.4). A client is considered malicious if the verification fails (see Section 3.1.1).

The PREDICT() function is designed such that it maximizes the number of simple requests and minimizes the number of cross-border requests. A simple request does not need any synchronization, thus they can be executed in parallel. However, cross-border requests needs to wait for all the relevant partitions until the same cross-border request becomes available at the head of the queue. Our approach uses application specific knowledge to analyze the request so that a better PREDICT() function can be implemented in order to maximize the number of simple requests.

The PREDICT() function should be optimized such that it predicts the partitions perfectly and distributes the request uniformly to all partitions (see Section 3.1.2.2). Please note that the perfect prediction does not mean that the function will not predict any cross-border requests. A perfect prediction means that all of the partitions and state objects that are going to be accessed by the request will be returned by the function. However, performance will be negatively affected if requests are biased towards a particular partition. Furthermore, the PREDICT() function should itself utilize minimum CPU resources. However, if the function utilizes a lot of CPU resources just to analyze the request, then this PREDICT() function will decrease throughput. We assume that the PREDICT() function perfectly predicts the set of partitions accessed by each request. We will relax this assumption in Section 3.2.7.

Algorithm 3.1 shows the prediction stage of a replica participating in a BFT ordering instance. The prediction stage calls the PREDICT() function (see line 6)

to verify the prediction. The request is forwarded to the agreement stage after verification (see line 11).

```
1  initialize:
2      bft_id as BFT agreement instance
3      P_mi as set of predicted partitions
4      par as single partition
5  upon receiving ⟨REQUEST, m, i⟩ at replica i do
6      P_mi := PREDICT(m) // predict partitions
7      for each par in P_mi
8          bft_id := instance(par)
9          // get responsible instance
10         if i equals leader(bft_id)
11             start agreement stage of ⟨REQUEST, m, i⟩
12         end if
13     end for
```

Algorithm 3.1: Prediction Stage on a replica

3.2.4 Agreement

Like conventional SMR services, MLBFT orders client requests through an agreement stage. The protocol differs from PBFT by running multiple ordering instances. We define number of ordering instances o as a system parameter. We assume that o is equal to the number of replicas n in the system (see Section 3.1.2).

We consider each BFT instance as a black box and do not alter the protocol. Thus, any conventional BFT protocol can be used with this approach. We assume that each ordering instance runs the classical PBFT (see Section 2.2) protocol. The input to the protocol is set of client request and the output is ordered set of client requests agreed to by all other replicas in the BFT instance. Algorithm 3.2 shows that request is categorized by analyzing the predicted number of partitions. After a request has been categorized as simple or cross-border request, it is enqueued into corresponding relevant partition. The request type is marked as EXECUTE if the request is a simple request. However, if a request is a cross-border request, then a partition is deterministically picked from the relevant partitions and we mark the request type as CROSS-BORDER-EXEC in the corresponding queue. All other relevant partitions mark the request type as CROSS-BORDER-SYNC in their corresponding queues.

As mentioned earlier the number of BFT ordering instances is equal to the number of replicas n. Each replica participates in all BFT ordering instances running in the agreement stage. Thus a replica has two identities: *leader* and

```
1 initialize:
2     no_of_partitions as total number of BFT instances
3     bft_id as current BFT agreement instance
4     worker as the BFT execution instance
5     PartitionQueues := Queue[no_of_partitions]
6 before execute ⟨REQUEST, m, i⟩ at instance bft_id do
7     if count(m.partitions) = 1
8         // this is a simple request
9         m.type := EXECUTE
10    else // this is a cross−border request
11        // pick a BFT instance deterministically
12        worker := exec_instance(m.partitions)
13      if worker equals bft_id
14        // request m should be executed
15        m.type = CROSS-BORDER-EXEC
16      else
17        // request m should be synchronised
18        m.type = CROSS-BORDER-SYNC
19      end if
20    end if
21    PartitionQueues[bft_id].enqueue(m)
```

Algorithm 3.2: Identifying request type and then enqueue them

follower. We assume that each replica is leader of a particular BFT ordering instance and a follower of all other instances. A replica may be selected as a leader of multiple BFT ordering instances in case of faults. Furthermore, all the BFT ordering instances are independent of each other. This means that BFT ordering instances do not communicate with each other. The communication within a particular BFT ordering instance is done over the network (see Section 2.2). Figure 3.3 shows that each replica is leader of a particular BFT instances as well as follower of other instances.

Typically, each identity of the replica is implemented by a system thread. Each replica has 4 identities when there are 4 replicas in the system (n = 4). This introduces at least 8 threads per replica (4 threads for ordering stage and 4 worker threads for execution stage). The scheduling cost will be higher if the replica machine is not a multi-core machine and throughput will be less than its single leader counterpart. However, if the machine has a multi-core CPU, then scheduling cost would be less than for a single core (as multiple threads can execute at a time) and throughput will be improved.

We have already established that there is one-to-one mapping between BFT ordering instances and partitions (see Section 3.1.2.1). A partition is assigned to a particular BFT ordering instance and only that instance is allowed to order

requests in the corresponding queue. This enables requests to access different partitions to be ordered in parallel by their respective BFT ordering instances. Section 3.2.4.1 and 3.2.4.2 explains ordering of client requests in detail.

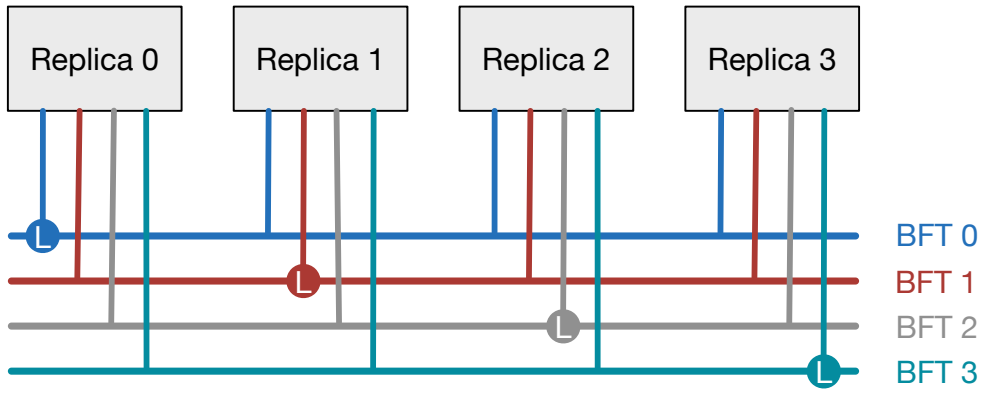

Figure 3.3: Multiple BFT instances

3.2.4.1 Partial Order

SMR requires total-ordering of requests to provide strong consistency and safety (see Section 2.1). MLBFT splits the total ordering of messages into multiple causally consistent partial orders. In particular, total-order is divided into o partial orders (o is number of BFT ordering instances. See Section 3.2.4). All requests the accessing same partition are considered to be dependent (see Section 3.1.1.2), thus they must be causally ordered in order to provide consistency. As we treat each BFT ordering instance as a black box, each ordering instance is perfectly capable of providing a causal order. A BFT ordering instance orders these dependent requests and adds them in the ordered queue of the corresponding partition. Only the corresponding BFT ordering instance is allowed to enqueue the request for the corresponding partition.

3.2.4.2 Total Order

Splitting total-order into multiple partial orders does not violate strong consistency, as all the partitions are disjoint (see Section 3.1.2.1) and only one BFT ordering instance orders the request accessing that partition. MLBFT provides safety and consistency by utilizing the safety and consistency properties of an underlying BFT ordering instance. Combining all partial orders will reflect the same state as

a single leader counterpart. Hence, safety is provided in all cases and the state objects will be consistent among all replicas.

Figure 3.4 shows how the total-order of requests ordered by a single leader L is translated by MLBFT (multiple leaders) into multiple partial orders. Requests with the same color are dependent requests and they must maintain the causal order to provide safety. Note that *Req* 3 and *Req* 6 maintain the causal order in both total-ordering and partial-ordering (as ordered by leader $L3$) of requests.

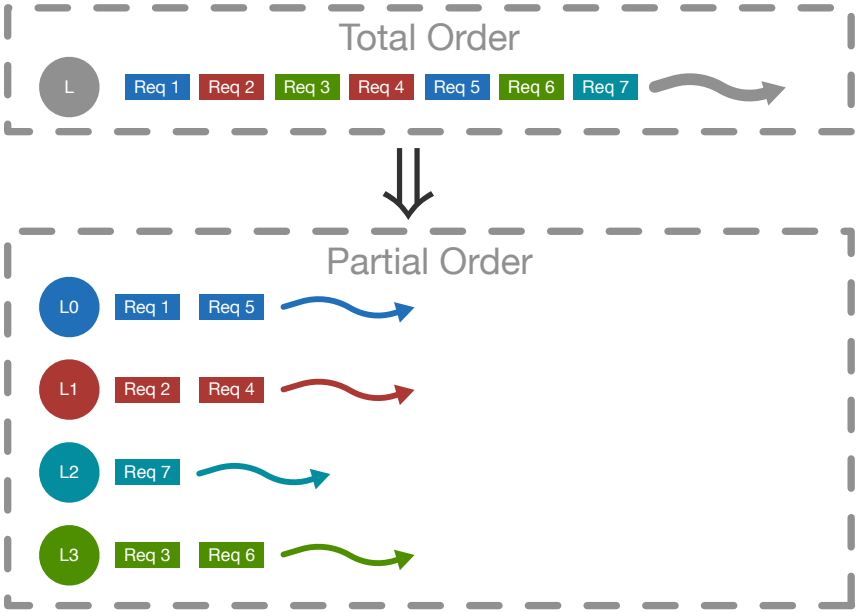

Figure 3.4: Splitting total-order into partial-orders

3.2.5 Execution

An ordered request is added to the ordered queue of the corresponding partition. Each queue is handled by a separate worker thread in the execution stage. A partition is assigned to a dedicated worker thread and only that worker thread operates on the assigned partition. A worker thread waits for the first available ordered request in the queue. In this discussion we assume that request is a simple request (see Section 3.2.6 for cross-border requests). A request is removed from the queue and becomes ready to execute. We call this operation EXECUTE if the request accesses single partition. An EXECUTE operation is executed and state objects are read or modified. Furthermore, a response is created and returned directly to the client. After the client receives a stable response (see Section 2.2), it sends its next request. Figure 3.5 shows a snapshot of a replica where requests

are executing in parallel. The requests $R0$, $R1$, $R2$, and $R3$ access different partitions so they are ordered in different queues. The worker threads $T0$, $T1$, $T2$, and $T3$ execute the corresponding requests in parallel. Each partition P_i (see Section 3.1.2.1) directly maps to a separate worker thread T_i. Algorithm 3.3 shows the idea of how a request is typically executed in a worker thread.

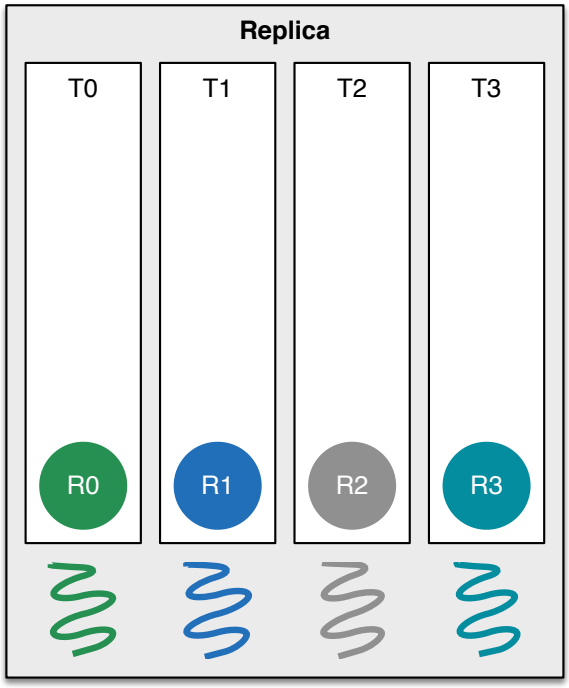

Figure 3.5: Parallel execution of requests in MLBFT

```
1 initialize:
2     req as a request
3 while executing at instance bft_id do
4     req := PartitionQueues[bft_id].peek()
5     if req.type is EXECUTE
6         execute(req)
7     else if req.type is CROSS-BORDER-EXEC
8         if detect_request_deadlock()
9             resolve_request_deadlock()
10        end if
11        wait for corresponding CROSS-BORDER-SYNC
12        executeCrossBorderRequest(req)
13    else if req.type is CROSS-BORDER-SYNC
14        wait for corresponding CROSS-BORDER-EXECUTE to execute request
15    end if
16    PartitionQueues[bft_id].dequeue()
```

Algorithm 3.3: Execution-stage worker thread

3.2.6 Handling Cross-Border Requests

In the previous section we assumed that requests will always be simple requests. This section addresses how cross-border requests (see Section 3.1.1.2) are handled by MLBFT. A cross-border request accesses objects that belong to more than one partition. Furthermore, a request can access an arbitrary number of objects and partitions. We established in Section 3.2.4 that a request can only be ordered by the corresponding BFT ordering instance. We extend this rule for cross-border requests. A cross-border request will be ordered by *all* relevant BFT ordering instances. The prediction stage will forward a cross-border request to the relevant BFT ordering instances and the request will be placed in ordered queues for *all* relevant partitions.

When the request has been ordered by each of the relevant BFT agreement instances and placed in the queues, then the request is retrieved by worker threads in the execution stage. Multiple worker threads will see the request as it has been placed in the queue by more than one BFT agreement instances.

When a CROSS-BORDER-EXECUTE request is available for execution it must wait for other threads. A CROSS-BORDER-EXECUTE request will only be executed when its corresponding CROSS-BORDER-SYNC request is available at *all* of the relevant partitions. The worker thread for a request marked as CROSS-BORDER-EXECUTE is the thread that will execute the request. All worker threads for requests marked with CROSS-BORDER-SYNC will be blocked until the request has been executed. This wait decreases the overall throughput of the system and this is why PREDICT() function must be implemented to minimize the number of cross-border requests. After the cross-border request has been executed, the worker threads of all relevant partitions move on to the next available request.

Figure 3.6 shows that both $R1$ and $R2$ are cross border requests. $R1$ is ordered in partitions 2, 3 (with worker threads $T2$ and $T3$) and $R2$ is ordered in partitions 0, 1 (with worker threads $T0$ and $T1$). Square notation for requests in queues 1 and 3 denotes that it is marked as CROSS-BORDER-SYNC. Requests $R1$ and $R2$ will be executed concurrently as both of them are available and independent of each other. Threads $T1$ and $T3$ will be blocked until the execution of their respective request is finished.

