Next generation state-machine replication protocols for data centers

Mohamad Jaafar Nehme

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE la Communauté UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Mohamad Jaafar NEHME

Thèse dirigée par **Vivien QUEMA**
et codirigée par **Kamal BEYDOUN**

préparée au sein **Laboratoire d'Informatique de Grenoble (LIG)**
et de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (EDMSTII)**

Next generation state-machine replication protocols for Data centers

Protocoles de réplication de machines á états de prochaine génération pour les centres de données

Thèse soutenue publiquement le **05.12.2017**,
devant le jury composé de :

Mr. Didier DONSEZ
Professeur, Grenoble Alps University, Président
Mr. Gael THOMAS
Professeur, Telecom SudParis, Rapporteur
Mr. Laurent RÉVEILLÈRE
Professeur, Université de Bordeaux, Rapporteur
Ms. Sonia BEN-MOKHTAR
CHARGE DE RECHERCHE, CNRS, Examinatrice
Mr. Vivien QUÉMA
Professeur, Grenoble INP, Directeur de thèse
Mr. Kamal BEYDOUN
Maître de conférences, Université Libanaise, Co-Directeur de thèse

Contents

List of Figures

List of Tables

Abstract

Many uniform total order broadcast protocols have been designed in the last 30 years. They can be classified into two categories: those targeting low latency, and those targeting high throughput. Latency measures the time required to complete a single message broadcast without contention, whereas throughput measures the number of broadcasts that the processes can complete per time unit when there is contention. All the protocols that have been designed so far make the assumption that the underlying network is not shared by other applications running. This is a major concern provided that in modern data centers (aka Clouds), the networking infrastructure is shared by several applications. The consequence is that, in such environments, uniform total order broadcast protocols exhibit unstable behaviors.

In this thesis, I present *MDC-cast* a new protocol for total order broadcasts that is optimized for multi-data center environments. *MDC-cast* combines the benefits of IP-multicast in cluster environments and TCP/IP unicast to get a hybrid algorithm that achieves very good performance in modern datacenters.

Keywords. State Machine, Datacenter, Cloud Computing, Replication, Total Order Broadcast.

Résumé

De nombreux protocoles protocoles de diffusion avec ordre total ont été conçus au cours des 30 dernières années. Ils peuvent être classés en deux catégories: ceux qui visent une faible latence, et ceux qui visent un haut débit. La latence mesure le temps nécessaire pour diffuser un seul message sans contention, alors que le débit mesure le nombre de diffusions que les processus peuvent réaliser par unité de temps (quand il y a contention). Tous les protocoles qui ont été conÃğus font l'hypothèse que le réseau n'est pas partagé par d'autres applications en cours d'exécution. Cette hypothèse n'est pas valide dans les centres de données modernes (appelÃľs Clouds), au sein desquels l'infrastructure réseau est partagée par plusieurs applications. La conséquence est que, dans de tels environnements, les protocoles de diffusion avec ordre total présentent des comportements instables.

Dans cette thèse, j'ai conçu et mis en œuvre un nouveau protocole pour la diffusion avec ordre total, appelé *MDC-cast*. Ce protocole optimise les performances lorsqu'il est exécuté dans des centre de données modernes (ou des groupes de centres de données). *MDC-cast* combine les avantages de la multidiffusion IP quand c'est possible de l'utiliser et l'efficacité des communications unicast TCP/IP quand nécessaire. Le protocole résultant est ainsi hybride et permet d'obtenir de très bonnes performances dans les environnements d'exécution modernes.

Mots-clés. Centres de données, Réplication, Diffusion avec ordre total.

Acknowledgements

I would first like to express my appreciation to the persons whom this thesis might not have been written without. In particular, I thank my thesis advisor Vivien QUEMA, for directing and guiding my work. I warmly thank him for the great effort he provided and for his constructive comments. His tremendous expertise and his constructive criticism helped me to complete this work. Also, I thank Kamal BEYDOUN, my second supervisor, who has given me the honor to be kindly the co-director of my thesis. I thank him for the freedom he has left me during my thesis, for his availability and his advices. I would also like to thank Nicolas PALIX who helped me during the first year of my PhD.

I would also like to thank Gael THOMAS and Laurent REVEILLERE who accepted to report on my work. Many thanks Sonia BEN-MOKHTAR and Didier DONSEZ who accepted to be part of my thesis committee. It is a great honor for me.

I must express my very profound gratitude to the funding received from the Islamic Center Association for Guidance and Higher Education.

As well, I would like to thank warmly my Parents, for their encouragement. They followed me since I was born and helped me carry out my projects until their completion. They have given me everything and I owe them everything. In addition, I want to thank my wife, Jinan, who accompanied me during the past years and

supported me with all her love in difficult times, the greatest gift of my thesis and my life.

Many aspects of this work are indebted to Abbass ZEINEDDINE because of the frequent and deep discussions he made with me to help me crystallize my thoughts and bypass some obstacles. Many thanks for my friends who have spent time in the administration procedures in spite of their many duties and responsibilities, notably Ibrahim SAFIEDDINE and Ali HALLAL.

Finally, I want to thank my family members and my colleagues for their constant enthusiasm.

Dedication

To the person living my heart...

To whom I harbor my sincere love...

To the grace withheld from my eyes...

To whom I sound eager to meet...

I dedicate my humble work

"Do not follow majority; follow the truth ..."

Imam Ali

Introduction

In this chapter, we first present the scientific context of this document, and more specifically, we focus on the performance of totally ordered broadcasts in multi-datacenter environments. We detail our objectives and the contributions of this research work. Finally, we give a brief description of the contents of this document.

Scientific Context The need for high accessibility and fault tolerance drives large enterprises to maintain multiple copies of their databases over several machines (Often called replicas). This mechanism is commonly referred to as State Machine Replication [Sch90] in the context of distributed systems. Coordinating messages in between replicas is not a trivial task. One of the primitives found for solving the issue is ordering messages among replicas. Total Order Broadcast [HT93] (Sometimes called Atomic Broadcast) is a primitive designed for coordinating the communication among data replicas. It asserts that messages are received in the same order from all the nodes. A Uniform Total Order Broadcast protocol ensures the following properties for all messages that are broadcast: (1) Uniform agreement: if a replica delivers a message m, then all correct replicas eventually deliver m; (2) Strong uniform total order: if some replica delivers some message m before message m_0, then a replica delivers m_0 only after it has delivered m.

Research Motivation Computing stepped out the boundaries of a single central processing unit into what is called a distributed system. Thereupon, tackling the performance of a system running over spatially apart servers became a real chal-

lenge especially with the dramatic increase of Internet users. Companies are ultimately aware for their data consistency. Loosing a piece of information or corrupting few bytes even may lead to catastrophic consequences. Achieving data consistency among several replicas requires reliable and totally ordered broadcast of requests or commands among them. This is the role of total order broadcast algorithms to provide this functionality.

Recently, [GLPQ10] declared the optimal throughput of TOB protocols in cluster environments. In the same article, they proposed an algorithm, LCR, and proved that it matches that throughput. From that time, the challenge has changed into finding an algorithm that matches that throughput in cluster environment and achieves better performance regarding some other metrics. For instance, FastCast [BQ13], proposed a solution where they achieve the optimal throughput in cluster environment but have a lower latency. Nowadays, with the distributed systems revolution brought by Clouds, the challenge has changed. Systems are no more limited to the bounds of clusters but rather distributed over the globe. More precisely, replicas are no more confined in the same datacenter but rather are spread over several datacenters distributed across the globe. Existing Total-Order Broadcast protocols have been designed for uniform environments (e.g. Clusters of machines). These protocols fail to achieve good performance in multi-datacenters environments which are characterized by non-uniform network connectivity.

Objectives and Contributions This thesis studies boosting the performance of systems operating total order broadcast protocols in datacenter environments. The thesis presents a novel and scalable Total Order Broadcast primitive *MDC-cast*, that not only achieves the optimal throughput of cluster environments, but also bypasses this throughput when encountering bottlenecks between sites. The performance of *MDC-cast* is assessed under several circumstances and on different testbeds showing significant improvement over other Total Order Broadcast primitives in multi-datacenter environments.

Organization of this document This thesis is organized into 3 chapters:

- **State of the Art**: Chapter 1 describes the state of the art on Total Order Broadcast protocols. We first define the Total Order broadcast notations and state the system model. Thereafter, we mention the related work and focus on the protocols that achieves the optimal throughput defined in [GLPQ10]. Finally, we analyze the performance of existing protocols.

- **MDC-cast**: Chapter 2 presents the *MDC-cast* protocol. In the first place, the chapter includes an overview of *MDC-cast*. Secondly, it includes a description of the system features and sub-protocols.

- **Performance Evaluation**: Chapter 3 presents the implementation of *MDC-cast* and its performance evaluation. Firstly, the experimental setup is described. Then, *MDC-cast* is evaluated under several circumstances and from various performance metrics. Finally, the algorithm is compared to *LCR* and *Ridge*.

At the end of the document, we conclude it and expand on the possible future works.

Chapter 1

State of the art on Total Order Broadcast Protocols

Several (Uniform) Total-Order Broadcast protocols have been designed in the past 10 years [GLPQ10, BCP15, BQ13, MPSP10, MPP12]. A Uniform Total Order Broadcast protocol is a building block for state-machine replication. It allows a set of replicas to broadcast messages and deliver them in the same (total) order at each replica. More precisely, a Uniform Total-Order Broadcast protocol ensures the following properties for all messages that are broadcast: (1) Uniform agreement: if a replica delivers a message m, then all correct replicas eventually deliver m; (2) Strong uniform total order: if some replica delivers some message m before message m_0, then a replica delivers m_0 only after it has delivered m.

The recent Total-Order Broadcast protocols that have been designed aim at achieving high throughput and low latency. They are actually very efficient, but have been designed for fully switched networks, such as those found in datacenters and clusters. Nevertheless, more and more companies want to deploy geo-replicated systems and are thus looking for protocols that work efficiently when used across several datacenters. Unfortunately, this is not the case of existing protocols, as we show in Section 1.3. There are mostly two reasons for this inefficiency in multi-datacenters environments. First, some algorithms [BQ13, MPP12] rely on IP-multicast for

broadcasting messages between servers. IP-multicast is usually not available across datacenters and must be replaced by ad-hoc, inefficient message dissemination patterns (e.g. a server can replace IP-multicast by sending the same messages to all servers using UDP or TCP). Second, because these algorithms target fully-switched environments, they equally balance the load among each network link, which is not optimal in multi-datacenters environments where inter-datacenter links are shared across nodes, unlike intra-datacenter links.

This chapter is organized as follows. Section 1.1 presents the system model, as well as notations and terms used throughout the chapter. Related works are described in Section 1.2. Section 1.3 describes the theoretical performance that would be achieved by the two best protocols in a multi-datacenters environment, before presenting a table that sumarizes all the studied protocols in Section 1.4.

1.1 Background and Model

In this section, I describe the notations and concepts used in the document, as well as the the system model we assume when designing a new total order broadcast protocol.

1.1.1 State Machine Replication

A State Machine is a virtual representation of the state of a system that starts at an initial state and changes after inserting some inputs into one or more other states. This concept was originally first pointed out with the works of Huffman [Huf55] and Moore [Moo58]. For example, a computer system could be seen as a very complex State Machine. State Machine Replication (SMR) [Lam78, Sch90, Pol] is a distributed protocols that aims to keep several replicas of a same State Machine consistent, robust, fault tolerant and available. Replicas (sometimes called processes too) communicate by message passing.

1.1.2 Broadcast Specifications

Firms possessing critical data are attentive for their data consistency. Mis-ordering messages may lead to catastrophic consequences. In a replicated database [CMZ04], executing $INSERT$ before $UPDATE$ instead of $UPDATE$ before $INSERT$ leads to unstable results. For example, in a bank account of balance one million dollar, deposing an amount of million dollar then having an interest 10% is different than getting 10% interest for the first million then deposing another million. The first means 2.2 million dollars while the latter means 2.1 million dollars.

[HT93] divided the communication patterns that interact by message passing in SMR systems into (1) $Point - to - point$ and (2) $Broadcast$. In order to have a consistent communication, they define some properties and classify guarantees according to their reliability level, as well to their ordering mechanism. Since my work lays under the topic of TOB, I discuss here the $Broadcast$ specication in general and then focus on Total Order Broadcast. They defined two primitives Broadcast and Deliver. When a process p sends a message m, this is called a Broadcast. When the system completes delivering m to all processes, this is called Deliver.

Reliability Guarantees

Reliable Broadcast is the weakest type of fault-tolerant broadcasts. It guarantees three properties (1)Validity: If a correct process Broadcasts a message m, then it eventually Delivers m. (2) Agreement: If a correct process Delivers a message m, then eventually all correct processes Deliver m. (3) Integrity: for any message m, any correct process p_j Deliver m at most once, and only if m was previously Broadcasted by some correct process p_i. A process is considered a correct process if it never fails. Uniform Broadcast protocol ensures the Uniform Agreement: If a replica (be it correct of faulty) delivers a message m, then all correct replicas eventually deliver m.

Ordering Guarantees

Several ordering guarantees can be ensured: FIFO Broadcast, Causal Broadcast, Total order Broadcast, Timed Broadcast, FIFO Total order Broadcast and Causal Total order Broadcast.

FIFO Broadcast is a reliable broadcast in addition to FIFO Order property: If a process Broadcasts a message m before it Broadcasts a message m', then no correct process Delivers m' unless it has previously Delivered m. Causal Broadcast is a reliable broadcast in addition to Causal Order property: If the Broadcast of a message m causally precedes the Broadcast of message m', then no correct process Delivers m' unless it has previously Delivered m. Total order Broadcast is a reliable broadcast in addition to Total Order property: For any two messages m and m', if any process p_i Deliver m without having delivered m', then no process p_j Deliver m' before m. Timed Broadcast is a reliable broadcast in addition to Timeliness property: There is a known constant δ such that if a message m is Broadcast at time t , then no correct process Delivers m after time $t + \delta$. FIFO Total order Broadcast: A combination between FIFO broadcast and Total order broadcast. And finally, Causal Total order Broadcast: A combination between Causal broadcast and Total order broadcast.

Uniform Total Order Broadcast

As a Conclusion, Uniform Total Order Broadcast is a primitive that allows implementing SMR protocols. Uniform Total Order Broascast allows ordering messages among replicas in order to ensure reliable and consistent data system. Uniform TOB satisfies the following properties:

- **Validity**: If a correct process p_i Broadcast a message m, then p_i eventually Deliver m.
- **Integrity**: For any message m, any correct process p_j Deliver m at most once, and only if m was previously Broadcast by some correct process p_i.

- **Uniform Agreement**: If any process p_i Deliver any message m, then every correct process p_j eventually Deliver m.
- **Total Order**: For any two messages m and m', if any process p_i Deliver m without having delivered m', then no process p_j Deliver m' before m.

1.1.3 Performance Metrics: Throughpuyt vs. Latency

In the context of this document, we will consider two performance metrics: Latency and Throughput. Latency represents the time needed to deliver one message, while Throughput is the number of delivered messages per time unit. A lot of TOB algorithms favor Latency over throughput [KT96, AFM92, Car85, GMS91, BvR96, WS95] while others favor throughput [GLPQ10]. Finaly, FastCast [BQ13] achieves optimal throughput in cluster environment and achieves also low latency as well as Ridge [BCP15].

To explain the difference between a latency-optimal and a throughput-optimal algorithm, I borrow Figure 1.1 from [GLPQ10] and describe it.

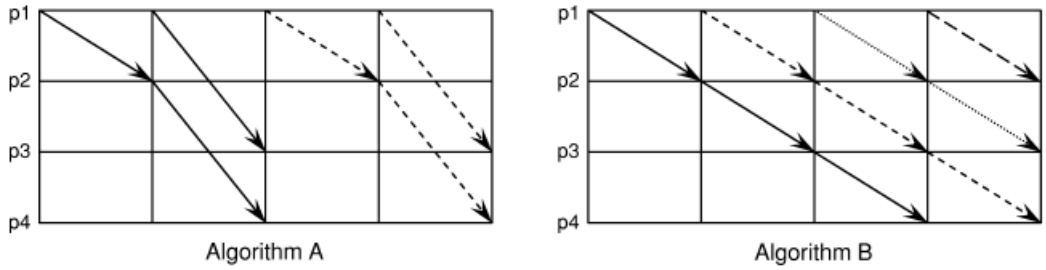


Figure 1.1: Latency Vs. Throughput

In *AlgorithmA*, p_1 sends a message m to p_2 which forwards it to p_4. In the second round, p_1 sends m to p_3. By now, m is broadcast from p_1 to all other nodes. Another message is broadcast in the third round and so on. Thus, it lasts two rounds for a message to be broadcast. The broadcast latency, hence, is two rounds. While, it is possible to broadcast one message each two rounds. The throughput, hence, is one message per two rounds (i.e. half message per round). On the other hand,

in $AlgorithmB$, p_1 sends a message m to p_2 which forwards it to p_3 and then from p_3 to p_4. As noticed, it needs three rounds for a message to be delivered, so the latency is three. While, three messages are delivered in three rounds which means that a message is delivered per round. Therefore, the throughput is one message per round.

1.1.4 System Model

MDC-cast is designed to work in a multi-datacenters environment. We assume a set $S = \{p_1, ...p_N\}$ of N processes (also called "machines") distributed over several datacenters. Each datacenter is composed of a local area network that contains a number of interconnected processes $G = \{p_i, ...p_j\}$. Nodes in different datacenters inter-communicate over a wide area network. We assume that machines can only fail by crashing (i.e. Byzantine failures are out of the scope of our interest), that crashes are rare, and that each node is equipped with a perfect failure detector (P) [CT96a]. The failure detector is implemented as follows: *MDC-cast* creates a TCP connection between each two nodes and maintains this connection during the entire execution of the protocol with setting the KeepAlive flag. The failure detector provides periodic heart-beating to specify whether the remote node is responding or not. When a connection fails or lasts long, the machine tries to re-establish it five times with an exponentially increasing delay between each connection attempt. If the connection cannot be re-established, we consider that the target node has crashed. If the node reappears before or during the recovery procedure, we force the node to crash.

1.2 Existing TOB Protocols

TOB is studied since the ends of 1970s. A wide number of algorithms have been designed and published since that time. Even in multi-datacenters situation, *MDC-cast*is not the first protocol used. Out of these systems, I discuss the most relevant work. I start the state-of-the-art by classifying systems according to their ordering

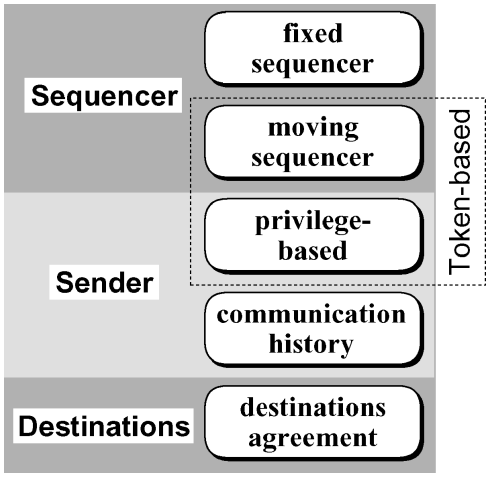


Figure 1.2: TOB Classification

mechanism. Then, I introduce some of them and discuss in details those that share some similarities with my work. I focus on their message transmission patterns and the communication techniques they use.

