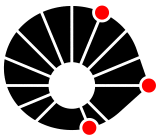

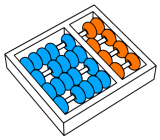

On-Time Fast Paxos

Daniel Cason *Luiz Eduardo Buzato*

Technical Report - IC-17-02 - Relatório Técnico
February - 2017 - Fevereiro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

On-Time Fast Paxos

Daniel Cason* Luiz Eduardo Buzato*

Abstract

Informally, total order broadcast protocols allow processes to send messages with the guarantee that all processes eventually deliver them in the same order. In this paper, we investigate the efficiency and performance of On-Time Fast Paxos a synchronous total order broadcast protocol for the crash-recover failure model that is built atop a broadcast-based asynchronous distributed system. On-Time Fast Paxos combines an asynchronous consensus protocol with a synchronous communication protocol while guaranteeing the original safety and liveness properties of Fast Paxos. The synchronous communication protocol relies on virtual global time to create the synchronicity necessary to make Fast Paxos work at its theoretical optimum, two communication steps. Experimental results allow us to conclude that On-Time Fast Paxos performs very well in terms of both throughput, reaches 960mbps in a 1Gbps network, and latency, with an averages below 2ms. Finally, its novel hybrid design, asynchronous layer driven by a synchronous layer, shows that, time and synchrony be used to improve the performance of total order broadcasts.

*Institute of Computing, University of Campinas, 13081-970, Campinas, SP, Brasil. Research partially founded by FAPESP under grants 2011/23705-4 and 2013/21651-0.

Contents

1 Introduction **3**

2 Total Order Broadcast and Consensus **4**

3 Model of Computation **6**
3.1 Distributed System 6
3.2 Virtual Global Time 7

4 Fast Paxos in a Nutshell **8**

5 On-Time Fast Paxos **11**
5.1 Activation 11
5.2 Normal Operation 12
5.3 Collisions 13
5.4 Tempo 14
5.5 Message Loss 15

6 Synchronous Communication Protocol **16**
6.1 Virtual Global Clock 16
6.2 Synchronous Message Dissemination 18

7 Experimental Evaluation **20**
7.1 Computing Environment 20
7.2 Asynchronous versus Synchronous Broadcast 20
7.3 Fast Paxos: Synchronous Siblings 22
7.4 Results and Discussion 23

8 Conclusion **26**

1 Introduction

Total order broadcast is a fundamental component of modern fault-tolerant distributed systems, from single mission-critical clusters to wharehouse-scale computers, as it ensures that messages sent to a set of processes are delivered by them in the same total order. Total order broadcast is useful to enable portions of the hardware and software in these machines to work in cooperation to efficiently deliver services that affect the performance of many other key Internet applications. An elegant way to implement total order broadcast is through reduction to consensus [4]; an algorithm that allows a set of processes, some of which can fail, to choose a single value out of a set of proposed values. Fast Paxos [12] is an optimal consensus algorithm for the asynchronous crash-recover model of computation because it can solve consensus in two communication delays [13]. So Fast Paxos would theoretically be the ideal consensus algorithm upon which to implement total order broadcast. Unfortunately, in practice, Fast Paxos can only maintain optimality when the values proposed for consensus do not collide, that is, none of the proposed values obtain enough votes and therefore no value is chosen as the value of consensus. In terms of the implementation of total order broadcast, this occurs when there is no *a priori* agreement among the processes as to the order in which messages should be delivered.

In the asynchronous model, the *a priori* order considered for the messages comes from the network, which would spontaneously tend to deliver broadcast messages in the same total order to the processes [16, 17]. This approach, however, has a number of practical limitations, which is attested by the lack of total order broadcast implementations based on Fast Paxos. In this paper we propose an alternative way to generate a *a priori* order for the messages from the use of *temporal order*, and use it to implement a synchronous variation of Fast Paxos, that we call On-Time Fast Paxos. To this end, we augment the asynchronous model with a *virtual global clock*, whose periodically generated clock ticks dictate the execution of the algorithm. Processes then seek agreement on the time, and not necessarily on the order, in which broadcast messages are received. The result is an original, very fast and latency conditioned total order broadcast algorithm.

The remaining of the paper is structured as follows. Section 2 contextualizes our research and points up our contribution to the area of total order broadcast protocols. Section 3 defines the model of computation assumed throughout the paper. The key algorithmic aspects of our On-Time Fast Paxos are covered from Section 4 to Section 7. Section 4 contains a summary of the original Fast Paxos; it has been included in the paper to make the explanation of our version of the algorithm self-contained. Section 5 defines On-Time Fast Paxos using a top-down approach, from the consensus layer down to the synchronous communication layer. Section 6 details the operation of the synchronous communication layer. Section 7 provides experimental evidence that the design and implementation of On-Time Fast Paxos represents a

significative advance in the state of the art of total order broadcast solutions. Section 8 concludes the paper.

2 Total Order Broadcast and Consensus

Chandra and Toueg [4] have shown that total order broadcast and consensus are equivalent by building a total order broadcast algorithm that relies on repeated executions of an asynchronous consensus algorithm. The algorithm starts with the assignment of a unique identifier to each message or, more generally, each set of messages to be broadcast. The pair unique identifier and set of messages is then proposed using any consensus algorithm. The reduction uses multiple independent, and possibly concurrent, instances of consensus; each instance chooses a single set of messages, among the possibly various sets of messages proposed to receive a given identifier. The output of the total order broadcast algorithm is formed by the set of messages decided in successive instances of consensus, taken in the order established by their identifiers. The delivery order is thus determined by the identifiers chosen for the messages; messages with the same identifier are delivered in some pre-agreed deterministic order—considering, for instance, the message ids.

The critical step in the reduction of total order broadcast to consensus is the assignment of unique identifiers to messages. This step can be implemented in a centralized or in a distributed manner. The usual way to generate identifiers is to rely on a distinguished process, namely a sequencer, that intercepts all messages broadcast and tags them with a unique identifier. This process accumulates the role of coordinator of the consensus algorithm, being responsible for selecting the value to propose in each instance. Consensus is therefore used to ratify the ordering for the messages generated by a sequencer, which is only replaced in case of failure. Examples of consensus algorithms that have been used to build this class of total order broadcasts are Chandra-Toueg's [4] and Paxos [10].

An alternative way to generate total order is to allow each process involved in the algorithm to propose identifiers for the messages. Identifiers are autonomously generated, and pairs set of messages and assigned identifier are proposed for consensus. Total order broadcasts opting for the distributed generation of identifiers do not require a sequencer because they tipically assume the existence of an *a priori* order for the messages. This means that, under favorable conditions, processes are likely to propose orders (assignments of identifiers to messages) that are identical, or at least not conflicting. Consensus in these circumstances becomes trivial [2], and is merely used to verify that the orders generated by each process are compatible, and only occasionally to solve conflicts. Examples of this approach are total order broadcasts based on prefix agreement [1], ordering oracles [17], and optimistic algorithms [16, 21], among them Fast Paxos.

The quest for algorithms that rely on a distributed procedure to generate total order is justified by the potentially lower communication cost provided by them, when compared with centralized algorithms. In fact, by eliminating the sequencer, the cost of total order broadcast is reduced to two communication delays, which is the best-case lower bound for consensus [5, 13]. Observe that centralized algorithms—such as Paxos or, equivalently, Fast Paxos operating in classic mode—can provide optimal latency for broadcasts initiated by the coordinator; a third communication delay however is necessary for messages broadcast by ordinary processes. In algorithms adopting a distributed approach, such as Fast Paxos, optimal latency can be achieved for broadcasts initiated by any process; possible exceptions occur when the *a priori* order assumption is not verified.

The nonobservance of the assumption that identifiers generated independently by the processes are order-compatible implies the proposal of conflicting values for consensus. Although conflicts are tolerated because safety is never violated, this situation is undesirable in terms of performance because liveness is hampered. For instance, when different identifiers are proposed for a message it is possible that either multiple or, worse, no identifiers are chosen for it. In this case, the resolution of conflicts may cost additional consensus instances to ensure that in the end an identifier is chosen for every message broadcast. Conversely, when the same identifier is assigned to distinct sets of messages, the corresponding instance of consensus is potentially affected as it may not admit a trivial solution; again additional communication may be required to ensure a successful completion of consensus. This scenario, common to all optimistic total order broadcasts is called *a collision* in Fast Paxos. Collisions thus denote failures to comply with the *a priori* order hypothesis, and constitute, at least in theory, the main adversary of total order broadcast implementations atop Fast Paxos.

