# Consensus Inside

Tudor David
EPFL, Switzerland
tudor.david@epfl.ch

Rachid Guerraoui
EPFL, Switzerland
rachid.guerraoui@epfl.ch

Maysam Yabandeh
Twitter
myabandeh@twitter.com

## ABSTRACT

Scaling to a large number of cores with non-uniform communication latency and unpredictable response time may call for viewing a modern many-core architecture as a distributed system. In this view, the cores replicate shared data and ensure consistency among replicas through a message-passing based *agreement* protocol.

In this paper, we present the first in-depth study of message-passing agreement on many-cores. In particular, we focus on the possibility of such a protocol being *non-blocking*. We highlight a number of optimizations that are specific to the many-core environment and present 1Paxos, a new non-blocking agreement protocol that takes up the challenges of this environment.

## 1. INTRODUCTION

The consistency of cached data in many-core systems is usually guaranteed by the hardware. Although this approach simplifies software design, a number of studies argued that it does not always scale to a large number of cores [4,34,38]. In future systems with heterogeneous cores, or larger non-uniform systems such as those using RDMA, hardware coherence may furthermore not even be present [4,13,35]. An alternative approach has been recently proposed: view the cores as nodes of a distributed system [4]. Accordingly, communication is not implicit, but rather explicit through a message passing layer. This approach discourages sharing. But when this is unavoidable, e.g. specific application state or configuration information need to be shared by multiple cores, multiple local replicas of the data are created and their consistency is ensured through an *agreement* protocol.

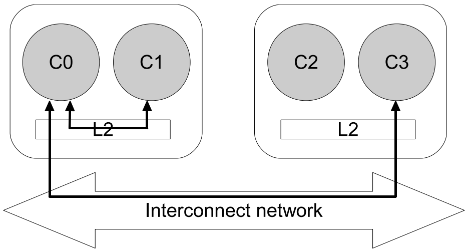

Figure 1: Non-uniform latency in inter-core communication; Cores $C_0$ and $C_1$ share the same last-level cache and communicate much faster than Cores $C_0$ and $C_3$, which have to go through the interconnect network.

Barrelfish [4] pioneered this approach by implementing a multikernel model where the kernel state is replicated on several cores. These cores exchange messages to execute a 2PC-like (two-phase commit) protocol [28], which sits as a middleware between the application and the OS and ensures the consistency of the replicated state among the kernels.[1] 2PC is known to be a *blocking* protocol, for it expects replies from all the participants in order to perform updates, which makes it vulnerable to even a single process being slow to respond. On many-cores, this can easily be caused by an unpredicted load or consecutive cache misses, especially in a large scale system, which is the precise target of the distributed system vision of a many-core. Indeed, upon a cache miss, loading the data from the memory takes around 100 ns,[2] i.e., ∼10 times longer than loading data from cache. If the data is swapped out to the hard disk by the virtual memory manager, the core has to wait until the corresponding memory page is swapped into the memory, which takes around 8 ms, i.e., ∼800K slower than a cache access. The process context switch latency is between 10 and 20 $\mu$s on average and can take much longer because of page faults. The inter-core latency is typically non-uniform in a large scale system. As illustrated in Figure 1, the cores located on the same CPU share the same last-level cache (LLC) and hence can communicate much faster than the cores located on different CPUs. On such an architecture, a blocking protocol (i.e., that needs responses from all the participants) is thus problematic: any such delay increases the overall waiting time.

Non-blocking agreement protocols, also called *consensus*

[1] We use the 2PC terminology here similarly to Baumann et al. [4] (we could also have used the term *primary-backup* [3]): in fact, 2PC is used in [4] solely in its agreement form, and not in its full transaction commitment sense (with disk storage etc.); it is this Barrelfish version of 2PC that we will be referring to throughout this paper

[2] The memory access time is highly dependent on the memory architecture and can range from 50 ns to 150 ns.

protocols [15], constitute an appealing alternative. These can progress with responses from a strict subset of replicas [6, 31], ignoring the slowest ones. Particularly desirable is a message-passing consensus that tolerates ”crashes”[3] as well as ”asynchrony”, namely arbitrary long delays in the communication between the nodes. The combination of the very notions of *crashes* and *asynchrony* models the communication scheme underlying many-core systems with *non-uniform* communication latency and *unpredictably* slow cores [18] quite well. By assuming that the nodes could be non-responsive for an arbitrarily long period of time, we capture cores that are running slow because of contention on shared resources, e.g., the CPU cycles, the cache, and the hard disk (after a page fault) and prevent the protocol from waiting for all of them.

A family of practical message-passing consensus protocols has been recently developed with IP networks in mind [8,22,24,27,32,33,36]. Multi-Paxos [27] is arguably the most efficient such protocol; it has been implemented in a wide variety of IP network settings [5,8,20,30]. Due to their non-blocking nature, any of these protocols may at first glance seem a good candidate for the many-core task at hand. However, a closer look reveals that none of them meet the very requirements of the many-core context. Roughly speaking, this is basically because, although it looks alike, a many-core is not a genuine distributed system in the classical IP sense: as we shown in Section 2.3, although Multi-Paxos deployed in a local area network (LAN) scales well to hundreds of clients, it saturates very quickly in a many-core setting, with only a few clients.

With message passing inside a many-core, the cycles of the sending/receiving core itself are taken as the transmission delay of the message. The more message transmissions per core are required by the agreement algorithm, the sooner the cores' processing power saturates. Since inter-core latency, i.e., the message propagation delay, is much lower in many-cores than in IP networks, a much higher rate of potential throughput is induced, which in turn magnifies the message transmission bottleneck for throughput scalability. Besides the typically large number of messages transmitted in agreement protocols, many such algorithms, e.g., Multi-Paxos, require that a request goes through a specific node that leads the agreement, making the message transmission load at the *leader* the bottleneck of the system [5].

As our experiments in Section 3 show, the ratio of *transmission delay* to *propagation delay* (*trns/prop*) inside a many-core is at least two orders of magnitude larger than in an IP setting. In other words, the transmission delay is a main contributor to the overall latency of the algorithms. A slightly smaller number of messages processed in the fast path of the algorithm has a high impact on the overall latency experienced by the client processes. The algorithms inside a many-core, including agreement algorithms, should therefore be designed with the objective of minimizing the number of messages produced. This is in contrast to the common practice of algorithm design in IP networks, where the main objective was to minimize the number of round trip delays (i.e., propagation delays) in the fast path of the algorithm. In addition, while in an IP setting the communication links are unreliable, this is currently not a problem on many-cores, where link failures and network partitions are not an issue.

Taking these observations into account, we designed and implemented 1Paxos, a consensus protocol tailored to many-cores. Very intuitively, 1Paxos reduces the agreement-related traffic in general, and more specifically that of a leader. A key insight underlying 1Paxos is the observation that the role of *acceptor* in Paxos-based protocols, i.e., to resolve conflicts among possibly multiple leaders, can be played by a single node.[4] We change the type of redundancy used in the acceptor role of Paxos to reduce the number of sent messages, thus enabling higher bandwidth and number of agreements achieved per second in many-cores. Making use of a single acceptor introduces some technical difficulties (that we discuss in the paper), but otherwise leads to much less traffic, yet without jeopardizing the consistency of the system. In terms of availability, using three cores, 1Paxos can progress even with one slow (or non-responsive) core, just like in the Paxos family of protocols. With a higher replication degree however, there is a trade-off: 1Paxos does not progress as long as neither the leader nor the active acceptor are responding. This, we believe, is less problematic in a many-core, where cores do not require manual recovery to start operating again, but might respond relatively slowly for a while, and network partitions do not appear (which might happen in a WAN).