1.2.1 TOB Classification

In order to classify TOB protocols, I summarize the taxonomy of [DSU04], where they classified the literature into five kinds [Figure 1.2] according to their synchronization mechanism:

- **Fixed-Sequencer**: As the name indicates, in fixed-sequencer systems, one node is elected as the sequencer of the group. A sequencer is in charge of ordering messages where each message has to take a sequence number from the sequencer. Examples of algorithms using a fixed-sequencer include [KT96, AFM92, Car85, GMS91, BvR93, WS95, Jia, CH, SH97, SNN, Reib, Reia, MPSP10, Ban07, BQ13, BCP15].

- **Moving-Sequencer**: They are based on the same principle as the Fixed-Sequencer algorithms. In the moving-sequencer systems [CM84, WMK94, KK97, CMA97], they avoid the bottleneck of the fixed sequencer by moving this role among participating nodes.

- **Privilege-Based**: Protocols [FR97, Cri91, ESU04, AMMS+95, GT89, ADMA+04a] rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. The sender should have a token in order to broadcast a message.

- **Communication History**: In protocols using Communication history [PBS89, MSS96, EMS95, Ng91, MMSA93], a sender has the ability to send every time. But, the privilege is needed on delivery instead of sending.

- **Destination Agreement**: In destinations agreement algorithms [PBS89, MSS96, EMS95, Ng91, MMSA93, CT96b, BJ87b, LG90, FIMR01, Anc97], as the name indicates, the delivery order results from an agreement between destination processes.

1.2.2 LCR

LCR [GLPQ10] is a Uniform TOB primitive for building SMR system that is efficient in failure free periods. It has been designed for small homogeneous clusters, in which machines are connected by a fully switched network. It is based on a ring topology and only relies on point-to-point interprocess communication where each node communicates just with its successor. The article defines a theorem that states the optimal throughput that can be achieved in cluster environments:

Theorem 1. Maximum Throughput. *For a broadcast protocol in a system with n processes in the round-based model used in [GLPQ06, GKLQ07], the maximum throughput μ_{max} in completed broadcasts per round is:*

$$\mu_{max} = \begin{cases} n/(n-1) & \text{if there are } n \text{ senders} \\ 1 & \text{otherwise} \end{cases} \tag{1.1}$$

Also, the authors propose an algorithm, LCR, that matches this optimal throughput. Its name is derived from the fact that it relies on **L**ogical **C**locks in addition to **R**ing

topology. The mechanism of LCR is as follows: In order to broadcast a message m, a process p_i will send m just once to its successor p_{i+1}. Each process p_j will forward m to its successor unless process p_{i-1} which creates an acknowledgement and sends it to the predecessor of p_i (Commonly represented as p_{i-1}). p_{i-1} creates an acknowledgement ACK and sends it to p_i. ACK will be forwarded by processes one by one until p_{i-2}. Each node can deliver the message as soon as it receives ACK due to the fact that it knows that all nodes have received the message correctly. Order in LCR is computed according to which messages are received by the last process in the ring, that is, process p_{n-1}. LCR assumes a perfect failure detector to which each process has access and that failures are rare. Processes in LCR are arranged in views. Each node that wants to participate will try to join the view. When a node crashes it will leave automatically the view. When a process joins or leaves the view, the *view_change* will be triggered and the view will be changed into another view that contains the new participating nodes. On every *view_change* some messages may got lost. To solve this issue the authors introduce a *recovery* method. Firstly, the nodes share their knowledge about pending messages. Then they deliver all relevant messages. Finally, they start over with the new view.

Figure 1.3 shows a network of five nodes $\{P1, P2, P3, P4\ and\ P5\}$ running LCR algorithm where process $P1$ broadcasts a message by transmitting it to its successor $P2$. $P2$ forwards the message to its successor $P3$ which forwards it to $P4$ and then from $P4$ to $P5$. $P5$ then generates an acknowledgment and transmit it to the message sender $P1$. The acknowledgment will be forwarded to $P2$ and successively to $P3$, $P4$ and $P5$.

TCP/IP does not provide fairness among sending sockets. Since LCR relies on TCP/IP communications, the bandwidth used for forwarding from one process can overwhelm other processes. In other words, if all processes broadcast messages, it is possible, in some cases, that the distribution of bandwidth among nodes will not be fair. LCR does thus provide a mechanism that ensures that each process will have equal opportunity to have its messages delivered by all processes. Each

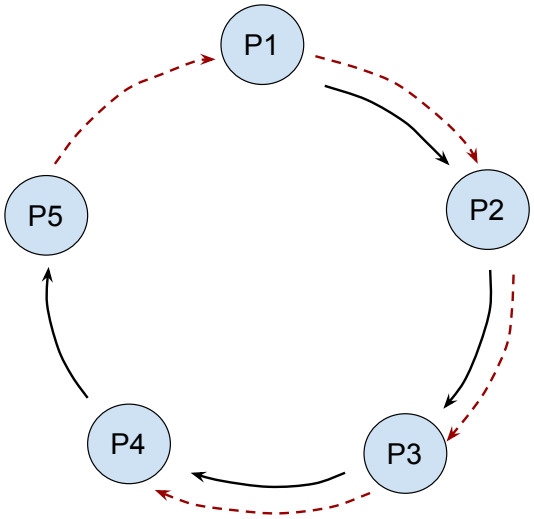


Figure 1.3: LCR Protocol: Dissemination Pattern Phase (Normal Arrows) and Acknowlegment Phase (Dashed Arrows)

process has two queues: *send_queue* that contains messages to be broadcast and *forward_queue* that contains messages broadcast by predecessors that should be forwarded. By this, each node will count the messages and assert that transmitted number of messages is fair between processes. The latency of the protocol is equal to $2n-2$ rounds due to the fact that a message needs $n-1$ rounds to be broadcast and $n-1$ rounds to be acknowledged.

The protocol was implemented in C. The implementation contains two network levels where both of them rely on TCP/IP: (1) The ring topology layer where each process establishes a connection with its successor and (2) The group membership layer where it relies on mesh topology and makes use of the Spread toolkit [ADMA+04a]. It is compared to two state-of-the-art systems: the Spread toolkit [ADMA+04a] and JGroups [Ban07] with various performance metrics: throughput, response time, fairness, and CPU consumption. LCR achieves optimal throughput for one sender as JGroups. For n senders, its throughput is also optimal and significantly better than the one achieved by Spread and JGroups. Its response time is reasonable as well as its CPU consumption which does not exceed 55 percent in worst cases.

Even though LCR is throughput optimal and scalable, it uses all of the links between

nodes equally and so it is not optimal in WAN systems which are prone to bandwidth bottlenecks, as we will see later in this document.

1.2.3 FastCast

FastCast is a total order broadcast based algorithm that achieves optimal throughput with low latency. It works in a fully switched network of inter-connected processes (machines). In order to broadcast a message, the process transmits data to all other processes directly relying on IP-multicast which is on top of UDP/IP. Then, acknowledgments are collected in a second step. The system is designed for small clusters of homogeneous machines interconnected by a local area network. *FastCast* assumes that machines do not partially fail, that crashes are rare and that each node is equipped with a perfect failure detector (the same as LCR).

In order to broadcast a message m, a process p_i multicasts m just once to all other nodes (Phase I). The *Leader* creates an acknowledgment ACK stamped by a sequence number and multicast it to all processes (Phase II). Thereafter, each process p_j creates an acknowledgement and multicasts it (Phase III). Processes in *FastCast* are arranged in groups. Each node that needs to participate joins the group. When a node crashes it leaves automatically the group. When a process joins or leaves the group, the *view_change* is triggered and the view is changed into another view that contains the new participating nodes. On every *view_change* some messages may got lost. To solve this issue *FastCast* introduces a *Recovery* method. Firstly, the processes exchange their knowledge about pending messages. Then, they deliver relevant messages and start with the new view.

Figure 1.4 shows a network of five nodes $\{P1, P2, P3, P4\ and\ P5\}$ running *FastCast* algorithm where $P4$ is the *Leader*. The figure illustrates the three consecutive phases of *FastCast* where process $P1$ broadcasts a message m. In the left part, which depicts the dissemination pattern phase, m is transmitted from $P1$ to other nodes directly using IP-multicast. Then, in the middle part, the *Leader* assigns a unique sequence

number to m and IP-multicast the message. Finally, in the right part, each message acknowledges m by IP-multicasting an acknowledgment message.

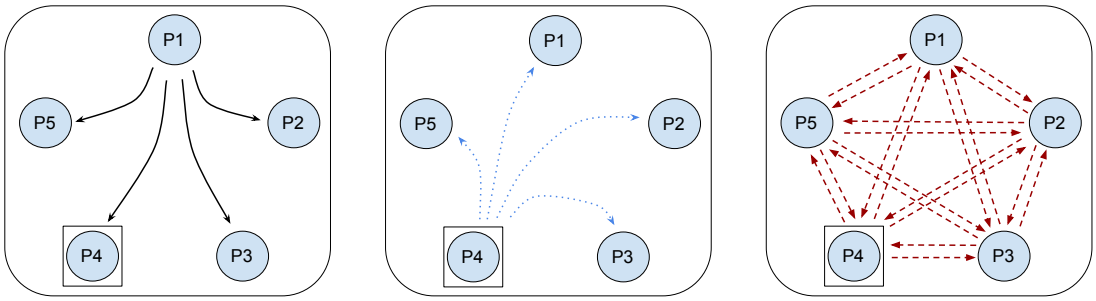


Figure 1.4: FastCast Protocol: Dissemination pattern (Phase I:left part); Assigning Sequence number (Phase II:middle part); Acknowledgment phase (Phase III:right part)

IP-multicast is prone to message losses. For that purpose, authors added a bandwidth allocation sub-protocol that imitates the congestion protocol of TCP. The goal of the bandwidth allocation protocol is to allocate bandwidth for each sending node in order to allow multiple nodes to simultaneously and fairly send IP multicast packets, while reducing message losses. To realize this mechanism, authors firstly assume that each node knows the bandwidth requirements of all other nodes and then they use a max-min fair bandwidth allocation algorithm [Bou00]. Nodes declare the changes in their bandwidth usage. More precisely, if a node requires to decrease its bandwidth, it can do it directly and then inform all other nodes about this change. If a node needs to increase its bandwidth, it needs to communicate with other nodes to check if this is possible. Then each node sends an acknowledgment. After receiving ACKs from all other nodes, the node can increase its bandwidth.

The protocol is throughput optimal because each $n-1$ rounds, there are n messages delivered. The latency is equal to three rounds due to the fact that a message needs one round to be broadcast, one round to be assigned a sequence number and one round to be acknowledged.

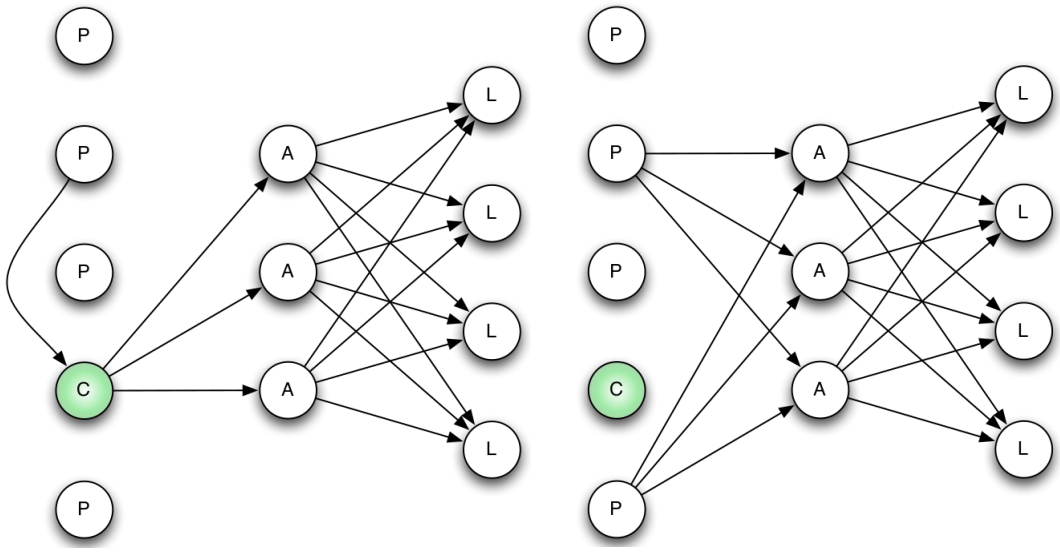
The protocol has been implemented in C++ using the same code base as the *Ring Paxos* protocol. It was compared with two state-of-the-art systems LCR [GLPQ10]

and Ring Paxos [MPSP10] with various performance metrics: throughput, response time and latency. *FastCast* achieves optimal throughput like LCR. However its response time and latency are lower than LCR and Ring Paxos.

Even though *FastCast* is throughput optimal and with low latency, it cannot scale to multi-datacenter environments because IP-multicasting is not supported between datacenters.

1.2.4 Paxos and Fast Paxos

Paxos has been designed to solve consensus, which is, roughly speaking, equivalent to total order broadcast. We explain the behavior of *Paxos* because it is used as a basis for several TOB protocols, e.g., *Ring Paxos* [MPSP10], *Multi-Ring Paxos* [MPP12] and Ridge [BCP15].



(a) Paxos Algorithm (b) Fast Paxos Algorithm

Figure 1.5: Paxos and Fast Paxos

We can explain the consensus problem by considering that a set of processes want to take a decision. One way to do that is to as follows. A process P_i would ask to take the token. When it gets the permission, it would propose something and the processes would vote for it (thus reaching consensus). Finally, the decision would be

announced to all processes. P_i is represented in *Paxos* by a *Proposer* while processes that vote are called *Acceptors* and the replicas are called *Learners*. It is sufficient to have a reply from a quorum (majority of *Acceptors*) in order to take a decision. A consensus ensures that only one proposed value is finally accepted and learned by every node even though messages can take arbitrarily long to be delivered, can be duplicated, and can be lost. *Paxos* nevertheless assumes that messages cannot be corrupted.

More precisely, the *Paxos* algorithm can be split into five tasks (noting that the first two tasks are called the selection phase):

- **Plebiscite**: The *Proposer* sends a $PREPARE$ message to a majority of *Acceptors*.
- **Allegiance**: *Acceptors* respond by showing their allegiance (The first two tasks are a sort of synchronization preamble).
- **Proposition**: After collecting a quorum of allegiance messages and being selected, the *Proposer* proposes its value to *Acceptors* by sending the message to the *Coordinator* (One node of the *Acceptors* elected as *Coordinator*) which forwards it to the *Acceptors*.
- **Learning**: When an *Acceptor* receives a new proposed value, it votes for it and informs *Learners* about it.
- **Delivery**: A message is delivered after getting a majority of votes.

Paxos algorithm is depicted in figure 1.5a without its selection phase (The first two tasks), a *Proposer* P proposes a value and send it to the *Coordinator* C. The *Coordinator* forwards the value to each *Acceptors*. Each *Acceptor* informs eventually each *Learner* about its decision.

Paxos was later improved. The resulting protocol, called *Fast Paxos* [Lam06] (Figure 1.5b) works by shrinking the steps of delivering a message. The key difference between the two algorithms is that instead of sending a $PREPARE$ message for

each instance, in *Fast Paxos* it is sufficient to send one $PREPARE$ message for a sequence of instances.

1.2.5 Ring Paxos

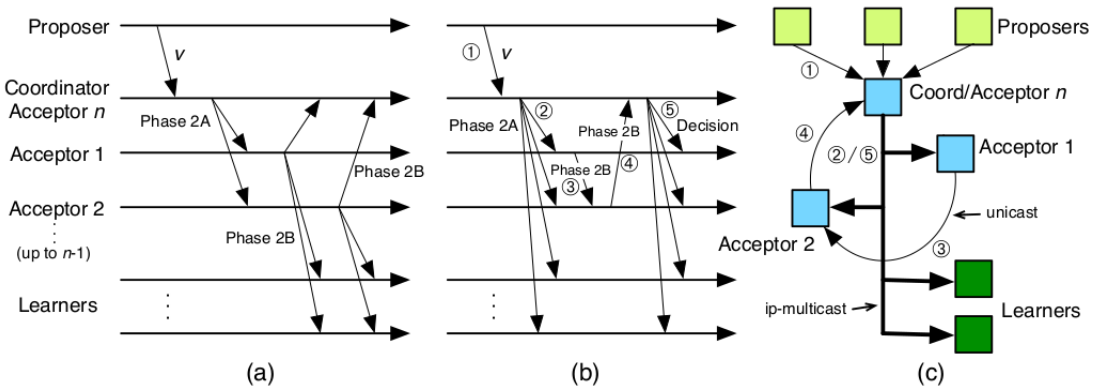


Figure 1.6: Comparison between *Paxos* and *Ring Paxos* provided in [MPSP10]

Ring Paxos [MPSP10] is a total order broadcast algorithm derived from *Paxos*. It provides some optimizations like the ring topology (as its name implies) and the IP-multicast. It has been designed for small homogeneous clusters, in which machines are connected by a fully switched network.

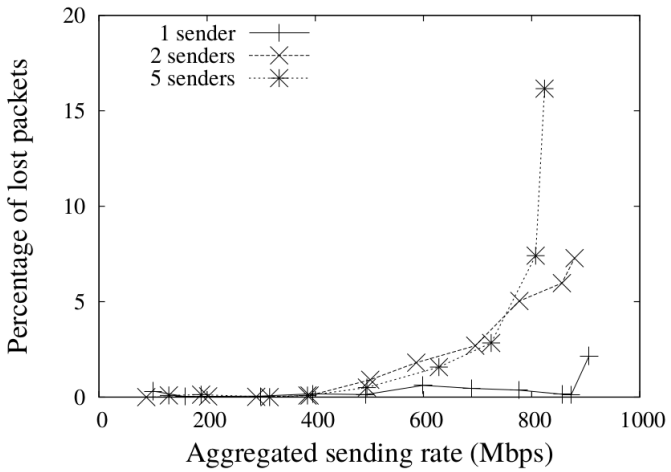


Figure 1.7: Percentage of packet losses when multicasting messages (1 to 3 senders). (Figure borrowed from [MPSP10]).

Like *Paxos*, there are three main roles in *Ring Paxos*: *Proposers*, *Acceptors* and

Learners in addition to the *Coordinator* which is one of the *Acceptors*. The procedure can be split into five phases like Paxos with a difference in the Learning and Delivery phases.

Figure 1.6 shows a comparison between *Paxos* and *Ring Paxos* [MPSP10]. In *Paxos*, when a *Proposer* proposes a value, it sends it to the coordinator. The *Coordinator* sends it to all *Acceptors*, then, to vote for it. However, in *Ring Paxos*, the *Coordinator* is in charge of multicasting the message directly to all nodes. The *Coordinator*'s successor then forwards it to its successor and so on until it comes back again to the *Coordinator*. After finishing its ring, the message will get in its final stage: Delivery. The *Coordinator* then multicasts again the message to all *Acceptors* and *Learners* to be delivered.

It is well known that IP-multicast is subject to message losses due to buffer overflow. To minimize these losses, *Ring Paxos* limits the throughput of multicasting from each sending node and configure the communication buffer sizes. But there is still a problem, multicasting from several simultaneous senders. Figure 1.7 (borrowed from [MPSP10]) shows the throughput impact when multiple senders multicast messages. It can be observed that when the system runs with 5 senders, the percent of lost messages is under the threshold of 5% until the aggregated sending rate meets around $800Mb/s$ where it faces a bottleneck. Authors treat this issue by replacing the way that *Acceptors* multicast messages by a point-to-point communication in a ring topology (they order *Acceptors* in a ring topology).