The asynchronous model of computation does not inherenly allows the existence of an *a priori* order for messages broadcast concurrently in a distributed systems. However, in practice, it is often the case that messages broadcast, especially in local-area networks, are received in total order with high probability. This property, known as *spontaneous total order* [16, 17] or, in a different context, consistent delivery order [7, 8], is at the core of asynchronous total order broadcasts atop Fast Paxos. By relying on its validity, processes tag messages so as to simply represent their reception order, proposing them in successive instances of consensus. More specifically, processes interpret each broadcast message as if it were the value proposed for the next, still unused instance of Fast Paxos [11]. This straightforward correspondence between messages and consensus values establishes an optimal consensus pace that is effective as long as the processes, specifically those with the role of acceptors in Fast Paxos, receive the same messages in the same total order.

When the spontaneous total order is not observed, either because part of the processes fail to receive some messages, or because messages are received by processes

in different orders, the conflicting values proposed for Fast Paxos may cause collisions. At the cost of additional communication delays, Fast Paxos can solve collisions, and decide which messages to assign each identifier. However, the optimal consensus pace may take some time to be restored because the straightforward correspondence has been disrupted by the collision and may further trigger collisions, negatively impacting performance. For example, when processes receive the same messages in permuted orders, a whole range of identifiers may be affected by collisions, even if most of such messages are received in the same order. Another situation occurs when a given message is received only by a part of the processes: the sequence of messages, indexed by identifiers, observed by processes that received such messages, and those did not, becomes displaced by one position. Thus, practical implementations of total order broadcast atop Fast Paxos are very sensitive to frequent violations of spontaneous total order. For example, violations tend to be frequent when the processes that broadcast messages are the same that are responsible for ordering them, making implementations of Fast Paxos practically unfeasible.

In this paper we propose an alternative way to generate total order from the use of *temporal order* and its use to implement a synchronous variation of Fast Paxos, namely On-Time Fast Paxos. To this end, we augment the asynchronous model with a virtual global clock, and use the values returned by this clock as identifiers for the messages. Order is then determined by the reception time of the messages as defined by a global clock, and not necessarily by the order with which they are received by the processes.

3 Model of Computation

Our model of computation assumes an asynchronous distributed system augmented with global time.

3.1 Distributed System

A distributed system is a set of n processes $\mathcal{S} = \{P_1, \ldots, P_n\}$ that exchange messages via a network. An *event* is an occurrence of a process. Events can be internal or external. Internal events only influence the state of the process that has executed it. External events can be the sending or receiving of messages, and time events. $\mathcal{S}$ adheres to the crash-recover model of distributed computation supplemented with a virtual global clock. Processes fail by crashing, and may later recover. Before crashing, processes behave as instructed by the protocol that specify their correct behavior. Processes communicate by unicast or broadcast message exchange, that is, we assume that there is a network primitive that allows a process to send a message to all process of $\mathcal{S}$ in a single operation. Communication channels are unreliable, so

both unicast and broadcast messages can be lost, duplicated, reordered, or arbitrarly delayed, but can not be corrupted.

3.2 Virtual Global Time

The measurement of time is essentially a process of counting. Any repeating phenomena, natural or artificial, can be used to count. The occurrences of the repeating phenomena are events; events take no time. The time passed between two consecutive events is a period of time or a duration. A clock is a device which is based on some periodic oscillation (phenomena) that can be used to measure the progression of time. Each oscillation of the clock mechanism generates an event called a *clock tick*. The granularity of a clock is the period between two consecutive clock ticks. The granularity of a given clock, say C, can only be measured if there is another clock, say a reference clock RC, available with a finer granularity. In this case, RC can be used to implement C.

In our model, every process of $\mathcal{S}$ has access to local physical clock. The local clocks are not equal, but it is supposed that they differ at most by the clock precision Π, and that they have granularity g. Local clocks use the network to exchange timekeeping information that allows them to synchronize with each other. The clock synchronization protocol governing the ensemble of local clocks is not synchronized to an external clock. Moreover, as $\mathcal{S}$ is asynchronous, no assumption can be made about the communication or process latencies, but nevertheless, it is assumed that during *good* conditions of execution synchronization is possible. This means that for all local clocks, the differences between their readings is smaller than a given ϵ, and the variability for interprocess message delays is such that it is possible to establish the maximum error in the estimation of the delay taken for a round-trip message exchange among the processes of $\mathcal{S}$. These assumptions are sufficient [9, 6] to allow the implementation of a virtual global clock.

The virtual global clock is used (i) to define the order of events and (ii) to coordinate processes so that they can execute an action at the same approximate real time instant. It implements an abstraction of time represented by an strictly monotonic increasing integer. The virtual global clock can be queried to obtain the current time and it can be advanced. Its granularity Δ is the period of oscillation of the clock, that is, the time period elapsed between two consecutive virtual global clock ticks. Processes can register with the virtual global clock to receive, in the form of software interruptions, the clock ticks it generates. The integer label of a clock tick is the time of its occurrence, so clock tick i is the event (instant) that marks the beginning of the time period (or time granule) i.

4 Fast Paxos in a Nutshell

This section presents a brief overview of Fast Paxos, a consensus algorithm; a complete description can be found in [12]. The consensus problem demands a set of processes to choose a single value. Fast Paxos is easier to explain in terms of three sets of agents that play complementary roles in the resolution of consensus: *proposers* are agents that propose values, *acceptors* ensure that a single value is chosen, and *learners* learn the value that has been chosen. Agents are implemented by processes, a single process can implement multiple agents playing different roles.

The consensus algorithm proceeds in rounds, aka as *ballots*, in which values are put to a vote. Each ballot is owned by a process, the coordinator of a ballot, and can be either a fast ballot or a classic ballot. Ballots are identified by positive integers, the ballot numbers. A ballot number identifies the coordinator of a ballot, and indicates whether the ballot is fast or classic. Ballots are not necessarly started in numerical order, some ballot numbers can be skipped, and multiple ballots may be executed concurrently. Each ballot progresses through two phases, with the following steps:

- a coordinator starts a ballot by choosing a ballot number b, larger than any ballot number it is aware of, and sending b to all acceptors in a Phase 1a message;
- an acceptor only reacts to the reception of a ballot b Phase 1a message if it has not participated of any ballot $b' \geq b$—otherwise, it ignores the request. The acceptor confirms its participation in the ballot b by sending a Phase 1b message to the coordinator. If the acceptor participated in the Phase 2 of a previous ballot, it attaches to the Phase 1b message the last vote it has cast: the highest ballot number b^* in which the acceptor voted, and the value v it voted for in ballot b^*;
- Phase 2 of is started when the coordinator receives Phase 1b messages from a quorum of acceptors. It is a requirement to not have yet started Phase 2 of ballot b, nor started any ballot $b' > b$. The action taken by the coordinator depends on whether acceptors reported to have cast votes in previous ballots. If they have, the coordinator is forced to propose in Phase 2 one of the values reported in Phase 1. To propose a value v the coordinator sends v in a Phase 2a message; the rule for selecting a value for Phase 2 is described in the following. If no acceptor reported to have cast votes in previous ballots, the coordinator is free to propose any value in Phase 2. From this point, the operation is different for classic and fast ballots. If b is a classic ballot, the coordinator uses the Phase 2a message to propose any value received from the proposers. If b is a fast ballot, this role is transfered to the acceptors, by sending to them a special Phase 2 message called Any message.

- the reception of the ANY message enables an acceptor to operate in fast mode in ballot b. The acceptor then treats a value v received from a proposer as if it was a PHASE 2A message from the coordinator proposing v for ballot b. The operation is the same as for PHASE 2A messages of classic ballots, as described below.

- upon receiving a ballot b PHASE 2A message an acceptor verifies whether it can participate in Phase 2 of that ballot: if it has not participated of any ballot $b' > b$, nor cast votes in any ballot $b' \geq b$. If so, the acceptor casts a vote for the value v carried by the PHASE 2A message. The acceptance of v in ballot b is registered, and informed to the learners by sending a PHASE 2B message to them.

A value v is decided when a quorum of acceptors vote for v in the Phase 2 of a ballot. This situation is identified by the learners, which monitor the `Phase 2B` messages sent by acceptors. Once a value is decided in a ballot, the algorithm ensures that no other value can be proposed by the coordinator of any subsequent ballot, as detailed below. Moreover, it is assumed agents remember the actions they performed. In particular, coordinators have to remember the last ballot they started, and acceptors the last ballot they have taken part in and the last vote they have cast. Since Fast Paxos agents may crash and recover, this information must be stored in stable memory.