3.2.7 Handling Mispredictions

In this section we drop the assumption that prediction must be perfect for all requests, MLBFT does not assume perfect prediction. However, perfect prediction will improve throughput. Misredictions introduce an extra step in execution that should be avoided as much as possible.

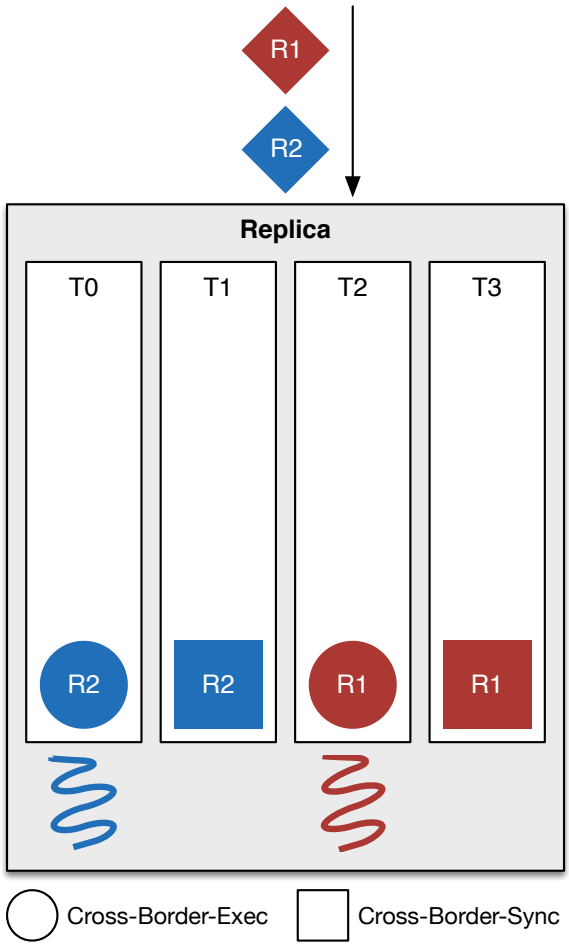

Figure 3.6: Execution of cross-border requests in MLBFT

The prediction stage passes the set of partitions for each request to the execution stage (see Figure 3.1). A part of the prediction stage maintains forecasts about all of the predicted requests. When a request is scheduled to execute at a worker thread in the execution stage, then this worker thread carefully monitors the execution and only grants access to the objects which were predicted in the forecast. If a request tries to access an object that is not in the forecast, then execution is halted immediately. The predictor component implements another function called RE-PREDICT(). This function takes the request as input and the set of objects which were not predicted before and produces a new set of partitions:

$$Set < partitions > \text{RE-PREDICT}(\ Request\ req\ ,\ Set < objects > new)$$

If the re-predicted partitions do not contain any new partition, then the execution is resumed as normal. Otherwise, the request should be ordered by the BFT instance of a new partition. The request goes through the agreement stage and is placed in the queue of a new partition. Execution continues when the request is available at the head of the queue of this new partition.

3.2.8 Deadlocks

Atomic multicast ensures ordered delivery of messages for all replicas. MLBFT does not rely on atomic multicast [12, 24, 26] of messages, thus messages can be received in different order by replicas. This causes MLBFT to be vulnerable to deadlocks.

A deadlock is possible if two or more cross-border requests accessing at least one common partition arrive in different orders at replicas. We classify deadlocks into two categories and explain how deadlocks are resolved in both cases.

3.2.8.1 Before Execution

In the execution stage when a worker thread encounters a CROSS-BORDER-EXECUTE request, it must wait for its corresponding CROSS-BORDER-SYNC on all of the relevant partitions. However, it is possible that the relevant partition is *also* in the waiting state. A cycle of requests waiting for each other will create a deadlock. At this point one of the relevant worker threads runs a deadlock detection algorithm and resolves any deadlocks encountered in finite time. This is called deadlock *before* execution because two or more requests are blocked and can not start execution.

All deadlocks are seen in exactly the same way by all replicas. It is impossible that these deadlocks look different a viewed by all of the different replicas. A deterministic resolution algorithm resolves all deadlocks when a deadlock is detected. The detection algorithm is only triggered when two or more requests cannot proceed to the execution stage because of synchronization. A resolution algorithm carefully looks at the partitions which are waiting for each other and detects possible cycles. If a deadlock is detected a CROSS-BORDER-SYNC request is deterministically picked and moved to the head of relevant partition to resolve the deadlock. The algorithm is optimised to move only the minimum number of requests required to resolve all of the deadlocks. As all of the relevant replicas run the same deadlock algorithm the result will be same on all of these replicas, hence the state will not diverge. A resolution algorithm may resolve multiple deadlocks at the same time. After a deadlock has been resolved at least one request will continue and starts its execution phase.

Figure 3.1 on page 20 showed the deadlock resolver component in the architecture. Figure 3.7 shows snapshots of two replicas in a deadlock situation. This deadlock is resolved by moving $R2$ on $T1$ to the head on both replicas. Algorithms 3.4 and 3.5 shows how to detect and resolve a deadlock before execution.

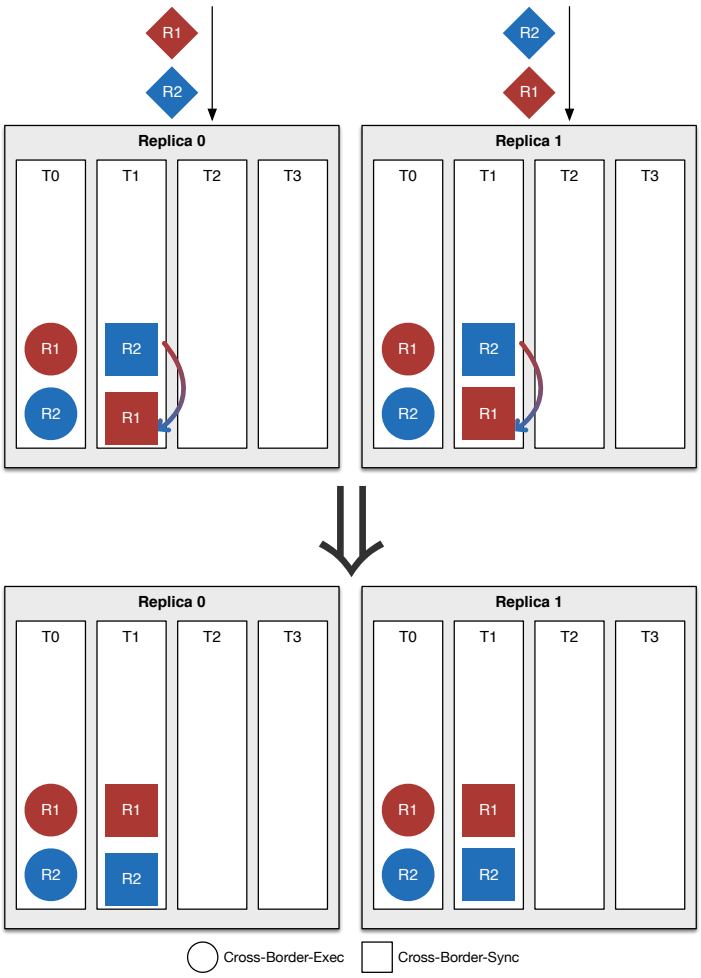

Figure 3.7: Deadlock detection and resolution before execution in MLBFT

3.2.8.2 After Execution

It is also possible that the execution stage encounters a deadlock *after* execution has already started by two or more cross-border requests. This is called deadlock *after* execution. If two ore more cross-border requests were wrongly predicted,

```
1 detect_request_deadlock() do
2     for all queue in PartitionQueues do
3         req := queue.peek()
4         if req.type is not EXECUTE
5             for all par in req.partitions do
6                 // detect cycle for cross−border requests in relevant partitions
7                 detected := detect_cycle(par)
8             end
9         end
10    end
11    return detected
```

Algorithm 3.4: Deadlock detection before execution

```
1 resolve_request_deadlock(Request req) do
2     for all par in req.partitions do
3         PartitionQueues[partition].move_to_head(req)
4     end
```

Algorithm 3.5: Deadlock resolution before execution

then they already have started the execution. In the middle of execution they try to access a common partition which was not predicted, thus execution must be halted. All new partitions will order these requests and assuming they are waiting for each other, then deadlock has occurred. This situation is very difficult to resolve. In this case the resolution algorithm will simply pick a request deterministically and put this request before the other partially executed request. Now the first request will be able to finish and control hands over to the second partially executed request which was halted. Now the second thread can continue and finish (without roll-backing) its execution as well. Execution continues normally when all deadlocks has been resolved.

3.2.8.3 Ordered queue

In order to resolve deadlocks our approach looks at the ordered requests in a queue. Usually, queues provide a First-in-First-out (FIFO) functionality with **`put()`** and **`take()`** methods. However, our implementation cannot rely on this interface as we may need to re-order events in the queue to resolve deadlocks. While resolving a deadlock it should be possible to re-order the requests inside a queue, while keeping rest of the order same. This ordered queue must provide an interface to find and move a particular request to the head of the queue. Additionally, all the queues must be thread-safe. A thread safe queue will

guarantee that only one thread (an ordering thread or a worker thread) performs an operation on the queue. Selection of appropriate data structures and locks are necessary to implement this functionality.

3.2.9 Safety and Liveness

The section discuss safety and liveness properties of the approach. The protocol uses an existing agreement stage (see Section 3.2.4 and Section 3.2.12.1) which guarantees safety [9, 14] and liveness when the *bounded fair links* [9, 14, 13] assumption holds. Furthermore, Section 3.2.8 explained deadlocks do get resolved deterministically within a finite time. Thus, even if there is a deadlock replicas will not diverge. Liveness is provided because at least one deadlock is resolved at a time and eventually all deadlocks will be resolved.

Section 3.2.4 explains that splitting total-order into multiple partial orders is equivalent to a totally ordered single leader approach. Partial order of requests ensures that causal relationship is not violated, hence the replicas are consistent in our approach.

3.2.10 Checkpoints

Checkpoints are useful in case of faults or when a new replica joins the system (see Section 2.1.4). In MLBFT there are two ways to create checkpoints. Either we can create a new checkpoint with all the state objects from all partitions or we can create checkpoints for individual partitions. The latter approach is useful if a particular partition is faulty. We will adopt the second approach in this thesis.

3.2.11 View-Change

In case of a faulty leader MLBFT utilizes the view-change protocol of the underlying BFT instance. When a new view is selected all replicas leave their current view and move to the next view [9, 14].

3.2.12 Implementation

This section explains some implementation details of the protocol. We explain two ways in which the protocol can be implemented regardless of any programming model. Both have advantages and disadvantages and tradeoffs have to be made by selecting one approach over the other. In Section 3.2.12.3 a comparison is made between these two approaches.

3.2.12.1 Extension of Conventional BFT protocols

As mentioned before we treat the existing agreement stage as a black box and simply utilize multiple BFT instances. We built MLBFT on top of an underlying agreement layer (of PBFT) and extended it for our purpose. This provides us the advantage that we can utilize any PBFT based single leader BFT agreement layer. However, this approach gives limited control over agreement stage.

3.2.12.2 Re-write MLBFT

Another approach is to re-write all the components of BFT. This approach gives us control over both the agreement and execution stages. Additionally, a large number of optimisations could be made to improve performance. For example, it is possible to reduce the memory footprint of MLBFT, hence utilizing less resources. The downside is that such a new implementation requires a lot more development time and resources than the first approach.

3.2.12.3 Comparison

Table 3.1 shows a comparison between the two implementation approaches discussed before.

Table 3.1: Comparison between implementation approaches

Extension of BFT protocols	**Re-write MLBFT**
Can utilize any existing PBFT agreement layer	Cannot utilize any existing PBFT agreement layer
Underlying agreement layers are encapsulated	Agreement layers are not encapsulated
Requires less development time and resources	Requires more development time and resources
Larger memory footprint because similar objects are created more than once between agreement layers	Smaller memory footprint because common objects between agreement layers have to be created only once
Opportunity for less improvements and optimizations	Opportunity for more improvements and optimizations
Ideal for prototyping	Ideal for deployment

Chapter 4

Evaluation

This chapter evaluates the impact of the MLBFT approach on throughput and compares it with a common single-leader approach. This chapter also provides implementation details, a selected case study, and benchmarking results.

4.1 Amdahl's law

In 1967, a computer architect named Gene Amdahl established a relationship between the number of processors and performance [33]. This relationship is known as "Amdahl's law". Amdahl's law is used to predict the maximum improvement to a system using multiple processors. We use Amdahl's law to predict the theoretical maximum performance that can be achieved by implementing our approach. Please note that the original paper did not contribute any mathematical equations to establish Amdahl's law, however others derived an equation from what he said in the fourth paragraph of his paper.

In this analysis we assume a system that can tolerate one faulty replica ($f = 1$), with $n = 4$ replicas. Let us assume a workload of 4 independent client requests as Amdahl's law can only predict improvement over fixed workloads. Furthermore, we assume that these 4 requests are simple requests and these requests are independent of each other. Let us denote the time to execute these requests on a single-leader approach by $T(1)$. By applying Amdahl's law we can predict the time to execute these requests using a multi-leader approach by following equation:

$$T(N) = T(1)(S_{ser} + \frac{S_{par}}{N})$$

Where N corresponds to number of BFT instances ($N = n = o$) and $S_{ser} + S_{par} = 1$. Furthermore, S_{ser} denotes the fraction of strictly serialized

execution and S_{par} (= 1 - S_{ser}) denotes the fraction of execution that can utilize multiple cores of a CPU. The theoretical performance improvement (denoted by $S(N)$) of MLBFT can be calculated by following equation:

$$\begin{aligned} S(N) &= \frac{T(1)}{T(N)} \\ &= \frac{T(1)}{T(1)(S_{ser} + \frac{S_{par}}{N})} \\ &= \frac{1}{S_{ser} + \frac{S_{par}}{N}} \\ &= \frac{1}{S_{ser} + \frac{(1-S_{ser})}{N}} \end{aligned}$$

We assumed 4 simple requests can be executed in parallel. Amdahl's law assumes that the parallel part of the execution can be infinitely parallelized, which is not true in our case. Parallelization is bounded by the number of BFT instances in our case, hence no more than 4 requests ($1 \leq N \leq 4$) can be executing in parallel at a time. We also assume that a single request cannot be further parallelized. This means that exactly one agreement layer can order a single request. As there are 4 independent requests, we can assign them to different partitions to maximize concurrency. An agreement instance will order each request in a separate partition. As a result we can order a single request in the serialized portion of the code ($S_{ser} = \frac{1}{4}$). We can calculate the maximum improvement by putting the values $N = 4$ and $S_{ser} = \frac{1}{4}$ (for 4 BFT instances) into above equation.

$$\begin{aligned} S(4) &= \frac{1}{\frac{1}{4} + \frac{(1-\frac{1}{4})}{4}} \\ &= 2.285 \approx 2.3 \end{aligned}$$

We can also compute the performance when number of cores are less than the total number of BFT instances ($2 \leq N < 4$):

$$S(2) = \frac{1}{\frac{1}{4} + \frac{(1-\frac{1}{4})}{2}} = 1.6$$

$$S(3) = \frac{1}{\frac{1}{4} + \frac{(1-\frac{1}{4})}{3}} = 2$$

The above calculations shows that MLBFT has potential to achieve over 100% higher throughput than single-leader approach in ideal cases. This model only predicts the maximum performance that can be achieved. However, in practice, the actual throughput improvement is usually less than theoretical performance. Section 4.3.3 uses the above model to compare to the performance improvements achieved by a microbenchmark.