The protocol was implemented in *C* language and compared experimentally to some state-of-the-art systems: LCR, Spread toolkit [ADMA+04a], LibPaxos [lib] and Paxos4sb [KA08]. The article shows a comparison between the Maximum Throughput Efficiency (MTE [1]) of the algorithms. Results show that the maximum throughput efficiency of *Ring Paxos* is very good as well as LCR but the algorithm has several advantages such as a better latency.

[1] They introduced the notion of *Maximum Throughput Efficiency* and define it by the rate between the maximum achieved throughput per receiver and the nominal transmission capacity of the system per receiver.

1.2.6 Multi Ring Paxos

As its name indicates, *Multi-Ring Paxos* is a protocol derived from Paxos that combines several instances of *Ring Paxos* algorithm into a new algorithm in order to improve the system performance. *Multi-Ring Paxos* uses multiple independent instances of *Ring Paxos* which are considered as groups of *Acceptors*.A *Proposer* can initialize a message broadcast and send it to one of the groups. Also, *Learners* subscribe to groups they would like to deliver messages from. If a *Learner* subscribes to groups g_{l_1}, g_{l_2} , ..., g_{l_k} , where $l_1 < l_2 < ... < l_k$, then the *Learner* could first deliver M messages from g_{l_1} , then M messages from g_{l_2} , and so on, where M is a parameter of the algorithm [MPP12]. Synchronization between groups is handled via a deterministic merge procedure.

The idea of Multi Ring Paxos is that a *Proposer* sends a proposition to one of the groups which will treat it as in Ring Paxos. Then the message will be sent to the subscribed learners. Inside a group, *Acceptors* act exactly like in Ring-Paxos: the *Coordinator* multicasts every message, and the message will be then forwarded in a ring topology between Acceptors.

In the enclosed figure (Figure 1.8), an illustrated scenario clarifies the system. The figure shows a system running *Multi-Ring Paxos* with two groups $g1$ and $g2$ where each group contains a set of *Acceptors*. The system includes two *Proposers* (*Proposer*1 and *Proposer*2) and two *Learners* (*Learner*1 and *Learner*2). *Proposer*1 initializes two message broadcasts ($m1$ and $m3$) and sends them to $g1$. *Proposer*2 initializes two message broadcasts ($m2$ and $m4$). $m2$ is sent to $g1$ while $m4$ is sent to $g2$. *Learner*1 subscribes to group $g1$ and *Learner*2 subscribes to groups $g1$ and $g2$. M is considered to be 1 which means that a *Learner* has the choice to switch between groups on each message.

The protocol was implemented assuming that decisions can be stored in main memory (and not on disk) and compared experimentally to some state-of-the-art systems: LCR, Spread and Ring Paxos using several performance metrics: Throughput, Latency and CPU usage.

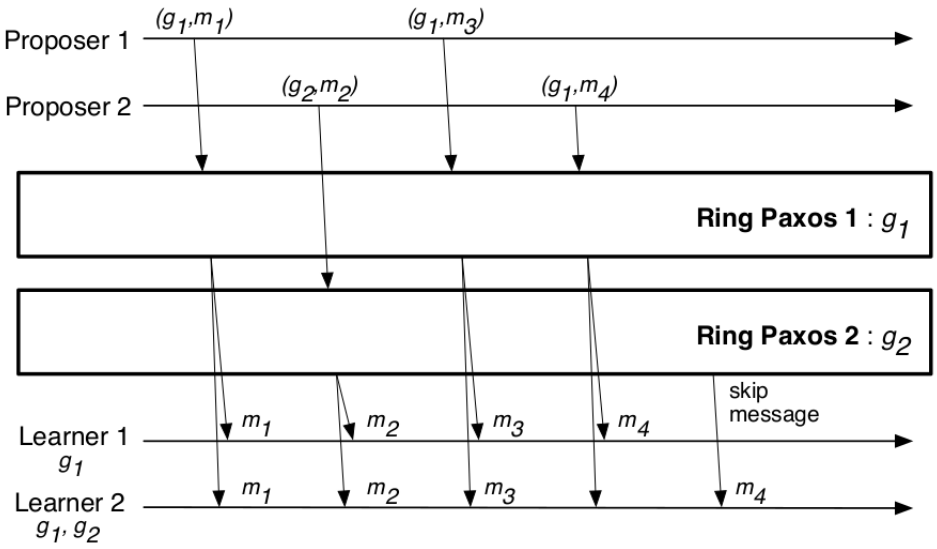


Figure 1.8: Multi Ring Paxos algorithm.

Even though *Multi-Ring Paxos* is throughput optimal, it is not adapted to multi-datacenters setups provided it relies on IP-multicasts.

1.2.7 Ridge

Ridge is another TOB algorithm from the *Paxos* family that differs from the former algorithms in the fact that it targets WAN systems. Roughly speaking, *Ridge* does not rely on IP-multicast and achieves better throughput than *Paxos* and *Fast Paxos*. It also intersects with Multi-Ring in the idea of having several intercommunicating groups called ensembles. Each ensemble contains $2f + 1$ *Acceptors*, where f is the maximum number of failures tolerated by the ensemble [BCP15]. We first explain *Ridge* with one ensemble. In order to explain *Ridge*, it is necessary to describe *Paxos* in more details. The consensus phase implemented by *Acceptors* can be split into two phases:

- **Election**: The *Coordinator* creates a new unique ID and sends it to all *Acceptors*. *Acceptors* will vote for it. If the *Coordinator* receives majority of votes, it starts the second phase.

- **Taking Decision**: The coordinator proposes a new value to all *Acceptors*. *Acceptors* will vote for it. If the *Coordinator* receives majority of votes, it notifies *Acceptors* about the new voted decision.

Ridge optimizes the second phase of *Paxos*. The *Coordinator* sends a message m to an *Acceptor* which forwards it to another one and so on until ensuring that a majority of *Acceptors* have received m. Then, the last *Acceptor* knows that m has been accepted by a quorum m. In order to take a decision, the last Acceptor sends m to a *Learner* with the help of a load balancer. This *Learner* is is in charge of distributing the decision by sending m to all other *Learners* directly. This phase (Phase II) could be split into two stages: *Phase IIa* which is similar to ring algorithms and *Phase IIb* which is multi-unicasting (i.e. sending from one node to each other node aside). The procedure can be split into several steps:

- **Proposer**: A *Proposer* proposes the message to one or more groups by sending the value to their *Coordinators*.
- **Coordinator**: The *Coordinator* proposes the message to the ensemble.
- **Acceptor**: the *Acceptor* takes its role in voting as described in one ensemble.
- **Learner**: The learner finally delivers the message according to its order.

Figure 1.9 shows a system running *Ridge* algorithm. The system is composed of six *Proposers*, five *Acceptors* and four *Learners* where a *Proposer* P broadcasts a message m. Firstly, *Ridge* runs the first phase of *Paxos*, then the *Coordinator* C sends m to an *Acceptor* A which forwards it to another *Acceptor* and so on until the majority of acceptors have received the message. The last *Acceptor* knows that a quorum has known about m so it forwards it to a *Learner* with the help of a load balancer (This *Learner* is called the *Distributing Learner*). The *Distributing Learner* then forwards the message to every other *Learner*.

In case of several ensembles, *Ridge* makes use of a merging algorithm that coordinates messages.

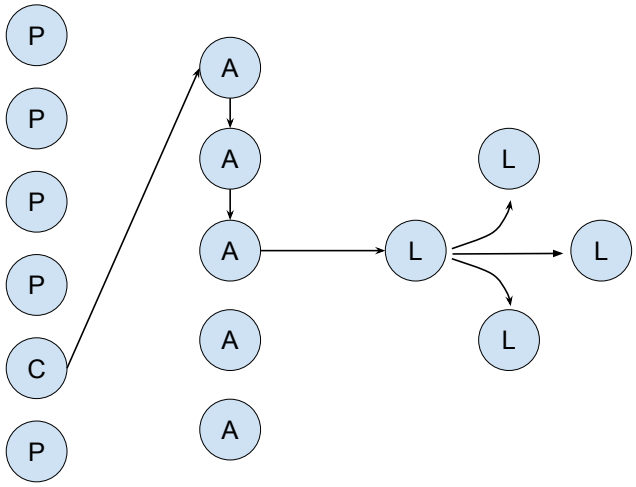


Figure 1.9: Ridge Protocol with one ensemble (Phase II).

In addition, *Ridge* provides an optimistic delivery which is strongly coupled with what is published in [BPGG13]. As its name indicates, using this mechanism, *Learners* deliver messages before being broadcast to *Acceptors*. The *Proposer* sends the message m directly to all *Learners*. Simultaneously, the proposer sends m to the *Coordinator* which proceeds with Phase II of *Paxos* for m. Once the message is accepted by $f + 1$ messages, the last message just notifies *Learners* about it.

Ridge has been compared against *Spread*, *LibPaxos* and *Ring Paxos* with one ensemble and varying the number of destinations. The evaluation shows that *Ridge* is better than other algorithms in general with varying message size and the number of destinations. Results show that latency is relatively good. As well, *Ridge* has been compared to *Spread*, *LibPaxos* and *Multi-Ring Paxos* with several ensembles. Results show *Ridge* improves both the latency and the throughput. Finally, the optimized-delivery algorithm was assessed and the percent of mistaken deliveries did not exceed 3% even under high throughput.

Even though *Ridge* has been designed to work in WAN systems, it is highly affected by bottlenecks that can be observed on network links, as we show in Chapter 3.

1.3 Performance of Existing Protocols in Multi-Datacenter Environments

We have presented several TOB protocols. Only some of them are throughput-optimal, namely: FastCast [BQ13], Multi-Ring Paxos [MPP12], *LCR* [GLPQ10], and *Ridge* [BCP15]. The first two protocols rely on IP-multicast and can therefore not be used efficiently in multi-datacenters environments. We therefore focus on the *LCR* and *Ridge* protocols.

Throughout this section, we consider the multi-datacenters settings presented in Figure 1.10. This settings comprises two datacenters (A and B), having two and three machines, respectively. Machines on each datacenter are interconnected via a switch (noted SW1 and SW2) using "network cables" $C1, C2, \cdots, C5$. The communications across datacenters use "network link" L1. Link L1 is shared by all machines belonging to datacenters A and B. Network cables are obviously not shared.

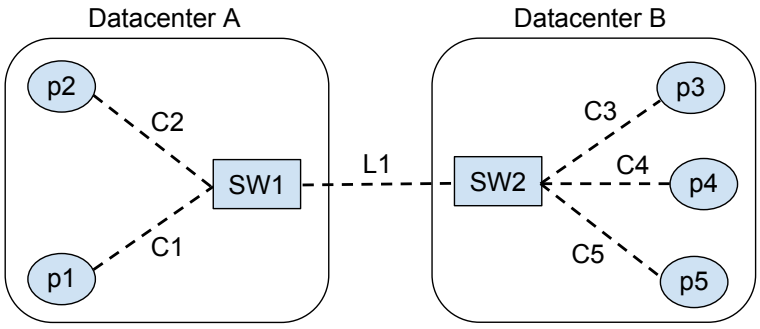


Figure 1.10: Example of a multi-datacenters environment comprising 5 machines located in two datacenters.

1.3.1 Message Pattern of LCR

The message pattern of the *LCR* protocol is depicted in Figure 1.11 in the case when process P_5 initiates a message broadcast. The process simply sends the message to each other nodes, using the TCP protocol. *LCR* uses a ring topology to disseminate messages. As the figure shows, the message is successively forwarded by processes

P_1, P_2, and P_3. When P_4 receives the message it does not forward it because it knows that the message originates from P_5. In order to ensure message ordering, *LCR* piggybacks some data on every forwarded messages (i.e. a vector clock) that allows acknowledging messages and defining the order in which messages must be delivered.

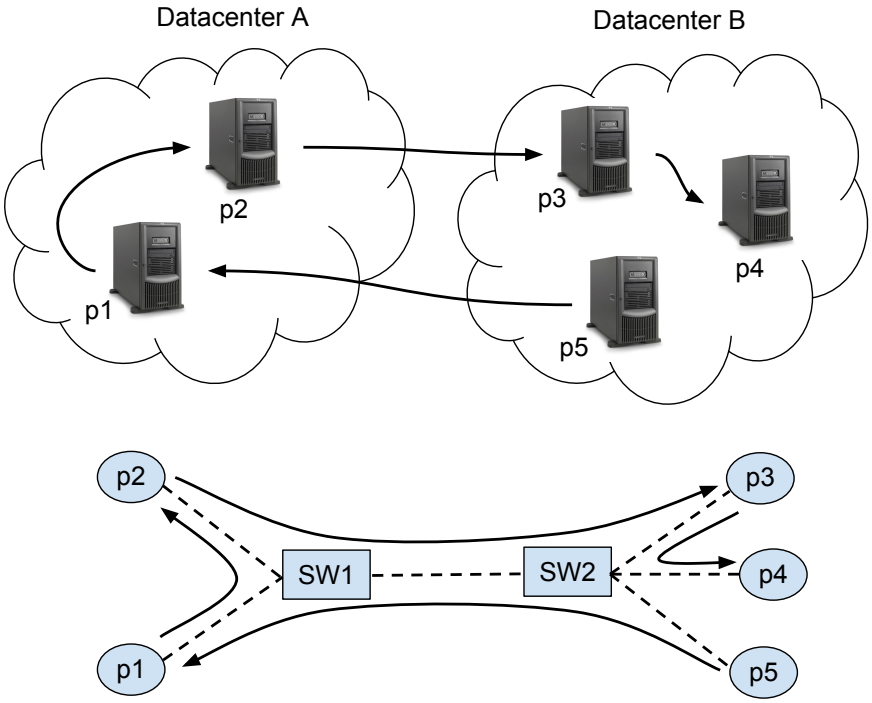


Figure 1.11: Broadcast pattern in the *LCR* protocol when process P_5 is initiating a broadcast: logical view (top part) and networking view (bottom part).

1.3.2 Message Pattern of Ridge

As explained before, *Ridge* improves the decision phase of Paxos in order to increase the system performance and adapt the protocol to WAN systems. The message pattern of the decision phase of the *Ridge* protocol is depicted in Figure 1.12. $Acceptor_1$ sends m to its successor $Acceptor_2$. $Acceptor_2$ knows that the quorum is achieved (two Acceptors out of three). It forwards the message to a Learner with the help of a load balancer (We assume that it is to $Learner_1$ which is the best case). $Learner_1$ sends m to all other Learners directly. In our example, there exists only one other

Learner which is $Learner_2$.

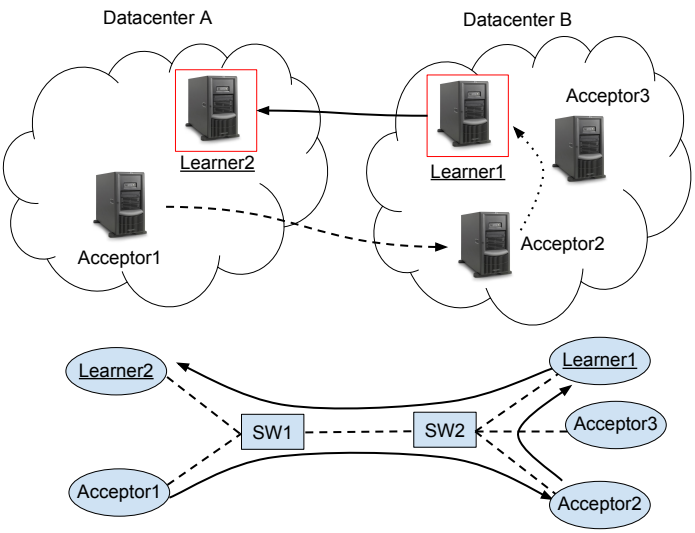


Figure 1.12: Broadcast pattern in the Ridge protocol when process $Acceptor_1$ is initiating a broadcast: logical view (top part) and networking view (bottom part).

The decision phase of the *Ridge* protocol at its core is composed of a quorum phase (Acceptors quorum) and a message dissemination phase (Among Learners). The dissemination pattern we present is more efficient than that of the actual *Ridge* protocol which we depict in Figure 1.13 where $Learner_5$ disseminates a message. In the actual prootocol, the process simply sends the message to all other nodes, using the TCP protocol.

Ridge implements a set of features that does not actually improve its throughput. The most interesting one is its ability to combine several components in one physical node. The message pattern of the *Ridge* protocol when roles are collocated in one process is depicted in Figure 1.14. A process can be an acceptor but also a Learner. $Acceptor_1$ sends m to its successor $Acceptor_2$. Then $Acceptor_2$ forwards m to its successor $Acceptor_3$. $Acceptor_3$ knows that the quorum is achieved (three Acceptors out of five). It forwards the message to a Learner with the help of a load balancer (We assume that it is to $Learner_5$ which is the best case). $Learner_5$ sends m to all other Learners directly.

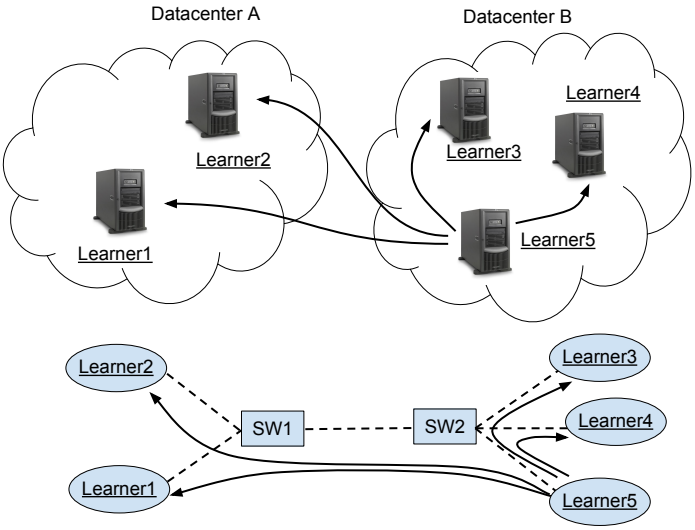


Figure 1.13: Broadcast pattern in the Ridge protocol when process $Learner_5$ is disseminating a message: logical view (top part) and networking view (bottom part).

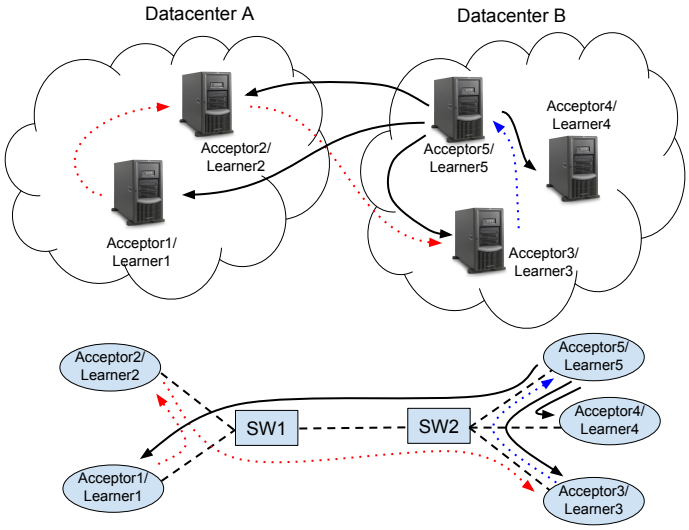


Figure 1.14: Broadcast pattern in the Ridge protocol when each Acceptor is also a Learner. $Acceptor_5$ initiates a broadcast: logical view (top part) and networking view (bottom part).