We consider various protocols and measure (1) their commit latency and throughput; (2) their scalability with the number of cores; and (3) their performance when a core becomes slow. We report on our evaluation on a machine with eight 2.1 GHz Six-Core AMD Opteron(tm) processors (48 cores in total). Our implementation of these protocols was itself technically challenging due to the lack of availability of an efficient framework for message passing inside a many-core.

In short, our results show that 1Paxos scales significantly better than Multi-Paxos and 2PC. Moreover, 1Paxos can progress with slow cores and in the worst case scenario where the leader is slow, 1Paxos replaces the leader and continues with the same throughput, whereas a 2PC update blocks as long as any node is not responding. Our results highlight another trade-off: indeed, reading replicated data can be performed faster with a blocking protocol, i.e. the read can be executed locally. However, as soon as the workload contains a small percentage of write operations 1Paxos becomes more appealing. For more relaxed read consistency guarantees, local reads may be performed even with non-blocking protocols.

To summarize, the main contributions of this paper are as follows:

1. We present the first in-depth study of agreement on many-cores and investigate the challenges of implementing message-passing consensus in this setting. We

[3] The notion of ”crash” used here does not necessarily mean the cores stopping any activities forever. It simply models slow ones.

[4] The presence of multiple leaders can typically be caused by asynchrony: a new leader might be elected if the former leader is non-responsive, even only temporarily.

support our analytic study with extensive evaluation on a 48-core machine.

2. We highlight fundamental differences between the network inside the many-core and the IP network.
3. We introduce 1Paxos, a variant of Paxos that takes up the challenges of the many-core environments by reducing the number of processed messages for achieving an agreement.

The rest of the paper is organized as follows. Section 2 recalls the message-passing vision on many-core systems as well as the design of basic blocking and non-blocking agreement protocols. Section 3 describes the main network characteristics of a many-core system, highlighting the differences with a LAN. Section 4 gives the key insight underlying 1Paxos. The detailed design of 1Paxos is presented in Section 5. Section 6 presents our message-passing framework. We present our experimental results in Section 7. Section 8 presents related work. Section 9 concludes the paper with some final remarks. Appendix A presents the pseudo code of 1Paxos which is followed by the correctness proofs of 1Paxos in Appendix B.

## 2. BACKGROUND

In this section we discuss consistency and the role of message-passing agreement in the context of many-core systems. We also give a brief overview of 2PC [28], as an example of a blocking agreement protocol and Paxos, as a way of achieving non-blocking agreement.

### 2.1 Consistency in Many-core Systems

A major scalability bottleneck in many-core systems is induced by the need to keep the cached data consistent among multiple cores. The developers expect to have the same view of data, independently of which core the processes are running on. However, two cores might have loaded the same data into their caches or local memory, and changes into the loaded data in one of the cores is not by default observable by the others. This gap, between the centralized view of the processor and the distributed implementation inside many-core systems, is typically bridged by hardware techniques, known as *cache coherence* protocols [17]. There are different kinds of such protocols, but the essence is that after a change into a memory address by a core, all cores that have loaded the same address are notified about the change, before doing any computation on that data. In essence, the hardware does not know which cores still need the data for future use. For a large number of cores, this sometimes implies long delays for change propagation and/or a large number of synchronous inter-core message transmissions.

The alternative is to have the software keep the replicated data consistent. This approach could be applied to both the kernel and the user levels when state needs to be shared. Barrelfish [4] applied this vision to the kernel design, exhibiting good scalability. According to this approach, the software handles the consistency of its own data by viewing the entire machine as a large distributed system whose nodes represent the actual cores. If the software assigns two separate cores to process the same data, each core gets its own copy of the data, i.e., replica. It is then the software's responsibility to maintain the consistency of the replicas by exchanging messages to run an *agreement* protocol among the cores hosting the replicas. In Barrelfish [4], the capability system is replicated on the cores and a 2PC protocol [28] keeps the replicated state consistent among the kernels.

### 2.2 2PC

The 2PC (two-phase commit) protocol, as its name suggests, has two phases. In the first phase, the coordinator (the leader) sends a prepare message to the replicas. Each replica locks its local copy of data and responds with an ack message if it is not already locked by another coordinator. The coordinator starts the second phase by broadcasting a commit message to the replicas, but only if it receives an ack from all of them. In this case, each replica executes the command of the commit message and releases its lock, which is followed by a commit_ack message back to the coordinator. Otherwise, the coordinator broadcasts a rollback message to the replicas. Upon receiving a rollback message, each replica releases its lock if it is already acquired by the corresponding ack message. As Section 7 will show, the many generated messages required by 2PC limit its efficiency inside a many-core.

In order to verify the impact of a highly loaded core on a blocking protocol, we measure the throughput of 2PC when the leader becomes slow, as well as the normal non-faulty case. In this setup, five clients are sending requests to three replicas. The experiment is run on a machine with four 2.4 GHz Dual-Core AMD Opteron(tm) processors (8 cores in total). The replicas are assigned to Cores 0 to 2, Core 0 being the coordinator, and run 2PC between them. We slow down Core 0 by running 8 CPU-intensive processes on it; each process is a bash script that continuously multiplies a number by itself. As expected, after Core 0 becomes slow, only a few requests can commit and the throughput drops to zero.

### 2.3 Consensus

Unlike 2PC, non-blocking agreement protocols, also called consensus protocols [31], tolerate slow processes. They require responses from only a majority of the nodes to progress, thus tolerating the *crash* of a minority. Whereas crashes are considered common in classical distributed systems, in a many-core environment, these model *slow* cores. As we pointed out in the introduction, asynchrony, on the other hand, models the tolerance to delayed messages. We recall below the celebrated Paxos consensus protocol [27] and its Multi-Paxos optimization, proposed in [27] and used in [8].

Using the underlying Synod consensus protocol, it assigns a total order to the commands issued by clients and guarantees that all nodes execute the commands in the same order (the term Paxos is often used to designate Synod as well).

*Basic-Paxos.* We now give a brief description of the original Paxos protocol [22, 27], which we call Basic-Paxos hereafter. The participant nodes in Basic-Paxos implement three different roles: *proposer*, *acceptor*, and *learner*. The proposers advocate the client commands, the acceptors resolve the contention between multiple proposers, and the learners learn the chosen values. The *leader* orchestrating the consensus is chosen among the proposers.

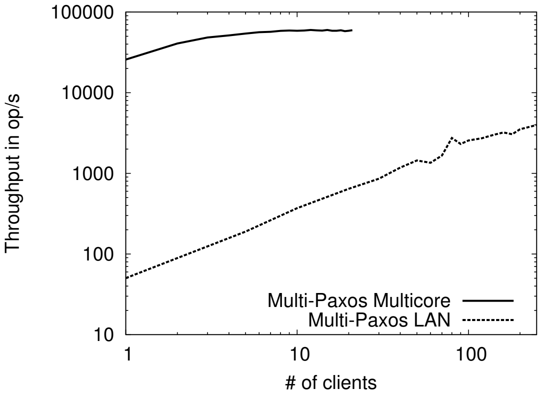

Figure 2: The scalability of Multi-Paxos in LAN compared to many-core systems.