4.2 Implementation

The prototype of MLBFT was implemented as an extension of Resource-efficient Fault and Intrusion Tolerance (REFIT) [34], a BFT library, written in Oracle Java 7. REFIT follows the common single-leader design based on the ideas of PBFT [9, 14] to implement the agreement stage. As mentioned in Section 3.2.12.1, we treat the agreement stage as a black box. This gives us the advantage that we can avoid the need to make complicated modifications to the agreement stage. We configured the system to instantiate single (in case of a single-leader) or multiple (in case of MLBFT) agreement protocols for later use in our evaluation. As a result a clean and well organized BFT library was developed which is capable of running either single or multiple leader protocols depending on its configuration. Moreover, both agreement and execution stages were implemented in separate threads (not processes) to maximize concurrency.

4.3 Microbenchmark

We evaluate the performance of MLBFT by implementing a simple Key-Value store for our micro benchmark. A Key-Value store is a good application to evaluate because of its simple design and limited number of operations. In addition to the Key-Value store, our application also implements a prediction function that can perfectly predict the state objects accessed in a request. These state objects are partitioned evenly across all BFT instances by the prediction function. In this case study we measure the performance for both simple and cross-border requests in terms of throughput and resources used by the replicas.

4.3.1 Key-Value Store

We implement the Key-Value store as a standard Java `MAP⟨K,V⟩` interface in the `java.util` package. Our implementation provides a basic `HASHMAP⟨K,V⟩` functionality with `Integer` keys and `String` datatypes as values. Moreover, the implementation provides three basic operations from the `Map` interface:

`get()` & **`put()`** as simple requests and **`putall()`** as a cross-border request. Additionally, this implementation also provides a `Predictor` object (see Appendix A.2). This predictor is capable of perfectly predicting state objects by parsing the client request. Furthermore, this predictor evenly distributes state objects to partitions based on the *key* which is generated by a client using Java's pseudorandom number generator (`java.util.Random`). We provide some of the source code for this implementation in Appendix A

4.3.2 Evaluation Setup

The evaluation setup consists of a small cluster of four physical machines representing four replicated state machines. This setup enables the service to tolerate a single faulty replica ($f = 1$ and $n = 4$). In addition to replicas, a separate physical machine was used as a client. This client machine is capable of running multiple client threads. Each thread is synchronous and can only send one request at a time. (see Section 3.1.1). Each physical machine is a Dell Optiplex 7010 machine equipped with Intel i7-3770 CPU (quad-core with hyper-threading enabled) clocked at 3.40 GHz and with 8 MB of L3 cache. Each of these machines is running Ubuntu Linux Server 14.04 64-bit and has 16 GB of physical memory installed. Furthermore, all of the machines are connected via a Gigabit Ethernet switch (Nortel BayStack 5520). Each of these machines is using the built-in network interface (Intel® 82579LM Ethernet LAN 10/100/1000) to communicate with and the interface is set to operate at 1 Gbps. All of these network interfaces are set to full-duplex mode.

4.3.3 Results

Figure 4.1 shows the performance results of MLBFT compared with the single-leader approach for simple requests. For a very small number of clients (less than 5 clients), throughput of MLBFT is roughly comparable to the throughput of a single-leader. A special case occurs for a single client when MLBFT performs a little worse than the single-leader. This happens because of the higher overhead of the multi-leader approach. However, when number of clients increases, it is observed that throughput of MLBFT increases up to a factor of two greater than the single-leader. Furthermore, the throughput of the single-leader starts to saturate at approximately 50 clients. This saturation point is seen at a much higher load (around 110 clients) in case of MLBFT.

In Section 4.1 we predicted that MLBFT can achieve up to 100% higher throughput than for the single-leader approach. Figure 4.1 shows a plot of this theoretical throughput (denoted by Max-ML). It is observed that the experimental performance achieved by MLBFT is nearly equal to the expected theoretical

throughput. In practice, the throughput is usually less than maximum achievable performance due to miscellaneous overheads. The maximum throughput that can be achieved is around 50 K requests/second in this configuration. MLBFT achieves slightly less throughput than the predicted model whereas the single-leader was only able to achieve around 20 K requests/second.

The increasing trend of the throughput can be estimated by a third order logarithmic equation. This model fits the experimental data very well. The first order logarithm will not be a better fit to the experimental data as first order logarithm always tend to increase (or decrease). These estimated equations and R^2 of the curves in Figure 4.1 are given below the figure.

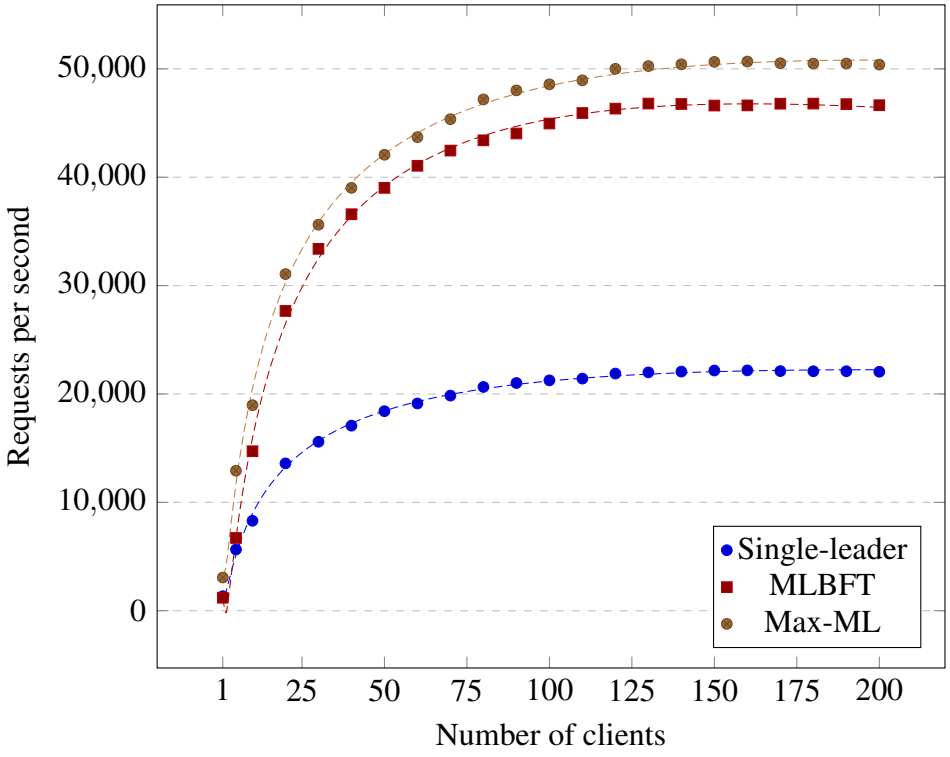

Figure 4.1: Throughput for simple requests

single-leader

$$y = 1413.68 - 1162.55\ ln(x) + 2675.51\ ln^2(x) - 323.57\ ln^3(x)$$

$$R^2 = 0.998$$

MLBFT

$$y = 1118.69 - 6998.35\ ln(x) + 8057.96\ ln^2(x) - 966.80\ ln^3(x)$$
$$R^2 = 0.998$$

Max-ML

$$y = 3230.27 - 2656.44\ ln(x) + 6113.54\ ln^2(x) - 739.37\ ln^3(x)$$
$$R^2 = 0.998$$

4.3.3.1 Payload

Figures 4.2 to 4.5 illustrate the relationship between payload size and throughput. As mentioned earlier, each machine uses a built-in network interface. The Maximum Transmission Unit (MTU) size was set to 1500 B and Jumbo Frames were not enabled (because the MTU was fixed to 1500 B). Unfortunately, we were not able to modify these settings because of restricted rights over the machines. The number of clients was set to 100, while the payload sizes vary from 50 B to 8 KB in this benchmark. The payload consists of application data. We separately measure both request and reply payloads.

Fragmentation overhead

Our approach does not introduce any additional dependencies on payload sizes. However, we briefly analyze the underlying network layer to show how it affects the throughput when we increase the payload size. The network messages are fragmented when they are larger than MTU, hence increasing the latency of protocol messages. All the messages greater than 1500 B were fragmented in our tests. The header size of a TCP/IP packet over Ethernet is 40 B (20 B for TCP and 20 B for IP). This means we can effectively use 1460 B (1500 B - 40 B) for the protocol messages without any fragmentation. For larger payload sizes e.g., 8 KB, we can calculate the overhead of fragmentation as follows:

$$\begin{aligned} no.\ of\ packets &= \left\lceil \frac{payload\ size}{effective\ frame\ size} \right\rceil \\ &= \left\lceil \frac{8192}{1460} \right\rceil \\ &= 6\ packets \end{aligned}$$

This means that 6 packets must be sent for a payload size of 8 KB. This fragmentation (and re-assembling) of packets will happen for each protocol message greater than MTU size. The overhead to transfer 1 message of size 8 KB at Transport layer will be $6 \times 40 = 240$ Bytes. Furthermore, there is additional overhead of Ethernet frames (4 Bytes CRC + 8 Bytes Preamble + 18 Bytes Ethernet header). The latency of transferring 1 packet can be calculated as:

$$\begin{aligned} latency &= \frac{MTU}{Network\ speed} \\ &= \frac{1530 \times 8\ bits}{1,000,000,000\ bits\ per\ second} \\ &= 0.01224\ ms \end{aligned}$$

The latency (ignoring the latency of the switch) to transfer a 8 KB packet will be 0.14688 ms (6 packets $\times$ 0.01224ms $\times$ 2 (switch hops)). Due to the increased overhead and latency per packet, it takes more time to deliver large messages.

Figure 4.2 shows that MLBFT outperforms single-leader by a factor of two (as expected, see Section 4.1) for all request sizes. Although, it is observed that throughput decreased with increasing request size. We have already explained that it takes longer to deliver large messages because of fragmentation for request sizes greater than 1500 B (i.e., 2 KB, 4 KB, and 8 KB in Figure 4.2). Furthermore, the increase in latency also increases the time to reach a consensus for each request, hence decreasing the throughput. The curves show a logarithmic trend that can be estimated by the following set of equations:

single-leader

$$\begin{aligned} y &= 37735.59 - 3728.14\ ln(x) \\ R^2 &= 0.969 \end{aligned}$$

MLBFT

$$\begin{aligned} y &= 76876.19 - 7592.09\ ln(x) \\ R^2 &= 0.967 \end{aligned}$$

Max-ML

$$y = 86791.86 - 8574.73\ ln(x)$$
$$R^2 = 0.969$$

From the equations of MLBFT and single-leader we observe that the ratio between constants and coefficients of logarithmic terms is roughly $2\times$. This shows that MLBFT outperforms single-leader by a factor of 2 for all request sizes. This is also consistent with the theoretical model we established in Section 4.1.

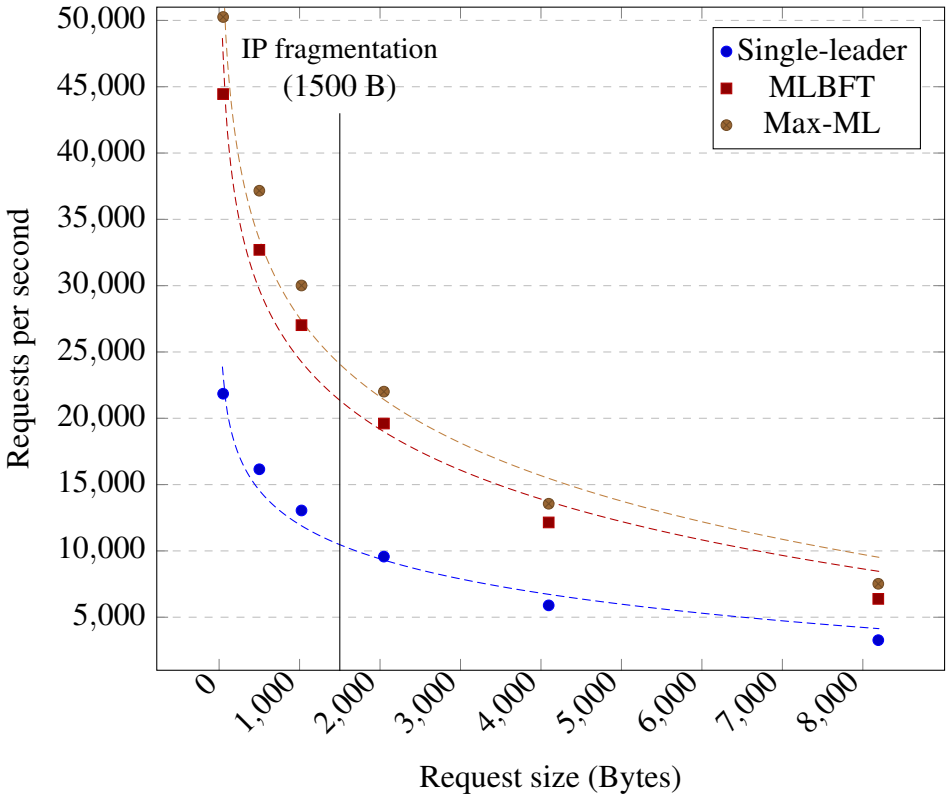

Figure 4.2: Throughput with different request sizes

Figure 4.3 shows the relationship between request size and bits per second. Table 4.1 shows the same results in the form of bandwidth used by each approach. The numbers represents Megabits per second for different request sizes in the table. The trends of Figure 4.3 can be estimated by following equations on log scale:

single-leader

$$y = -4194606470.31 + 1024367199.42\ ln(x)$$
$$R^2 = 0.956$$

MLBFT

$$y = -8284210070.05 + 2044367082.9\ ln(x)$$
$$R^2 = 0.953$$

Max-ML

$$y = -9647594881.45 + 2356044558.63\ ln(x)$$
$$R^2 = 0.956$$

Table 4.1: Bandwidth usage for different request sizes (Mbps)