Fast Paxos ensure that ballots do not choose different values. This guarantee relies on the role for selecting values for Phase 2 and on requirements for the intersection of quorums. For selecting the Phase 2 value we consider the simplified rule for Fast Paxos proposed by [18]. As in the original rule [12], the coordinator ranks the received votes by ballot number and selects the values associated to the highest ballot b^* reported. If the selected values contain a single element, which should always happen when b^* is a classic ballot, this value must be proposed again. Otherwise, if multiple values were selected, b^* is necessarly a fast ballot and the value most often present among the selected ones must be proposed.

Quorums are minimal sets of processes whose participation in a ballot is a condition for it to proceed. Each type of ballot has a set of quorums associated with it, classic quorums for classic ballots and fast quorums for fast ballots. Quorums are used to guarantee that if a value was or might ever be chosen in a ballot, it is going to be reported on the Phase 1 of any successive ballot. The requirements for that are [12]: (i) any two quorums must have non-empty intersection, and (ii) any two fast quorums and any classic or fast quorum have a non-empty intersection. An optimal classic quorum can then contain a simple majority of processes, i.e. $\lfloor N/2 \rfloor + 1$ processes in a system with N processes. In this case, the corresponding fast quorum must contain $\lceil 3N/4 \rceil$ processes. For an optimal fast quorum, fast and classic quorums must have the same size of $\lfloor 2N/3 \rfloor + 1$ processes [18].

The option for fast or classic ballots is made by the coordinator, based on its perception of the state of the algorithm. If the coordinator believes that no value has been yet proposed, it starts a fast ballot. If it was indeed the case, no previous votes will be reported by the acceptors, which are then instructed to operate in fast mode. This is the expected behavior for Fast Paxos, and allows proposed values to be chosen, without the intervention of the coordinator, in two communication steps. Classic ballots are reserved for ballots where the coordinator suspects that an active ballot may not succeed in choosing a value. In this scenario, acceptors are expected to report previous votes, from which the coordinator selects the value it proposes for consensus. The algorithm operates in recovery mode, and can benefit from the potentially smaller quorums required by classic ballots.

Ballots may not succeed for several reasons, such as the failure of their coordinators, message loss, or the concurrence of higher-numbered ballots. To detect these situations, often indistinguishable from each other, processes monitor the various algorithm messages and resort to failure detectors. To deal with these situations that tend to hamper progress, different recovery mechanisms can be implemented, but ultimately the intervention of a coordinator may be necessary to start a new ballot. To avoid competition, the role of starting ballots is restricted to a single coordinator, elected among all agents, and replaced only when it crashes. Fast Paxos is safe in presence of multiple coordinators, so correctness is not compromised by mistakes of the election mechanism. However, the eventual election of a correct process as the only active coordinator is a progress requirement.

In addition to the reasons described below, fast ballots may also not succeed due to *collisions*. A collision occurs when acceptors accept, in the Phase 2 of a fast ballot, values from different proposers, so that no value is accepted by a quorum of acceptors. From the learners' point of view, the votes are distributed among several values, and none of them can reach enough votes to get chosen. To solve a collision it is necessary to start a new ballot, incurring the cost of operating in recovery mode in a failure-free scenario. Thus, altough some optimizations are possible, recovery from collisions is very costly to the algorithm. In fact, depending on how often collisions happen, there is can be no advantage operating in fast mode.

The procedure presented above refers to the operation of a single *instance* of consensus. Fast Paxos supports the execution of multiple consensus instances, identified by positive integers. Instances are independent, in terms of values proposed and chosen in each of them, but share the same set of agents and the same coordinator. The coordinator, once elected, selects a fast ballot number and tries to activate it in *all* consensus instances. This is done by sending a single `Phase 1A` message to all acceptors. At this initial moment, each acceptor only informs the coordinator the last (highest-numbered) consensus instance in which it has participated. Based on the reports of a quorum of acceptors, the coordinador identifies the last instance in which values have been proposed, the last active instance. All subsequent instances

are inactive, which means that no value can have been chosen for them. The coordinator then sends a `Any` message enabling acceptors to operate in fast mode for all instances above the last active one. Next, acceptors report their previous votes for the active instances, which are handled individually by the coordinator. Hopefully, most of those instances will have already been decided, so that Phase 1 needs only to be executed for few of them.

5 On-Time Fast Paxos

On-Time Fast Paxos is a total order broadcast algorithm that combines two protocols: a synchronous communication protocol and an asynchronous consensus protocol. The synchronous communication protocol is responsible for (i) the on-time dissemination of messages, and (ii) the use of a virtual global clock to generate timestamps that establish a timeline for the broadcast messages and are used to uniquely identify them. The unique identifiers assigned to the broadcast messages and their timeline are then used to establish a total order. Subsequently, this total order is ratifyed by the asynchronous consensus protocol, essentially an implementation of Fast Paxos, which main function is to render fault tolerance to the synchronous communication protocol.

Akin to other total order broadcasts based on (Fast) Paxos, in our algorithm clients first interact with proposers that are, in their turn, responsible for initiating broadcasts. Delivery of totally ordered messages is the responsibility of learners, with acceptors playing the role of building and rendering persistent the total order. In a nutshell, On-Time Fast Paxos differs from other existing asynchronous total order broadcasts based on Fast Paxos by its use of a synchronous communication protocol to condition the behavior of the consensus agents. The critical aspect of our design is determined by how the agents of the asynchronous consensus protocol interact with the components of the synchronous communication protocol to materialize our total order broadcast algorithm. In the next sections we detail these interactions, starting with a description of the procedure required to activate On-Time Fast Paxos, and proceeding through the normal operation and the exceptional operation: collision handling, lost messages, and flow control.

5.1 Activation

As we have seen in section 4, Fast Paxos uniquely identifies instances of consensus by positive integers, and their operation in fast mode must be enabled by the coordinator. Fast Paxos has been designed for the asynchronous model, On-Time Fast Paxos honors the original asynchronous design of Fast Paxos while making their agents work on-time, that is, synchronously. It is the responsibility of the synchronous

communication protocol to offer mechanisms that can be used by the consensus protocol to achieve total order on time. So the first problem the design of On-Time Fast Paxos has to solve is how to activate the protocol, that is, how to bootstrap a correspondence between Fast Paxos instance identifiers and the identifiers used by the synchronous communication protocol. On-Time Fast Paxos manages a mapping between the timestamps generated by the virtual global clock and instances of consensus. The map is straightforward: virtual global clock timestamps are used to uniquely identify instances of Fast Paxos. The second problem to be solved is how to enable the concerted, paced, operation of the agents of Fast Paxos; this is achieved by synchronously delivering sets of messages to the agents.

The activation of On-Time Fast Paxos is initiated by its coordinator, which as soon as elected selects a fast ballot number b and sends it to the acceptors using a special PHASE 1A message that does not refer to any particular instance of consensus. If b is a high enough ballot number, each acceptor informs the coordinator the last (higher-numbered) instance of consensus for which it has cast a vote. Upon receiving replies from a quorum of acceptors, the coordinator can establish the activation point [20]: the first (lower-numbered) instance of consensus i from which no value may have been decided in ballots previous to b. It then enables the fast operation mode starting from instance i by sending a ballot b ANY message to all processes. When received by a proposer, the ANY message instructs it to send broadcast messages directly to the acceptors, through the synchronous communication protocol. For an acceptor, in its turn, it enables the acceptance of sets of proposer's messages delivered by the synchronous communication protocol as values proposed for instances of consensus. Acceptance of consensus instances is enabled for instances greater than or equal to i, as indicated by the ANY message, or, equivalently, for broadcast messages timestamped from i onwards. Virtual clock timestamps and instances of consensus are placed in a one-to-one correspondence by advancing the virtual global clock to a future time equal to the value established by the consensus activation procedure as the point of activation.

5.2 Normal Operation

During the normal operation On-Time Fast Paxos proceeds through the following steps. To broadcast a message m, a proposer uses the synchronous communication protocol to send m to all destinations. Along with m, the proposer sends its process identifier (pid) and a counter, whose concatenation form the message identifier. The synchronous layer timestamps every message it transports with the current global time. At the receivers, this is especially important for the acceptors, messages received through the synchronous layer are grouped according to their timestamps. Let i be the timestamp of the messages in the set M_i, received from clock tick i to immediately before clock tick $i + 1$; at the clock tick $i + 1$, the pair $\langle i, M_i \rangle$ is delivered to the

consensus protocol.