The ultimate goal of Basic-Paxos is to assign totally ordered instance numbers to client commands. To associate values (client commands) with instance numbers, Basic-Paxos requires two phases. In the first phase, a proposer attempts to become the leader for a particular instance number by broadcasting a prepare_request message to the acceptors. Upon receiving a prepare_response message from a majority of acceptors, the proposer becomes the leader of that instance number. In the second phase, the leader proposes a value to the acceptors and the acceptors broadcast the corresponding message to all the learners. A learner learns the proposal after receiving the message from a majority of acceptors. All message transmissions related to a particular order constitute a separate *instance* of Basic-Paxos.

Although each role can be implemented by a separate node, usually a single node implements all three roles (Collapsed Paxos).[5] According to the liveness property of Basic-Paxos [24] a value will be eventually chosen, given that enough nodes are running. For example, in Collapsed-Paxos deployed on 3 nodes, the liveness property holds as long as 2 of the 3 nodes are running. Basic-Paxos guarantees the following two safety properties [27]: (i) non-triviality: only proposed values can be learned; and (ii) consistency: two different learners cannot learn two different values.

***Multi-Paxos.*** After a proposer $p$ takes the leadership position for one instance $in$, it could be more efficient if $p$ assumes this position for the next Paxos instance $in'$ ($in' > in$) as well. The other proposers can still try to become leaders when they suspect that the last leader has failed. Multi-Paxos [22] is the version of Paxos that implements the mentioned optimization.

Multi-Paxos [27] is considered one of the most efficient protocols of the Paxos family; it has been implemented in a wide variety of IP settings [5, 8, 20, 30]. Figure 2 depicts the throughput of Multi-Paxos with three replicas in a local area setting (LAN). By increasing the number of clients, the overall throughput increases. As depicted in the figure, up to a hundred clients the throughput increases (the scale on the X axis in Figure 2 is logarithmic).

This is not the case in a many-core system: after the third client, the throughput changes only marginally by adding more clients to the system. This is because the processing power of the cores gets saturated very quickly by the many generated messages. In other words, although Multi-Paxos performs well in IP networks, it has a very limited scalability in many-core systems.

## 3. MANY-CORE: A NETWORK VIEW

Applying consensus protocols designed in an IP setting to a many-core does not lead to efficient protocols according to our experiences. The reason is that the characteristics of the networks are different. Here, we measure the main network characteristics of a many-core system, namely the transmission delay and the propagation delay, which we compare to those of a LAN. By transmission delay, we refer to the amount of time necessary to place a message on the propagation medium, while by propagation delay we refer to the time necessary from the moment the sender has written the message, and until the receiver is notified of its arrival. Analyzing the differences, we conclude that reducing the number of messages per core should be the main objective in the design of distributed protocols in many-cores.

In the first experiment, we measure the transmission delay for a message on a many-core using our framework (presented in Section 6). To achieve this, we use a sender process assigned to core 0 repeatedly issuing messages to an unbounded queue. The average duration needed to send a message approximates the transmission delay. The measurements show a transmission delay of $0.5\mu$s in the many-core scenario.

To measure the propagation delay, we consider the following experiment: we again use a sender and a receiving process (which is placed on core 1), this time using a queue that can only hold a single message. Therefore, the sender pauses until it learns that the last message has been read. On the receiver side, the process simply dequeues the requests as they arrive, performing no further computation. In our framework, the amount of processing required to receive a message is very similar to that required to send it. Therefore, we approximate the latency between two consecutive messages being sent in this experiment using the following formula: $latency \simeq 2 * trans + 2 * prop$, where the two transmission delays account for sending and receiving the message, and the two propagation delays account for the time required (i) for the message to reach the receiver and (ii) for the new value of the queue head pointer to propagate back to the sender once the message is dequeued. The value we measured for the latency in this experiment is $2.1\mu$s. Hence, using the value for the transmission delay obtained in the previous experiment, we approximate the propagation delay to $0.55\mu$s. The ratio between the transmission delay and the propagation delay is therefore very close to 1 in this case.

Even in a simplified setting, where the system of message queues would not be necessary, a core sending a message has to at least fetch the corresponding cache line before it can write (which can be taken as the transmission delay), after which the receiver has to in turn fetch it in order to

[5] The advantage is avoiding messages between two roles that are located on the same node.

read [11]. This results in a similar *trans/prop* ratio to the one observed above.

We perform similar experiments for the scenario where the two participants are placed on different machines, and communicate via message passing over a LAN. To measure *prop*, we modify the receiver in the 2nd experiment to immediately enqueue a reply after receiving a request. We therefore measure the interval from the moment a request is sent and until a reply is received. We can estimate this latency as $latency \simeq 4 * trans + 2 * prop$.

The measured transmission delay in the LAN scenario is around 2 $\mu$s, while the propagation delay is around 135 $\mu$s. Therefore, the ratio between the two is of about 0.015. However, network propagation delays are often unpredictable, and the value of this ratio may become much smaller than in this ideal case.

As the numbers presented above indicate, the ratio between the transmission delay and the propagation delay is much larger in the case of a many-core when compared to an IP setting. In other words, the transmission delay is a main contributor to the overall latency of the protocols. Even a small reduction in the number of messages processed in the fast path of the protocol has a high impact in the overall latency experienced by the client processes. Therefore, the protocols inside a many-core, including consensus protocols, should be designed with the objective of minimizing the number of messages produced.

# 4. RETHINKING PAXOS

In this section, we rethink the design of Paxos based on the observations we made in Section 3: the number of processed messaged per core should be minimized. To be able to do so, we need to first analyze the internals of Paxos and see which of the components could be changed to achieve the targeted optimization. Based on this analysis, we devise a new consensus protocol, 1Paxos, that reduces the number of messages the leader has to process. A major specificity of 1Paxos is the use of only one active acceptor at a time. In the following, we first distill the different roles of classical Paxos, including the acceptor. We then explain the rationale behind the replication of each role before highlighting the main insight of 1Paxos: in short, not replicating the role of the acceptor.

## 4.1 The Roles in Paxos

There are three major roles in Paxos: (i) proposer, (ii) acceptor, and (iii) learner. The proposer role is to advocate the client's command. This is essential for the scalability of the system. By relinquishing this task to the proposers, the consensus is required among only a few nodes and is thus more scalable with the number of clients. The learner is the actual long-term memory of the system. When a Paxos instance is completed successfully and its value learned, this value is kept in the multiple available learners. The clients can then read this value from each of the learners.

The acceptor is somehow the main safety-guard in Paxos. If multiple proposers want to propose values for the same Paxos instance, the acceptor is key to resolving the contention between the competing proposers. Suppose some acceptors accept value $v_0$ from proposer $P_0$ and, for some reason, the Paxos instance does not complete successfully. Now, to finish the instance, proposer $P_1$ must first read the *accepted value* by the acceptors (i.e. $v_0$) and propose the *same value*. It implies that the acceptors play the role of the short-term memory for the system; they must remember a few values during the short period of one Paxos instance.