Payload / **Approach**	**50 B**	**500 B**	**1 KB**	**2 KB**	**4 KB**	**8 KB**
Max-ML	57.5	425.2	703.4	1031.6	1270.8	1410
MLBFT	50.8	374	633.3	918.9	1138.7	1197
Single-leader	25	184.8	305.8	448.5	552.5	613

We observe the ratio between constants and coefficients of equations for MLBFT and single-leader is approximately $2\times$. This ratio also fits our theoretical model. At a message size of around 8 KB we observe a saturation in both approaches. MLBFT and single-leader attain their maximum throughput at this point. Both approaches rely on the agreement layer to reach a consensus for each request. The agreement layer consists of three rounds of messages (see Section 2.2.2). It takes a minimum of 18 messages to reach a consensus. The

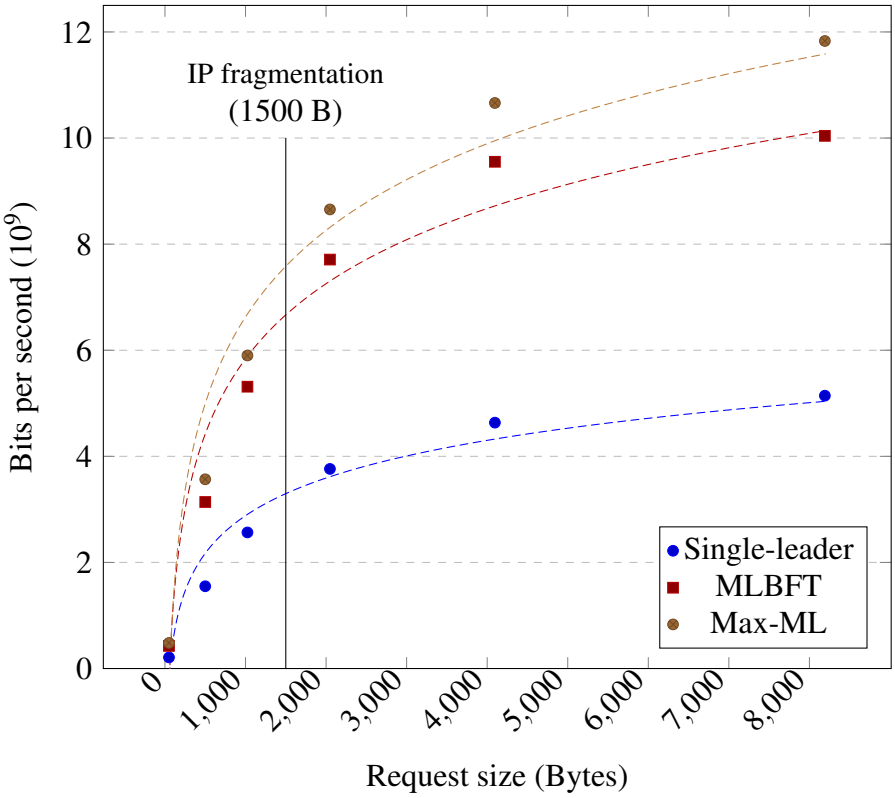

Figure 4.3: Bits per second with different request sizes

detailed measurements of the overhead to reach a consensus is outside the scope of this thesis because we consider the agreement layer as a black box (see Section 3.2.4). However, we have shown a rough calculation of overhead and latency of the messages in this section.

Form the Figure 4.3 and Table 4.1 it is clear that bandwidth can become the bottleneck for larger request sizes. In our configuration we use Gigabit Ethernet as mentioned before. It is observed that Ethernet bandwidth becomes the bottleneck for request size of more than 2 KB. This is because PBFT (see Section 2.2) generates 24 internal protocol messages for a single incoming client-request. As the number of clients (and requests) increases, protocol messages saturate the available bandwidth hence limiting the throughput. The bandwidth can be increased (if required) by switching to 10 Gigabit Ethernet or by installing additional network interfaces.

Figure 4.4 shows the measurements for different reply sizes. The multi-leader approach outperforms the single-leader approach again for all reply sizes. Measurements show a gradual decrease in throughput for smaller reply sizes (less than 2 KB). The performance decreases with the increase in reply size. This happens because of network fragmentation for messages explained earlier. A reply is generated in the last round of messages and only f + 1 similar replies are required to verify that the reply is stable. These messages are 6 times less than the internal communication messages of the protocol (see Section 2.2.2). This is why performance is decreased a lot by increasing the request size as compared to the effect on performance by increasing the reply size. The trend can be estimated by following linear equations:

single-leader

$$y = 21155.17 - 1.615\,x$$
$$R^2 = 0.943$$

MLBFT

$$y = 43908.62 - 3.89\,x$$
$$R^2 = 0.980$$

Max-ML

$$y = 48656.89 - 3.71\,x$$
$$R^2 = 0.943$$

The ratio of coefficients and constants between the equations of MLBFT and single-leader is roughly 2 as expected (theoretically). It is also observed that MLBFT cannot match the throughput of the predicted model because of the fragmentation for larger reply sizes. Similar results can be observed in Figure 4.5. It is shown that performance of both approaches is degraded for larger reply sizes (larger than MTU). Following logarithmic equations of order two estimates the curves in Figure 4.5.

single-leader

$$y = 599364178.87 - 265464191.99\,ln(x) + 29289334.79\,ln^2(x)$$
$$R^2 = 0.999$$

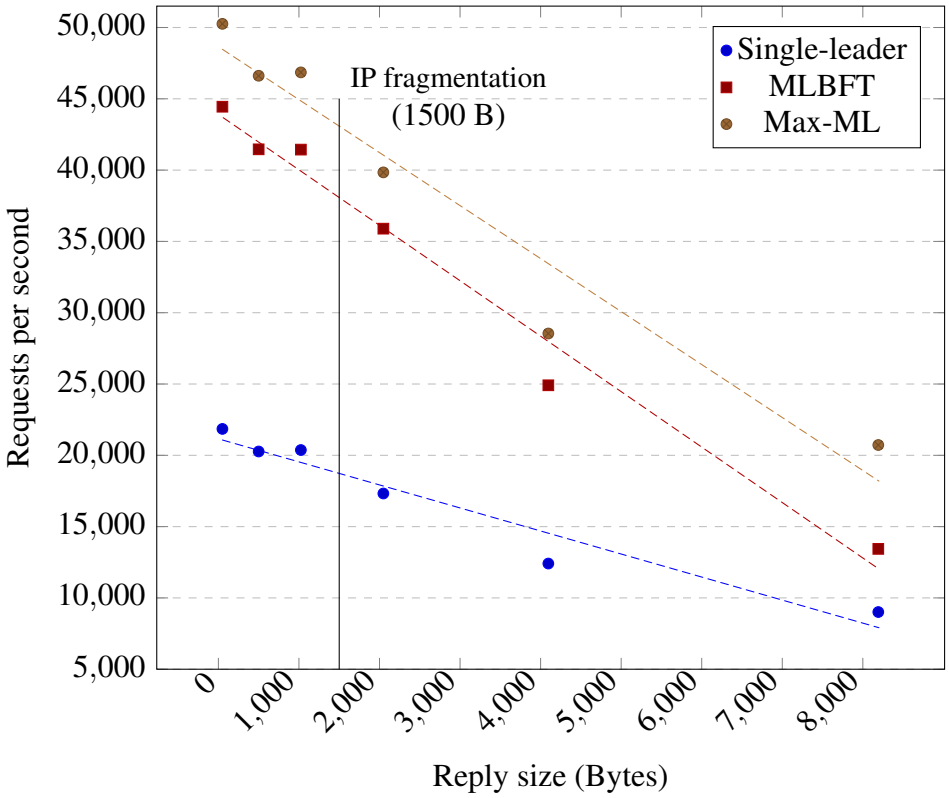

Figure 4.4: Throughput with different reply sizes

MLBFT

$$y = 389968679.72 - 223892341.58\ ln(x) + 31802917.18\ ln^2(x)$$
$$R^2 = 0.965$$

Max-ML

$$y = 1378537611.40 - 610567641.59\ ln(x) + 67365470.02\ ln^2(x)$$
$$R^2 = 0.999$$

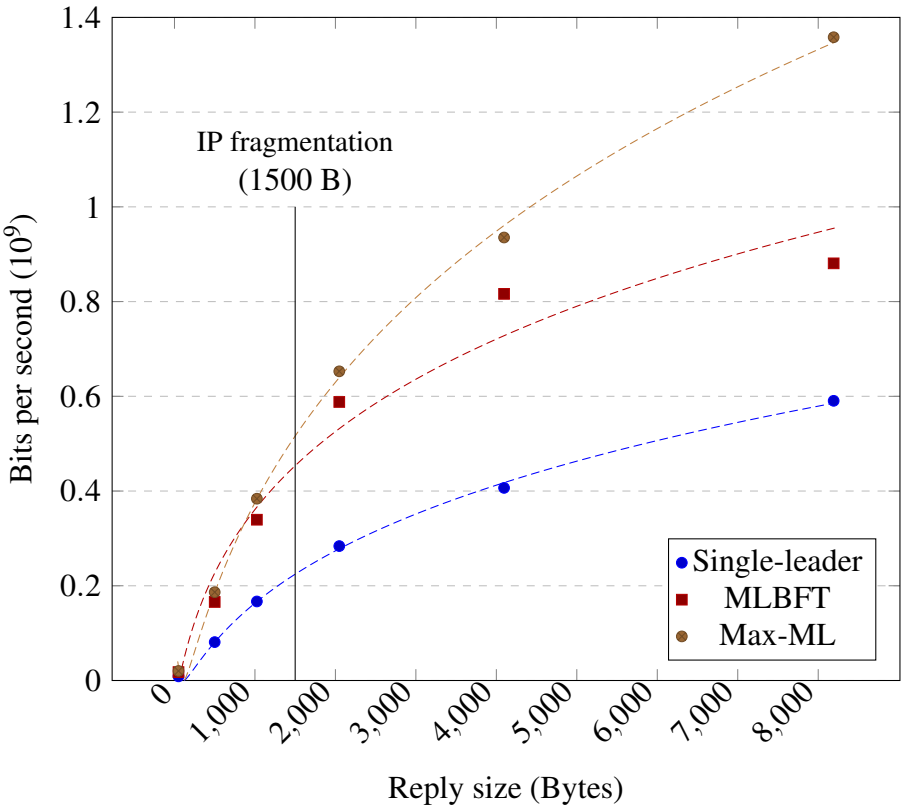

Figure 4.5: Bits per second with different reply sizes

We also measure the effect of response times while varying payload (request and reply) sizes. In all cases MLBFT performs better than the single-leader setup (see Figures 4.6 and 4.7) with respect to the measured response time. MLBFT performs up to twice as fast as the single-leader approach for smaller payload sizes. Both of these benchmarks show that performance is bounded by the payload sizes which can become a bottleneck for networks with lower MTU. This is because messages are fragmented when the payload size is greater than MTU. It is recommended to use larger MTU values and enable Jumbo frames (if available) for larger payloads (request and reply). Figure 4.6 and 4.7 shows that response time is not greatly affected by payload size when there is no IP fragmentation. The response time increases linearly for all payloads larger than MTU because of fragmentation. Furthermore, it takes longer to deliver large messages, hence increasing the response time.