An acceptor interprets the pair $\langle i, M_i \rangle$ received from the synchronous protocol as if it was a PHASE 2A message that proposes M_i as the value for the instance i of Fast Paxos; the message is assumed to refer to the last (high-numbered) fast ballot b activated by the coordinator for instance i. During normal operation, it is expected that no ballot $b' > b$ has been started, and no value has been accepted in the instance i—in particular, due to the mapping of consensus instances to the timestamps produced by the virtual global clock that are monotonically increasing. The acceptor then endorses the assignment of the unique identifier i to the set of messages M_i, proposed by the synchronous layer, by casting a vote and sending it to the learners in a ballot b PHASE 2B message.

Learners and the coordinator of Past Paxos also receive the proposer's messages that are broadcast using the synchronous communication layer. This allows the PHASE 2B messages sent by acceptors to carry only the message's ids, instead of the full payload. Upon receiving identical PHASE 2B messages from a quorum of acceptors, and learning the ids of the proposer's messages decided for that instance, the learner retrieves from the synchronous layer the corresponding full messages (payloads). The set of proposer's messages retrieved, taken in a pre-agreed order of their ids, form a sequence of messages; the sequences of messages formed for every instance of consensus, taken in the natural order of their identifiers, define the total order for the messages. Observe that learners only keep a cache of proposer's messages, which are permanently stored by the acceptors, and can, if necessary, be retransmitted by them, as detailed in section 5.5.

5.3 Collisions

When acceptors disagree on the set of messages to which a given timestamp is affixed, the corresponding instance of consensus may end in a collision [12]. The situation can be detected by the coordinator, that by also playing the role of learner listens to PHASE 2B messages. Upon detecting a collision, the coordinator starts a classic ballot to enforce the decision of a single set of messages chosen from the distinct sets of messages proposed in the conflicting fast ballot. Assume the collision occurs in a fast ballot b owned by the current coordinator, which also owns the subsequent classic ballot $b+1$. When this is the case, PHASE 2B messages of ballot b can be interpreted as if they were the votes reported by the acceptors in PHASE 1B messages of ballot $b + 1$. This allows the starting of the collision-recovery ballot $b + 1$ from Phase 2, thus saving the two communication steps required for Phase 1: a procedure known as coordinated recovery [12].

Recall that acceptors' PHASE 2B messages do not carry full proposer's messages, but only message's ids. The same applies to PHASE 2A messages sent by the coordinator in collision-recovery ballots. The point is that acceptors may even send only

references to their votes in PHASE 2B messages, but can not accept PHASE 2A messages referencing proposer's messages they have not yet received. The protocol must then intercept each PHASE 2A message, extract the set of message's ids proposed therein, and retrieve the corresponding proposer's messages, so that it can offer a set of full messages to the acceptor. Hopefully, in most cases the acceptor has already received the referenced proposer's messages—even if associated with different timestamps. Otherwise, the retransmission of the missing proposer's messages is requested, and their reception becomes a condition for the acceptor to process PHASE 2A message. Thus, in addition to the intervention of a coordinator in the consensus layer, collisions may require the exchange of proposer's messages in order to resynchronize the processes in the communication layer.

5.4 Tempo

From the activation onwards the tempo of the consensus algorithm, that is, the pace at which new instances of consensus are created, are determined by the clock ticks of the virtual global clock. Optimistically, once initiated most instances will end with no collisions and no need for retransmissions, in a single communication delay. Some instances may take longer, requiring extra communication steps for the algorithm to reach a decision, or for the synchronous exchange of the proposer's messages. On average it is expected that all agents of the consensus algorithm work at approximately the same tempo, implying that learners, acceptors, and proposers are subject to stable workloads. The funcion of the tempo mechanism is to ensure maximize the periods of the agent's stable work.

Instances that take too long to complete represent probably the main cause of tempo variability because of unsuccessful ballots or lost messages, that, in their turn, are caused by periods of asynchronicity of the distributed system. Late instances form *gaps* in the sequence of completed consensus instances. The existence of a gap delays the delivery of already totally ordered messages that succeed the gap until it is closed. Delivery delays clearly penalize the latency of those messages, while blocking the end of the corresponding total order broadcasts and subsequent release of the resources associated with them. In summary, instances that take too long to complete generate a backlog of tasks that propagate work back along the agents—learners do not deliver messages, acceptors have to retransmit messages and try to advance via asynchronous Paxos ballots, proposers pace may further overload acceptors—in a domino effect with clear negative impact on performance. It is impossible to guarantee workload and latency stability in an asynchronous distributed system. So, to try to cope with unfavourable setups on-time fast paxos resorts to lowering the proposer's tempo.

We can not prevent atypical delays from affecting instances of consensus, but we can restrict the start of new instances in order to restrain the backlog.

▷ *How are clock instances re-enabled?*

This is achieved by controlling the generation of ticks by the instances of the global clock—which are in part operated by software. Initially, consider that upon being activated an instance of the global clock is enabled to generate a given number of clock ticks. When the number of clock ticks an instance was enabled to generate is reached, the instance is disabled. This means that in the processes governed by that clock instance ticks are no longer generated, and so new instances are no longer initiated. In order to remain in operation, a clock instance should be opportunely *wound*, that is, enabled to generate an additional number of clock ticks.

5.5 Message Loss

The synchronous communication layer connects all processes and allows proposers to send messages directly to all destinations, both acceptors and learners. Proposer's messages carry application messages, possibly multiple of them, and can therefore be particularly large: if Phase 2 messages had to include several proposer's messages, this step would easily become a bottleneck. By sending only message's ids in Phase 2 messages, however, it becomes possible for a process to receive a message's id from the consensus layer but not the corresponding full proposer's message from the synchronous layer. This requires a simple protocol for retrieving proposer's messages from message's ids, which possibly includes asking for retransmissions from other processes.

The primary source of messages circulating on the synchronous communication layer are the proposers. However, due to their possibly transient participation in the protocol they are not good candidates for retransmitting any messages possibly lost during the original broadcast. Acceptors are better suited to the function of message retransmission and storage as Fast Paxos requires the reception of messages by at least a quorum of acceptors. So, in our solution missing messages are handled by acceptors. Any process can request the retransmission of a set of proposer's messages by sending their ids to all acceptors. If the requested messages were, or might have been, decided in a consensus instance, then necessarily a quorum of acceptors has already received them, and at least one non-failed acceptor will be able to fulfill the retransmission request.

Observed that in regular operation, several acceptors will be able to retransmit each requested proposer's message. To avoid the retransmission of multiple copies of the same message, the space of message's ids is partitioned among acceptors, and each acceptor only retransmits messages with ids in a certain range. In addition, retransmissions are sent to all processes, and an acceptor does not retransmit a message if has already done so recently. On the other side, if retransmissions are not received after a maximum delay, the request is resent with increased counter. The request counter shifts message's ids so that to address different acceptors. This avoids situations in which the target acceptor for a given id is irresponsive, or has not received

the corresponding message.

6 Synchronous Communication Protocol

In this section we detail the operation of an essential component of On-Time Fast Paxos: the synchronous communication protocol. As referred in Section 5 it is responsible for the on time dissemination of broadcast messages. This role comprises: (i) the transport of messages, whose broadcast is initiated by proposers, to their destinations, acceptors and learners; and (ii) the assignment of timestamps to the broadcast messages, which in turn determine their delivery order. This is a question of establishing a preliminary order for the broadcast messages, which is then submitted to Fast Paxos for ratification.

The synchronous communication protocol thus combines the transport of broadcast messages with timing functions. Its operation is governed by a virtual global clock, that periodically generates clock ticks at every process. A simple implementation of a virtual global clock, that does not rely on specialized hardware or real-time software, is presented in Section 6.1. Section 6.2 then progressively construct a synchronous dissemination protocol, governed by the virtual global clock, for the broadcast of messages.

6.1 Virtual Global Clock

Modern computers keep track of the passage of time using a battery-operated CMOS clock circuit driven by a quartz resonator. This simple physical clock does not have the precision or accuracy for real-time processing, but is sufficient to implement a fairly accurate virtual global clock [6]. The resonators of the local clocks are not equal, but it is supposed that they differ at most by the clock precision Π. Furthermore, the granularity g of the virtual global clock is defined as its period of oscillation, expressed as a multiple of the frequency of oscillation generated by the resonators of the local clocks.