## 4.2 Replication in Paxos

At the heart of the efficiency of 1Paxos lies the observation that replication can be used for different purposes. In general, we have two types of replication: (i) replication of service and (ii) replication of data. Replication of service increases the availability of the system. In other words, when a client requests for the service, we want to make sure that there is at least one responding node, ready to receive the client commands. The replication of data, however, is for increasing the reliability of the system. In other words, it decreases the chance of data loss by missing some nodes (after permanent failures). The roles in Paxos are replicated, but each one for a different purpose.

The replication of the proposers is to increase availability, as the proposers provide a service to the clients, i.e., advocating their request. In contrast, the learners store the data of the system, and the purpose of their replication is to enhance reliability.[6]

The acceptor replication is partly for service availability and partly for data reliability. The proposers start the consensus procedure by contacting the acceptors. These require the availability of the provided service. In addition, as mentioned before, there are a few data kept by the acceptors such as the accepted value and the promised proposal number, which should be kept during the Paxos instance. However, this data is required only for the active Paxos instance, and in the case of failure, we can think of some workaround solutions. An important insight in the design of 1Paxos, which will be explained later, comes from the following observation: the replication of the acceptor role is mainly for availability, and if its availability is provided via other mechanisms, then the replication of the acceptor is no longer necessary.

## 4.3 One Acceptor is Enough

In the following, by comparing 1Paxos with Multi-Paxos, the most efficient variation of Paxos used in practical settings [8], we explain why the design of 1Paxos is appropriate in a many-core system. Figure 3 depicts message transmission in a collapsed Multi-Paxos setup that consists of three nodes. The messages that cross the node boundary must be included in the total number of messages.

A crucial parameter is the number of sent/received messages by the leader node. The leader exchanges more messages compared to the other nodes and hence, when it gets saturated, the system cannot process more client commands.

[6] From performance perspective, one can take advantage of replication to increase scalability as well. For example, Mencius [32] uses proposer replication to enhance the scalability. Moreover, if the application does not demand the very last state of the system, its read traffic can be directly serviced from either of the replicated learners.

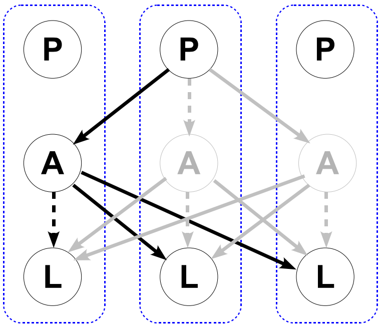

Figure 3: The reduced number of messages in 1Paxos compared to collapsed Multi-Paxos deployed on three nodes. The dotted box represents the node boundary. P, A, and L represent the proposer, acceptor, and learner roles, respectively. The grayed acceptors and consequently the communications to/from them are eliminated in 1Paxos.

As depicted in Figure 3, reducing the number of acceptors to one decreases the number of messages processed by the leader.

As we explained earlier, the availability of the acceptor role can be provided in different ways. One approach, which is taken by Multi-Paxos, is the replication of the acceptor. A side-effect of this approach is the increase in the number of exchanged messages between acceptors and other roles. An alternative approach is to rely on *backup acceptors*, and replace the failed (or suspected to be failed) acceptor with a new fresh one. The backup acceptors do not participate in the normal execution of the protocol and do not, hence, increase the message complexity of the protocol. This idea is the main insight underlying 1Paxos, which reduces the number of produced messages by a factor of two. This is however not the only concept that separates it from Multi-Paxos: although the use of backup acceptors addresses the problem of the acceptor availability and yet provides better performance, it poses the non-trivial problem of the reliability of acceptor's data, which we discuss now.

Recall that the acceptors also keep a few data, which is necessary during the short-term period of a single Paxos instance to address the possible contention between multiple proposers. Missing this data, by switching from the active acceptor to a fresh backup acceptor in the middle of a Paxos instance, can violate system consistency. For instance, if the active acceptor promises not to take any proposal number less than *pn*, then a fresh new acceptor would not be aware of this promise and might accept proposal numbers less than *pn*. Nevertheless, if the proposers get properly notified of this data loss, they can safely restart the Paxos instance without risking consistency. For example, upon receipt of the failure notification of the active acceptor, the proposers know that the promised sequence number by the previous acceptor is no longer held.

We explain in Section 5 that, if we assume that the leader and the active acceptor nodes do not fail at the same time, then there exists a procedure through which the leader can safely notify the other proposers of the active acceptor switch. This is the same assumption that is already made by Paxos in the common setup that consists of three nodes implementing three proposer, three learner, and one acceptor roles. By carefully placing the proposer and acceptor roles in a way that the leader and the active acceptor are separated, we can make the assumption that the leader and the active acceptor do not fail at the same time. This assumption cannot be violated unless two of the three physical nodes fail. In this case, we would be left with one node which is less than the minimum required nodes for Multi-Paxos to progress ($min > total/2$).

## 5. 1PAXOS: THE PROTOCOL

In this section, we detail our protocol, 1Paxos. As mentioned in Section 4, the main idea is to use only one active acceptor and ensure availability via backup acceptors. Care must be taken to preserve the reliability of the acceptor's data when the active acceptor is replaced. We start this section by describing the communication scheme underlying 1Paxos in the failure-free case where messages are received in a timely fashion. We then discuss the backup cases executed when the cores are faulty.

### 5.1 The Failure-free Case

The roles in 1Paxos and the interaction between them for the failure-free case is depicted in Figure 3.

1. Proposer $P$ decides to take the position of the leader. It first obtains the Id of the active acceptor, $A$ (we will explain how we obtain this Id in the next subsection), and sends a prepare_request message including a proposal number, *pn*, to acceptor $A$. By doing so, the proposer asks the acceptor to recognize it as the leader.

2. If the proposal number, *pn*, is greater than all previous proposal numbers received by the acceptor, the acceptor sends a prepare_response message back to $P$. By doing so, the acceptor promises not to accept any proposal number smaller than *pn*. Notice that these two steps are necessary only the first time a proposer contacts the acceptor. After that, the proposer becomes leader and skips these two steps.

3. $P$ then sends to $A$ an accept_request message including proposal number *pn* as well as a proposed value.

4. When acceptor $A$ receives the accept_request message corresponding to the proposal number to which it has given its promise, it accepts the proposal and broadcasts a learn message to all the learners.

### 5.2 Switching Acceptor

We consider the scenario in which the active acceptor $A$ fails (does not respond in a timely manner). When the active acceptor fails, the leader is the only node that is allowed to replace it with another backup acceptor $A'$. This change, however, must be confirmed by a majority of nodes. This is necessary to avoid having multiple instances of active acceptors running in the system, compromising consistency. The scenario is illustrated in Figure 4.