1.3.3 Latency of Ridge and LCR

In table 1.1, we provide the theoretical latency achieved by *LCR* and *Ridge* in a cluster environment. The network assumed is composed of N homogeneous nodes interconnected by a switch. The latency is showed in terms of t, the time needed for a message to be sent from one node to another. The table studies two cases (1) when one sender only Broadcasts a message and (2) when each node is broadcasting a message. First, we study when just one sender Broadcast a message. Using *LCR*, when a process P Broadcast a message m, it sends m to its successor process. m is forwarded from each process to its successor until it arrives to the predecessor process of P. So, the time needed to disseminate the message is $(N-1)*t$. Using *Ridge*, when a process P Broadcast a message m, it sends it directly to each other process. So, the overall latency of the Broadcast of m using *Ridge* is t. On the other hand, in the case of N senders, both algorithms have the same latency $(N-1)*t$. Because the two protocols have optimal throughput, they both need $N-1$ rounds to deliver N messages as stated in [GLPQ10] ($N-1$ rounds). Thus, the time needed to Broadcast N messages is $(N-1)*t$. This is due to the fact that using *Ridge*, it is impossible to send a message to two different processes at the same instant but rather messages are sent sequentially.

	LCR	*Ridge*
One sender	$(N-1)*t$	t
N senders	$(N-1)*t$	$(N-1)*t$

Table 1.1: The theoretical assessment of the latency of *LCR* and *Ridge*.

To confirm the above results, we illustrate with two figures (Figure 1.15 and Figure 1.16) the dissemination pattern when five messages are sent simultaneously. In both protocols, the number of rounds needed to Broadcast five messages using is equal to four.

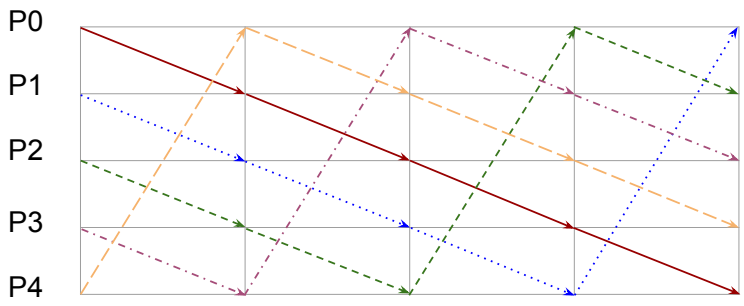


Figure 1.15: The latency of *LCR* with N senders.

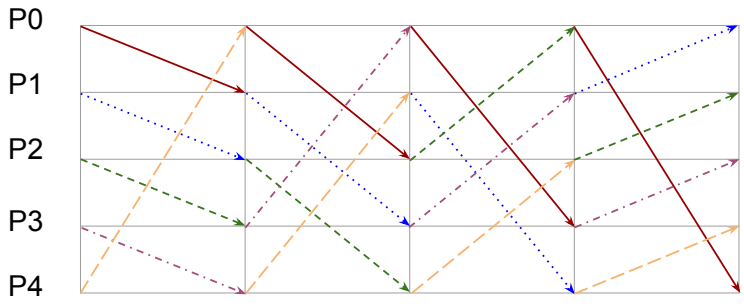


Figure 1.16: The latency of *Ridge* with N senders.

1.3.4 Throughput of Ridge and LCR

In this section, we study the throughput achieved by Ridge and LCR. We performed experiments using two setups. In the first setup, five machines are located in the same datacenter and communicate using a fully-switched network (using a $1Gb/s$ ethernet network). In the second setup, 5 machines are spread in two datacenters, as depicted in Figure 1.10. Figure 1.17 compares the performance of *LCR* and *Ridge*.

.

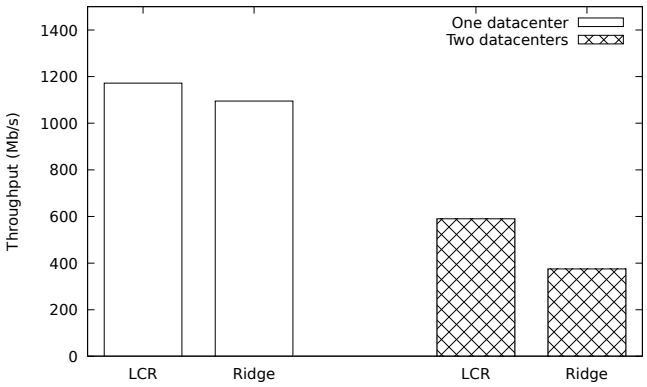


Figure 1.17: Throughput comparison between *LCR* and *Ridge* in one datacenter (left two bars) and in a multi-datacenters environment (right two bars).

Within each datacenter, machines communicate using a fully-switched $1Gb/s$ network, whereas across datacenters, machines communicate using link $L1$ that has a bandwidth of $500Mb/s$. The reason why link $L1$ has a lower bandwidth available for the protocol is that it can be simultaneously used by all machines belonging to datacenters A and B. In both cases, all machines initiate message broadcasts.

In the "one datacenter" setup, *Ridge* and *LCR* both achieve optimal throughput as defined in [GLPQ10]: $\frac{N}{N-1} \times 1Gb/s$ with $N = 5$ machines. In the multi-datacenters setup, *LCR* and *Ridge* achieve a much lower throughput. We explain this result in the remainder of this section.

To understand the performance depicted in Figure 1.17, we analytically study, the number of messages that transit on each network cable and link for the two protocols (using the setup depicted in Figure 1.10).

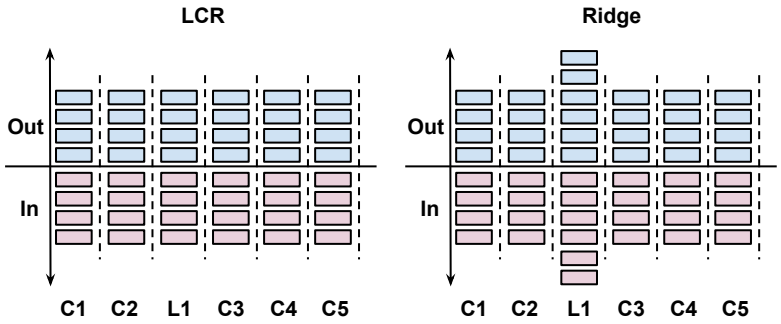


Figure 1.18: Network usage in *LCR* and *Ridge* when all processes initiate message broadcasts (topology depicted in Figure 1.10).

Figure 1.18 depicts the number of messages that are sent and received on each network cable and link when 5 messages are broadcast (one by each process). As network cables and links are bidirectional, we distinguish the two directions: "In" and "Out". We first observe that all nodes are receiving 4 messages in the two protocols ("In" direction of each network cable). This is expected provided that each machine must receive each broadcast message once.

Using *LCR*, each process generates its own message in addition to forwarding three other messages. Consequently, each machine sends 4 messages on the "Out" direction of its network cable. $L1$, the link connecting the two datacenters, is used to send messages from P_2 to P_3 and from P_5 to P_1. This explains why 4 messages transit on this link in the two directions.

Using *Ridge*, each machine sends to all other machines the messages it broadcasts. Consequently, each node sends 4 messages on the "Out" direction of its cable. $L1$, the link connecting the two datacenters, is used to send messages from each node in datacenter A towards each node in datacenter B, and vice versa. So, in total, 6 messages are sent over $L1$ in both directions.

This analysis explains the performance drop observed in the multi-datacenters setup. For both protocols link $L1$ is the bottleneck. More precisely, for *LCR*, this link conveys as many messages as other cables, but has half the bandwidth than the latter. For *Ridge*, besides having half the bandwidth of other cables, link $L1$ also conveys 50% more messages than cables (6 messages against 4). This explains why

Ridge achieves lower performance than *LCR* in the multi-datacenters setup.

LCR requires sharing $L1$ between four nodes (Each message is forwarded three times) on each lane so the theoretical throughput of generating messages is limited to $125Mb/s$ for each node. Thus, the overall theoretical expected throughput is $625Mb/s$. The achieved throughput is around $600Mb/s$ which is very cloe. *Ridge*, on the other hand, sends two messages from datacenter A on $L1$ and three messages from datacenter B on L1 per round. A node in datacenter A is allowed to transmit $250Mb/s$ to nodes in datacenter B over $L1$ which means generating around $83Mb/s$. A node in datacenter B is allowed to transmit $166Mb/s$ to nodes in datacenter A over $L1$ which means generating around $83Mb/s$. Thus, the whole system is able to generate theoretically around $415Mb/s$ which is near to the experimental results shown in the figure.

Intra-datacenter link usage

In table 1.2, we provide the usage of intra-datacenter links in the *LCR* and *Ridge* assuming that each process P_i utBroadcasts one message. We assume a system containing a set $S = \{p_1, ...p_N\}$ of N processes distributed over several datacenters. Each datacenter contains a group of processes $G = \{p_i, ...p_j\}$. In *LCR*, each node transmits its own messages and forwards all other messages except those initiated from its successor in the ring. Thus, in *LCR* each intra-datacenter link transmits $N - 1$ messages. Similarly, each node receives messages from all other nodes (that is $N - 1$ messages). Concerning *Ridge*, each node sends its messages to all other nodes directly, which means that each node transmits $N - 1$ messages and receives $N - 1$ messages.

	LCR	Ridge
Out	$N-1$	$N-1$
In	$N-1$	$N-1$

Table 1.2: Intra-datacenter link usage

Inter-datacenter link usage

In table 1.3, we provide the usage of inter-datacenters links with *LCR* and *Ridge* assuming that each process P_i utBroadcasts one message. Since *LCR* relies on a ring topology and each node transmits fairly equal bandwidth to its successor, a node transmits over a cable exactly as much as it transmits over a link, that is $N-1$ messages. Using *Ridge*, each node in group, D_j sends a message to each node in group D_k. Hence, a link in between D_j and D_k transmits $G_j * G_k$ and respectively receives $G_k * G_j$ where G_i represents the number of nodes is D_i.

	LCR	*Ridge*
Out	$N-1$	$G_j * G_k$
In	0	$G_k * G_j$

Table 1.3: Inter-datacenter link usage

1.4 Conclusion

We have studied existing TOB protocols. Our study shows that only two protocols are throughput optimal and are able to work in multi-datacenter environments: LCR and Ridge. A deeper study of those two protocols has highlighted the fact that their usage of inter-datacenter network links is not optimal, thus yielding poor throughput in this setup. In the remainder of this document, we describe a new protocol that outperforms those two protocols in multi-datacenter setups.

Chapter 2

The *MDC-cast* protocol

In this chapter we present *MDC-cast*, a Total Order Broadcast protocol specifically designed for multi-datacenters environments.

This section is organized as follows. Section 2.1 presents the description of the protocol. TSection 2.2 proves the correctness of our approach. Section 2.3 presents the bandwidth allocation mechanism. In Section 2.4, we describe some optimizations to *MDC-cast*. Finally, section 2.5 concludes the chapter.

2.1 Protocol description

2.1.1 Message dissemination

Description of the dissemination pattern

In order to achieve high performance in multi-datacenters environments, it is necessary to reduce the traffic as much as possible. Thus, we rely on IP-multicast. But IP-multicast is not supported among inter-datacenter links. So, we use IP-multicasting within datacenters and unicast communication across datacenters (I.e. on inter-datacenter links). In each datacenter, a node is delegated as an *Importer* which is in charge of forwarding messages coming from nodes in other datacenters

into nodes in its datacenter. In other words, we use IP-multicast where possible (i.e. between nodes in the same datacenter) and forward messages over TCP/IP between datacenters. This mechanism is handled by two machines in each datacenter, called *Exporter* and *Importer*. Other nodes are called *standard nodes*.

The message dissemination pattern is depicted in Figure 2.1. The figure illustrates a network of five processes $\{P_1, P_2, P_3, P_4 \textit{ and } P_5\}$ distributed over two datacenters: A and B. The figure contains both the logical view in the top part and the networking view in the bottom part. It shows the dissemination pattern of message m when process P_5 broadcasts a message with *MDC-cast*. The Broadcast primitive is implemented in two phases: the first phase is an intra-datacenter phase, whereas the second phase is an inter-datacenter phase. More precisely, when a node Broadcasts a message m, it first multicasts it inside its datacenter (phase 1). Then, the datacenter's *Exporter* forwards m to all the other datacenters by sending m directly to *Importers* located in other datacenters over TCP/IP (phase 2). Each *Importer*, then, IP-multicasts m inside the datacenter it belongs to (phase 3). We denote the *Exporter* of a datacenter by the node holding the smaller ID number in the datacenter while the *Importer* is the node holding the largest ID.

Specifically, in Figure 2.1, P_5 IP-multicasts m locally to its neighbors[1] P_3 and P_4. P_3, which is the *Exporter* of datacenter B, exports m to the other datacenter A by sending it directly to the *Importer* of A: P_2. P_2 receives the message and forwards it to its neighbors using IP-multicast.

Illustration of the link usage

Figure 2.2 depicts the number of messages that are sent on each network cable and link when five messages are broadcast (one by each process) using the *MDC-cast* protocol. In datacenter A, P_1 is the *Exporter* and P_2 is the *Importer*. In datacenter B, P_3 is *Exporter* and P_5 is *Importer*. We borrow the same system and configurations depicted in figure 1.18 to ensure a fair comparison. We observe that,

[1]We denote two processes as neighbors if they are both localized in the same datacenter

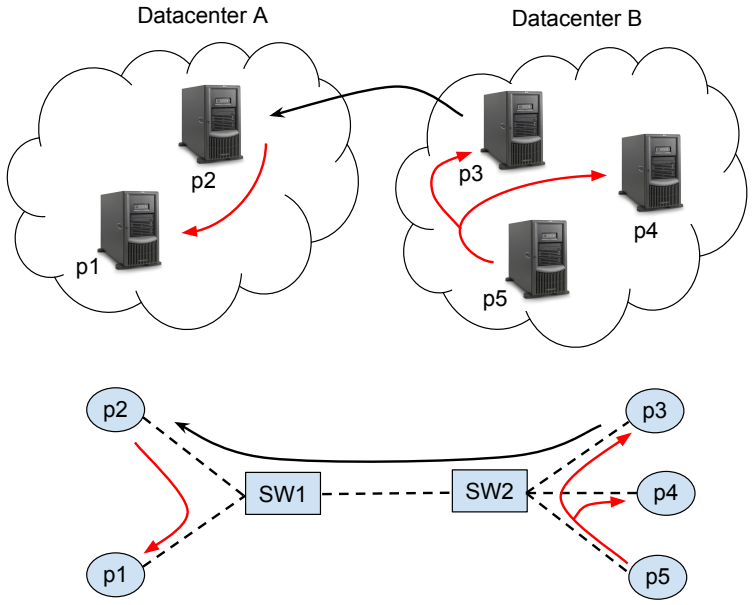


Figure 2.1: Broadcast pattern in the *MDC-cast* protocol when process P_5 is initiating a broadcast: logical view (top part) and networking view (bottom part). IP-Multicast messages are depicted in red, whereas unicast TCP messages are depicted in black.

like other protocols, each node is receiving four messages. Moreover, each node is generating its own message. *Exporters* forward the messages produced within their datacenter to other datacenters: for instance, P_1 outputs three messages: m_1 that it multicasts and exports, and m_2 that it exports. The *Importer* is in charge of forwarding messages received from other datacenters. For instance, P_2 outputs four messages: m_2 that it multicasts, and three messages received from datacenter B that it forwards: m_3, m_4 and m_5. As a conclusion, we can say that unlike other protocols, *MDC-cast* optimizes the usage of inter-datacenter links. Indeed, only two (respectively three) messages are sent from datacenter A to datacenter B (respectively from datacenter B to datacenter A).

Theoretical assessment of the link usage

Table 2.1 shows the link usage of *MDC-cast* in a network composed of five processes. The figure studies the broadcast of a message represented in figure 2.1. The case is

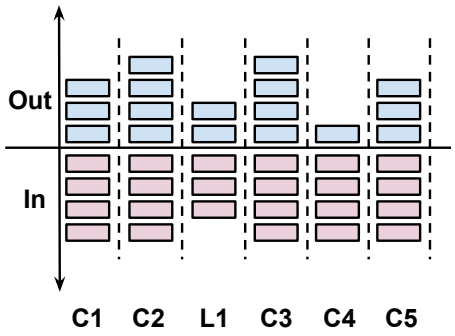


Figure 2.2: Network usage in *MDC-cast* when all processes initiate message broadcasts (topology depicted in Figure 1.10).

analyzed in two dimensions: inter-datacenter and intra-datacenter.

	Links (Inter-datacenter)	Cables (Intra-datacenter)
Out	G_j	Exporter: $G_k * (D-1)+1$ Importer: $N - G_k + 1$ Others: 1
In	G_k	$N-1$

Table 2.1: MDC-cast theoretical link usage

Internally, in each datacenter nodes are of three types *Exporter*, *Importer* and a normal node. A normal node transmits one message. An *Exporter* in D_k is in charge of forwarding G_k messages to each datacenter. That is $G_K * (D-1)$ in addition to generating its own message which means that an *Exporter* transmits $G_K * (D-1)+1$ messages. An *Importer* receives from each *Exporter* in other datacenters and forwards the received messages $N-G_k$ times. In addition, an *Importer* generates its own messages which means that an *Importer* transmits $N-G_k+1$ messages. Finally, each node receives $N-1$ messages, a message from each node. A link connecting two datacenters D_j and D_k is used to forward messages from the *Exporter* of D_j into the *Importer* of D_k and on the other lane from the *Exporter* of D_k towards the *Importer* of D_j. Thus, the link usage is G_j on the first lane and G_k on the other.

The link usage of *MDC-cast* is compared to the link usage of *LCR* and *Ridge* and shown in table 2.2. We observe from the table that *MDC-cast* uses inter datacenter links less than *LCR* and *Ridge*.

		Links (Inter-datacenter)	Cables (Intra-datacenter)
LCR	IN	$N-1$	$N-1$
	OUT	0	$N-1$
Ridge	IN	$G_k * G_j$	$N-1$
	OUT	$G_j * G_k$	$N-1$
MDC-cast	IN	G_k	$N-1$
	OUT	G_j	Exporter: $G_k * (D-1)+1$ Importer: $N-G_k+1$ Others: 1

Table 2.2: MDC-cast theoretical link usage

2.1.2 Message ordering

For ordering messages, *MDC-cast* uses the well-known fixed-sequencer pattern [DSU04]. More precisely, a fixed *Sequencer* is in charge of ordering messages by assigning a sequence number to each message. The *Sequencer* is elected by other processes using a group membership system explained in section 2.1.4.

2.1.3 Pseudo-code of the dissemination and ordering mechanisms

The pseudo-code is provided in Figure 2.3. The protocol works as follows. Each node multicasts a message in its datacenter (line 12), the *Exporter* of this datacenter forwards this message to other datacenters (line 20) and the message follows the path designed and explained before. The *Importer* is in charge of receiving messages from other datacenters and multicasting these messages to its datacenter (line 22). In order to ensure uniform agreement on message delivery, the sequencer assigns a novel and unique ID for each message (line 26) and multicasts an acknowledgment holding the new sequence number (line 28). In addition, every node acknowledges about the reception of messages and the sequence numbers associated with them (line 28). In order to deliver a message m, it should be received by all nodes first. So, for delivering a message m, each node waits m to be acknowledged by each other

node (lines 43 and 44). That way, a node is sure that m is already known and it holds a sequence number. If a message is not delivered after some amount of time (a specific timeout), the message is resent again (line 49).