A simple way to implement a virtual global clock is to have a set of time daemons, one per process, organized in a master-slave hierarchy. One of the time daemons is elected to play the role of *synchronizer*, the process responsible for the implementation of the virtual global clock. The synchronizer has granularity Δ, meaning that a synchronization message m is broadcast to every process of the system every Δ unities of its local clock. Every synchronization message carries a timestamp $\mathsf{m.t}$, a positive integer incremented by the synchronizer at each broadcast. It represents a discrete instant of the global time base the virtual global clock is supposed to implement. Note that the synchronizer can be activated, enabling the broadcast of a given number of synchronization messages, its timestamp can be advanced to given value, or can be

deactivated if necessary.

The slave time daemons listen to the synchronization messages periodically broascast by the synchronizer. At every message reception a slave, say s compares its clock value s.c with the timestamp m.t received from the synchronizer. If m.t > s.c then clock tick m.t is generated and the clock value is updated s.c ← m.t. Otherwise the message is just discarded and the value of the global clock at that slave remains unchanged. Thus, the virtual global clock synchronization protocol does not fully implement any distributed time synchronization algorithm, it implements a mechanism that is sufficient to generate a timeline where global clock ticks, represented as integers, are about Δ apart.

The virtual global clock is fault tolerant The virtual clock global tolerates the failure of the synchronizer which is, in this case, replaced by a new synchronizer elected among the remaining correct slaves. The detection of the synchronizer's failure and the subsequent election of the new synchronizer is carried out using a mechanism similar to the one implemented in Ω failure detector [14]. In the case of the virtual global clock, the synchronization messages serve an additional purpose, they double as *heartbeat* messages of an Ω failure detector. Piggybacked onto the synchronization messages is the following information: an epoch number and the process identifier of the current synchronizer.

Any slave daemon that suspects that the master daemon may have failed becomes a candidate and tries to elect itself as the new synchronizer. The candidate generates a new epoch number and takes the role of synchronizer using as the initial timestamp of the new epoch the current value of its own clock. In the presence of concurrent candidates to the role of synchronizer, the candidate with the largest epoch number is elected. The remaining candidates are demoted to slaves as soon as they receive a synchronization message containing an epoch number larger than their own. Time daemons conform to a synchronization message from a different epoch provided it is greater than the current epoch; and ignore synchronization messages with epochs smaller than the current epoch. As a consequence, the virtual global clock eventually converges toward a state where a single synchronizer exists and all the remaining daemons behave as slaves.

Precision and Accuracy of the Global Clock The quality of the virtual global clock obtained by this approach, centered around the figure of a synchronizer, is determined by three factors related to the synchronism presented by the system. First, the oscillation period of the physical clock read by the synchronizer, that defines the granularity of the virtual global clock. This factor is measured by the drift rate of the synchronizer's clock when compared to a reference clock or to the other processes' local clocks. Second, the accuracy with which the synchronizer broadcasts synchronization messages in relation to the instants of its physical clock scheduled for such. Together

with the clock drift rate, the accuracy of the synchronizer is decisive for the broadcast of synchronization messages to occur at intervals close to the desired granularity Δ. Third, the variability of latencies for the delivery of synchronization messages to every process. This variability ultimately defines the precision of the virtual global clock, that is, the duration of the real-time interval during which different processes generate the same clock tick.

We will not delve into the analysis and experimental evaluation of the three factors mentioned, in particular because we already presented in a previous work [3]. For now, it is enough to say that the performance achieved by On-Time Fast Paxos is considerably due to the accuracy and precision of a synchronizer-based virtual global clock.

6.2 Synchronous Message Dissemination

Having presented a simple protocol to implement the virtual global clock, we now discuss how the on-time dissemination of messages occur, and how it is related to the assignment of timestamps to the broadcast messages. Recall that the virtual global clock mechanism is responsible for the periodic generation of clock ticks at every process. Clock ticks are ideally generated with periods close to the configured granularity Δ, and all non-crashed processes are expected to generate the same timeline. The global virtual clock therefore allows to induce in the processes the lock-step operation of a synchronous system.

Considering first processes that are destinations of the broacast messages, acceptors in particular, the lock-step operation consists on the interleaving of message receiving and message processing phases. Messages received are not immediately processed but buffered until the next clock tick, when all buffered messages are processed together. As a result of this processing phase relative, to say, time instant i the synchronous communication protocol delivers a pair $\langle i, M_i \rangle$, where M_i is *a priori* the set of messages received from clock tick i to immediately before clock tick $i + 1$, as discussed in Section 5.2. Observe that so far we have made no assumptions about the behavior of proposers.

Asynchronous and synchronous proposers On the side of processes that initiate broadcasts, the proposers, there are two possible modes of operation. The original Fast Paxos proposers are asynchronous, in the sense that whenever they receive a broadcast request, the message is immediately submitted to the acceptors. Proposers are allowed then to access the network at any time, without any external form of contention. This is the case, for example, when proposers are considered clients, external to the total order algorithm. An alternative approach is, since the system is equiped with a global clock, to make proposers synchronous. A synchronous proposer does not immediately submits a message whose broadcast is requested by a client; instead,

the message is enqueued to be submitted, together with other messages in the same situation, at the next clock tick. The broadcast of messages is then conditioned by the pace dictated by the virtual global clock, so that each proposer can broacast a single batch of messages per clock tick.

An experimental argument to support the adoption of synchronous proposers is presented in Section 7.2. A more comprehensive argument concerns the ability to control the load applied by proposers to the network through the broadcast of messages. In fact, as proposers are synchronous, it is possible to limit the network usage by restricting the number of bytes each proposer is allowed to broadcast at every clock tick. It is assumed that the number of concurrent proposers is known, and so the granularity of the global clock, allowing to accurately predict the maximum load applied to the network. As this could be a very restrictive policy, we relax it by defining an *optimal batch size* with which proposers try to comply when considering the average message size per clock tick.

A synchronous transport Assuming that proposers have access to the virtual global clock, how could they participate more actively on the generation of the total order? An alternative would be to transfer the role of message ordering to the proposers, which can assign the timestamp of each message at the time of its broadcast. A proposer can then read its clock value at the broadcast time and add to it a number κ of time units that depends on the clock granularity Δ and the broadcast message size. An acceptor a that receives a message m from a proposer, compares the current value of its clock $\mathsf{a.c}$ with the message timestamp $\mathsf{m.t}$. If $\mathsf{m.t} < \mathsf{a.c}$ the message was early received, and its delivery is postponed until instant $\mathsf{m.t}$. Otherwise, if $\mathsf{m.t} > \mathsf{a.c}$ the message was received late, and for the sake of synchronous principles must be discarded. This is actually the major limitation of the proposer-oriented ordering approach, when applied to a system that is not stricly synchronous. If a proposer chooses κ wrongly, in particular a very low value, a broadcast message may end up being discarded by most of the acceptors.

We opted for a solution that represents a compromise between the proposer-oriented and the acceptor-oriented ordering approaches. Proposers do assign timestamps for the messages they broadcast but acceptors do not discard messages received late, since we do not assume strict upper bounds for message transmission. The timestamps assigned have then the role of establishing a minimum latency κ for the delivery of broadcast messages. Early messages have their delivery postponed to the instant designated by the proposer, while the other messages are assigned, as final timestamp, the instant at which they are received by the acceptors. As detailed in Section 7.4, the adoption of this approach allowed a significant reduction in the rate of collisions and a consequent improvement on the performance of On-Time Fast Paxos. The main reason for the improvement is the reduction of time uncertainty as to the global time instant at which different acceptors receive certain messages that

are likely to be received around a clock tick—before for some, after for others.

The approach based on a minimum latency parameter allows the deployment of a process that colocates two Paxos agents usually never placed together in Fast Paxos: a proposer and an acceptor. This in only possible because our synchronous approach defines a minimum latency for the reception and processing of the messages processed by the acceptor.

7 Experimental Evaluation

This section contains the experimental evaluation of On-Time Fast Paxos. First, it specifies the cluster of computers and network used in the experiments. Second, it shows that synchronous broadcasts can be used to uplift the performance of protocols in contrast with asynchronous broadcasts. Third, it presents and discusses a progression of total order broadcast protocol designs that lead to On-Time Fast Paxos. All designs are discussed in the light of the experimental results.

7.1 Computing Environment

The experiments reported in this Section were performed in a cluster with Supermicro X8DTL machines equipped with two quad-core Intel Xeon E5620 processors running at 2.4 GHz, and with 12GB of main memory. Machines ran a 64-bit Gnu/Linux Debian 8.0, and were connected to a 3Com 4200G Gigabit Ethernet switch, with round-trip time of approximately 0.2ms. The algorithms were implemented in Java atop of the Treplica toolkit [19], and experiments were carried out using the OpenJDK 1.7 Server VM.