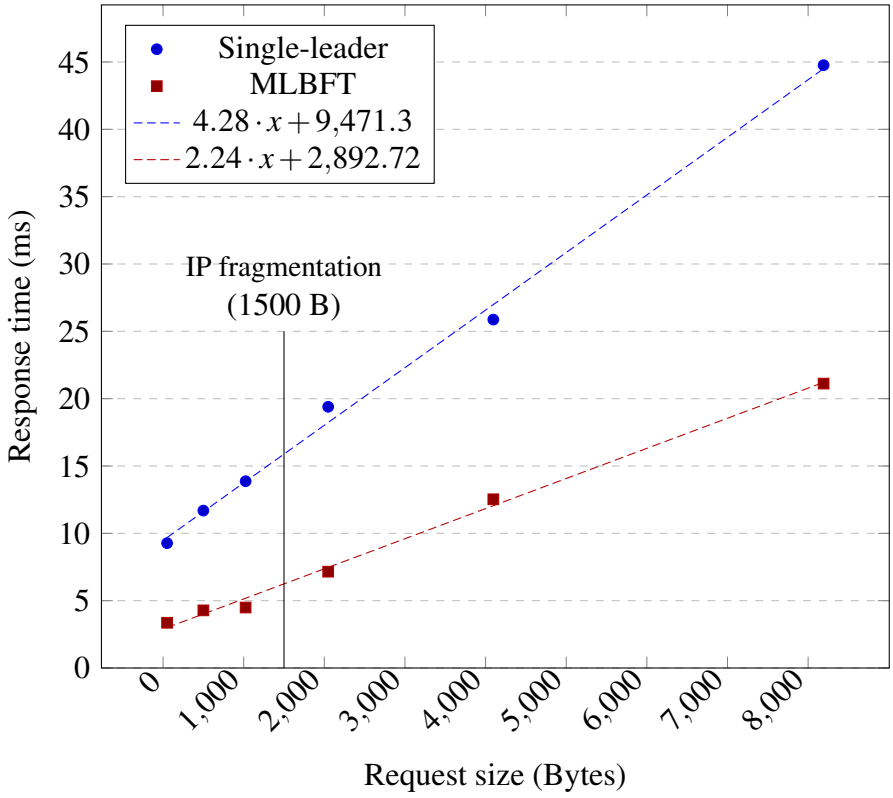

Figure 4.6: Response time with different request sizes

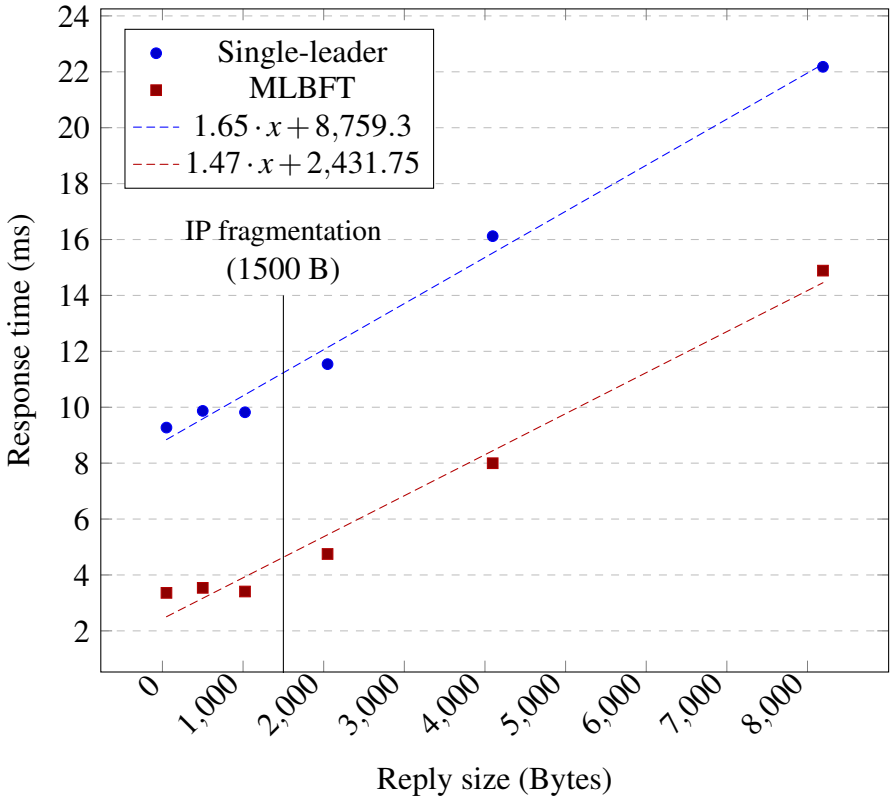

Figure 4.7: Response time with different reply sizes

4.3.3.2 Response time

Figure 4.8 shows the average response time of a request with different numbers of clients. It is shown that in nearly all cases single-leader approach has higher response time as expected than MLBFT. This happens because single-leader provides a total-order of messages. However, MLBFT can perform multiple consensus at the same time exploiting a partial-order of messages. Benchmarks show that MLBFT can perform up to twice as fast as single-leader. For a special case when there is a single client, we observe nearly 11% decrease in the performance of the MLBFT approach than a common single-leader approach. This decrease in performance is because of higher overhead of MLBFT, while performing prediction at both client and server side to decide upon the partition, hence BFT agreement will order the request. The regression lines show a linear trend in increase of response time for both approaches. We observe that the slope of MLBFT (13.57) is lower than single-leader (42.98). This means that increase in response time for MLBFT approach is much lower than increase in response time for the single-leader approach, when there is an increase in number of clients. This ratio shows that MLBFT can serve roughly $3\times$ more clients than single-leader approach for a given response time.

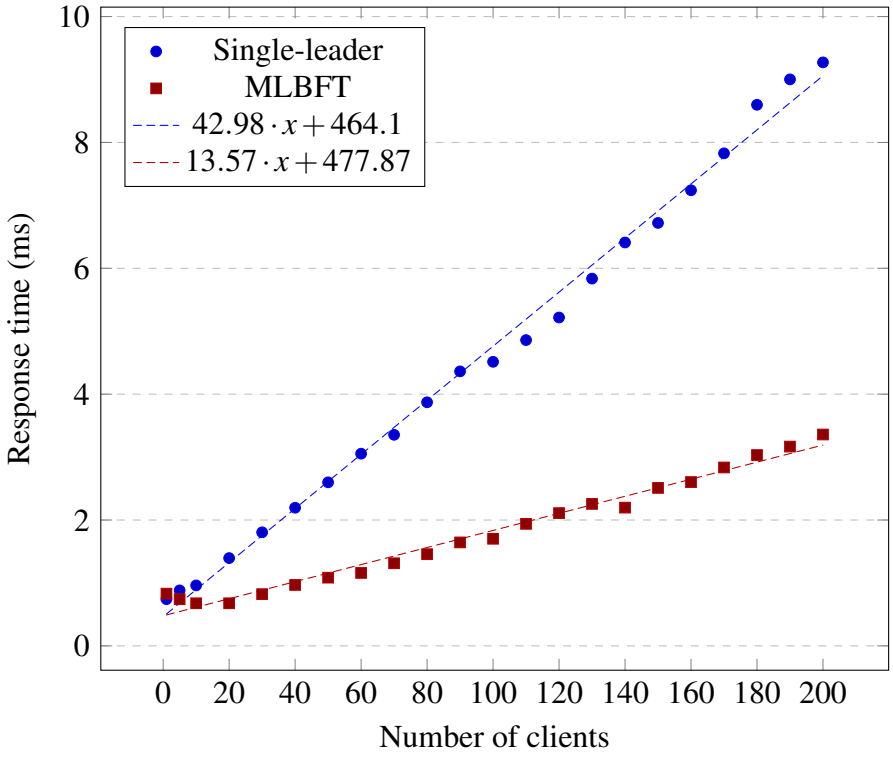

Figure 4.8: Average response time

4.3.3.3 CPU usage

A benchmark was designed to measure individual CPU usage for each replica. CPU usage is compared between MLBFT and single-leader approach in Figure 4.9. As expected, the leader (Replica 0) of single-leader approach consumes 11% more CPU than other replicas. This increase in CPU usage is a limitation of single-leader approach. On the other hand, MLBFT distributes work load to all replicas, hence CPU usage is similar on all machines. We can observe that the 11% more CPU usage of single-leader is evenly distributed in multi-leader approach ($\sim \frac{11\%}{4} = 2.75\%$). Hence, a slight increase in CPU usage ($45\% + 2.75\% = 47.75\% \approx 47\%$ usage for all replicas in MLBFT) can be seen in the Figure 4.9.

In a single-leader approach every request has to go through the leader and there is no parallel part in the execution ($S_{ser} = 1$). However, in the case of MLBFT the workload is distributed among all replicas equally (in ideal case). This means that for a given workload, a single replica will order $\frac{1}{4}$ of total requests (as there are 4 replicas). This further explains that why $S_{ser} = \frac{1}{4}$ in Section 4.1.

In our benchmarks we used 8 core (4 physical cores with hyper-threading) machines for replicas. Our configuration uses 4 BFT instances as there are 4 replicas. This means that we are essentially using only 4 cores of each replica as one core of each replica participates in a dedicated BFT instance. Each core handles $\frac{1}{4}$ of total workload. The operating system reports that system is using almost 50% of CPU usage in the benchmarks. This is because only 4 cores which are participating in BFT instances are busy to handle the workload. The rest of the cores are idle and there is no more work to do. Even if more client requests comes MLBFT will schedule those requests on 4 cores that are already busy. This is why we can only utilize 50% of the CPU usage on the 8 core machine. To utilize full CPU power either number of BFT instances should be increased or the system must be configured for $f > 1$. This is why the number of cores should be equal to number of BFT instances (ideally).

The CPU time is spend on calculating and verifying signatures for protocol messages, calculating request digests and executing client requests. Furthermore, there is additional CPU overhead of prediction, partitioning and deadlock detection & resolution. It is observed that most of the CPU time is spent on the calculation of signatures and message digests. Our implementation uses SHA-1 [35] algorithm as HMAC to verify protocol messages. Moreover, our implementation uses MD5 [36] algorithm to calculate request digests. Table 4.2 shows the speed of these hash algorithms on the CPU used for our benchmarks. The numbers represents 1000s of Bytes processed per second for each algorithm on the given CPU. It is observed that SHA-1 performs faster than MD5 in all scenarios.

Table 4.2: Bytes processed per second for different hash algorithms

Block Size / Algorithm	16 B	64 B	256 B	1024 B	8192 B
MD5	69166 K	203310 K	452872 K	656342 K	752943 K
SHA-1	80329 K	227637 K	491785 K	700090 K	815169 K

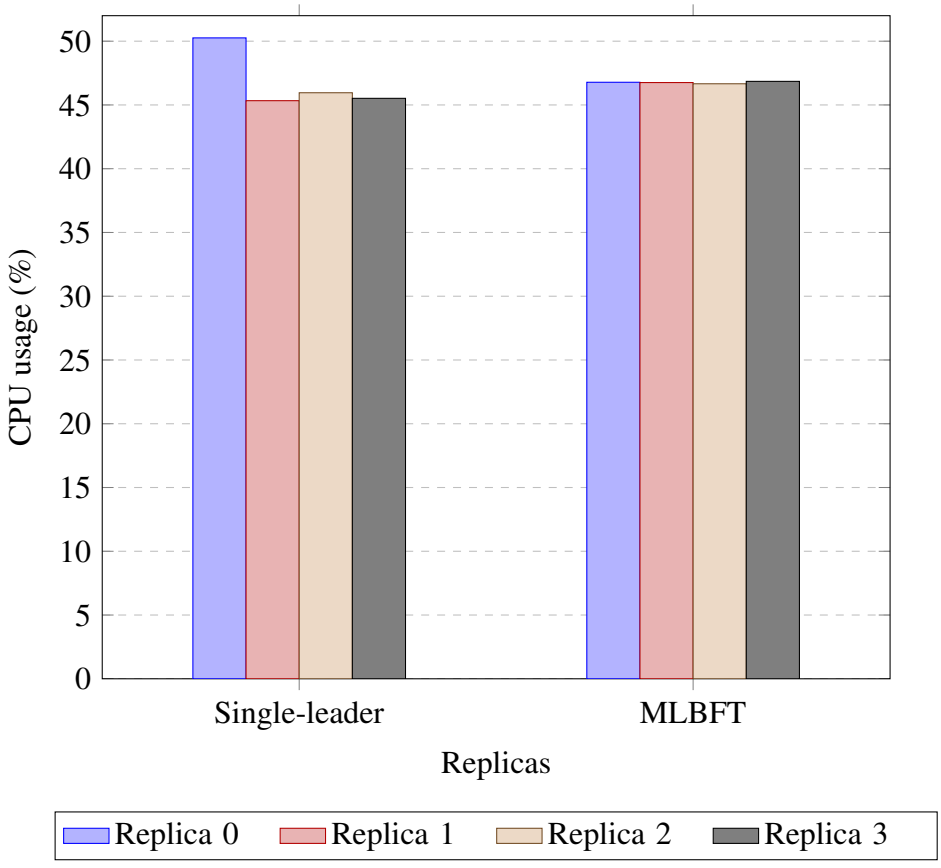

Figure 4.9: CPU Usage

4.3.3.4 Cross-border requests

Performance of cross-border requests is also a focus of MLBFT. We compare the throughput of cross-border requests with single-leader approach in Figure 4.10. The benchmark generates cross-border requests accessing 2 partitions in different proportions. It is observed that MLBFT outperforms the single-leader solution in most cases. Furthermore, it is observed that throughput is decreased with an increase in the proportion of cross-border requests increases. The performance is similar to single-leader when all of the requests are cross-border requests. In addition, un-even changes in throughput are also marked in the figure. These changes happened because of greater number of deadlocks that occurred while running the benchmark. It should be noted that the prediction function must be designed in a way that it minimizes the proportion of the cross-border requests as well as number of partitions that are accessed in each request. A slightly modified version of the prediction function was used in this benchmark to control the proportion of cross-border requests from the configuration. Please note that perfect prediction does not mean there are no cross-border requests. A perfect prediction means that all of the state objects (and partitions) that are going to be accessed by the request will be returned by the PREDICT() function (see Section 3.2.3).

Each cross-border request is ordered by more than one BFT instances. This means that parallelization will decrease, hence S_{ser} will increase, when proportion of cross-border requests increases. We observe a speedup of nearly a factor of 1.8 with 1% cross-border requests in Figure 4.10. If we solve the equation discussed in Section 4.1 for a speedup of 1.8 (neglecting the overhead), we get $S_{ser} \approx 0.41$. Furthermore, if the proportion of cross-border requests is increased to 100%, we get $S_{ser} \approx 1$. This means that MLBFT will behave similar to the single-leader approach when all of the requests are cross-border requests. The increase in proportion of S_{ser} is observed as exponential as we see an exponential decrease in throughput while increasing the proportion of cross-border requests and from the equation we know that S_{ser} is inversely proportional to speedup. Additionally, the decrease in throughput in case of cross-border requests can be estimated by following equation:

$$y = 29453.26\ (1+x)^{-0.133}$$
$$R^2 = 0.967$$

This equation shows an inverse relationship between throughput and cross-border requests ($0 \leq x \leq 100$). The equation also shows that MLBFT approach cannot perform worse than single-leader approach even when all of the requests ($x = 100$)

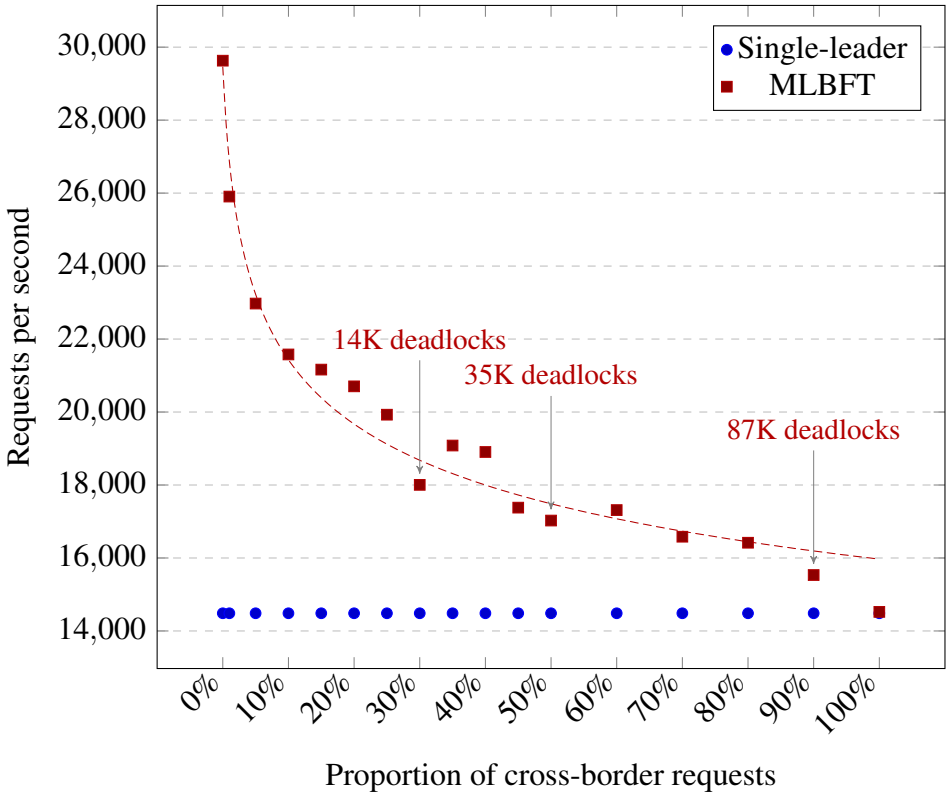

Figure 4.10: Throughput of cross-border requests

are cross-border requests (with 2 partitions). MLBFT performs 2 times better than the single-leader approach in ideal case ($x = 0$).