Obtaining the confirmation of a majority of the proposers is a separate consensus problem that can be solved by any Paxos-like protocol (a similar strategy of falling back to a different protocol when needed has been used by Guerraoui

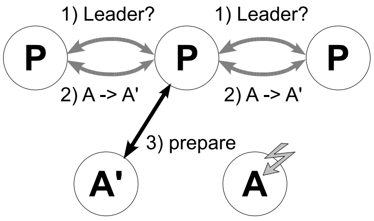

Figure 4: The interaction between nodes in 1Paxos to replace failed acceptor $A$ with another backup acceptor $A'$. In Step 1, the leader verifies that it is still known as the leader by a majority of nodes. Then in Step 2, it announces the change of the active acceptor. Finally in Step 3, it sends a prepare_request message to the new active acceptor $A'$.

et al. [16] in the context of BFT protocols). Although it is possible to merge this consensus into the main operation of 1Paxos, for the sake of simplicity of presentation, we assume that the consensus over the new active acceptor is achieved by a separate basic implementation of Paxos, which hereafter is called PaxosUtility. Notice that the PaxosUtility instance that handles consensus over the new active acceptor is totally separate and independent from the 1Paxos protocol that we are explaining here. Moreover, running PaxosUtility does not require any extra nodes; it runs on the same nodes as 1Paxos. Beside the Id of acceptor $A'$, the leader also includes the uncommitted proposed values into the message sent to the PaxosUtility. This is to cover the cases where acceptor $A$ has received an accept_request message with value $v_{in}$ for instance number $in$, but the corresponding issued learn message is not received by the other nodes yet. In this way, it guarantees that the next leader will try to propose the same value as $v_{in}$ for instance number $in$. After finishing the consensus over the active acceptor, the leader switches from acceptor $A$ to acceptor $A'$, i.e., the new active acceptor. Because the acceptor node has changed, the leader must start over with a prepare_request message to take the leadership of the new acceptor.

## 5.3 Switching Leader

In principle, every proposer could spontaneously try to take the leadership position by sending a prepare_request message to the acceptors. In practice, this usually happens when the current leader is non-responsive (i.e., fails). Similarly, when the leader fails in 1Paxos, any proposer can try to take its position by sending a prepare_request message to the active acceptor. Assume that proposer $P'$ suspects the failure of leader $P$ and decides to become the leader. The active acceptor Id, $A$, can be obtained by inquiring a majority of the nodes. This is due to the fact that the last leader does not use the new active acceptor unless it obtains agreement from a majority of nodes. The sequence of exchanged messages is depicted in Figure 5.

Care must be taken to ensure that, in the meanwhile, active acceptor $A$ is not replaced by the last leader. Otherwise, we end up with two leaders which use two different active acceptors. To this aim, proposer $P'$ uses PaxosUtility to start a consensus instance to announce its leadership position by assuming $A$ as the active acceptor. Accordingly, every leader must always check for this announcement before switching the active acceptor. If the leader observes this announcement, it must consider its position as relinquished. This step is marked as Step 1 in Figure 4.

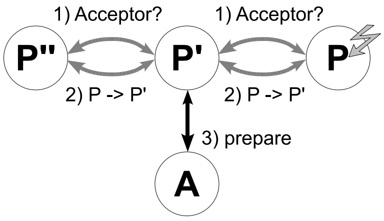

Figure 5: The interaction between nodes in 1Paxos when proposer $P'$ takes the leadership position from leader $P$. In Step 1, proposer $P'$ inquires for the active acceptor Id. It then announces itself as leader in Step 2. Finally in Step 3, it sends a prepare_request message to the active acceptor.

## 5.4 Switching both Leader and Acceptor

If the active acceptor fails, the leader is in charge of replacing it with a fresh backup acceptor. On the other hand, if the leader fails, then any proposer can safely take its position, given that the active acceptor is still running. The only remaining case to handle is when both the leader and the active acceptor fail together.

As mentioned in Section 4, to handle this scenario we carefully assign the 1Paxos roles to the nodes in a way that the leader and the active acceptor are located in 2 separate nodes. Assume that we have $N$ nodes available and each node implements all the roles: proposer, acceptor, and learner. 1Paxos has only one active acceptor, and hence we have the option to pick the node that will also play the active acceptor role. This deployment is depicted in Figure 3. The idea is to assign the active acceptor and leader roles to 2 separate nodes. In this way, the failure of the leader and the active acceptor cannot occur together, unless 2 of $N$ nodes fail at the same time.

Notice that, in the usual setup of consensus, which consists of three nodes, this failure scenario implies that two of the three nodes have failed. On the other hand, (asynchronous) consensus protocols, e.g., the Paxos family, cannot progress with just one running node out of three. Consequently, we can assume that if the failures of the leader and the active acceptor occur at the same time, there is only one node left. In this situation, neither Paxos family of protocols nor 1Paxos can progress.

It is worth noting that, for $N > 3$, the failure of the leader and the active acceptor at the same time does not jeopardize the consistency of the system. It only slows down the progress of consensus until any of these two cores start responding again. In other words, while both the leader and the active acceptor are not responding, it is the *liveness* of the system that is affected, but not its *safety*. Nevertheless, the failure probability of two particular nodes, i.e. the leader and the acceptor, is much less than failure probability of two arbitrary nodes, which makes this failure scenario very rare.[7] The detailed pseudo code of 1Paxos as well as the

[7] For example, if the failure probability of a core is $s$, then the failure probability of two particular cores is $s^2$, and the

correctness proof is covered in a Appendix A and Appendix B respectively.

# 6. MESSAGE PASSING FRAMEWORK

This section presents the architecture of QC-libtask, the message-passing framework that we developed on top of an inter-process shared memory communication in order to support our agreement protocols. A challenge here is to avoid synchronization locks when writing into the communication channels. Another challenge in the design of QC-libtask is to prevent the operating system from being involved in the message-passing process, since system calls induce expensive context switches. Our framework is implemented at the user level using standard C++ libraries, and hence is portable to various operating systems, including Barrelfish [4]. For the purpose of performance evaluation in this paper, similarly to the approach taken by previous work [4], we implement a message-passing paradigm on top of shared memory.[8] Changes made by a process into a shared memory address are first applied to the cache of the core that is running the process. Thanks to the cache coherence mechanism implemented in hardware, the changes in the cache of the source core are *only* propagated into the cache of the destination core.

Notice that although our implementation transmits the unicast messages via a cache coherence mechanism, it is still faithful to the distributed vision of a many-core system, as there are separate channels per pair of cores. In a centralized implementation on top of a cache coherence mechanism, a message would be written into the memory and read by all the cores in the system, resembling a broadcast message. As pointed out in [4], this approach does not scale with the number of cores, since it induces a burst of messages, whereas in our distributed implementation, the software makes use of its knowledge about the application internal to efficiently decide to *where* and *when* each message must be sent. In the following, we describe our messaging system and its integration into a user-level thread library for the efficient delivery of messages.

## 6.1 Message Queuing

As mentioned above, we make use of the cache coherence mechanism by writing/reading to a shared memory address, created using the *shm_open* system call. To implement asynchronous message passing, we use more than one slot (seven by default) for sending messages. The size of each slot is 128 bytes, which is twice the cache line size. Matching the cache line size reduces the number of cache misses for transferring the message. The multiple slots are wrapped into a queue. As illustrated in Figure 6, there are two queues between each two processes $p_i$ and $p_j$: one for writing by $p_i$ and reading by $p_j$ and the other for reading by $p_i$ and writing by $p_j$. Because of separate queues, there is no need for operating system locks to access the queues, which makes the design simple as well as efficient. Each queue has a *head* and a *tail* pointer. The head pointer is moved by the reader and the tail by the writer. The reader process verifies the equality of head and tail pointers to check for new messages.