7.2 Asynchronous versus Synchronous Broadcast

In order to place asynchronous broadcast in contrast with synchronous broadcast, the following experiment has been devised. A set of processes, in the role of clients of the ip-multicast network (Ethernet), broadcast 8kB messages to generate increasingly larger workloads (mbps) with the goal of using the full bandwidth of the network (1Gbps). The total workload grows linearly with each process being responsible for the generation of an equal fraction of it. Identical workloads—the throughput, measured in messages or bits per second—are generated by two different means: asynchronously and synchronously. The asynchronous workload is generated by allowing processes to execute their broadcasts at any time. The synchronous workload is generated by requiring processes to broadcast messages periodically at the same time, with the support of the virtual global clock.

Figure 1 Asynchronous (left) shows what happens to message loss as a function of the workload. A single process, purple line, is able to reach full bandwidth usage

without any message loss. Message loss reaches approximately 0.10% for a workload of 650 mbps (10,000 8kB-messages/s) generated by two concurrent processes. If four and eight broadcasters are used to generate the workload, then message loss reaches, respectively, 0.28% and 0.36%. For a workload of 750 mbps message loss grows by almost an order of magnitude at every scale; the loss rates for two, four, and eigh broadcasters are, respectively, 1.0% (10 times higher), 2.2% (8 times higher), and 2.5% (7 times higher). As the workload grows, the asymptotic behavior of message loss becomes evident, it grows exponentially. It is worth observing that message loss, even at small rates, have a negative impact on the performance of any broadcast-based protocol, more specifically of Fast Paxos, because they imply message retransmissions, and message retransmissions imply extra work by the protocol. The message loss effect observed in Figure 1 Asynchronous has previously been been observed and used as a motivation to develop a variant of Paxos [15].

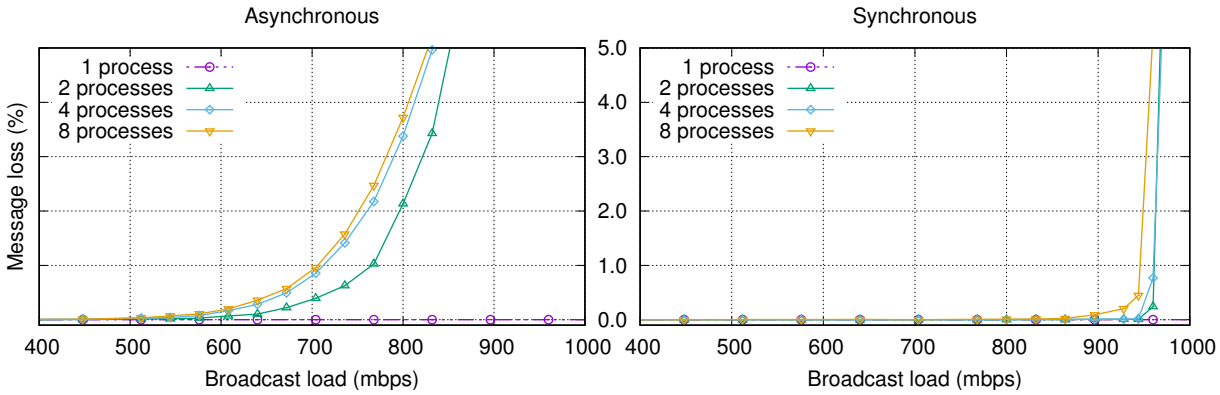

Figure 1: Rates of message loss measured when asynchronous (left) and synchronous (right) processes ip-multicast 8kB messages in a Ethernet-based local area network.

By coordinating the processes and restricting the ip-multicast of messages exact the same instants of time, the results are considerably improved. Figure 1 Synchronous shows that message loss remains below 0.10% up to an workload of approximately 900 mbps for scales 2, 4, and 8 processes. If a single process (purple line) is used to generate the workload then, obviously, there is no communication contention and the full 1Gpbs bandwidth is used. As the workload reaches approximately 960 mbps, message loss rates becomes relevant. In the graph it is clearly noticeable that an inflection point has been reached for the configurations of broadcasters with 2, 4, and 8 processes within a very small workload interval, from aproximately 945 mbps to 960 mbps, message losses depart from 0.0% to reach 0.24%, 0.77%, and 5.4%, respectively. From this point on, message loss increases rapidly. The lesson learned from the comparison of the message losses for the asynchronous and synchronous workloads is clear: synchronous broadcasts make better use of the bandwidth available. In fact,

as detailed next, when proposers are synchronous, On-Time Fast Paxos can sustain throughputs up to 930 mbps.

7.3 Fast Paxos: Synchronous Siblings

In the next section we analyse the performance of four total order broadcasts based on Fast Paxos in order to evaluate the impact of

different designs of the communication substrate. The four implementations thus share the same asynchronous consensus protocol

but differ in the way broadcast messages are disseminated, and mapped to instances of consensus. The first implementation corresponds to the traditional, asynchronous way of obtaining total order atop Fast Paxos, where acceptors allocate messages to successive instances of consensus, so as to represent their reception order. The other three are implementations of our On-Time Fast Paxos, in which broadcast messages are mapped to the instance of consensus corresponding to the instant of global time at which they are received. In the three implementations acceptors operate atop the synchronous communication protocol, which proposes at each clock tick a (possibly empty) set of messages for consensus. They differ then by the way proposers operate to initiate the broadcast of messages.

	Agent Synchrony	
Protocol	Proposer	Acceptor
Original Fast Paxos (FP)	asynchronous	asynchronous
Timed Acceptor Fast Paxos (TA)	asynchronous	synchronous
Timed Acceptor-Proposer Fast Paxos (TP)	synchronous	synchronous
On-Time Fast Paxos (OT)	synchronous	κ-synchronous

Table 1: Synchrony and Fast Paxos

In the first synchronous protocol based on Fast Paxos, that we henceforth call Timed Acceptor Fast Paxos (TA), proposers operate asynchronously, that is, in the same way as in the Original Fast Paxos (FP). Namely, proposers are asynchronous broadcasters that can ip-multicast messages at any time. This approach has advantage of being more generic, since proposers do not necessarily need to be part of the protocol, but also disadvantages in terms of the reliability of communication, as presented in Section 7.2. In the Timed Acceptor-Proposer Fast Paxos (TP) protocol, in turn, proposers are also synchronous (see Section 6.2), with operation dictated by the virtual global clock. So at each clock tick proposers may ip-multicast a batch of messages which broadcast was requested during the inter-ticks interval, if any. Finally, the On-Time Fast Paxos (OT) protocol is the most synchronous of the three,

as proposers not only are synchronous broadcasters but also participate in the mapping of messages to instances of consensus. As detailed in Section 6.2, proposers may stipulate a minimum latency κ for the delivery of the messages they broadcast, so that the temporal information provided by the virtual global clock not only to pace their operation, but also to generate ordering.

7.4 Results and Discussion

In the experiments presented in this Section a fixed setup for the agents was used for the four protocols. The number of acceptors was set to four, the least replication factor for which Fast Paxos tolerates failures. The size of fast and classic quorums were then set to three acceptors, so that to tolerate the failure of one acceptor. The coordinator was allocated to a fifth process, which in the case of the timed protocols accumulate the role of synchronizer (the master of the virtual global clock protocol, see Section 6.1). The workload is generated by other four processes, which play the roles of proposers and learners. Having proposer and learner co-allocated enables a more accurate measurement of latencies, since the broadcast and delivery of messages occur in the same process. A client thread in each of these processes requests the broadcast of 1 kB random messages to the associated proposer at exponentially distributed intervals. The experiments had duration of 60 seconds, and the data collected in the first 30 seconds, when the broadcast rate is progressively increased up to the target rate, is disregarded. In the experiments in which proposers are synchronous broadcasters, the optimal batch size (see Section 6.2) was set to 14 kB. Finally, in the experiments with timed agents the granularity of the virtual global clock was set to $\Delta = 500\mu s$, that is, 2000 clock ticks per second.

The summary of the data collected during the experiments is depicted in Figures 2, 3, and 4. The x-axis of the graphs present the average throughput measured for each configuration (essentially the target broadcast rate) by the learners. Each point depicted in the graphs is generated from the data of at least 5 executions of the experiment with a given configuration. The graphs present relevant aspects of total order broadcasts based on Fast Paxos. Figure 2 summarizes the performance of the four implementations, under the form of the average latency as a function of the achieved throughput. In Figure 3 we depict the rate of collisions, relative to the number of consensus instances initiated, measured by the coordinator. As we observed two very different behaviors, with low rates less than 0.5% and high rates that can exceed 80%, we use logarithmic scale on the y-axis. Figure 4 depicts the average CPU usage in the acceptors, relative to the capacity of a single core (machines have 8 cores), as a function of the rate of messages (or bits) they are able to order per time unit. It is a relevant metric when we consider, in particular, that performance can be severely hampered by an excessive CPU usage.