Figure 4.11 shows the relationship between throughput and number of clients. We compare throughput of single-leader (SL) with 0%, 1%, 15%, and 50% (denoted by ML, ML-1, ML-15, and ML-50 in the figure respectively) cross-border requests. It is shown that MLBFT with cross-border requests performs better than single-leader solution for larger number of clients. As the proportion of cross-border requests increases, we observe that the curve becomes closer to the curve of single-leader approach. These curves and there R^2 can be estimated by following logarithmic equations of order three:

single-leader

$$y = 1083.97 + 483.30\ ln(x) + 1505.38\ ln^2(x) - 190.99\ ln^3(x)$$
$$R^2 = 0.997$$

ML-50

$$y = 893.57 + 763.34\ ln(x) + 1621.16\ ln^2(x) - 193.71\ ln^3(x)$$
$$R^2 = 0.996$$

ML-15

$$y = 893.4 - 425.82\ ln(x) + 2415.95\ ln^2(x) - 249.5\ ln^3(x)$$
$$R^2 = 0.997$$

ML-1

$$y = 899.16 + 356.26\ ln(x) + 1569.71\ ln^2(x) - 13.86\ ln^3(x)$$
$$R^2 = 0.998$$

MLBFT

$$y = 978.35 + 97.27\ ln(x) + 1871.35\ ln^2(x) - 11.53\ ln^3(x)$$
$$R^2 = 0.998$$

Comparison of throughput of cross-border requests and response time is shown in Figure 4.12. It is observed that throughput of single-leader is saturated

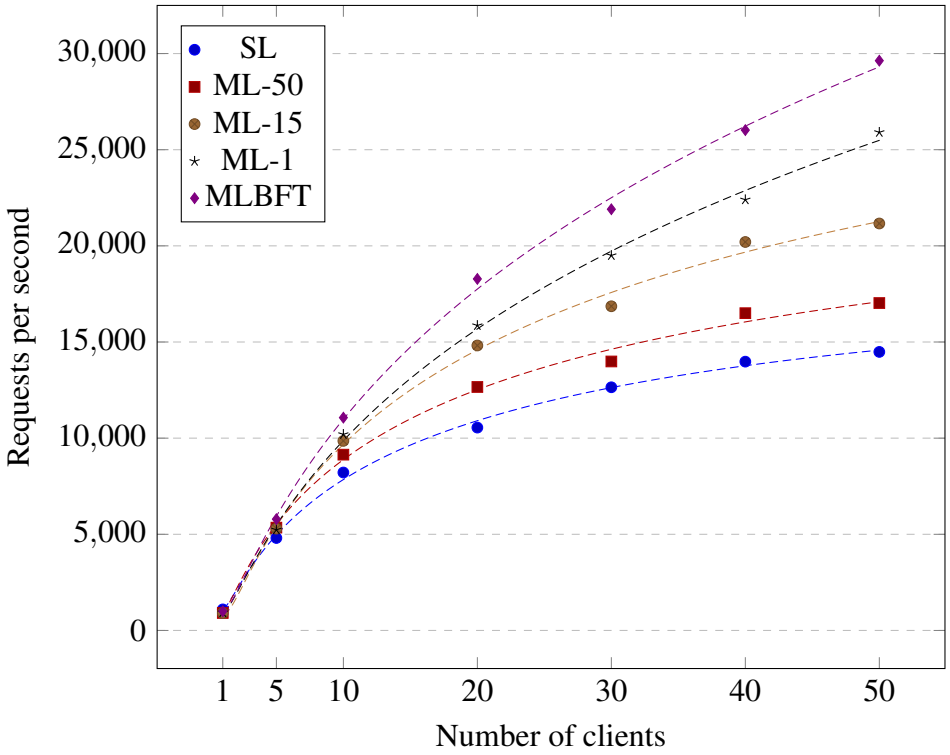

Figure 4.11: Throughput with cross border

at 20,000 requests per second while the response times increases with the increase in number of clients. On the other hand, when the proportion of cross-border requests is low (0% and 1%), the response time of MLBFT approach is much lower than for the single-leader approach. The response time increases with an increase in the proportion of cross-border requests to 15% or 50%. All approaches are bounded by an upper limit of the throughput. The response times becomes a vertical line when that upper limit of throughput is reached by both approaches. The curves can be estimated by following equations:

single-leader

$$y = \frac{1}{2.376 - 1.37 \times 10^{-04}\, x + 1.62 \times 10^{-09}\, x^2}$$

$$R^2 = 0.904$$

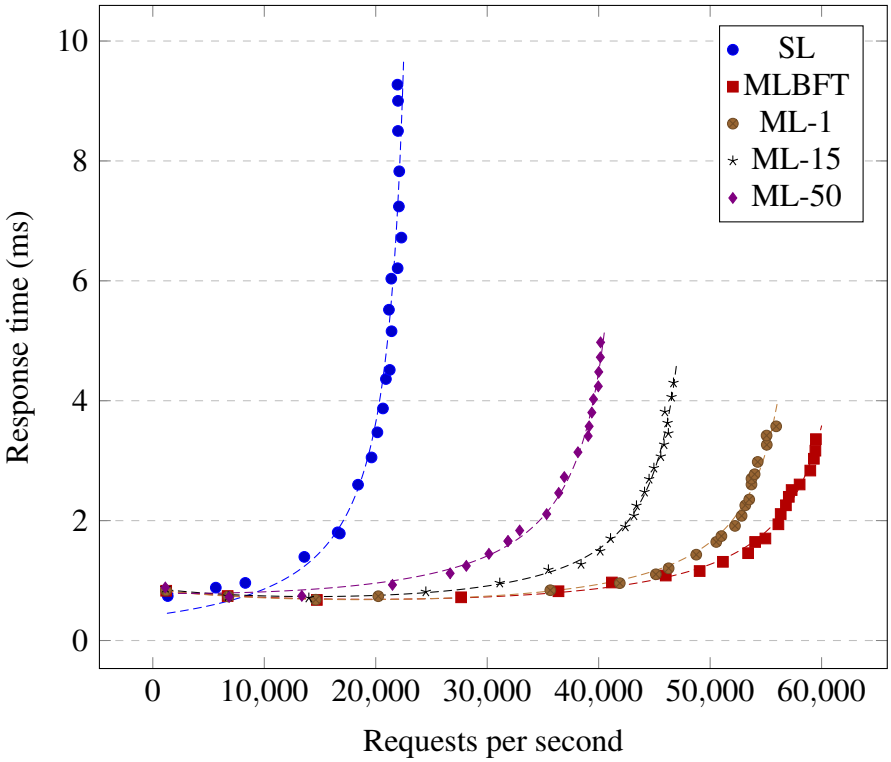

Figure 4.12: Throughput versus response time

ML-50

$$y = \frac{1}{1.182 + 2.78 \times 10^{-05}\, x - 7.146 \times 10^{-10}\, x^2}$$

$$R^2 = 0.993$$

ML-15

$$y = \frac{1}{1.176 + 3.16 \times 10^{-05}\, x - 8.57 \times 10^{-10}\, x^2}$$

$$R^2 = 0.99$$

ML-1

$$y = \frac{1}{1.14 + 3.07 \times 10^{-05}\, x - 1.07 \times 10^{-09}\, x^2}$$
$$R^2 = 0.982$$

MLBFT

$$y = \frac{1}{1.27 + 3.85 \times 10^{-06}\, x - 7.489 \times 10^{-10}\, x^2}$$
$$R^2 = 0.99$$

4.3.3.5 Deadlocks

Deadlocks cannot be avoided when there are large number of cross-border requests. Figure 4.13 shows the increase in the number of deadlocks when there is an increase in cross-border requests. The number of deadlocks increase exponentially when proportion of the cross-border requests is less than 50%. The number of deadlocks increase almost linearly when more than 50% cross-border requests are encountered. Eventually these deadlocks decrease the throughput of the MLBFT which can be observed in Section 4.3.3.4.

The increase in deadlocks can be estimated by the following equation for the curve in Figure 4.13:

$$y = 103.13 + 18.14\, x^2 - 8.27 \times 10^{-02}\, x^3$$
$$R^2 = 0.999$$

4.3.3.6 Memory usage

MLBFT uses more memory than traditional approaches. This overhead is because of prediction, deadlock detection, state partitioning, and an increase in number of threads. This overhead also depends on the configuration and implementation level details. In practice memory overhead should be fixed when there are a constant number of clients. Unfortunately, it is difficult to measure memory usage due to Java's memory management policy. However, we present a rough comparison of memory usage between single-leader and MLBFT in Figure 4.14. This figure shows that memory usage is increased when we run MLBFT and single-leader approach with the same system configuration. It is possible to improve the implementation of the Key-Value store so that it uses less memory

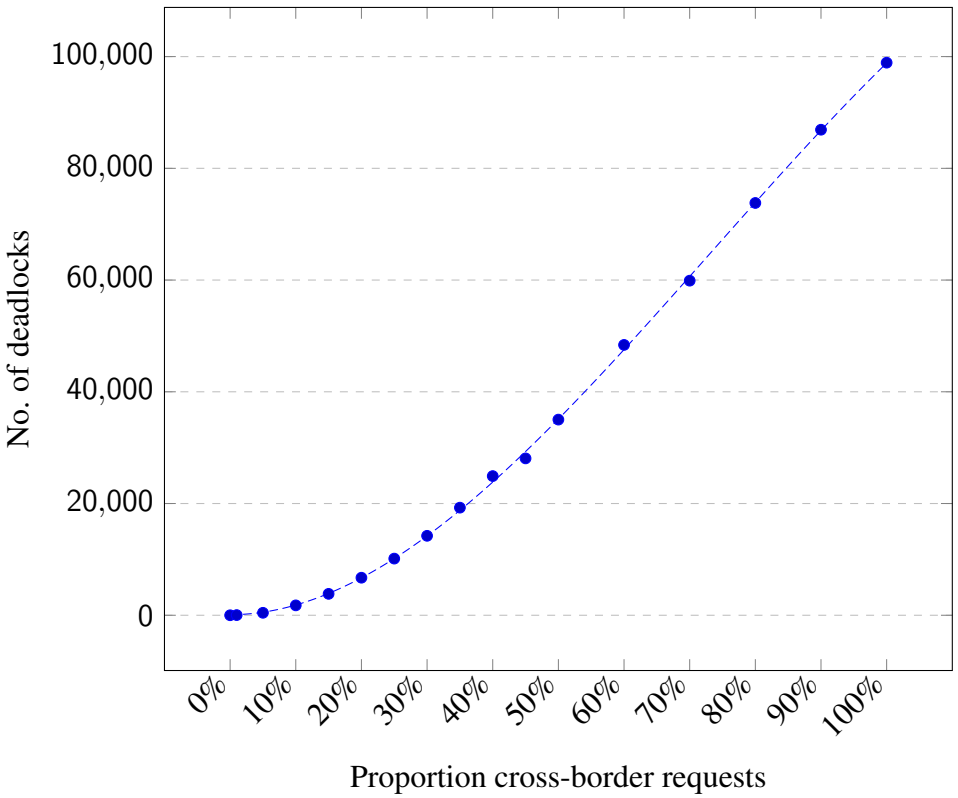

Figure 4.13: Deadlocks versus cross-border requests

than shown in the figure. For example, instead of using Java's `HashMap<K, V>`, an application can use Trove's[1] HashMap which uses less memory. Furthermore, the memory overhead can be reduced by implementing the components of MLBFT as described in Section 3.2.12.2.

[1] Trove, `http://trove.starlight-systems.com` (Last accessed: 2015-06-09)

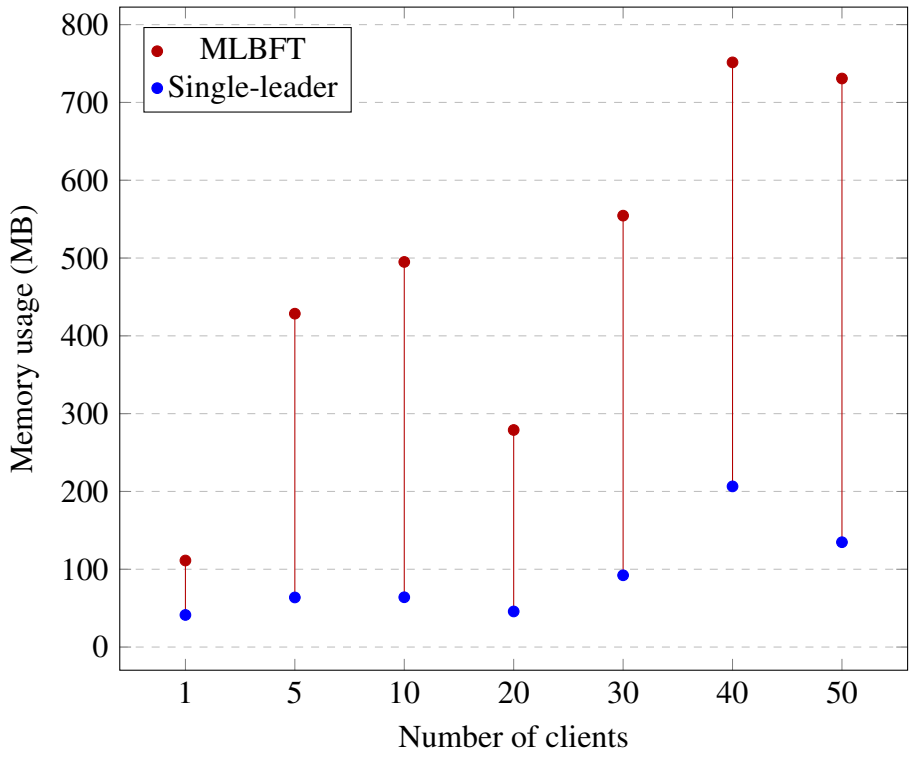

Figure 4.14: Memory usage

4.3.3.7 Multicore CPU

Figure 4.15 measures throughput of MLBFT and the single-leader solution on a multicore CPU. As expected both solutions performs best when the number of BFT instances is equal to number of cores of the CPU. It is shown that single-leader performs best on a single core machine because there is only one BFT instance running on the replicas. The single-leader approach even outperforms MLBFT on a single-core configuration. On the other hand, MLBFT performs best at 4 cores CPU because in this setting we use 4 BFT instances. The throughput is decreased a little if number of cores are more than BFT instances. This happens because the threads are going to be schedule on virtual cores of the CPU after all physical cores are busy. Furthermore, only 4 cores are busy in handling the workload (see Section 4.3.3.3). There is no more work that can be scheduled on rest of the 4 cores and we do not see any improvement in throughput when scheduling the replica threads on all 8 cores. This is why MLBFT utilizes only 50% of the CPU. On the other hand, if number of cores are less than number of BFT instances then throughput is decreased a lot. This is because there is at least one thread which is waiting to be scheduled on CPU. The results from the figure match the predicted model of MLBFT analyzed in Section 4.1. MLBFT performs worse on single core than the single-leader approach because of the overhead mentioned before. For 2 cores we see an improvement of 150% which matches the theoretical speedup $S(2) = 1.6$. Similarly, for 3 cores ($S(3) = 2$) we see an improvement of 180% as expected.