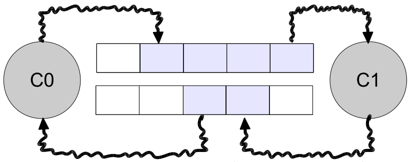

Figure 6: Two separate queues between each pair of cores.

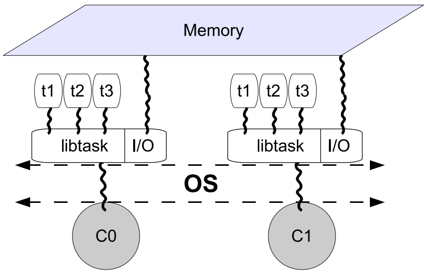

Figure 7: The architecture of QC-libtask.

## 6.2 Message Delivery

As explained above, a process that communicates with $n$ other processes must check for new messages from $n$ separate read queues. After reading a message, the corresponding thread must be notified to process it. To implement this efficiently, we make use of *libtask* [29], a user-level thread library. By doing so, we reduce the cost of delivering the message to that of a lightweight user-level context switch. The architecture of our implementation is depicted in Figure 7. Upon reading a request from each queue, the requested thread blocks and its reading destination is added to the waiting list of the scheduler. The scheduler checks for all waiting reads and, upon receiving a message, loads the context of the corresponding reading thread. In other words, the developer takes advantage of the simple blocking read interface, while the back-end benefits from the asynchronous message-passing implementation to gain performance.

Following the messaging standard in *libtask*, a replica waits for the clients to connect (by *netlisten* function). Afterwards, the replica creates (i) the send and the receive queues for future communications with the node and (ii) a thread for reading the messages from the open connection. The thread will block by calling *fdread* on the connection and process the received message after the scheduler wakes it. Note that while a user-level thread is blocked, the replica could still progress by processing other messages in other threads. Since we have implemented standard interfaces provided by the library, the implemented protocols in our framework can be easily ported to a network system with no change. (The library already supports TCP and UDP implementation of the messaging interfaces.)

# 7. EVALUATION

In this section, we report on the evaluation of 1Paxos, which we compare with Multi-Paxos and 2PC. Multi-Paxos is arguably the most efficient consensus protocol to date. As pointed out earlier, 2PC is used here in the sense of [4], i.e. in its agreement form (and not in its full transaction commitment sense). We basically explore: (1) The improved

failure probability of two arbitrary cores is $\binom{N}{2}.s^2$. Then for $N = 7$, this failure scenario is 21 times less likely than the failure of two arbitrary cores.

[8] We expect to have standard inter-core channels in upcoming computer architectures [9].

commit latency and throughput by using 1Paxos; (2) The scalability of 1Paxos with the number of cores; and (3) The performance of 1Paxos when a core becomes slow.

## 7.1 Experimental Setup

We made use of a machine with eight 2.1 GHz Six-Core AMD Opteron(tm) processors (48 cores in total) and 32 GB of RAM. The L1 cache size is 128 KB, the L2 cache is 512 KB, and each of the processors has a 6 MB L3 cache. The machine runs Linux (Ubuntu 12.04 64 bit, kernel version 3.4.2).

In our experiments, we refer to the nodes having the replicated data and participating in the agreement protocol as servers, and to the nodes which make use of this service as clients. It is shown by the many years of research in state machine replication that to make agreement scalable, it must only be achieved between a few servers and the other nodes should behave like clients. We have applied the same lesson by using three replicas in all protocols (we also present an experiment showing the lack of scalability when increasing the number of replicas), which are each assigned a separate core from 0 to 2, via the *taskset* command, a simple assignment that is fair to all the protocols. The clients are assigned to cores 3 to 47. The clients start sending requests to the replicas after receiving a message from the load manager which is run on Core 47. There is no payload added to the requests and responses. In all experiments, a client sends a request to Core 0, waits for the commit ACK, and then sends another request, until it finishes 100 requests. We run each experiment three times and report the average values.

## 7.2 Latency

In this experiment, only one client is used, which is assigned to Core 3. We measure the average commit latency experienced by the client. The commit latency is the delay between the times the client sends the request and receives the reply. The throughput is the number of received replies per unit of time.

1Paxos has the lowest latency (around 16 $\mu$s) whereas 2PC has the highest (21.4 $\mu$s), because of the more message copy operations induced by sending more messages. Although both Multi-Paxos and 2PC transmit the same number of messages per request, Multi-Paxos has a slightly better latency (19.6 $\mu$s) since it progresses right after receiving the response from a majority of nodes whereas 2PC has to wait for all the responses to be received. The same pattern applies to throughput, since the higher the latency of each commit is, the fewer requests will be sent by the client per unit of time.

## 7.3 Scalability

We evaluate the scalability of the protocols by increasing the number of clients from 1 to 45 (we have 48 cores in total).[9] Figure 8 depicts the average commit latency vs. the total achieved throughput by the clients. 1Paxos is the most scalable protocol, since the throughput improves by a factor of two as the number of clients is increased from 1 to 13. Multi-Paxos scales up to only two clients. Afterwards, by adding more clients, the throughput improves slightly, while the latency increases steeply. This makes its throughput stop at 68070 op/s with 6 clients, 52% of the maximum throughput of 1Paxos. Similar to Multi-Paxos, 2PC throughput rises by adding the first client and slightly improves up to 7 clients, where the throughput is 48% of that of 1Paxos. As the number of clients increases after the point that the cores are saturated, all protocols suffer a small decrease in throughput due to the imposed overhead.

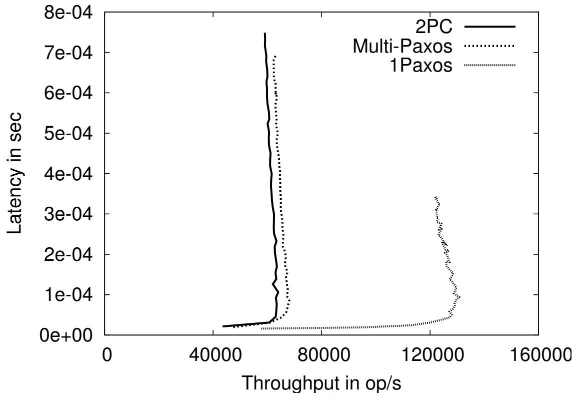

Figure 8: The latency vs. throughput w.r.t the number of clients in a 48-core machine.

## 7.4 Degree of Replication

This experiment explores the alternative to achieving agreement among replicas, i.e., running the agreement protocol directly between the clients. In other words, each client is also a replica. Figure 9 plots the commit throughput as a function of the number of nodes for the three modified protocols 2PC-Joint, Multi-Paxos-Joint, and 1Paxos-Joint. The initial leader is Core 0 for all three protocols and remains unchanged during the experiment. To avoid the contention between multiple concurrent agreement instances, all the clients forward their commands to the leader, which runs the agreement and returns the result back to the client. After receiving a reply, a client waits 2 ms before issuing a new request.