The first relevant aspect observed in Figure 2 is that all three timed protocols

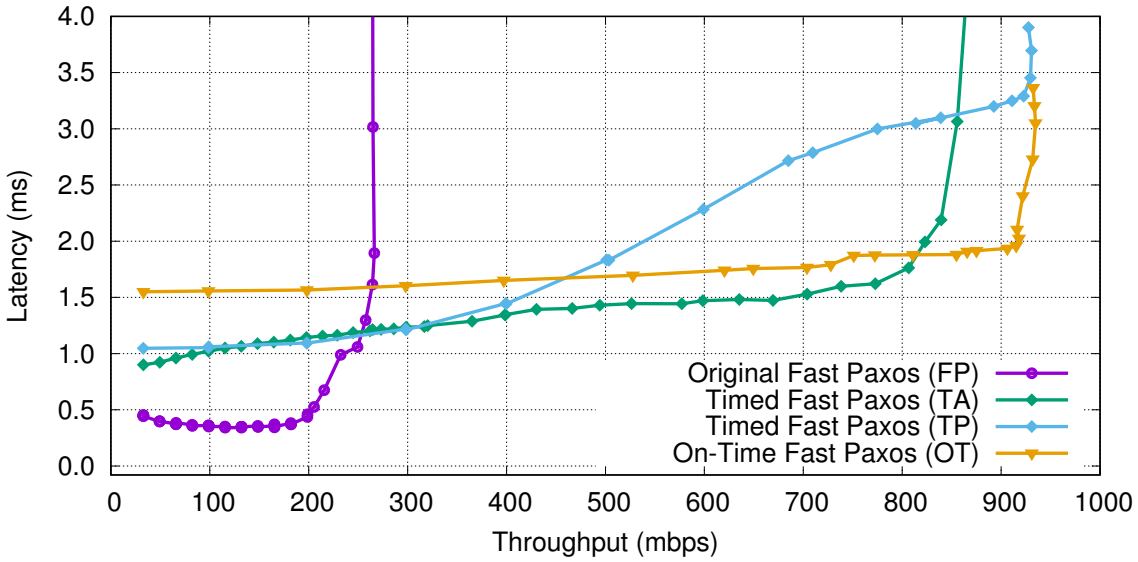

Figure 2: Performance of the four total order broadcasts based on Fast Paxos with 4 proposers and 4 acceptors, in the form of average latency in function of throughput.

based on Fast Paxos achieve high throughput. On-Time Fast Paxos slightly exceeds 930 mbps, the TP protocol hits 930 mbps, and the TA protocol can reach 850 mpbs of throughput, with corresponding average latencies in the range 3.0 to 3.5 ms. Original Fast Paxos, in turn, can only maintain a reasonable latency behavior up to about 265 mpbs, when the average latency increases very rapidly. It is also true that under low load, up to 200 mbps Original Fast Paxos presents latencies substantially lower than the timed protocols: with no contention, FP latency is tipically below 0.4 ms; with timed acceptors, TA and TP have latencies up to 3 times higher (around 2Δ); finally, with a minimum latency $\kappa = 2$ On-Time Fast Paxos has latency about 4 times higher than FP, slightly above 3Δ. It is interesting to note that, at least up to 300 mbps, TA and TP protocols have very similar latencies. Recall that TP differs from TA because it conditions not only acceptors but also proposers; so, in theory, the latency of TP (expected to be around 2Δ) would be higher than that of TA (with average expected to be up to 1.5Δ). This is not the case and the probable reason is the higher collision rates of TA when compared with TP, especially at these low loads, where the difference is more than one order of magnitude.

To better understand what happens to the timed protocols TA and TP in the central range of load, between about 300 mpbs and 800 mpbs, is interesting to move to Figure 3. Note, first, that the execution of the two protocols, TA in particular, are dominated by collisions: at 300 mpbs of load the collision rate for TA was about 57% and 7% for TP; at 700 mbps, rates reach their picks of 88% for TA and 64% for TP. In the case of TA protocol, the very high collision rates is a result of the

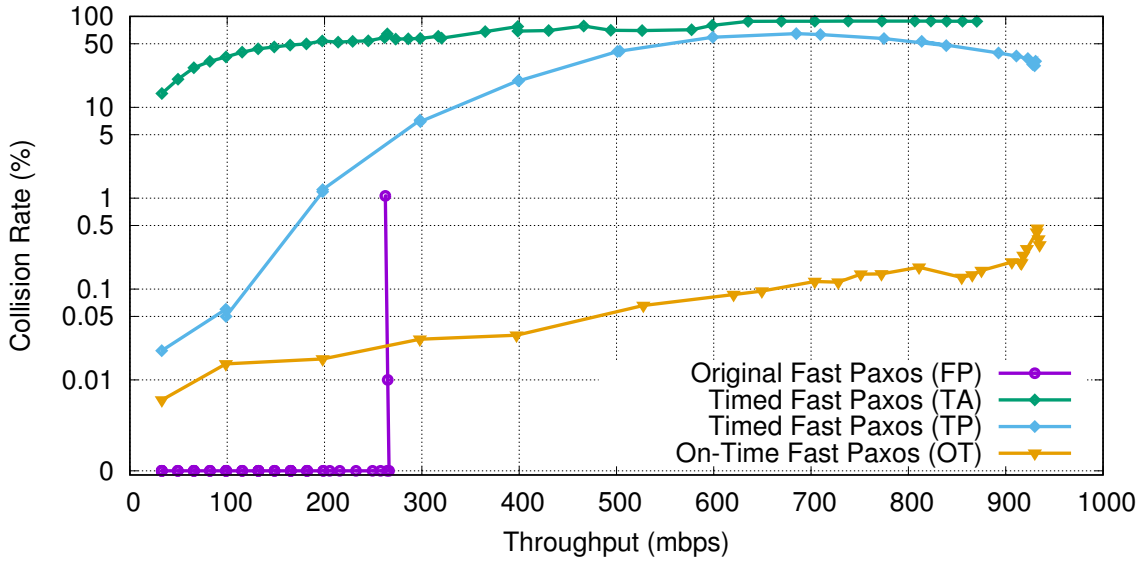

Figure 3: The rate of collisions, relative to the number of instances of consensus started, in log scale, as a function of the throughput achieved with 4 proposers and 4 acceptors.

high rate of relative small messages the proposers broadcast: between 19 and 48 per clock tick ($500\mu s$), on average. Just one such message received at different clock ticks by proposers is enough to cause, not only one but probably two, collisions. As TP protocol restricts broadcasts to discrete instants of time, the number of messages per clock tick can not exceed the number of proposers, but the average size of the batches of messages broadcast increases with increasing load. The point is that depending on the broadcast message size, the network load, and the number of concurrent proposers, the expected latency may be close to multiples of the clock granularity Δ. This triggers a scenario where part of the acceptors receives a given message just before a clock tick, while others a little later the same clock tick. It is a special situation in which minor latency variations may trigger collisions.

The adoption in On-Time Fast Paxos of a minimum latency for broadcast messages, related to the clock granularity, stems from the observation of this situation in the TP protocol. As a result, collisions affect less than 0.2% of the instances of consensus in the OT protocol up to 915 mbps; an impressive improvement from a simple modification of the communication protocol. However, it is worth noting that despite the significant collision rates observed for the TA and TP protocols, they manage to achieve high throughput, while providing increasing but still fairly reasonable latencies. This is possible because, despite the broadcast load, the number of instances of consensus started per time unit remains unchanged: one per clock tick. Thus, there is time to recover instances affected by collisions, without actually compromising the

effectiveness of newly started instances. But as observed in Figure 2, there is an important impact on latency as a result of such collision rates, in particular for TP protocol due to its contention mechanisms.

Protocol	Throughput	Av. Latency	Collisions Rate	CPU usage
FP	198.72 mpbs	0.44 ms	0.000%	222%
TA	806.55 mpbs	1.76 ms	88.8%	189%
TP	910.98 mbps	3.25 ms	36.7%	134%
OT	915.17 mpbs	1.96 ms	0.195%	120%

Table 2: Performance for configurations with maximum throughput/latency ratio.