Procedure executed by any process P_i in datacenter G_j

```
 1: procedure initialize(initial_view)
 2:    pendings[] ← ∅
 3:    seqnos[] ← ∅
 4:    acks[][] ← ∅
 5:    snToDeliver ← 0
 6:    sequencer = p_0
 7:    exporter = p_s where s = min k, p_k ∈ G_j
 8:    importer = p_t where t = max k, p_k ∈ G_j
 9:    sn ← 0

10: procedure utoBroadcast(m)
11:    id_m ← hash(p_i, m)
12:    multicast ⟨DATA, id_m, m⟩ to G_j
13:    if P_i = exporter then
14:       forward ⟨DATA, id_m, m⟩ to all Importers
15:    pendings[id_m] ← m
16:    SetTimeout ⟨id_m⟩

17: upon Receive ⟨DATA, id_m,m⟩ from P_j do
18:    if P_j ∈ G_j and P_i = exporter then
19:       for each imp ∈ Importers do
20:          forward ⟨DATA, id_m, m⟩ to imp
21:    if P_j ∉ G_j and P_i = importer then
22:       multicast ⟨DATA, id_m, m⟩ to G_j
23:    if P_i = sequencer then
24:       if ∄ seqnos[id_m] then
25:          seqnos[id_m] ← sn
26:          sn ← sn + 1
27:          acks[id_m][p_i] = 1
28:       multicast ⟨ACK, id_m, seqnos[id_m]⟩ to G_j
29:    pendings[id_m] ← m

30: upon Receive ⟨ACK, id_m, sn_m⟩ from P_j do
31:    if P_j ∈ G_j and P_i = exporter then
32:       for each imp ∈ Importers do
33:          forward ⟨ACK, id_m, m⟩ to imp
34:    if P_j ∉ G_j and P_i = importer then
35:       multicast ⟨ACK, id_m, m⟩ to G_j
36:    if P_j = sequencer and ∃ pendings[id_m] then
37:       multicast ⟨ACK, id_m, sn_m⟩ to all processes
38:       acks[id_m][p_i] = 1
39:    seqnos[id_m] ← sn_m
40:    acks[id_m][p_j] = 1
41:    tryDeliver()

42: procedure tryDeliver()
43:    while ∃ id_m s.t. (seqnos[id_m] = snToDeliver and sum(acks[id_m]) = n) do
44:       Deliver(m)
45:       snToDeliver ← snToDeliver + 1
46:       pendings ← pendings − pendings[id_m]

47: upon Timeout⟨id_m⟩ do
48:    if ∃ pendings[id_m] then
49:       multicast ⟨DATA, id_m, pendings[id_m]⟩ to all processes
50:       SetTimeout ⟨id_m⟩
```

Figure 2.3: Pseudo-code of the dissemination and ordering mechanisms.

2.1.4 Membership management

Machines are prone to failures and crashes. In order to handle machines joining and leaving the system, the *MDC-cast* protocol is built on top of a group communication system [BJ87a] relying on an external perfect failure detector ($\mathscr{P}$) that guarantees strong accuracy (No process is suspected before it crashes) and strong completeness (Eventually every process that crashes is permanently suspected by every correct process) [CT96a].

Figure 2.4 presents the membership management subprotocol. When a process leaves or joins the group, the *view_change* procedure is triggered. Firstly, every process completes the execution of all other procedures (if any) which is described in Figure 2.3. Then it starts the recovery part of the view change procedure. The functions make use of two primitives Rsend (reliable send) and Rreceive (reliable receive) that implement reliable communication channels over TCP/IP. Each process synchronizes its pending lists by sending each message in its pendings along side with its sequence numbers (seqnos) to all processes in the new view (line 2).

Upon receiving these arrays, every process updates its own pendings and seqnos arrays using those received from all other processes (lines 17 and 19). Then, the processes send back an ACK_RECOVER message (line 20). Processes wait until they receive ACK_RECOVER messages from all processes (line 3) before sending an END_RECOVERY message to all (line 4). When a process receives END_RECOVERY messages from all processes (line 5), it means that the message is ready to be delivered when its sequence number matches the waited sequence number, snToDeliver (lines 21 to 26). So, after the view change procedure finishes, all processes belonging to the new view will have delivered the same messages in the same order. Each process then empties its pendings, seqnos and acknowledgments (acks) arrays (lines 8 to 10). Moreover, each process uses as new *Sequencer* the first process in the new view (line 11). Each datacenter uses the first process as the *Exporter* (line 12) and the last process as the *Importer* (line 13).

```
Procedures executed by any process P_i
 1: upon view_change(new_view) do
 2:    Rsend ⟨RECOVER, P_i, pendings, seqnos⟩ to all P_j ∈ new_view
 3:    Wait until received ⟨ACK_RECOVER⟩ from all P_j ∈ new_view
 4:    Rsend ⟨END_RECOVERY⟩ to all P_j ∈ new_view
 5:    Wait until received ⟨END_RECOVERY⟩ from all P_j ∈ new_view
 6:    forceDeliver()
 7:    view ← new_view
 8:    pendings[] ← ∅
 9:    seqnos[] ← ∅
10:    acks[][] ← ∅
11:    sequencer = first process in view
12:    Exporter = first process in its datacenter
13:    Importer = last process in its datacenter
14:    sn ← nextToDeliver

15: upon Rreceive ⟨RECOVER, pendings_{p_j}, seqnos_{p_j}⟩ from P_j do
        Z   for each [id_m] ∈ pendings_{p_j} do
16:        pendings[id_m] ← pendings_{p_j}[id_m]
18:        if ∃ seqnos_{p_j}[id_m] then
19:           seqnos[id_m] ← seqnos_{p_j}[id_m]
20:    Rsend ⟨ACK_RECOVER⟩ to p_j

21: procedure forceDeliver()
22:    for each id_m ∈ seqnos[id_m], ordered by increasing sequence number do
23:       if ∃ pendings[id_m] and seqnos[id_m] ≥ snToDeliver then
24:          Deliver(pendings[id_m])
25:          pendings ← pendings − pendings[id_m]
26:       snToDeliver ← seqnos[id_m] + 1
27:    for each id_m ∈ keys(pending[id_m]), ordered by increasing id_m do
28:       Deliver(pendings[id_m])
29:       pendings ← pendings − pendings[id_m]
```

Figure 2.4: Pseudo-code of the membership management sub-protocol.

2.2 Correctness of the protocol

In this section, we prove that *MDC-cast* is a uniform total order broadcast protocol. We proceed by successively proving that *MDC-cast* ensures the four properties mentioned at the beginning of Section 1.1: validity, integrity, uniform agreement and total order.

Lemma 1. Validity*: if a correct process P_i* Broadcast *a message m, then P_i eventually* Deliver *m.*

Proof. If a correct process P_i Broadcast a message m, m is added to P_i's pending list (Line 15 of Figure 2.3). If there is a membership change, m will be in the new view (Line 7 of Figure 2.4). This view change guarantees that pending messages enter a recovery procedure. Message are resent again and they are eventually delivered. Let us now consider the case when there is no membership change. When a process P_j receives a message m from a process P_k, this message is added to P_k's pending list (Line 29 of Figure 2.3). The dissemination procedure ensures that, in the free-failure case, m is forwarded to every process in the current view through *Exporters* and *Importers*. Thereafter, the sequencer assigns a new unique ID (Called Sequence Number) to m and sends an acknowledgment to every process. When a process P_j receives an acknowledgment from the sequencer, it multicasts an acknowledgment ACK and Broadcast it in order to inform all other processes about receiving m (Line 37 of Figure 2.3). When a process P_i receives an acknowledgment ACK about a message m from each process in the system which is already in the pending list, it knows that all other nodes have received m with its sequence number and m can be delivered. The message m is then delivered if its sequence number matches the expected sequence number (Line 44 of Figure 2.3). If m is not received by some process, then the sender P_i will wait for an acknowledgment for some period, consider it as undelivered and re-broadcast it after some timeout (Line 49 of Figure 2.3). Consequently, all messages undelivered will be delivered. Eventually, m is delivered.

□

Lemma 2. Integrity: *for any message m, any correct process P_j* Deliver *m at most once, and only if m was previously* Broadcast *by some correct process P_i.*

Proof. When a message m is Deliver, the sequence number is incremented by one (Line 45 of Figure 2.3) and the next message to be delivered should hold the new unique sequence number. Also, the message is removed from the pending list (Line 46 of Figure 2.3). Similarly, when there is a membership change, Line 17 of Figure 2.4 guarantees that process P_j will not deliver messages twice.

□

Lemma 3. Uniform Agreement: *if any process P_i* Deliver *any message m in the current view, then every correct process P_j in the current view eventually* Deliver *m.*

Proof. We consider a message m_k initiated by process P_k and delivered by P_i in the current view. We study two cases with and without membership change. On the first hand, if m_k is not delivered on a membership change. The protocol ensures that m_k is received and acknowledged by every correct process before being delivered by P_i. Each process sends acknowledgment for m_k and it will be transmitted to every correct process. Thus, every correct process delivers m_k when it becomes the first entity in the pending list (Line 44 of Figure 2.3). Consequently, all correct processes will eventually deliver m_k. On the other hand, if m_k is delivered by process P_i during a membership change, then P_i should have m_k in its pending list before executing line 24 or line 28 of Figure 2.4. However, correct processes synchronize their pending lists frequently. So, processes that has not delivered m_k yet still have m_k in their pending lists before executing line 6 of Figure 2.4. Consequently, all correct processes in the current view will eventually deliver m_k. □

Lemma 4. Total Order. *for any two messages m and m', if any process P_i* Deliver *m without having delivered m', then no process P_j* Deliver *m' before m.*

Proof. Let m be a message Deliver by P_i before m'. Then m should have been matched the expected sequence number before m' (Line 44 of Figure 2.3). This is

to say that the sequence number of m is smaller than the sequence number of m'. However, the sequencer is the only process that issues sequence numbers (Line 25 of Figure 2.3) in strictly increasing order (Line 26 of Figure 2.3). Thus, each process delivers messages according to the same ordering which is maintained by the sequencer and checked on the delivery (Line 43 of Figure 2.3). Hence, the message that holds the smaller sequence number will be delivered before on every process. Therefore, m will be delivered before m'. So, for any two messages m and m', if any process P_i Deliver m without having delivered m', then no process P_j Deliver m' before m. □

Theorem 2. Total Order. *MDC-cast is a uniform total order broadcast protocol.*

Proof. By Lemma 1, Lemma 2, Lemma 3, and Lemma 4, we can derive the fact that the *MDC-cast* protocol ensures validity, integrity, uniform agreement, and total order. Thus, it is a uniform total order broadcast protocol. □

2.3 Bandwidth Allocation Mechanism

2.3.1 The need for a bandwidth allocation mechanism

Since *MDC-cast* uses IP-multicasting (which is unreliable) inside datacenters, packets can get lost if the network throughput exceeds the available bandwidth over intra-datacenter cables. For instance, a cable of bandwidth $1Gb/s$ is capable of receiving packets up to $1Gb/s$ and drops additional packets. Figure 2.5 shows the overall throughput of a Gigabit network of five processes $\{P_1, P_2, P_3, P_4, P_5\}$ where each process IP-multicasts messages in a fixed throughput rate. The throughput is increasing until each process multicasts $250Mb/s$. This is due to the fact that each server is able to receive $1GB/s$. When the sending throughput is higher, the overall throughput dramatically decreases. This is what we observe for instance when each process tries to IP-multicast at throughput rate $400Mb/s$ (exceeding $150Mb/s$). This behavior motivates the development of a bandwidth allocation mechanism.

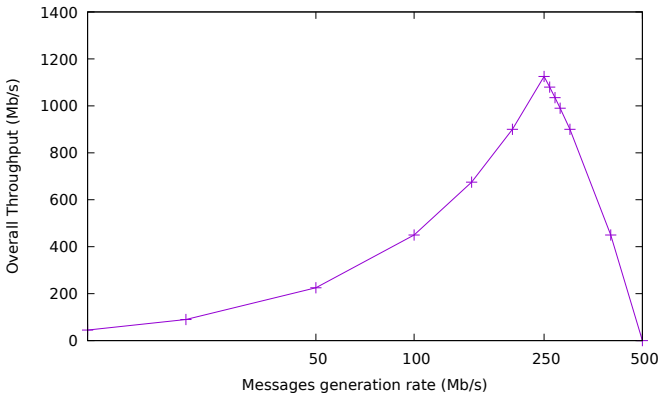


Figure 2.5: Throughput when five processes IP-multicast messages as a function of the individual multicasting rate.

Several group communication systems has been developed [BvR93, vRBM96, BVG+96] that have rate and congestion control mechanisms. They commonly address the IP-multicast issue using one of the three following techniques. In the first technique, they inhibit the throughput of each process [BC94]. The second technique is to imitate what is developed in TCP/IP by dynamically varying window sizes. They create virtual windows in the application space and IP-multicast them as ordinary messages. They distinguish between small windows for basic flow control and large windows for total ordering. The third technique is to to keep the gap between concurrent messages. This is done by limiting the number of concurrent senders in each group. This technique is used for large networks where the number of senders is relatively big.

In this section, we describe a bandwidth allocation mechanism that works as follows. The bandwidth allocation mechanism is managed by the *Sequencer*. It uses the TCP protocol to communicate with all machines in the system. Using TCP, the *Sequencer* asks each machine the throughput at which it wants to broadcast messages including resent messages. It then uses a max-min fair bandwidth allocation algorithm [Bou00] which shares at its core the same concept of inhibiting the throughput of each process [BC94]. We start by describing an example. We then provide the detailed pseudo-code.

2.3.2 Bandwidth allocation example

We illustrate this mechanism in the case of the topology described in Figure 1.10. In this topology, there are two datacenters: A has two nodes $\{P_1, P_2\}$ and B has three nodes $\{P_3, P_4, P_5\}$. Nodes inside each datacenter are interconnected by a $1Gb/s$ Ethernet switch. Moreover, for the purpose of this example, we consider that the inter-datacenter link $L1$ has a bandwidth of $300Mb/s$.

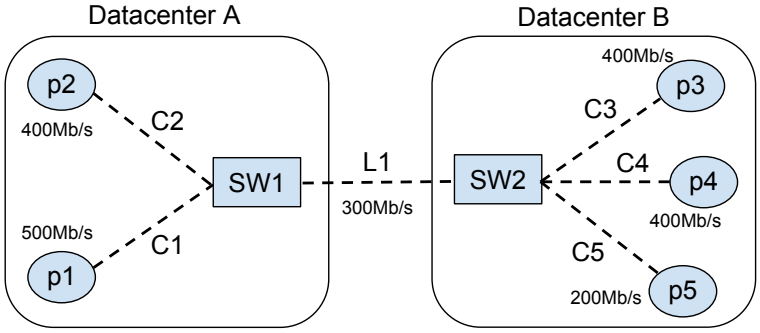


Figure 2.6: Bandwidth allocation example - intra-datacenters level.

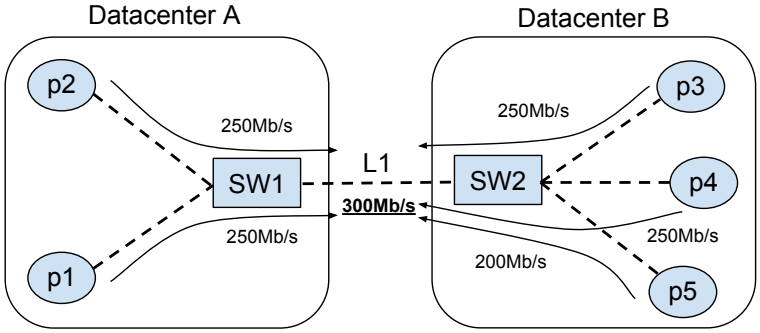


Figure 2.7: Bandwidth allocation example - inter-datacenters level.

As depicted in Figure 2.6, we assume that P_1 wants to broadcast $500Mb/s$, P_2, P_3 and P_4 want to broadcast $400Mb/s$, and P_5 wants to broadcast $200Mb/s$. In a first step, each machine deterministically computes the following fair bandwidth allocation: $250Mb/s$ for nodes $\{P_1, P_2, P_3, P_4\}$, and $200Mb/s$ for P_5 (see Figure 2.7). It is indeed not possible to allocate more than $250Mb/s$ to nodes $\{P_1, P_2, P_3, P_4\}$. Otherwise, P_5 would have to receive more than its bandwidth capability ($1Gb/s$), which is impossible.

In a second step, machines take into account the bandwidth of link $L1$ ($300Mb/s$ in our example). With the bandwidth allocation described in the previous paragraph, $L1$ would need to send $500Mb/s$ and receive $700Mb/s$, which is not possible provided that it has a capability of $300Mb/s$. Consequently, the two sites have to decrease their export to $300Mb/s$ each. This leads to decrease the bandwidth of $\{P_1, P_2\}$ to $150Mb/s$, and the bandwidth of $\{P_3, P_4, P_5\}$ to $100Mb/s$.

2.3.3 Detailed pseudo-code of the bandwidth allocation mechanism

Figure 2.8 provides the bandwidth allocation mechanism executed by standard processes (not the *Sequencer*). If a node wants to decrease its throughput (line 4), it decreases it directly then notifes the *Sequencer* about this decrease (line 10). If a node wants to increase its throughput, it sends a demand to the *Sequencer* (line 16) and waits for the response to update its throughput (line 18) if possible.

```
Procedures executed by any process P_i unless Sequencer
 1: procedure initialize(initial_view)
 2:     currentBW ← 0
 3:     required_bandwidth ← 0

 4: procedure decrease_BW(amount)
 5:     required_bandwidth ← currentBW − amount
 6:     currentBW ← required_bandwidth
 7:     if Sequencer then
 8:         BW_allocation()
 9:     else
10:         Rsend ⟨DECR, amount⟩ to Sequencer

11: procedure increase_BW(amount)
12:     required_bandwidth ← currentBW + amount
13:     if Sequencer then
14:         BW_allocation()
15:     else
16:         Rsend ⟨INCR, amount⟩ to Sequencer

17: upon Rreceive ⟨new_throughput⟩ from P_j do
18:     currentBW ← new_throughput
19:     required_bandwidth ← 0
```

Figure 2.8: Pseudo-code of the bandwidth allocation protocol executed by any process.

Figure 2.9 shows the pseudo-code executed by *Exporters*. In order to calculate the

available bandwidth between datacenters, *Exporters* push messages (that are to be forwarded) into a forwarding queue. During their transmission, they monitor the sending throughput. An *Exporter* will update the *Sequencer* (line 7) on significant changes (at least 10%).

Figure 2.10 gives the pseudo-code of the bandwidth allocation protocol executed by the *Sequencer*. On a throughput change, each node sends its required bandwidth to the *Sequencer*. The *Sequencer* is in charge of managing the throughput of each node. Firstly, the *Sequencer* stores the bandwidth requirements of other nodes in the bwRequirements array and their current bandwidth in the currentBW variable. In addition, the *Sequencer* stores the required_throughput field that is used when a node wants to increase its bandwidth. The required_throughput stores the required increase (before being processed). In order to manage the usage of links between datacenters, the *Sequencer* stores the available bandwidths of each link in the available_bandwidths field.

```
Procedures executed by any Exporter
 1: procedure initialize(initial_view)
 2:     BW_of_adjacent_links ← default_bandwidth

 3: upon change_Available_bandwidth(BW_of_adjacent_links) do
 4:     if Sequencer then
 5:         BW_allocation()
 6:     else
 7:         Rsend ⟨BW_of_adjacent_links⟩ to Sequencer
```

Figure 2.9: Pseudo-code of the bandwidth allocation protocol executed by *Exporters*.