4.3.3.8 Read requests

Figure 4.16 shows the relationship between throughput and proportion of read-only requests. It is observed that throughput is not affected much by a read-heavy or write-heavy clients. However, throughput is increased a little if all of the requests perform read operations on the state objects in both approaches. This happens because state objects must be created (if they don't exists) in a write request. Creating state objects adds a little overhead in overall throughput. In all cases MLBFT (with no cross-border requests) outperforms single-leader approach up to a factor of two.

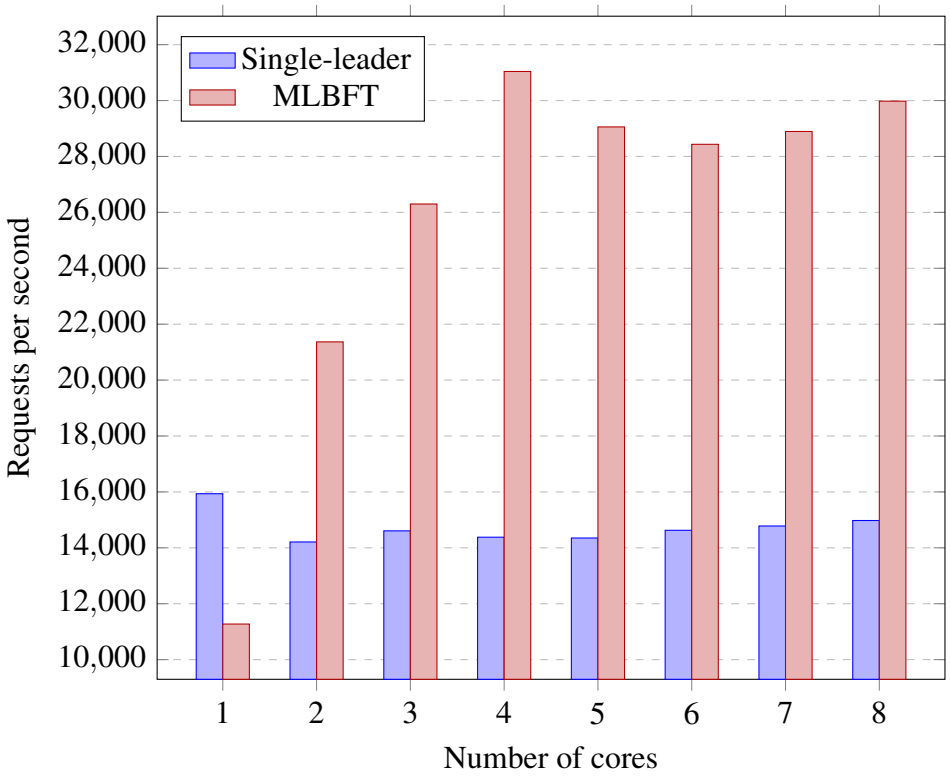

Figure 4.15: Throughput on multicore CPU

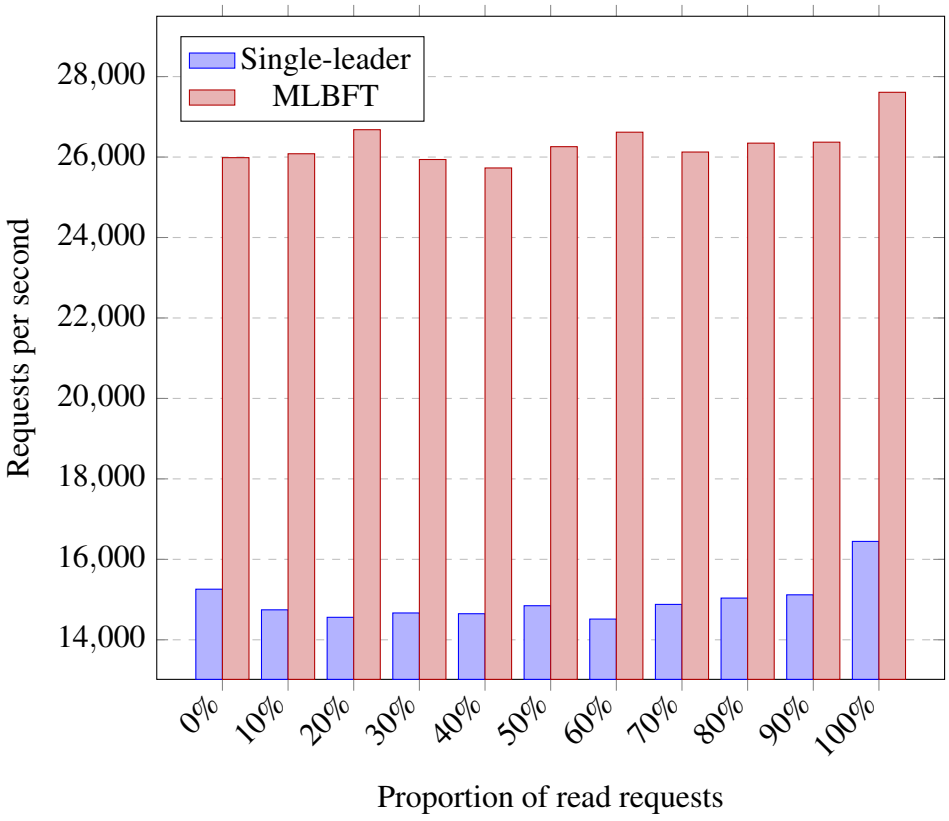

Figure 4.16: Throughput of read requests

Chapter 5

Conclusions

In this thesis, we have presented MLBFT, a multi-leader approach to improve the throughput of BFT systems. This chapter summarizes the findings presented in previous chapters and proposes a number of improvements and extensions that may be of interest in order to continue this work.

5.1 Conclusion

High resource consumption and overhead is one of the most important reasons why BFT systems have not yet been adopted by industry [3, 4]. So far research has improved the performance by introducing concurrency in the execution stage. However, the agreement stage remains an open area to be explored. This thesis introduces some improvements to the agreement stage and shows that throughput can be improved up to a factor of two in ideal cases.

5.1.1 Goals

We have presented the following approach to introduce concurrency in the agreement stage. Following this description of our by basic approach, we summarize the main research questions in this thesis.

Basic Approach

Traditionally, the single-leader approach assumes that all requests are dependent on each other and must provide a total-order of requests to ensure consistency. In practice, not all the requests are dependent on each other and only dependent requests should be ordered (partial-order). MLBFT exploits this idea by using application knowledge to split the state objects into partitions. Each partition is managed by an independent BFT agreement

instance establishing a partial-order on dependent requests. Furthermore, MLBFT provides a way to handle requests which need to access multiple partitions.

Research Questions

Given the basic idea of splitting state objects into partitions, this thesis has followed two main research questions:

- Can the suggested approach enable a Byzantine Fault Tolerant system to deliver **better throughput**?
- Can the suggested approach enable a Byzantine Fault Tolerant system to handle **simple** as well as **cross-border** requests?

With regard to first research question, we have shown (in Chapter 4) that MLBFT can achieve better throughput as compared to the common single-leader approach. In ideal cases it is possible to increase the throughput up to a factor of two by implementing this approach. However, it is observed that multicore machines are required to gain higher performance. Furthermore, evaluation shows that this approach can produce increased overhead on a single-core machine, hence decreasing overall throughput.

Regarding the second research question, evaluation shows that MLBFT can handle both simple and cross-border requests. In the case of simple requests we have seen improved throughput. However, to achieve better throughput in the case of cross-border requests, predictor should be configured to minimize the proportion of cross-border requests. Although, results have shown that MLBFT produces slightly better throughput than single-leader approach even with 90% of cross-border requests (accessing two partitions), it is recommended to keep the proportion of cross-border requests as minimal (ideally 0%) as possible. Furthermore, partitioning the state objects also plays an important role in improving the performance of MLBFT.

5.1.2 Insights

We have observed from our evaluation that the single-leader approach cannot fully utilize resources like CPU and network bandwidth. The single-leader approach can only utilize single core on a multi-core CPU. Furthermore, the single-leader approach nearly utilizes 60% of the network bandwidth (for 8 KB request) on a Gigabit Ethernet. MLBFT, on the other hand, can utilize more resources than the single-leader approach. MLBFT can take advantage of multiple cores (4 cores in our configuration) on a multi-core CPU. Furthermore, MLBFT can utilize 100% of the network bandwidth (for large request sizes) on a Gigabit

Ethernet. It is also observed that MLBFT approach is bounded by both CPU and network bandwidth. In our experimental setup, the CPU becomes a bottleneck by executing cryptographic functions (to verify signatures and to calculate request digests) and bandwidth becomes a bottleneck for large requests (larger than 2 KB). The CPU and network usage give some insights about the limitations of multi-leader approach.

5.1.3 Sustainable Development

The traditional BFT systems lack the performance which is desired by modern software industry. The single-leader approach is not able to provide the desired throughput because the single-leader approach under-utilize the available resources. We have shown that MLBFT can achieve over 100% higher throughput (in ideal cases) by properly utilizing the available resources. In fact, MLBFT outperforms single-leader approach in most scenarios when both approaches run on the same infrastructure (multi-core machines and network). Although, MLBFT utilizes more resources than single-leader approach, it can sustain the life of existing BFT services without incurring any additional costs. Furthermore, MLBFT can execute client requests in less time than traditional approaches. This performance improvement not only provides a better user experience, but it also saves time and energy. These improvements bring both **economic** and **environmental** benefits to our society and support sustainable development for the future.

5.1.4 Challenges

We encountered a few challenges in the context of implementing the proposed MLBFT approach. This section highlights two of the most important challenges we faced in this research.

5.1.4.1 Deadlocks

As we have already established that we do not rely on atomic multicast in Section 3.2.8, hence deadlocks cannot be avoided. Traditional deadlock prevention schemes cannot be applied because of their limitations. For example, a circular wait cannot be avoided because requests can be received by replicas in any order. Another approach is not to schedule a cross-border request in a queue which might cause a deadlock. In this case we have to scan all ordered requests in the relevant partition queues and compare it with the incoming request. This is not a good approach because it will be extremely expensive to compare each request with already ordered requests. Additionally, we want to implement a deadlock

resolution scheme which is free of rollbacks. For instance, while executing a request if we encounter that a lock is required which is already acquired by another thread, then we might have to rollback a partial execution. Furthermore, we can avoid rollbacks by executing the request on a copy of data. This is not an ideal solution because sometimes it is impossible (or impractical) to copy state objects (e.g., large state objects or if the state objects are located remotely) and then operate on it. Given these constraints it is a challenge to come up with a solution which can resolve deadlocks deterministically.

5.1.4.2 Mispredictions

In Section 3.2.7 we established that a request can be mispredicted by prediction layer. We have also established that the system can encounter a deadlock in the case of two or more mispredictions when execution has already been started (see Section 3.2.8.2). In this scenario it is a challenge to resolve a deadlock. To solve this case all partially executed requests (in a deadlock) must be halted immediately. An algorithm must be designed that produces a sequence of requests after analyzing dependencies between these requests such that sequential execution of these requests does not violate linearizability. Solving this problem deterministically can be a challenge given the constraints mentioned in the previous section.

5.2 Future work

This section mentions some of the work which might be of interest in order to continue and improve this research.

5.2.1 What has been left undone?

We have provided a solution to handle mispredictions in Section 3.2.7. Unfortunately, due to time constraints this solution was not implemented and evaluated. Therefore, it is highly recommended that the proposed solution be implemented and its performance evaluated. It can be guessed that mispredictions will decrease the throughput. This decrease will happen because the request must be ordered in all the new partitions before the request can continue its execution. However, it would be interesting to observe whether the performance degrades slightly or quite a lot.

Furthermore, the prediction layer can be designed to keep a record of all the previous mispredictions and then learn from them to improve its rate of correct predictions. It would be interesting to make the prediction layer

dynamic rather than statically predicting requests. However, care should be taken when implementing this functionality as the prediction function must produce a deterministic response on all machines (including clients see Section 3.1.1) which can be a challenge.

5.2.2 Next obvious things to be done

There is a lot of room for improvements and future development in this research. We propose some obvious improvements that can be applied to this approach.

5.2.2.1 More Case Studies

We explored key-value store as a case study, but there is a need to evaluate other case studies. Evaluating other case studies will give more insights into this approach which can then be used to improve the protocol. Exploring other case studies will also give a better idea of possible applications of this approach in the software industry.

5.2.2.2 Deadlock Resolution

We spend quite some time developing the deadlock resolution algorithm. Despite this effort, there is still room for improvements to develop a faster deterministic deadlock resolution algorithm. Moreover, this research lacks an implementation of deadlock after execution (see Section 3.2.8.2) algorithm which can be researched further.

5.2.2.3 Fault Handling

Another improvement would be to evaluate the throughput in case of a faulty replica. Our guess is that the throughput of this approach would still be better than its single-leader counterpart. This is because leadership of only one BFT agreement instance would be transferred to next replica. All other BFT agreement instances will continue to work normally.

5.2.2.4 Batching and reply Digests

One of the common ways to improve performance is to introduce batching. As a result multiple requests can be batched together and passed on to the agreement stage for ordering. This enables the service to run only one consensus round per batch, hence improving performance. It is recommended to enable batching and then evaluate it to see how it affects performance. Furthermore, another optimization is to return a digest of replies [9] instead of a full result. The digest

enables the client to verify the result while reducing network bandwidth for large replies.

5.2.2.5 Thread pinning

In Section 4.3.3.7 we evaluate the performance of MLBFT on multicore CPUs. We observed that maximum performance is achieved when the number of cores is equal to number of BFT ordering instances. However, in that evaluation we did not pin any of the threads to specific CPU cores, hence thread scheduling overhead was not affected. Another approach would be to pin ordering and worker threads to a specific core of the CPU. For example, the ordering thread for BFT-0 can be pinned to the first core of CPU. This will help reduce the scheduling cost as this thread will always be scheduled only on a specified CPU core (the first core). It would be interesting to observe whether this approach can help achieve improved performance or not.