We summarize the performance of the four protocols assessed in Table 2, by selecting the configuration with the best trade-off between throughput and average latency for each of them. In this critical configuration, the ratio throughput/latency is maximal, meaning that subsequent increases in throughput come with the cost of a considerable increase in latency. Examples of this behavior are observed for Original Fast Paxos (FP), considered in the last configuration with latency below 0.5 ms, and for the TA protocol, considered before the inflection of latencies noticed from throughput 800 mbps. It is also relevant the very high throughput achieved by On-Time Fast Paxos, of about 91% of the network capacity, while sporting a reasonably low latency, below 2.0 ms. The TP protocol reaches similar throughput but with 66% higher latency, which is nevertheless impressive given the cost of dealing with collisons in 36% of the instances of consensus executed.

Figure 4 allows us to make an argument for the relatively poor performance observed for the Original Fast Paxos. Observe that out of the four protocols, FP has from the outset a high CPU usage, and it grows very fast (slope of purple curve) as the workload is increased. Figure 3 can be used in conjunction with Figure 4 to observe that there is a point after which the collision rates jump to very high values. What happens is that the treatment of collisions requires extra CPU usage, that in its turn, increases the chances of new collisions, as the acceptors become increasingly overloaded, thus more asynchronous. At the limit, the cascading effect of increased CPU usage coupled with increased collision rates stalls the acceptors as they become very busy trying to process the ever increasing backlog of pending consensus instances and the reception and ordering of new proposals.

8 Conclusion

This paper has presented On-Time Fast Paxos, a synchronous total order broadcast that results from the principled application of time and synchrony to design an algo-

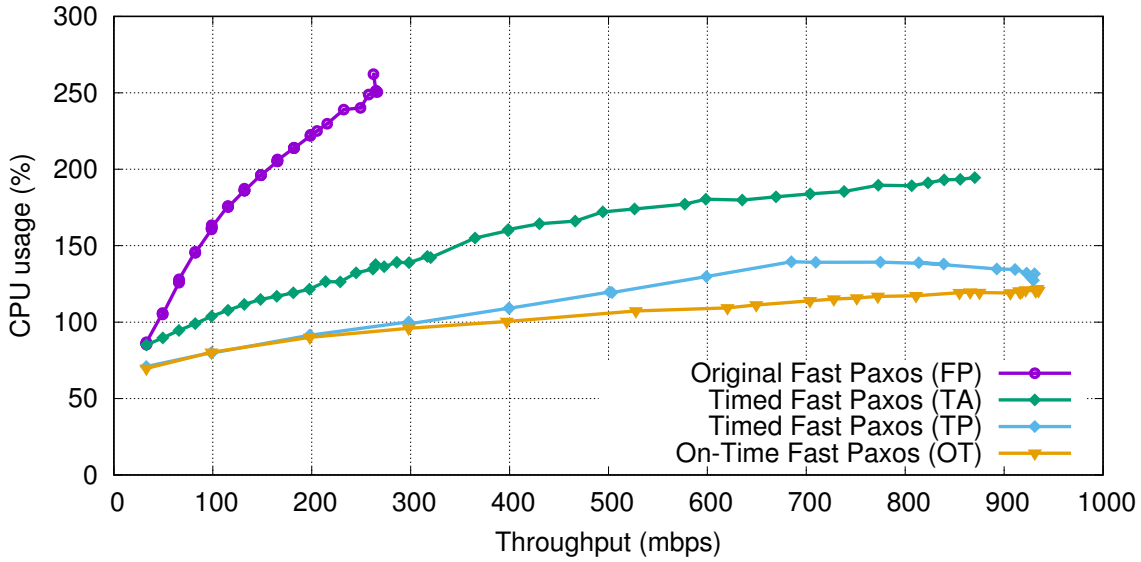

Figure 4: Average CPU usage relative to a single core measured for 4 acceptors as a function of the rate of messages (bits) from 4 proposers they can order per time unit.

rithm that paces the operation of Fast Paxos, an asynchronous consensus protocol for the crash-recover failure model. The paper has briefly contextualized the use of consensus to implement total order broadcast and summarized the Fast Paxos algorithm. On-Time Fast Paxos has been described using a top-down approach. The topmost layer of On-Time Fast Paxos consists of a Fast Paxos modified to follow the tempo of timed broadcasts generated by the synchronous communication protocol; the lowest layer of On-Time Fast Paxos. The core of the protocol lies in the mechanisms for the activation, operation, tempo maintenance, and message loss that had to be designed and implemented to allow the on-time generation of total order broadcasts using Fast Paxos. Also key to the design of the protocol is the use of a virtual global clock and synchronous communication layer capable of dynamically adapting themselves to the requirements of the timely execution of consensus instances.

The experimental design created to assess the characteristics of On-Time Fast Paxos considers two sets of experimentos. The first set has been designed to compare the performance of asynchronous versus synchronous ip-multicasts (broadcasts) over an Ethernet network; probably the most common local area network used in the world. The results show that message loss is much better behaved when processes make synchronous broadcasts. This is key to our argument that conditioned synchronous broadcasts can be used to implement fast performing total order broadcasts. The second set of experiments assess the behavior of On-Time Fast Paxos through the following metrics: throughput (mbps) versus latency (ms), throughput (mbps) versus collision rates (% of message loss), and throughput (mbps) versus CPU

usage (% of a single processor). Four variations of total order broadcast protocols are assessed: the Original Fast Paxos (FP), a Timed Acceptor Fast Paxos (TA), a Timed Acceptor-Proposer Fast Paxos (TP), and On-Time Fast Paxos (OT). What sets these versions of total order broadcast apart is the use of time and synchrony in ther implementations, FP is asynchronous, then application of synchrony grows in the following order: TA, TP, and OT. In summary, what the results show is that OT has the best throughput (915mbps), smallest collision rate (0.195%) and smallest CPU usage (less then half of the CPU usage of FP, our benchmark protocol), with a latency similar to the raw round-trip latency of the network, approximately 2ms. The figures obtained allow us to conclude On-Time Fast Paxos is indeed very efficient vindicating the use of time and synchrony in the design of total order broadcasts.

References

[1] Emmanuelle Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing (FTCS '97)*, pages 292–301, June 1997.

[2] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *Proceedings of the 6th International Conference on Parallel Computing Technologies (PaCT '01)*, pages 42–50, London, UK, September 2001. Springer-Verlag.

[3] Daniel Cason and Luiz Eduardo Buzato. Time hybrid total order broadcast: Exploiting the inherent synchrony of broadcast networks. *Journal of Parallel and Distributed Computing*, 77(0):26–40, March 2015.

[4] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[5] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus. *Journal of Algorithms*, 51(1):15–37, April 2004.

[6] Riccardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by TEMPO in berkeley UNIX 4.3BSD. *IEEE Transactions of Software Engineering*, 15(7):847–853, July 1989.

[7] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2011.

[8] Hermann Kopetz, G. Grünsteidl, and J. Reisinger. *Dependable Computing for Critical Applications*, chapter Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System, pages 411–429. Springer Vienna, 1991.

[9] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Computers, IEEE Transactions on*, C-36(8):933 – 940, aug. 1987.

[10] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[11] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

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

[13] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, June 2006.

[14] Mikel Larrea and Cristian Martín. Quiescent leader election in crash-recovery systems. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '09)*, pages 325–330, Shanghai, China, November 2009. IEEE Computer Society.

[15] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems Networks (DSN '10)*, pages 527–536, Chicago, USA, June 2010.

[16] Fernando Pedone and André Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, January 2003.

[17] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing (EDCC '02)*, volume 2485 of *Lecture Notes in Computer Science (LNCS)*, pages 44–61, Toulouse, France, October 2002. Springer, Berlin, Heidelberg.

[18] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. On the coordinator's rule for Fast Paxos. *Information Processing Letters*, 107:183–187, August 2008.

[19] Gustavo Maciel Dias Vieira and Luiz Eduardo Buzato. Treplica: Ubiquitous replication. In *Proceedings of the 26th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC '08)*, Rio de Janeiro, Brazil, May 2008.

[20] Gustavo Maciel Dias Vieira, Islene Calciolari Garcia, and Luiz Eduardo Buzato. Seamless Paxos coordinators. *Cluster Computing*, 17(2):463–473, June 2014.

[21] Piotr Zieliński. Optimistic generic broadcast. In Pierre Fraigniaud, editor, *Proceedings of the 19th International Symposium on Distributed Computing (DISC '05)*, volume 3724 of *Lecture Notes in Computer Science (LNCS)*, pages 369–383. Springer, Berlin, Heidelberg, Cracow, Poland, September 2005.