Before going into the details of the protocol, let us remark that the limit_cables function (line 30) implements a classical max-min fair bandwidth allocation algorithm [Bou00]. The only important point to mention is that it uses a variable, called link_availableBW, that represents the maximum capability of a network cable inside a datacenter. On the other hand, the limit_links function (line 21) covers the case of inter-datacenters links.

The *Sequencer* receives either an increase demand (line 11) or a decrease demand (line 8) from a standard node or the available bandwidth from the *Exporters* (line 14). If a node wants to increase or decrease its bandwidth, the *Sequencer* will update

the bwRequirements with the amount (lines 8 and 11) and then recompute the bandwidth allocation on all nodes and cables (lines 9 and 12). The new bandwidth allocation is then sent to all the nodes (line 44). Finally, the *Sequencer* updates its currentBW (line 20).

2.3.4 Illustration of the bandwidth allocation mechanism

We provide three illustrations of the bandwidth allocation protocol in Table 2.3, Table 2.4, and Table 2.5. We consider a system of two datacenters $g1$ and $g2$, where $g1$ has three nodes $\{p_0, p_1, p_2\}$ and $g2$ has two nodes $\{p_3, p_4\}$. P_0 is the *Exporter* of $g1$ (respectively P_3 the *Exporter* of $g2$), P_2 is the *Importer* of $g1$ (respectively P_4 the *Importer* of $g2$) and P_0 is the *Sequencer*. Nodes inside each datacenter are interconnected by a 1Gb/s Ethernet switch whereas we suppose that datacenters are connected to each other by a cable of 300Mb/s uniformly on the two lanes. In each table, we describe a set of steps that happen in the system and we illustrate how the different fields of the processes are updated. Initially, processes have a null bandwidth (currentBW is equal to 0 in Table 2.3, step S1).

In Table 2.3 we depict what happens when from this initial state P_1 calls increase_BW(400) and P_4 calls increase_BW(300). Processes reach a state (step S6) in which P_1 has its currentBW variable equal to 400Mb/s and P_4 has its currentBW variable equal to 300Mb/s. From that state (also depicted in Table 2.4, step S7), Table 2.4 depicts what happens when the *Exporter* of $g1$ declares that it wants to decrease its throughput. Processes reach a state (step S12) in which P_1 has its currentBW variable equal to 400Mb/s and P_4 has its currentBW variable equal to 300Mb/s. From that state (also depicted in Table 2.5, step S13), Table 2.5 depicts what happens when P_1 calls decrease_BW(200) and P_3 calls increase_BW(250). Processes reach a state (step S18) in which P_1 has its currentBW variable equal to 200Mb/s, P_3 has its currentBW variable equal to 150Mb/s and P_4 has its currentBW variable equal to 150Mb/s.

Procedures executed by the Sequencer of a system of n nodes

```
1: procedure initialize(initial_view)
2:     bwRequirements[] ← [0, ··· , 0]
3:     new_BWs[] ← [0, ··· , 0]
4:     BW_summation ← default_bandwidth ∗ n/(N − 1)
5:     link_capacity[] ← [0, ··· , 0]
6:     currentBW ← 0

7: upon Rreceive ⟨Decr, amount⟩ from P_i or decrease amount do
8:     bwRequirements[P_i] ← bwRequirements[P_i] − amount
9:     BW_allocation()

10: upon Rreceive ⟨Incr, amount⟩ from P_i or increase amount do
11:    bwRequirements[P_i] ← bwRequirements[P_i] + amount
12:    BW_allocation()

13: upon Rreceive ⟨BW_of_adjacent_links⟩ from Exporter of datacenter g_j or change BW_of_adjacent_links
    do
14:    link_capacity[g_j] ← BW_of_adjacent_links
15:    BW_allocation()

16: function BW_allocation()
17:    limit_links()
18:    limit_cables()
19:    distribute_new_allocation()
20:    currentBW ← new_BWs[Sequencer]

21: function limit_links()
22:    group_members ← all the members of group G
23:    temp_link_capacity ← link_capacity
24:    for P_j in group_members do
25:       if bwRequirements[P_j] ≤ temp_link_capacity[G]/size(group_members) then
26:          group_members ← group_members − P_j
27:          temp_link_capacity[G] ← temp_link_capacity[G] − bwRequirements[P_j]
28:          new_BWs[P_j] ← bwRequirements[P_j]
29:          allocated = true

30: function limit_cables()
31:    nodes ← the biggest (N − 1) values in bwRequirements
32:    link_availableBW ← B
33:    do
34:    allocated = false
35:    for P_j in nodes do
36:       if bwRequirements[P_j] ≤ link_availableBW/size(nodes) then
37:          nodes ← nodes − P_j
38:          link_availableBW ← link_availableBW − bwRequirements[P_j]
39:          new_BWs[P_j] ← bwRequirements[P_j]
40:          allocated = true
41:    while nodes ≠ ∅ and allocated = true do
42:       if P_i ∈ nodes then
43:          new_BWs[P_j] ← link_availableBW/size(nodes)

44: function distribute_new_allocation()
45:    nodes ← all processes unless Sequencer
46:    for P_j in nodes do
47:       Rsend ⟨new_BWs[P_j]⟩ to P_j
```

Figure 2.10: Pseudo-code of the bandwidth allocation protocol executed by the *Sequencer*.

Table 2.3: A first example execution of the bandwidth allocation protocol.

step	*group*	*process*	*bwRequirements*	*currentBW*	*required_throughput*	*BW_of_adjacent_links*	*link_capacity*	*new_BWs*	
S1	g1	p_0 † ‡	$[0,0,0,0,0]$	0	0	300	$[300,300]$	$[0,0,0,0,0]$	Initial state
		p_1	–	0	0	–	–	–	
		p_2*	–	0	0	–	–	–	
	g2	p_3†	–	0	0	300	–	–	
		p_4*	–	0	0	–	–	–	
S2	g1	p_0	$[0,0,0,0,0]$	0	0	300	$[300,300]$	$[0,0,0,0,0]$	p_1 calls increase_BW(400) p_4 calls increase_BW(300)
		p_1	–	0	**400**	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	0	**300**	–	–	–	
S3	g1	p_0	$[0,\mathbf{400},0,0,\mathbf{300}]$	0	0	300	$[300,300]$	$[0,0,0,0,0]$	*Sequencer* Rreceives $\langle \text{INCR}, 400\rangle_{p_1}$ *Sequencer* Rreceives $\langle \text{INCR}, 300\rangle_{p_4}$
		p_1	–	0	400	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	0	300	–	–	–	
S4	g1	p_0	$[0,\mathbf{100},0,0,\mathbf{0}]$	0	0	300	$[300,300]$	$[0,\mathbf{300},0,0,\mathbf{300}]$	*Sequencer* calls limit_links() for $g1$ *Sequencer* calls limit_links() for $g2$
		p_1	–	0	400	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	0	300	–	–	–	
S5	g1	p_0	$[0,100,0,0,0]$	0	0	300	$[300,300]$	$[0,300,0,0,300]$	*Sequencer* calls limit_cables() for $g1$ *Sequencer* calls limit_cables() for $g2$
		p_1	–	0	400	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	0	300	–	–	–	
S6	g1	p_0	$[0,100,0,0,0]$	0	0	300	$[300,300]$	$[0,300,0,0,300]$	p_1 Rreceives $\langle 300\rangle_{Sequencer}$ p_4 Rreceives $\langle 300\rangle_{Sequencer}$
		p_1	–	**300**	**0**	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	**300**	**0**	–	–	–	

Table 2.4: A second example execution of the bandwidth allocation protocol.

step	*group*	*process*	*bwRequirements*	*currentBW*	*required_throughput*	*BW_of_adjacent_links*	*link_capacity*	*new_BWs*	
S7	g1	p_0	$[0, 100, 0, 0, 0]$	0	0	300	$[300, 300]$	$[0, 300, 0, 0, 300]$	Initial state (equal to S6 in Table 2.3)
		p_1	–	300	0	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	300	0	–	–	–	
S9	g1	p_0	$[0, 100, 0, 0, 0]$	0	0	**500**	$[\mathbf{500}, 300]$	$[0, 300, 0, 0, 300]$	p_0 calls change_Available_bandwidth$\langle 500 \rangle$
		p_1	–	300	0	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	300	0	–	–	–	
S10	g1	p_0	$[0, \mathbf{0}, 0, 0, 0]$	0	0	500	$[500, 300]$	$[0, \mathbf{400}, 0, 0, 300]$	*Sequencer* calls limit_links() for $g1$
		p_1	–	300	0	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	300	0	–	–	–	
S11	g1	p_0	$[0, 0, 0, 0, 0]$	0	0	500	$[500, 300]$	$[0, 400, 0, 0, 300]$	*Sequencer* calls limit_cables() for $g1$
		p_1	–	300	0	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	300	0	–	–	–	
S12	g1	p_0	$[0, 0, 0, 0, 0]$	0	0	500	$[500, 300]$	$[0, 400, 0, 0, 300]$	p_1 Rreceives $\langle 400 \rangle_{Sequencer}$
		p_1	–	**400**	0	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	300	0	–	–	–	

Table 2.5: A third example execution of the bandwidth allocation protocol.

step	*group*	*process*	*bwRequirements*	*currentBW*	*required_throughput*	*BW_of_adjacent_links*	*link_capacity*	*new_BWs*	
S13	g1	p_0	$[0, 0, 0, 0, 0]$	0	0	500	$[500, 300]$	$[0, 400, 0, 0, 300]$	Initial state (equal to S12 in Table 2.4)
		p_1	–	400	0	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	0	300	–	–	
		p_4	–	300	0	–	–	–	
S14	g1	p_0	$[0, 0, 0, 0, 0]$	0	0	500	$[500, 300]$	$[0, 400, 0, 0, 300]$	p_1 calls decrease_BW(200) p_3 calls increase_BW(250)
		p_1	–	**200**	**−200**	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	**250**	300	–	–	
		p_4	–	300	0	–	–	–	
S15	g1	p_0	$[0, \mathbf{-200}, 0, \mathbf{250}, 0]$	0	0	500	$[500, 300]$	$[0, 400, 0, 0, 300]$	*Sequencer* Rreceives $\langle \text{DECR}, 200 \rangle_{p_1}$ *Sequencer* Rreceives $\langle \text{INCR}, 250 \rangle_{p_3}$
		p_1	–	200	−200	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	250	300	–	–	
		p_4	–	300	0	–	–	–	
S16	g1	p_0	$[0, \mathbf{0}, 0, \mathbf{100}, \mathbf{150}]$	0	0	500	$[500, 300]$	$[0, \mathbf{200}, 0, \mathbf{150}, \mathbf{150}]$	*Sequencer* calls limit_links() for $g1$ *Sequencer* calls limit_links() for $g2$
		p_1	–	200	−200	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	250	300	–	–	
		p_4	–	300	0	–	–	–	
S17	g1	p_0	$[0, 0, 0, 100, 150]$	0	0	500	$[500, 300]$	$[0, 200, 0, 150, 150]$	*Sequencer* calls limit_links() for $g1$ *Sequencer* calls limit_cables() for $g2$
		p_1	–	200	−200	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	0	250	300	–	–	
		p_4	–	300	0	–	–	–	
S18	g1	p_0	$[0, 0, 0, 100, 150]$	0	0	500	$[500, 300]$	$[0, 200, 0, 150, 150]$	p_1 Rreceives $\langle 200 \rangle_{Sequencer}$ p_3 Rreceives $\langle 150 \rangle_{Sequencer}$ p_4 Rreceives $\langle 150 \rangle_{Sequencer}$
		p_1	–	**200**	−200	–	–	–	
		p_2	–	0	0	–	–	–	
	g2	p_3	–	**150**	250	300	–	–	
		p_4	–	**150**	0	–	–	–	

2.4 Optimizations

In order to improve the performance of *MDC-cast*, we implemented an optimization that is very often used in existing protocols: it consists in batching small messages in order to improve the network usage. The second optimization is relative to retransmitted messages. Basically, when a process P_i sends a message m, some $ACKs$ may get lost as well as m itself. In this case, P_i will wait the timeout and resend m again. In the datacenter environment, we propose to organize the retransmission mechanism in three steps. From P_i to *Exporter*, then from *Exporter* to another datacenter (The path between datacenters), then inside the receiving datacenter.

Assume that a message m is generated by a process P_i located in a datacenter D. The message firstly is IP-multicast in D and then forwarded from D to other datacenters. If m is received patially by some nodes in a datacenter D', but not received by all nodes in D', then there is no need to get the message back from D. It is sufficient to get it from a process that already received it in D' (At least the importer has received it). We define two timeouts, $intra\_timeout$ which is short because it is used inside the datacenter, and *timeout* for the complete procedure of the message broadcast. P_i sends a message and waits ACKs from its datacenter and from other datacenters. It will wait $intra\_timeout$ to get ACKs from its datacenter and wait *timeout* to get ACKs from other processes that should pass through the *Importer*. Similarly, if *Importer* receives at least an ACK from another datacenter, this means that m has arrived to that datacenter via its *Importer*. This *Importer* will wait to get ACKs from all members of the local datacenter D'. After $intra\_timeout$, the *Importer* of D' will re-multicast the message again.

2.5 Conclusion

Existing optimal total order broadcast protocols target fully switched networks. In this chapter, we have presented *MDC-cast*, a Total Order Broadcast protocol that specifically targets multi-datacenters environments. *MDC-cast* optimizes the use of inter-datacenters links and decrease the impact of background traffic. In addition is makes use of a bandwidth allocation mechanism to ensure that the network is not congested.

Chapter 3

Performance Evaluation

In this chapter, we assess the performance of *MDC-cast* against two state of the art protocols: *LCR* [GLPQ10] and *Ridge* [BCP15]. Firstly, we present a theoretical assessment in section 3.1. Then, we describe the experimental setup in section 3.2 and our methodology to ensure that we achieve the best possible performance in Section 3.3. Finally, we present the experimental evaluation results in section 3.4. Finally, the chapter is concluded in 3.5.

3.1 Theoretical assessment

MDC-cast achieves the optimal throughput defined in [GLPQ10] in both cluster and multi-datacenter environments. In this section, we prove that the throughput of *MDC-cast* is optimal assuming that there is no background traffic. Then, we study *MDC-cast* in the presence of background traffic.

3.1.1 Optimal throughput

First, we show that the throughput of *MDC-cast* is optimal and that no other broadcast protocol can obtain strictly higher throughput. We do this by relying on the upper bound of the performance of any Total Order Broadcast protocol defined

in [GLPQ10] and show that *MDC-cast* matches this bound. Then, we show a case study and analyze it. First, we recall the maximum throughput for a Total Order Broadcast protocol in a system with n processes stated in [GLPQ10]:

$$\mu_{max} = \begin{cases} n/(n-1) & \text{if there are } n \text{ senders} \\ 1 & \text{otherwise} \end{cases} \tag{3.1}$$

Theorem 3. *The throughput of MDC-cast matches the optimal throughput for Total Order Broadcast algorithms.*

Proof. The Broadcast is composed of three phases: (1) *Phase I*: The IP-multicast inside the sender's datacenter (Also called Origin Datacenter). (2) *Phase II*: The export phase, which is to forward the message from one datacenter to other datacenters. The export phase is to forward the message through the *Exporter* of the origin datacenter to each *Importer* in other datacenters. (3) *Phase III*: The IP-multicast from each *Importer* to other nodes inside its datacenter. We first consider the case with N senders. In a system of D datacenters each containing G nodes, *phase I* needs $G-1$ rounds to be accomplished where each node multicast one message. Then, *phase II* and *phase III* run simultaneously. An *Importer* is able to receive messages from other datacenters in $(D-1)*G$ rounds. As well, non importers are able to receive IP-multicasted messages coming from other datacenters in $(D-1)*G$ rounds. So, *phase II* and *phase III* needs $(D-1)*G$ rounds to be accomplished. Hence, delivering N messages needs $N-1$ ($D*G-1$ rounds) rounds. In the case with less than N senders, a non-sender process receives one and only one message per round. Thus, it delivers one message per round. So, the maximum throughput is equal to 1 for less than N senders. □

Illustration of the throughput optimality

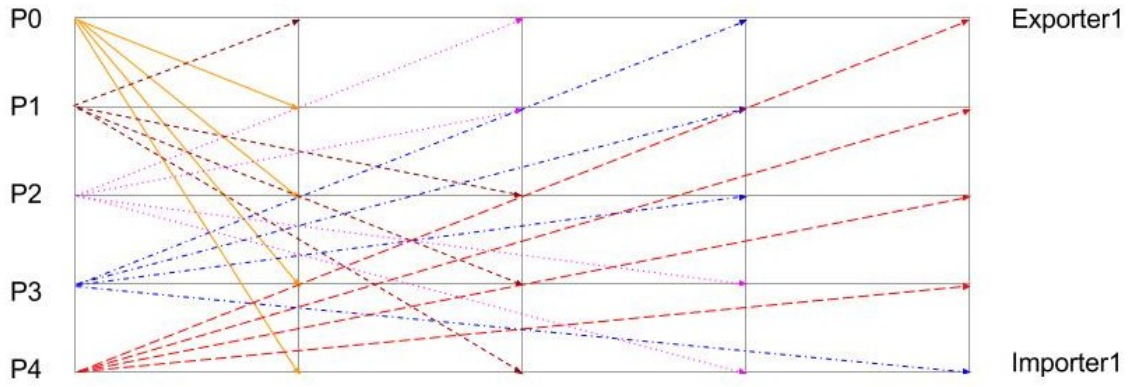


Figure 3.1: Phase I of *MDC-cast*

We study an illustrating scenario to clarify the issue. Figure 3.1 illustrates the first phase of the Broadcast. The figure studies a network composed of three datacenters each containing five processes running *MDC-cast* algorithm where each process Broadcasts one message. The figure shows the dissemination pattern inside only one datacenter because other datacenters perform the same pattern and Broadcast simultaneously. The first datacenter contains five nodes $\{P0, P1, P2, P3\ and\ P4\}$ where P_0 is the *Exporter* and P_4 is the *Importer*. Each round is represented by a vertical slot while a horizontal slot represents a process. IP-multicast allows the node to send a message to several node at the same time, but a node is not able to receive two messages in the same time slot. The first round shows the Broadcast of five nodes simultaneously. The second round shows that $\{P1, P2, P3\ and\ P4\}$ receive the message Broadcasted by P_0. P_0 is able to receive a message, it receives the message of P_1. Nodes receive the message Broadcasted by P_1 in the third round while P_1 receives the message of P_2. In the fourth round, nodes receive the message Broadcasted by P_2 while P_2 receives the message of P_3. In the last round, nodes receive the remaining messages. The message Broadcasted by P_4

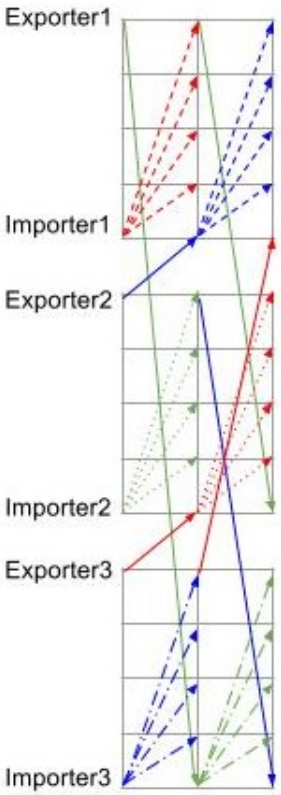


Figure 3.2: *Phase II* and *phase III* of *MDC-cast*

is received by all nodes while P_4 receives the message of P_3. Thus in five rounds, 15 messages are IP-multicasted each in its datacenter.