Figure 8 shows that the processing power of the replicas is the bottleneck for scalability. As a result, after they become saturated, increasing the rate of client commands would only results into increased commit latency due to more buffering delay. Figure 9 shows that when the agreement is run directly among clients, once the nodes become saturated, the throughput actually decreases when more processes are added: the number of messages per second the nodes process remains constant, but because by adding nodes the number of messages per consensus round increases, the throughput in terms of commits decreases. This behavior is exhibited by Multi-Paxos-Joint and 2PC-Joint, who reach the saturation point around 20 nodes. In contrast, the low number of messages transmitted in 1Paxos-Joint allows the throughput to increase linearly up to 47 nodes, thus scaling much better with the number of cores compared to the alterna-

[9] We avoided using the cores allocated to the replicas for the client processes. That would make the analysis of the results more complicated both because of the added load on the replica cores and the lower latency between the clients and the replicas.

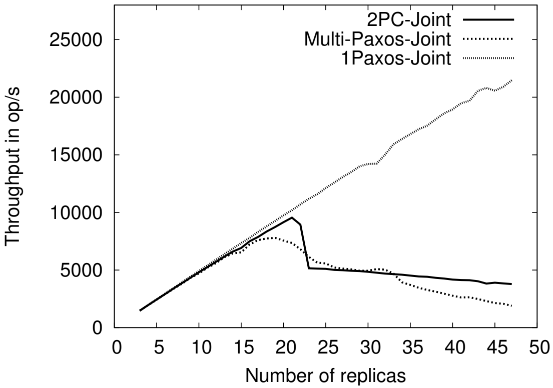

Figure 9: The throughput w.r.t the number of replicas in a 48-core machine.

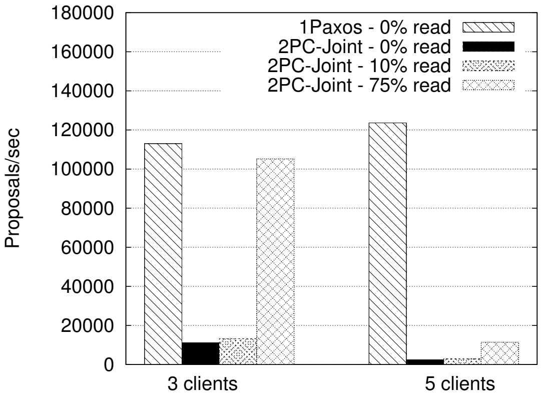

Figure 10: Throughput of 2PC-Joint, which is run directly among the clients.

tives. Because of the lower number of message exchanges per agreement, 1Paxos-Joint also provides a much lower latency: as an example, with 15 processes, before the nodes become saturated in any of the protocols, 1Paxos-Joint has a commit latency of 32 $\mu$s, while Multi-Paxos-Joint requires 190 $\mu$s per client request and 2PC-Joint requires 125 $\mu$s.

## 7.5 Read Workload

In the Paxos family of protocols, messages are issued as a result of each client command, which is the type of traffic targeted by 1Paxos. In general, read requests also cause Paxos protocol messages to be issued. This is because the read requests often require the last updated data, which is not necessarily updated in every learner, including the leader node. Thus, the read traffic can be treated as normal client command traffic.

2PC in particular could handle the read traffic more efficiently if it is run directly between the clients, i.e., each client is also a replica (2PC-Joint). In this way, a client can locally service the read requests if it is not received in the gap between two phases of 2PC. However, direct agreement between the clients implies an increase in the number of transferred messages with increase in the number of clients, which negatively impacts the protocol scalability.

As depicted in Figure 10, without any read traffic, 2PC-Joint throughput is much lower than 1Paxos's. By adding more read traffic, the throughput of 2PC-Joint improves: by 75% read traffic, 2PC-Joint running between three clients keeps up with 1Paxos. However, by adding a few more clients the 2PC throughput quickly drops: by 75% read traffic, the throughput of 2PC-Joint is much lower than that of 1Paxos. This shows that the optimization for read workloads that one could apply in 2PC is not scalable with the number of nodes, and could not, hence, be employed in our application.

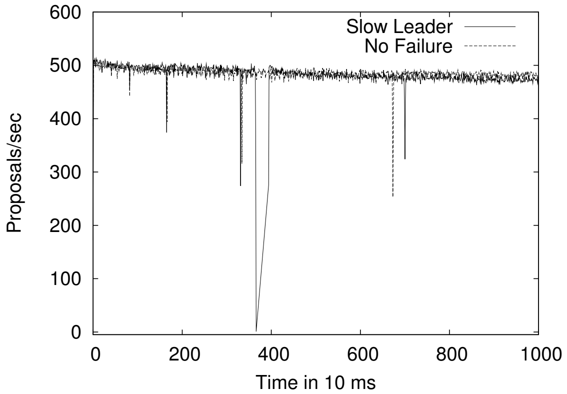

Figure 11: The changes in throughput achieved by 1Paxos when the leader is slow.

## 7.6 Throughput with Slow Cores

As described in Section 2, since 2PC is blocking, no requests can commit after any replica including the leader is unavailable, and the throughput drops to zero. Here, we run an experiment to see how 1Paxos tolerates faulty cores. Figure 11 plots the throughput of 1Paxos when the leader becomes slow, as well as the normal non-faulty case. The experiment is run on a machine with four 2.4 GHz Dual-Core AMD Opteron(tm) processors (8 cores in total). 5 clients are sending requests to 3 replicas, from which the initial leader is assigned to Core 0. We slow down Core 0 by running 8 CPU-intensive processes on it; each process is a bash script that continuously multiplies a number by itself. Once the clients detect the slow leader, they send their requests to other nodes. After receiving the clients' request, the non-leader node tries to become leader in PaxosUtility. After that, it sends the proposals to the active acceptor. During the leader change process, the throughput drops to zero.

## 8. RELATED WORK

Barrelfish [4], an implementation of a multikernel model, pioneered the idea that a many-core should be viewed as a distributed system. Key information of the kernel is replicated on several cores and a 2PC-like (i.e. blocking agreement) protocol ensures the consistency of the replicas. Barrelfish exploits the cache hierarchy inside the processors to efficiently broadcast messages via multicast trees. A slow node in the multicast tree can however delay the propagation of the message to the rest of the nodes. This approach contrasts with that of our 1Paxos consensus (a non-blocking agreement) protocol, which is precisely designed to address the problem of slow cores, and hence does not use any multicast tree to broadcast messages to the replicas. (Otherwise, an unresponsive core would cause all its child nodes under the multicast tree to be unresponsive as well.)

Mencius [32] was derived from Multi-Paxos to distribute the load of client commands among multiple leaders [25]. Assuming a balanced load of client commands received by the leaders, it partitions the space of Paxos instance numbers among the leaders: each leader proposes the received client commands only for its range of instance numbers. By doing so, the leaders can, in total, process more aggregate commands from clients. Yet, each leader still has to communicate with all acceptors to make a proposal. If the load is not balanced on the leaders, the loaded leader could forward its traffic to the other under-loaded leaders, which causes higher delays. The under-loaded leaders also have to skip their share of the instance space, which would not help the load balancing objective. In contrast, 1Paxos targets the load on each leader individually, and is not limited by assuming a balanced load on the leaders. By reducing the number of messages exchanged between servers (non-client nodes), each leader in 1Paxos can process more client commands. The main insight of 1Paxos can be applied to any protocol of the Paxos family. In fact, we conducted experiments of 1Paxos over an IP network and observed a factor of 2.88 improvement over Multi-Paxos ( similar to what is depicted in Figure 2). Mencius could also benefit from the main insight of 1Paxos and increase the system throughput further. This would enable a Mencius leader to be assigned to a single separate acceptor, and indeed increase the overall throughput. However, the exact conditions under which 1Paxos is preferable to Multi-Paxos or Mencius in an IP network need to be further studied.