5.3 Required Reflections

Our proposed approach targets network-based software services that require strong availability and reliability. The research presented in this thesis serves as a vital effort towards achieving improved performance in the aforementioned services. For example, implementing this research in banking and e-commerce services, where unavailability can cause economic losses, would not only increase the quality of service, but it will also improve the performance. Additionally, this research also targets applications whose reliability is crucial for other services (e.g., Key-Value stores or Network File Systems). Taking advantage of this research leads to lower response time for these services, hence improved client satisfaction. These two applications reflects positive **social** and **economic** effects of this research.

Furthermore, BFT systems are an excellent candidate for services where a single fault can lead to life-threatening disasters (e.g., Railway dispatch and control systems). This research can improve the availability and reliability of such services if implemented correctly, where software vendors are hesitant to use BFT systems because of their poor performance. These applications of BFT services reflects desirable **ethical** effects of this thesis. Furthermore, for ethical reasons, this research does not disclose any kind of confidential information of the lab at Technische Universität Braunschweig (where the research was conducted).

Bibliography

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. doi: 10.1145/357172.357176. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[2] A. Shoker and J.-P. Bahsoun, "BFT for three decades, yet not enough!" 2009. [Online]. Available: http://haslab.uminho.pt/ashoker/files/shokerbft3decadestr.pdf

[3] C. Ho, "Reducing costs of Byzantine fault tolerant distributed applications," 2011-08-31. [Online]. Available: http://ecommons.library.cornell.edu/handle/1813/30754

[4] P. Kuznetsov and R. Rodrigues, "BFTW3: Why? When? Where? workshop on the theory and practice of byzantine fault tolerance," vol. 40, no. 4, pp. 82–86, 2010-01. doi: 10.1145/1711475.1711494. [Online]. Available: http://doi.acm.org/10.1145/1711475.1711494

[5] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: optimistic deterministic multithreading," in *Proceedings of the Sixth international conference on Hot topics in system dependability*. USENIX Association, 2010, pp. 1–8. [Online]. Available: http://static.usenix.org/events/hotdep/tech/full_papers/Kapitza.pdf

[6] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *2004 International Conference on Dependable Systems and Networks*, Jun. 2004. doi: 10.1109/DSN.2004.1311928 pp. 575–584.

[7] T. Distler and R. Kapitza, "Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011. doi: 10.1145/1966445.1966455. ISBN 978-1-4503-0634-8 pp. 91–106. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966455

[8] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, and others, "All about eve: Execute-verify replication for multi-core servers." in *OSDI*, vol. 12, 2012, pp. 237–250. [Online]. Available: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-190.pdf

[9] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999. ISBN 1-880446-39-1 pp. 173–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296824

[10] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990. doi: 10.1145/98163.98167. [Online]. Available: http://doi.acm.org/10.1145/98163.98167

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. doi: 10.1145/359545.359563. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[12] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," 2003. [Online]. Available: http://infoscience.epfl.ch/record/49913

[13] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003. doi: 10.1145/945445.945470. ISBN 1-58113-757-5 pp. 253–267. [Online]. Available: http://doi.acm.org/10.1145/945445.945470

[14] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002. doi: 10.1145/571637.571640. [Online]. Available: http://doi.acm.org/10.1145/571637.571640

[15] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, pp. 824–840, Oct. 1985. doi: 10.1145/4221.214134. [Online]. Available: http://doi.acm.org/10.1145/4221.214134

[16] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient

byzantine fault tolerance," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012. doi: 10.1145/2168836.2168866. ISBN 978-1-4503-1223-3 pp. 295–308. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168866

[17] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis, and others, "Zeno: Eventually consistent byzantine-fault tolerance." in *NSDI*, vol. 9, 2009, pp. 169–184. [Online]. Available: https://www.usenix.org/event/nsdi09/tech/full_papers/singh/singh_html/

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, Jan. 2010. doi: 10.1145/1658357.1658358. [Online]. Available: http://doi.acm.org/10.1145/1658357.1658358

[19] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMaRt," *Department of Computer Science, University of Lisbon, Tech. Rep*, 2013. [Online]. Available: http://www.di.fc.ul.pt/~bessani/publications/dsn14-bftsmart.pdf

[20] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 277–290. [Online]. Available: http://dl.acm.org/citation.cfm?id=1629602

[21] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 153–168. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558988

[22] J. B. Postel, "User datagram protocol," Internet Engineering Task Force, RFC 768, Aug. 1980, published: Internet RFC 768. [Online]. Available: http://www.rfc-editor.org/rfc/rfc768.txt

[23] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985. doi: 10.1145/3149.214121. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[24] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," *arXiv preprint arXiv:1311.6183*, 2013. [Online]. Available: http://arxiv.org/abs/1311.6183

[25] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," *arXiv preprint arXiv:1404.6721*, 2014. [Online]. Available: http://arxiv.org/abs/1404.6721

[26] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6263916

[27] G. Santos Veronese, M. Correia, A. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *28th IEEE International Symposium on Reliable Distributed Systems, 2009. SRDS '09*, Sep. 2009. doi: 10.1109/SRDS.2009.36 pp. 135–144.

[28] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," in *OSDI*, vol. 8, 2008, pp. 369–384. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855767

[29] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. doi: 10.1145/279227.279229. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[30] J. Behl, T. Distler, and R. Kapitza, "Scalable BFT for multi-cores: Actor-based decomposition and consensus-oriented parallelization," in *10th Workshop on Hot Topics in System Dependability (HotDep 14)*. Broomfield, CO: USENIX Association, Oct. 2014. [Online]. Available: http://blogs.usenix.org/conference/hotdep14/workshop-program/presentation/behl

[31] R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001. doi: 10.1145/502034.502037. ISBN 1-58113-389-8 pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/502034.502037

[32] J. B. Postel, "Transmission control protocol," Internet Engineering Task Force, RFC 793, Sep. 1981, published: Internet RFC 793. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[33] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967. doi: 10.1145/1465482.1465560 pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[34] T. Distler, "Resource-efficient fault and intrusion tolerance," Ph.D. dissertation, Friedrich-Alexander-Universität Erlangen-Nuremberg, 2014.

[35] D. Eastlake, 3rd and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," United States, 2001.

[36] R. Rivest, "The MD5 Message-Digest Algorithm," United States, 1992.

Appendix A

Java Source Code

```
package refit.agreement.multileader;

import refit.agreement.primary.PrimaryProtocol;
import refit.config.REFITConfig;

/**
 * A class that implements primary protocol. This class is used
 * to select the primary replica given a view id and BFT instance id.
 * Initially all BFT instances will start with view id = 0. This
 * implementation will select replica-0 as a leader of BFT-0,
 * replica-1 as a leader of BFT-1 and so on.
 *
 * BFT will move to the next view id if the primary replica fails.
 * This will select the next replica as the leader of BFT.
 *
 * Created by Zeeshan
 */
public class MLPrimaryProtocol implements PrimaryProtocol {
    private final int   viewID;
    private final short bftInstanceId;

    public MLPrimaryProtocol(int viewID, short bftInstanceId) {
        this.viewID        = viewID;
        this.bftInstanceId = bftInstanceId;
    }

    @Override
    public short getPrimary() {
        return (short) (
                    (viewID + bftInstanceId) % REFITConfig.TOTAL_NR_OF_REPLICAS
        );
    }
}
```

Code A.1: Primary selection protocol

```java
package refit.replica.multileader;

import java.util.Set;

/**
 * Predictor interface.
 * All predictor implementations must implement this interface as the base.
 * Predictor must provide the implement of given methods using the application
 * specific knowledge. For example, an implementation for key-value store
 * can look at the given request, parse it and find the key and then use that
 * key to return the set of partions the request will access.
 *
 * Created by Zeeshan
 */
public interface Predictor {
    /**
    * parse the given request using application knowledge and return set
    * of objects
    */
    public Set<Object>  predictObjects    ( byte[]      request    );
    /**
    * for given set of objects find the set of partitions containing
    * given objects
    */
    public Set<Short>   getPartitions     ( Set<Object> objects    );
    /**
    * for given set of partions find a BFT instance that will actually
    * execute the request through a deterministic rule
    */
    public short        execBFTInstance   ( Set<Short>  partitions );
    /** a mapping function which returns the partition id of given object **/
    public short        objectToPartition ( Object      object     );
}
```

Code A.2: Prefictor interface

```java
package refit.replica.execution.multileader;

/**
 * An enumeration to distinguish different request types. Each ordered request
 * in queue will have this property to identify whether the request is a
 * simple-request or a cross-border request. If it is a cross-border request
 * then identify whetherit is cross-border exec or a cross-border-sync
 * request.
 *
 * Created by Zeeshan
 */
public enum MLOperationType {
    /** simple request **/
    EXECUTE,

    /** cross-border request that will be executed **/
    CROSS_BORDER_EXECUTE,

    /**
     * cross-border request that will be synchronised with corresponding
     * cross-border-execute request
     */
    CROSS_BORDER_SYNC;

    public static MLOperationType fromOrdinal(int ordinal) {
        return values()[ordinal];
    }
}
```

Code A.3: MLBFT request types

```java
package refit.replica.multileader.concurrent;

/**
 * A wait/notify signal class to capture missed signals. If
 * a thread goes into waiting state after it has received
 * a notify signal then the notify signal will be missed.
 * Java monitors do not keep method calls to wait/notify.
 * This class avoid objects to keep waiting forever because
 * notify signal was missed.
 *
 * Created by Zeeshan
 */
public class WaitNotifySignal {
    private Object  monitor;
    private boolean signaled;

    public WaitNotifySignal() {
        this.monitor  = new Object();
        this.signaled = false;
    }

    public void doWait() {
        synchronized (this.monitor) {
            while (!this.signaled) {
                try {
                    this.monitor.wait();
                } catch (InterruptedException e) {}
            }
            this.signaled = false;
        }
    }

    public void doNotify() {
        synchronized (this.monitor) {
            this.signaled = true;
            this.monitor.notifyAll();
        }
    }
}
```

Code A.4: Wait/Notify signal object for partition queue

```
package refit.replica.execution.multileader;

import refit.config.REFITConfig;
import refit.config.REFITLogger;
import refit.message.REFITRequest;
import refit.replica.execution.REFITExecutor;
import refit.scheduler.REFITScheduling;

import java.util.HashSet;
import java.util.Set;

/**
 * A worker thread that will pick the request at head ordered in the queue and
 * execute it. It will distinguish between different types of request.
 * For example, simple-request does not need any synchronization so it will
 * just execute normally. In case of cross-border request it must synchronise
 * all the relevant partitions and worker thread operating on them.
 * After synchronization the request will be executed and worker will move on
 * to next request. If there is a deadlock then worker thread will delegate
 * the responsibility to deadlock-resolver and wait until the deadlock has
 * been resolved.
 *
 * Created by Zeeshan
 */
public class MLExecWorker extends Thread {
    private final short            bftInstanceId;
    private final MLOperationQueue queue;
    private final REFITExecutor    executor;

    public MLExecWorker(short bftInstanceId, MLOperationQueue queue,
    REFITExecutor executor) {
        this.bftInstanceId = bftInstanceId;
        this.queue         = queue;
        this.executor      = executor;
        this.setName("MLExec-" + this.bftInstanceId);
    }

    private void executeRequest(MLOperation request) {
        if (this.queue.lockIfAtHead(this.bftInstanceId,
    request.getRequest())) {
            this.executor.processRequest(request.getRequest(),
    request.getAgreementSeqNr());
            this.queue.unlockAndRemoveHead(this.bftInstanceId,
    request.getRequest());
        }
        this.queue.notifyOtherPartitions(this.bftInstanceId);
    }
```

Code A.5: Execution stage worker thread

```
private void executeCrossBorderRequest(MLOperation request) {
    try {
        boolean execute = waitForOtherPartitions(request);
        if (!execute) {
            return;
        }

        this.executor.processRequest(request.getRequest(),
request.getAgreementSeqNr());
        crossBorderRequestFinished(request);
        this.queue.notifyOtherPartitions(bftInstanceId);
    } catch (Exception e) {
        REFITLogger.logError(this, "Error executing cross-border request:
" + e.getMessage());
    }

}

private void executeCrossBorderSync(MLOperation request) {
    this.queue.notifyOtherPartitions(bftInstanceId);
    queue.waitUntilAtHead(bftInstanceId, request.getRequest());
}

private boolean waitForOtherPartitions(MLOperation request) {
    Set<Short> partitions = request.getPartitions();
    while (!requestAtHead(partitions, request.getRequest())) {
        boolean resolved = this.queue.detectAndResolveDeadlock();
        if (resolved) {
            this.queue.notifyOtherPartitions(bftInstanceId);
        } else {
            this.queue.waitUntilSignaled(bftInstanceId);
        }

        /* after deadlock resolution request might have been re-ordered */
        MLOperation operation = queue.peekFirst(bftInstanceId);
        if (!operation.getRequest().equals(request.getRequest())) return
false;
    }
    return true;
}

private void crossBorderRequestFinished(MLOperation request) {
    Set<Short> partitions = request.getPartitions();
    for (short partition : partitions) {
        queue.unlockAndRemoveHead(partition, request.getRequest());
    }
}
```

Code A.6: Execution stage worker thread (*cont'd*)

```
    private boolean requestAtHead(Set<Short> partitions, REFITRequest
    request) {
        Set<Short> lockedPartitions = new HashSet<>();
        for (Short partition : partitions) {
            boolean locked = queue.lockIfAtHead(partition, request);
            if (locked) {
                lockedPartitions.add(partition);
            } else {
                unlockPartitons(lockedPartitions);
                return false;
            }
        }
        return true;
    }

    private void unlockPartitons(Set<Short> partitions) {
        for (Short partition : partitions) {
            queue.unlockPartition(partition);
        }
    }

    @Override
    public void run() {
        while(true) {
            try {
                MLOperation request = queue.peekWait(bftInstanceId);

                switch (request.getType()) {
                    case EXECUTE:
                        executeRequest(request);
                        break;
                    case CROSS_BORDER_EXECUTE:
                        executeCrossBorderRequest(request);
                        break;
                    case CROSS_BORDER_SYNC:
                        executeCrossBorderSync(request);
                        break;
                    default:
                        throw new RuntimeException("Unknown request type: " +
    request.getType());
                }
            } catch (Exception e) {
                REFITLogger.logError(this, "Error executing request: " +
    e.getMessage());
            }
        }
    }
}
```

Code A.7: Execution stage worker thread (*cont'd*)

TRITA-ICT-EX-2015:137