Figure 3.2 discusses both the second and the third phase of the dissemination pattern because the two phases are performed simultaneously. The second and the third phase are composed of five identical stages, thus the figure depicts just the first stage. Roughly speaking, each *Exporter* is in charge of exporting five messages to each datacenter (i.e. 10 messages) and each *Importer* is in charge of IP-multicasting five messages coming from each datacenter (i.e. 10 messages). The figure discusses the export procedure of one message to each datacenter and how the importer receives the message and IP-multicast it. *Exporter1* forwards a message to *Importer2* and *Importer3* which needs two consecutive rounds because over TCP/IP, a process is unable to send more than one message per round. Respectively, *Exporter2* forwards a message to *Importer1* and *Importer3* in two rounds and *Exporter3* forwards a message to *Importer1* and *Importer2* in two rounds too. When an *Importer* receives a message, it IP-multicasts it to each node inside it datacenter. This stage is repeated five times. Thus, the broadcast of 15 messages costs 14 rounds (4 rounds for *phase I*, and 10 rounds for both *phase II* and *phase III*).

3.1.2 Performance in Presence of Background Traffic

Links connecting data centers are impacted by background traffic and prone to saturation. In this section, we study the impact of the background traffic on links connecting datacenters. The remaining available bandwidth on a lane of link connecting two datacenters is noted AVB. Figure 3.3 depicts an example of a network composed of five nodes distributed over two datacenters where the link connecting the two datacenters is exposed to background traffic.

We consider a system S composed of N processes distributed over several datacenters each containing G nodes. The overall throughput of S when running *MDC-cast*

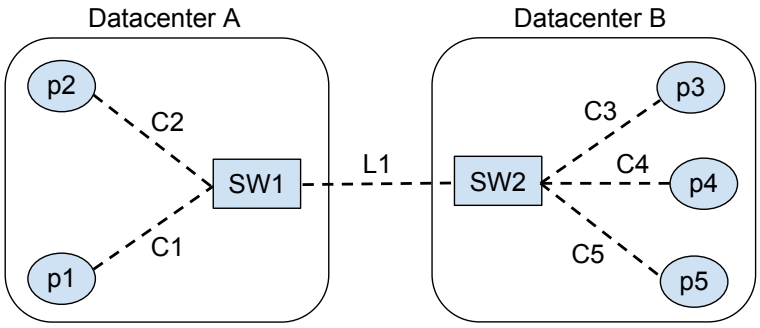


Figure 3.3: A network composed of five nodes distributed over two datacenters

protocol with N senders is stated as follows:

$$\begin{cases} \text{BW*(N/(N-1))} & \text{if } AVB \geq \min_{AVB} \\ \min_{AVB} * N/G_j & \text{otherwise} \end{cases} \tag{3.2}$$

where G_j is the number of nodes inside the datacenter and $\min_{AVB} = G_j * BW/(N-1)$.

The equation can be considered in three distinct situations: (1) if there is no background traffic on the links in between datacenters, (2) If there is background traffic but not affecting the overall throughput of the system (i.e. the AVB on all links is sufficient to achieve the optimal throughput), and (3) If there is background traffic on the links in between datacenters that affects the overall throughput of the system.

The first two situations together mean that if the available bandwidth on links connecting datacenters is more than or equal to the needs, then it is possible to achieve the optimal throughput (the first line of the equation); otherwise the overall throughput is proportionally affected (The second line of the equation).

3.1.3 Latency

The latency of *LCR* and *Ridge* in cluster environments is presented in table 1.1. Nonetheless, their latency is not the same in multi-datacenter environments due to the heterogeneity of links. The latency of a message transmitted between two

processes each in a distinct datacenter is represented by O. We assume that the time needed to send messages between processes in the same datacenter is homogeneous but negligible. This is due to the fact that in our experiments, the latency intra-datacenter almost does not exceed 5% of the latency inter-datacenters.

	One sender	N senders
LCR	$(D-1)*O$	$(N-1)*O$
Ridge	O	
MDC-cast	O	

Table 3.1: The theoretical assessment of the latency of *LCR*, *Ridge* and *MDC-cast* in multi-datacenter environments.

In table 3.1, we provide the latency of *LCR*, *Ridge* and *MDC-cast* in multi-datacenter environments. The studied network is composed of N homogeneous nodes distributed over D datacenters. Nodes inside datacenters are interconnected by switches while datacenters are interconnected over one or several routers. The table studies two cases (1) when one sender only Broadcasts a message and (2) when each node is broadcasting a message. First, we study when just one sender Broadcast a message. Using *LCR*, when a process P Broadcast a message m, it sends m to its successor process. m is forwarded from each process to its successor until it arrives to the predecessor process of P. The message traverses all the datacenters and returns back to its origin datacenter (The datacenter that contains the sender). Thus, the time needed to disseminate the message is $(D-1)*O$. Using *Ridge*, when a process P Broadcast a message m, it sends it directly (and in parallel) to all other processes. So, the overall latency of the Broadcast of m using *Ridge* is O. Using *MDC-cast*, the Broadcast is done in three phases: a message is IP-multicast, then forwarded to the *Importers* in other datacenters which IP-multicast them in their datacenters. As for *Ridge*, the sending of m to all other datacenters can be performed in parallel. Consequently, the overall latency of the Broadcast of m using *MDC-cast* is O.

On the other hand, in the case of N senders, the three algorithms have the same latency $(N-1)*O$. Because the protocols have optimal throughput, they all need

$N-1$ rounds to deliver N messages. Thus, the time needed to Broadcast N messages is $(N-1)*O$.

3.2 Experimental Setup

The experiments focus on the failure-free case that is the common case in the targeted systems. We describe firstly the experimental setup of the testbed before characterizing the performance of the three algorithms from two points of view: throughput and latency.

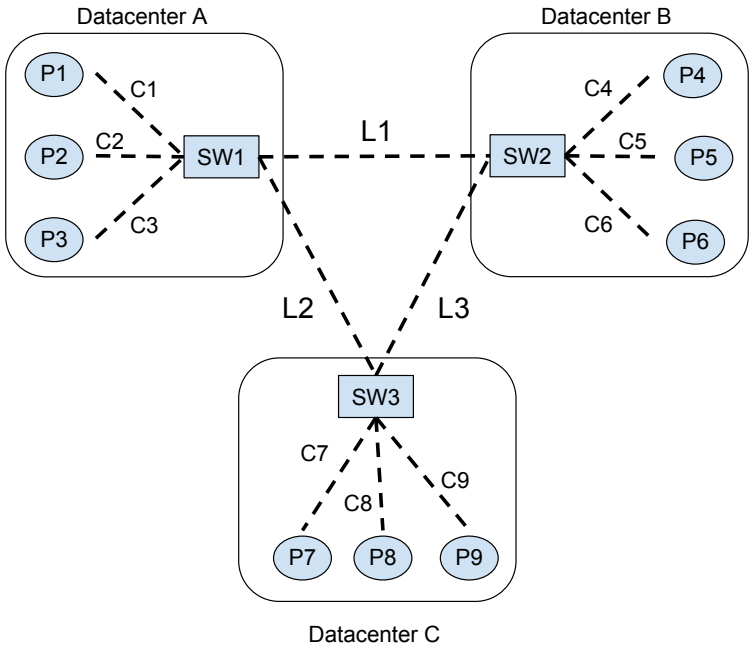


Figure 3.4: The basic topology used in the experiments

Experiments were conducted on Grid'5000 [gri17][1]. We use servers comprising 8 cores, 16GB of RAM, and running the Linux 3.2.0-4-amd64 kernel. Nodes belonging to the same datacenter are interconnected by a Gigabit switch. The raw bandwidth over IP is measured using Iperf [DEM+10]: 940Mb/s between any two

[1]Experiments presented in this chapter were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

servers. Links between sites are shared by all machines of the Grid'5000 testbed. Grid'5000 is a real testbed, hence the background traffic is almost variable and unstable. In order to obtain stable and deterministic results, we limit to $500Mb/s$ the bandwidth of inter-datacenters links using the *tc* Linux tool. In order to ensure a fair evaluation among the three prototypes, we implemented *MDC-cast*, *Ridge* and *LCR* using the same networking libraries (in C++). Indeed, there exists some group communication toolkits that we could have relied on to design our system such as Spread [ADMA+04b] but we preferred to create our own communication mechanism to have stronger control over packets transmission. Our performance evaluations basically study a network topology of nine nodes distributed over three datacenters. Nodes inside each datacenter are intended to be within the same cluster and connected to the same switch. Datacenters are interconnected by shared links. Figure 3.4 illustrates the topology we use in our experiments.

Since experiments are done on distributed servers, we adjusted the window size to be able to scale up to the needed size. The presented experiments were preceded by a warm-up phase to ensure that all links and buffers were filled up.

3.3 Configuration Methodology

3.3.1 The Need For a Configuration Methodology

As we have seen in the previous sections and chapters, the performance of a protocol are often strongly depending on the bandwidth available on links inter and intra-datacenters. Let us consider a system S composed of five processes $\{p_1, ...p_5\}$, where each process p_i is connected to the network via a cable called C_i respectively. Processes are distributed over two datacenters A *and* B, interconnected by link $L1$. Datacenter A contains processes p_1 *and* p_2, while datacenter B contains processes p_3, p_4 *and* p_5. Processes inside the same datacenter are interconnected by a Gigabit local area network, while datacenters communicate over a wide area network. We

assume that the network is shared and exposed to background traffic. Figure 3.5 shows S, the described system with the available bandwidth on each link in Mb/s. A link contains two lanes: lane IN which is colored blue and underlined, and lane OUT which is colored red and bolt.

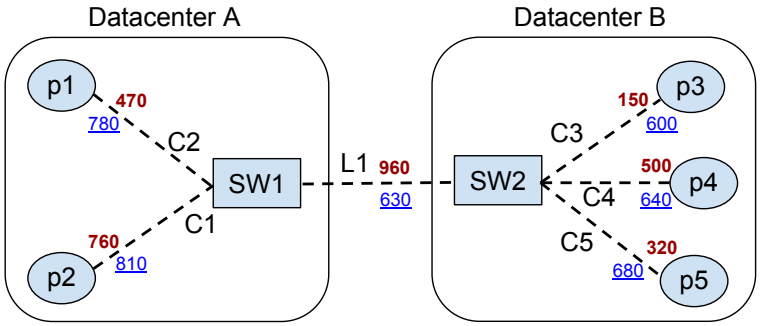


Figure 3.5: The available bandwidth on links in a system of two datacenters.

We assume that *MDC-cast* is running over S. Unfortunately, the throughput required by *MDC-cast* (showed in Figure 3.6) is different than the available bandwidth on links. The theoretical overall throughput of S with N senders if not exposed to background traffic should be $1.125Gb/s$ which is the optimal throughput. In the depicted case, the bottleneck link is $C3$ that has an available bandwidth of $150Mb/s$, but is supposed to transmit messages at $1000Mb/s$. Our goal was to design a methodology ensuring that we would find the deployment (i.e. where to place importers and exporters in the case of *MDC-cast*) that achieves the best possible performance.

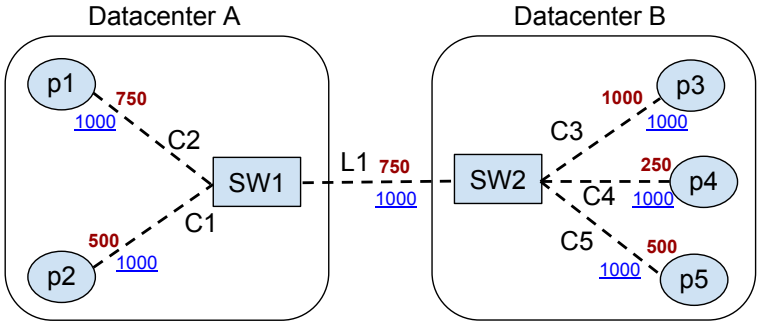


Figure 3.6: The Required bandwidth on links when using *MDC-cast*.

3.3.2 Describing the Configuration Methodology Using an Example

In this section, we will describe the configuration methodology using an example. More precisely, we will explain how to tune the number of *Acceptors* in the *Ridge* protocol. The configuration methodology comprises three successive steps:

- **max_lane:** we first detect the bottleneck link and more specifically the bottleneck lane of the link [2] which we call max_lane. That value represents the number of messages that transit on the bottleneck link for each step of an algorithm.
- **max_gen:** we then calculate the maximum throughput that could be generated from the bottleneck node if it initiates message broadcasts. This throughput, noted max_gen is equal to BW/max_lane.
- **possible_throughput:** we finally calculate the overall possible throughput, if several nodes are broadcasting messages: $possible\_throughput = max\_gen*$ G (where G is the number of generators).

We provide an illustrative scenarios to explain our methodology. Let us consider a system of nine servers interconnected by a Gigabit fully-switched network. Let us assume that the system runs *Ridge* [BCP15]. Finally, let us assume that there are five *Acceptors* named $\{A_1, ...A_5\}$ and four *Learners* named $\{L_1, ...L_4\}$. Briefly, in *Ridge*, when an *Acceptor* receives a request, it forwards it to its successor *Acceptor* and so on until getting a majority of *Acceptors* which is three successive nodes in our example. Then the message will be sent to a *Learner* which the load balancer chooses. This *learner* is in charge of broadcasting the message using TCP/IP messages to all other *Learners*.

So, when $A1$ receives a new request, it forwards it to $A2$ as shown in figure 3.7. $A2$ forwards it to $A3$ and $A3$ notices that the message is acknowledged by a majority

[2] A lane is a term commonly used for one direction of a link

of *Acceptors*. So, $A3$ forwards the message to a *Learner* that, we assume, is $L1$. Finally, $L1$ sends the message to $L2$, $L3$ and $L4$.

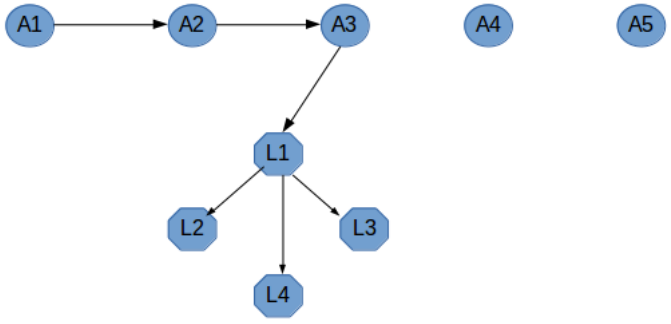


Figure 3.7: The *Theoretical_Needs* of *Ridge*

The link usage for this scenario is shown in table 3.2. As we can notice, the link $L1$ is used three times due to the fact that it sends the message to the three *Learners*: $L2$, $L3$ and $L4$. So, we conclude that OUT of $L1$ is the max_lane.

	A1	A2	A3	A4	A5	L1	L2	L3	L4
In	0	1	1	0	0	1	1	1	1
Out	1	1	1	0	0	3	0	0	0

Table 3.2: Links usage of a system running *Ridge* with one sender: $A1$

We can conclude that the maximum throughput of this system, if limited to one *Acceptor* working at any given time, is $333Mb/s$:

- $max\_lane = 3$
- $max\_gen = (1Gb/s)/3 = 333Mb/s$
- $possible\_throughput = 1 * 333Mb/s = 333Mb/s$

Let us now assume that there are five *Acceptors* concurrently working. The load balancer is going to distribute messages over four *Learners* uniformly. There will remain one message which will be forwarded to a *Learner* that already receives a message before. Hence, each *Learner* will receive one message unless one of them

will receive two messages, let us say it is $L1$. So, $L1$ is going to broadcast two messages and the OUT lane of the link connected to $L1$ will be used to send six messages as shown in table 3.3.

	A1	A2	A3	A4	A5	L1	L2	L3	L4
In	2	2	2	2	2	5	5	5	5
Out	3	3	3	3	3	6	3	3	3

Table 3.3: Links usage of a system running *Ridge* with five senders

In this scenario, the maximum throughput of this system will be $834Mb/s$:

- $max\_lane = 6$
- $max\_gen = (1Gb/s)/6 = 166Mb/s$
- $possible\_throughput = 5 * 166Mb/s = 834Mb/s$

As a conclusion, the maximum throughput of the system depends on the network topology and the way the algorithm is deployed

3.4 Experimental Evaluation

To assess the throughput of the three protocols, we deploy N nodes that initiate and broadcast messages. The message size is fixed to 10KB. Each node periodically computes and reports the delivery throughput. The throughput is calculated as the ratio of delivered bytes over the time elapsed since the end of the warm-up phase. The plotted throughput is the average of the values computed by each process.

Throughput comparison against *LCR* and *Ridge*

We first compare the performance of *MDC-cast* against *LCR* and *Ridge*. The results are shown in Figure 3.8. We observe that *MDC-cast* achieves a similar throughput to *LCR* in the one datacenter setup, and a much higher throughput than other protocols

in a multi-datacenters setup. In the multi-datacenters setup, the throughput of *LCR* and *Ridge* are degraded by about 50%. This is due to the impact of the background traffic. *MDC-cast* is not affected by this amount of background traffic (which is around $500Mb/s$). This result was expected and is explained by the fact that *MDC-cast* optimizes the utilization of inter-datacenters link, as explained in Section 1.3. If the the background traffic on links varies slightly, *MDC-cast* is not affected. Actually, *MDC-cast* in such a topology is not affected until the available bandwidth on links goes down $375Mb/s$ as the equation 3.2 states.

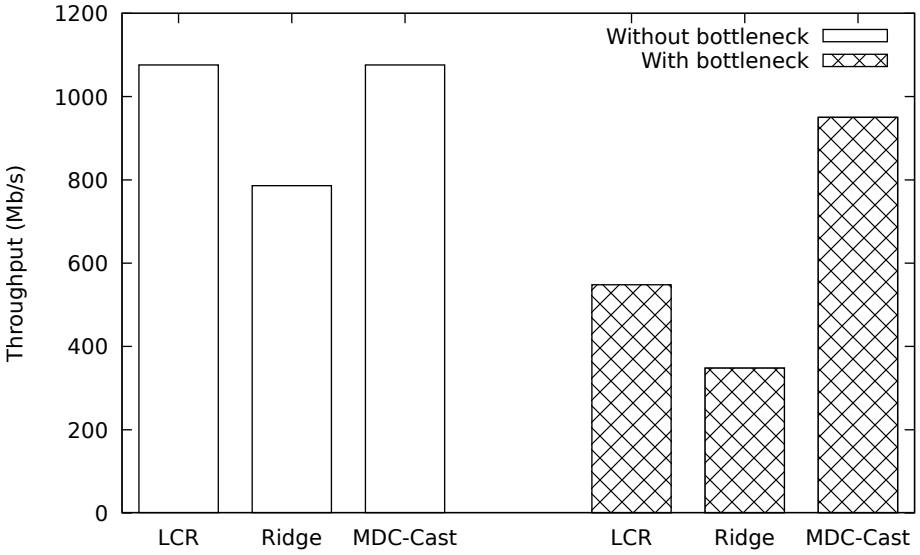


Figure 3.8: Comparison between *MDC-cast*, *LCR* and *Ridge* in one datacenter (left three bars) and in a multi-datacenters environment (right three bars).

Throughput when varying the number of nodes per datacenter

This experiment makes use of the topology depicted in figure 3.4. The topology contains three datacenters with three nodes in each datacenter. In this experiment we vary the number of nodes inside each datacenter (X axis) respectively. Figure 3.9 shows the throughput of *LCR*, *Ridge* and *MDC-cast* in a setup comprising three datacenters, when varying the number of nodes per datacenter. We observe that when the number of nodes increases inside the datacenter the throughput of *LCR*

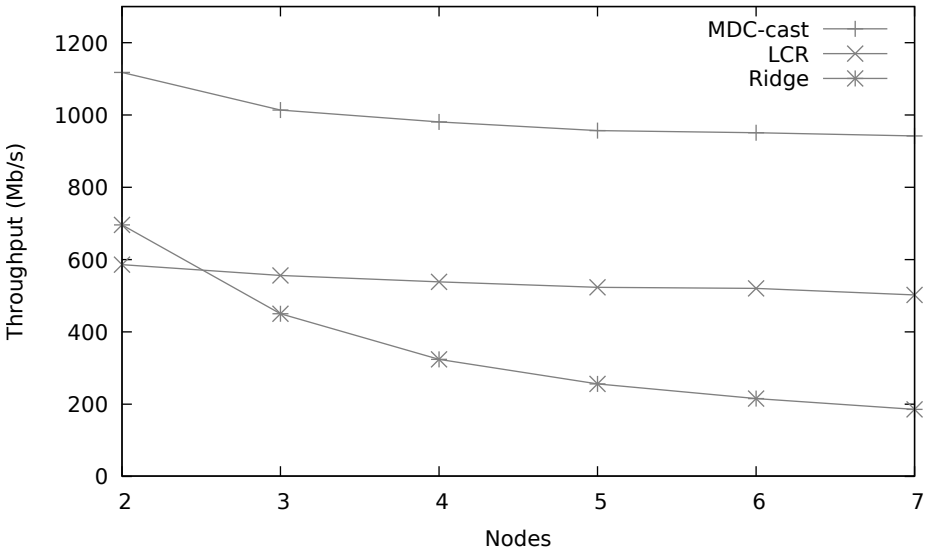


Figure 3.9: Performance of *LCR*,*Ridge* and *MDC-cast* in a setup comprising three datacenters, when varying the number of nodes per datacenter.

and *MDC-cast* only decreases a little bit while the throughput of *Ridge* decreases dramatically, due to the message dissemination pattern it employs.

Throughput when varying the message size

In this experiment, we use nine machines spread over three datacenters. We vary the message size and compute the resulting throughput. Results are depicted in Figure 3.10. We compare two variants of each protocol: with and without batching. We observe that, when batching is *not* enabled, *MDC-cast* is more impacted than other protocols by small messages. This is due to its use of IP-multicast inside datacenters that yields poor results with small messages. With batching enabled, the three protocols obtain stable results, whatever the message size.

Latency assessment

The last experiment assesses the latency achieved by *LCR*, *Ridge* and *MDC-cast*. The round-trip time between datacenters is measured using *ping* and varies between $2.3ms$ and $18.2ms$ (see Table 3.4). Figure 3.11 shows the latency of *LCR*, *Ridge*

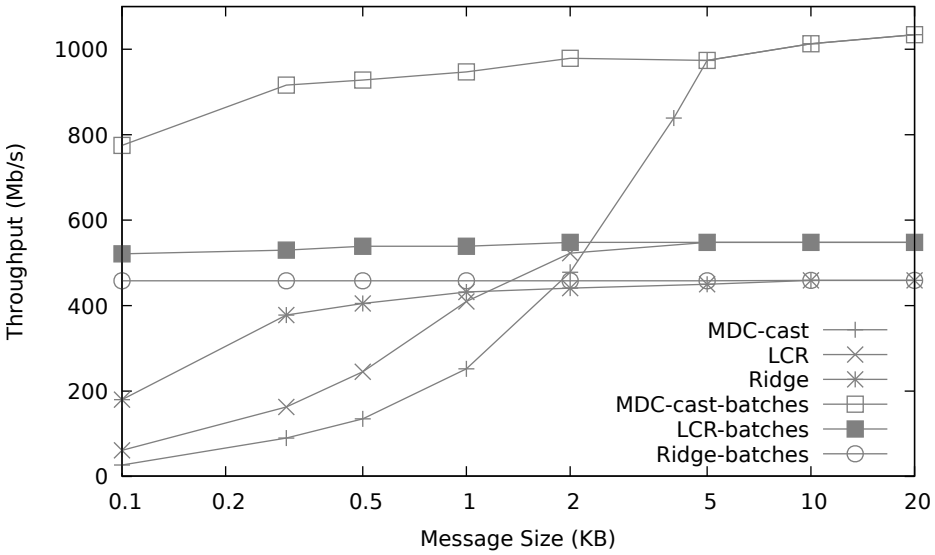


Figure 3.10: Throughput as a function of message size for *LCR*, *Ridge* and *MDC-cast* (with and without batching).

and *MDC-cast* with only one sender that periodically initiates message broadcasts. The figure shows the latency of the three algorithms when varying the number of datacenters. Results show that both *MDC-cast* and *Ridge* have almost stable latency and are not impacted by varying the number of datacenters. While the latency of *LCR* is proportional to the number of datacenters, it increases as the number of datacenters increases.

	Luxembourg	Nancy	Sophia	Grenoble
Luxembourg	0	2.3	18.2	13.7
Nancy	2.3	0	16.1	11.6
Sophia	18.2	16.1	0	9.74
Grenoble	13.7	11.6	9.74	0

Table 3.4: The response time between datacenters in micro seconds measured using *ping* tool

Figure 3.12 shows the latency of *LCR*, *Ridge* and *MDC-cast* with N senders in three datacenters, the topology of figure 3.4. We observe that *LCR*, *Ridge* and *MDC-cast* achieve close latencies (the Y axis starts at $16ms$). This is not surprising as in this case, the latency is limited by the throughput. As *MDC-cast* achieves the higher

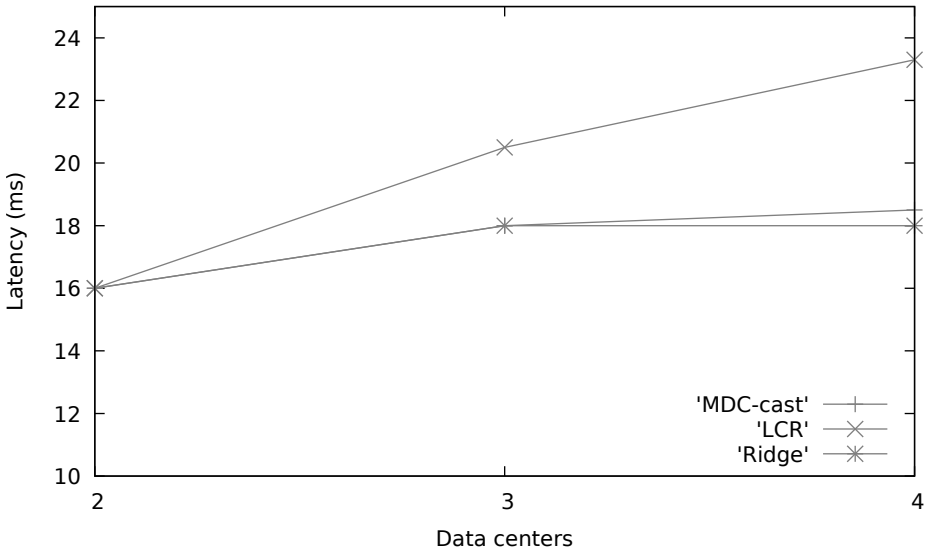


Figure 3.11: Latency of *LCR*, *Ridge* and *MDC-cast* with one sender

throughput, it exhibits the lowest latency.

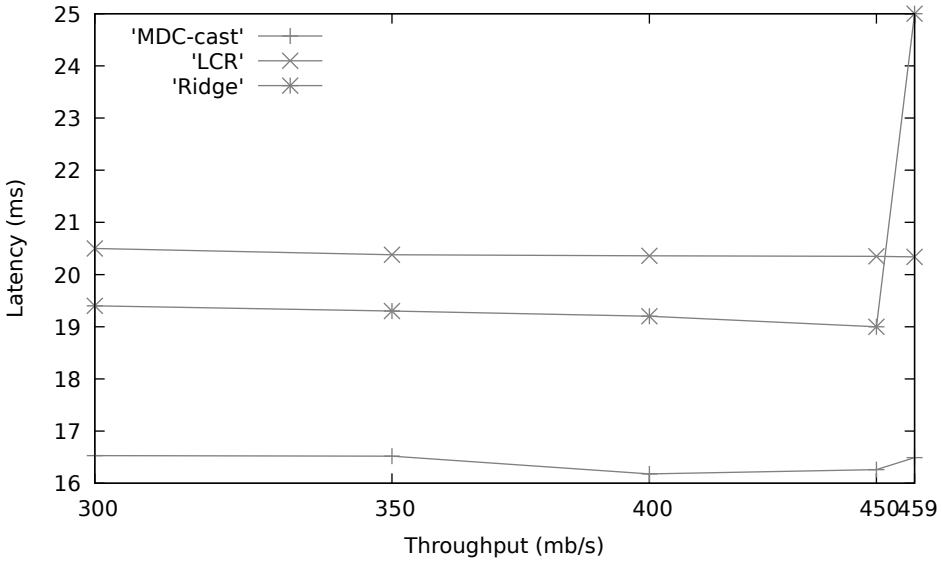


Figure 3.12: Latency of *LCR*, *Ridge* and *MDC-cast* with N senders

3.5 Conclusion

In this Chapter, we have presented a theoretical assessment of *MDC-cast*. We have also evaluated *MDC-cast* in a real-life settings (the Grid'5000 testbed) and we have

compared its performance to that achieved by two state-of-the-art protocols: *Ridge* and *LCR* from several points of view. Our performance evaluation shows that *MDC-cast* significantly outperforms other protocols in the datacenter environment.

Conclusion

Summary During my thesis, I worked on the topic "Total Order Broadcast in datacenter environments". After having studied the related works, we noticed that existing algorithms were not able to perform well in the context of multi-datacenter environments. We therefore decided to design a new total order broadcast protocol. That protocol is original in the sense that it combines the use of IP-multicast within datacenters and the use of TCP between datacenters. We have proved that the protocol is correct and we have performed both an analytical and a practical performance evaluation on Grid 5000. Besides, as the studied environments are not homogeneous, we have described the methodology we used to choose the best possible deployments of protocols on network topologies.

Our evaluation shows that the protocol we propose, namely *MDC-cast*, achieves similar performance than existing protocols in the context of homogeneous clusters, and significantly outperform the in the context of multi-datacenter environments.

Future Work There are several topics for future work that I list below:

- we plan to study the impact of this protocol on real systems, such as Zookeeper.
- we plan to extend the protocol to deal with more severe failures, such as crash recovery failures or even byzantine failures.
- we plan to extend our configuration methodology to other distributed systems requiring high throughput. Indeed, we believe that many existing systems

(e.g. Big Data systems comprising various components such as Spark, Kafka, Cassandra) would benefit from automated ways to find the best possible deployment.

Bibliography

[ADMA+04a] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical report, CNDS-2004-1, Johns Hopkins University, 2004.

[ADMA+04b] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical report, CNDS-2004-1, Johns Hopkins University, 2004.

[AFM92] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. RFC 1301, IETF, 1992.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. ACM Transactions on Computer Systems, 13(4):311–342, 1995.

[Anc97] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97), Washington, DC, USA, 1997. IEEE Computer Society.

[Ban07] Bela Ban. JGroups – A Toolkit for Reliable Multicast Communication. http://www.jgroups.org, 2007.

[BC94] Kenneth P. Birman and Timothy Clark. Performance of the isis distributed computing toolkit. Technical report, 8725 John J. Kingman Road, Virginia, Fort Belvoir, USA, 1994.

[BCP15] Carlos Eduardo Bezerra, Daniel Cason, and Fernando Pedone. Ridge: high-throughput, low-latency atomic multicast. 34th Symposium on Reliable Distributed Systems (SRDS), IEEE, pages 256–265, Oct 2015.

[BJ87a] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP'87), pages 123–138, New York, NY, USA, 1987. ACM Press.

[BJ87b] K. Birman and T. Joseph. Reliable communication in the presence of failures. ACM Trans. Comput. Syst., 5(1):47–76, 1987.

[Bou00] Jean-Yves Le Boudec. Rate adaptation, congestion control and fairness: A tutorial, 2000. Ecole Polytechnique Fédérale de Lausanne, 2012.

[BPGG13] C. E. Bezerra, F. Pedone, B. Garbinato, and C. Geyer. Optimistic atomic multicast. In 2013 IEEE 33rd International Conference on Distributed Computing Systems, pages 380–389, July 2013.

[BQ13] Gautier Berthou and Vivien Quéma. Fastcast: a throughput- and latency-efficient total order broadcast protocol. ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, pages 1–20, Dec 2013.

[BVG+96] K. Birman, W. Vogels, K. Guo, M. Hayden, T. Hickey, R. Friedman, R. van Renessen, and Al Vaysburd. Moving ensemble communication system to nt and wolfpack. In Proceedings of the USENIX Windows NT Workshop, August 1996.

[BvR93] K. Birman and R. van Renesse. Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, 1993.

[BvR96] K. Birman and R. van Renesse. Software reliability for networks. Scientific American, 274(5), May 1996.

[Car85] R. Carr. The tandem global update protocol. Tandem Syst. Rev. 1, pages 74–85, jun 1985.

[CH] Ge-Ming Chiu and Chih-Ming Hsiao. A note on total ordering multicast using propagation trees. IEEE Transactions on Parallel and Distributed Systems, 9:217.

[CM84] J.-M. Chang and N. Maxemchuk. Reliable broadcast protocols. ACM Trans. Comput. Syst., 2(3):251–273, 1984.

[CMA97] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. Distrib. Syst. Eng. J., 4(2):109–128, jun 1997.

[CMZ04] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Cjdbc: Flexible database clustering middleware, 2004.

[Cri91] F. Cristian. Asynchronous atomic broadcast. IBM Technical Disclosure Bulletin, 33(9):115–116, 1991.

[CT96a] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 43(2):225–267, 1996.

[CT96b] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. J. ACM, 43(2):225–267, 1996.

[DEM+10] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. Iperf. `http://iperf.fr/`, 2010. [online: ver. 2.0.5 pthreads.

[DSU04] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv., 36(4):372–421, 2004.

[EMS95] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: a fault-tolerant group communication protocol. In Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95), Washington, DC, USA, 1995. IEEE Computer Society.

[ESU04] R. Ekwall, A. Schiper, and P. Urban. Token-based atomic broadcast using unreliable failure detectors. In Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04), pages 52–65, Washington, DC, USA, 2004. IEEE Computer Society.

[FIMR01] U. Fritzke, P. Ingels, A. Mostefaoui, and M. Raynal. Consensus-based fault-tolerant total order multicast. IEEE Trans. Parallel Distrib. Syst., 12(2):147–156, 2001.

[FR97] T. Friedman and R. Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97), Washington, DC, USA, 1997. IEEE Computer Society.

[GKLQ07] Rachid Guerraoui, Dejan Kostic, Ron R. Levy, and Vivien Quéma. A High Throughput Atomic Storage Algorithm. In The 27th IEEE International Conference on Distributed Computing Systems (ICDCS'07), Toronto, Canada, 2007.

[GLPQ06] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. High Throughput Total Order Broadcast for Cluster Environments. In IEEE International Conference on Dependable Systems and Networks (DSN'06), Philadelphia, PA, USA, 2006.

[GLPQ10] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput optimal total order broadcast for cluster environments. ACM Trans. Comput. Syst., 28(2):5:1–5:32, July 2010.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. ACM Trans. Comput. Syst., 9(3):242–271, 1991.

[gri17] Grid'5000. `http://www.grid5000.fr/`, 2017.

[GT89] A. Gopal and S. Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In Proceedings of the 3rd International Workshop on Distributed Algorithms, pages 110–123, London, UK, 1989. Springer-Verlag.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. pages 97–145, 1993.

[Huf55] David A. Huffman. The synthesis of sequential switching circuits. Cambridge, Mass. : Research Laboratory of Electronics, Massachusetts Institute of Technology, [1954], 1955.

[Jia] Xiaohua Jia. A total ordering multicast protocol using propagation trees. IEEE Transactions on Parallel and Distributed Systems, 6:617.

[KA08] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware, LADIS '08, pages 3:1–3:6, New York, NY, USA, 2008. ACM.

[KK97] J. Kim and C. Kim. A total ordering protocol using a dynamic token-passing scheme. Distrib. Syst. Eng. J., 4(2):87–95, jun 1997.

[KT96] F. Kaashoek and A. Tanenbaum. An evaluation of the amoeba group communication system. In Proceedings of the 16th International

Conference on Distributed Computing Systems (ICDCS '96), Washington, DC, USA, 1996. IEEE Computer Society.

[Lam78] L. Lamport. The implementation of reliable distributed multiprocess systems. Comput. Netw. 2, page 95, 1978.

[Lam06] Leslie Lamport. Fast Paxos. Distributed Computing, 19(2):79–103, October 2006.

[LG90] S. Luan and V. Gligor. A fault-tolerant protocol for atomic broadcast. IEEE Trans. Parallel Distrib. Syst., 1(3):271–285, 1990.

[lib] libpaxos. `http://libpaxos.sourceforge.net/`.

[MMSA93] L. Moser, P. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. SIAM J. Comput., 22(4):727–750, 1993.

[Moo58] Edward Moore. Gedanken-experiments on sequential machines. The Journal of Symbolic Logic 23, page 60, 1958.

[MPP12] 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, pages 1–12, 2012.

[MPSP10] P.J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on, pages 527–536, 2010.

[MSS96] L. Malhis, W. Sanders, and R. Schlichting. Numerical performability evaluation of a group multicast protocol. Distrib. Syst. Enj. J., 3(1):39–52, march 1996.

[Ng91] T. Ng. Ordered broadcasts for large applications. In Proceedings of the 10th IEEE International Symposium on Reliable Distributed

Systems (SRDS'91), pages 188–197, Pisa, Italy, 1991. IEEE Computer Society.

[PBS89] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. ACM Trans. Comput. Syst., 7(3):217–246, 1989.

[Pol] Stefan Poledna. Replica determinism in fault-tolerant real-time systems. Real-Time Syst. 6, page 289.

[Reia] Michael K. Reiter. Distributing trust with the rampart toolkit. Magazine Communications of the ACM, 39:71.

[Reib] Michael K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. Proceedings of the 2nd ACM Conference on Computer and communications security, page 68.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv., 22(4):299–319, 1990.

[SH97] Shiu-Pyng Shieh and Fu-Shen Ho. A comment on "a total ordering multicast protocol using propagation trees". IEEE Transactions on Parallel and Distributed Systems, 8:1084, Oct 1997.

[SNN] S. Chanson S. Navaratnam and G. Neufeld. Reliable group communication in distributed systems. Distributed Computing Systems, 8:439.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. Commun. ACM, 39(4):76–83, 1996.

[WMK94] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In Selected Papers from

the International Workshop on Theory and Practice in Distributed Systems, pages 33–57, London, UK, 1994. Springer-Verlag.

[WS95] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In Proceedings of the 14TH Symposium on Reliable Distributed Systems, Washington, DC, USA, 1995. IEEE Computer Society.