Some protocols of the Paxos family target the latency of client commands [12, 23, 24]. In Basic-Paxos, each client command takes four message delays between the servers. Multi-Paxos, which has been successfully integrated into a number of practical deployed [5,8,30] systems, behaves similarly to Basic-Paxos for the first command, but requires only two message delays between servers for the next commands. This does not include the RTT delay between the client and the leader. Fast Paxos [24], using more replicas $(3f + 1)$, saves the delay between the leader and the acceptors by allowing the client to optimistically send the accept_request messages directly to the acceptors. Collisions between commands from different clients can be resolved by spending more steps. The average latency can be lower if the rate of collisions is low. If collisions are frequent, classic Paxos actually outperforms Fast Paxos [24].

In scenarios where the throughput of the system is a bottleneck, the number of client commands is very high, and the probability of collisions increases accordingly. 1Paxos is designed for high-throughput systems and reducing the number of consensus phases is not targeted by the protocol. Fast Paxos cannot outperform the throughput of Multi-Paxos, as the number of sent/received messages to/from each acceptor does not change; although the leader-to-acceptor messages of Multi-Paxos are eliminated in Fast Paxos, the messages must be sent to more acceptors, $3f+1$. For $f = 1$, the message/node is equal to six per command, which is the same number as Multi-Paxos. BFT protocols [7, 10, 21] tolerate not only crashes but also Byzantine faults: these include arbitrary faults and malicious behavior. BFT protocols, because of aiming stronger guarantees, are more expensive than the widely deployed consensus protocols [5, 8, 30] (to which 1Paxos belongs).

Since Paxos requires only a majority $(f+1)$ of the replicas to progress, in scenarios where all cores are responsive, $f$ of the nodes can be excluded from an execution. This observation is leveraged by Cheap Paxos [33] to improve the throughput. Yet, this optimization comes with liveness penalties. For example, with 3 replicas $r_1$, $r_2$, and $r_3$, if $r_1$ fails and afterward $r_2$ fails, then the system cannot progress until $r_2$ recovers. In other words, the recovery of $r_1$ does not help since $r_2$ has the crucial last state of the system. In comparison, 1Paxos can progress as long as any 2 of the 3 replicas are responding. For example, in the above scenario, 1Paxos progress as soon as either $r_1$ or $r_2$ starts responding. In this sense, 1Paxos does not jeopardize the liveness of Paxos and yet offers higher performance.

Vertical Paxos [26] addresses the scenario in which only a subset of the available servers should be responsible for replicating a particular piece of data. Vertical Paxos uses so-called read and write quorums of acceptors, which must intersect. Their sizes can be adjusted, but as the size of the read quorums is reduced, the size of the write quorums must be increased. At the expense of a somewhat lower degree of fault tolerance with more than three replicas, 1Paxos allows an arbitrary degree of replication using a single acceptor in the entire process, thus requiring a lower number of message exchanges.

The underlying message passing mechanisms in the context of many-cores also represent an active area of research. Barrelfish [4] uses unidirectional one-to-one message-passing channels, avoiding unnecessary cache line sharing. Aublin et al. [2] propose ZIMP, a one-to-many communication mechanism for cache-coherent many-cores, addressing situations in which messages need to be broadcast to multiple receivers. In addition, architectures such as Tilera processors [38] and the Intel SCC [37] expose interfaces to some forms of hardware-implemented message passing, which either complement or replace cache coherence. In QC-libtask, we employ one-to-one communication in order to avoid scalability limitations due to cache line sharing between a large number of cores. By preventing threads from spinning unnecessarily when waiting for messages, QC-libtask also provides efficient support for multiprogramming.

# 9. CONCLUDING REMARKS

It is important to note that our paper does not argue for a message-passing vision of a many-core architecture, nor does it argue against on-chip cache coherence protocols. In particular, we do not claim that consistency in a multi-core architecture should be ensured in software through some middleware agreement message-passing based layer rather than in hardware through a cache coherence protocol. Both approaches will be further studied and refined and might very well coexist.

Considering situations where consistency is indeed ensured in software using a message-passing agreement protocol, the motivation of this paper was to carefully analyze such protocols, and in particular explore the feasibility of making them non-blocking. An efficient implementation of such protocols could benefit a number of systems besides Barrelfish. These include 0MQ [1] and Jetlang [19] for instance, which use message passing to solve notoriously hard software engineering problems that developers have to face when implementing parallel applications, such as avoiding locks (and deadlocks).

Our 1Paxos protocol was specifically designed with a many-core system in mind: basically, it transmits fewer messages than alternative protocols, which reduces the load on the cores. Roughly speaking, 1Paxos can be viewed as a Paxos-like agreement protocol that uses only a single acceptor, which is replaced only in case of non-responsiveness. Our experimental results conveyed the fact that, on a many-core system, non-blocking agreement achieved using 1Paxos outperforms a similar approach based on Multi-Paxos, the most efficient practical non-blocking agreement protocol to date, and even a classical (blocking) 2PC-like protocol (as adopted in [4]). 1Paxos might block if both the leader and the acceptor are not responsive during the exact same period. In this scenario, which is arguably very rare, 1Paxos progresses after at least one of them starts responding. When using three cores, 1Paxos and the Paxos family of protocols can tolerate the same number of slow cores, i.e., exactly one.

While in this paper we focused on a traditional many-core, 1Paxos and similar protocols can be expected to become even more useful in future architectures. First, as transistor density continues to increase, their reliability is expected to suffer [14]. This may lead to certain inconsistencies or failures in hardware. Although in this paper we modeled failures as delayed cores, 1Paxos can tolerate crashes as well. In addition, hardware heterogeneity is expected to increase in future architectures [4, 14]. In such a scenario, not all the computing nodes are expected to be connected through a cache coherent shared memory. For essential data that is commonly accessed, 1Paxos would represent an ideal candidate to ensure consistency. Moreover, the scale of NUMA is increasing. For example, an emerging trend at rack-scale level is the use of remote direct memory access (RDMA) [13,35]. In this setting, multiple machines operate on a common address space, but there is no cache coherence protocol between them. 1Paxos could represent a solution for ensuring coherence (where needed) at a software-level.

Our QC-libtask message passing framework, as well as our implementations of 1Paxos, 2PC and MultiPaxos are available at `https://github.com/LPD-EPFL/consensusinside`.

**Acknowledgements.** We wish to thanks the anonymous reviewers for their insightful comments which helped improve the paper. This work is supported by the Swiss National Science Foundation (project number 147067).

## 10. REFERENCES

[1] 0mq. http://www.zeromq.org.

[2] P Aublin, S Mokhtar, Gilles Muller, and Vivien Quéma. Zimp: Efficient intercore communications on manycore machines. Technical report.

[3] Ö. Babaoğlu and S. Toueg. Distributed systems (2nd ed.). chapter Non-blocking atomic commitment, pages 147–168. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[4] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. SOSP 2009.

[5] M Burrows. The Chubby lock service for loosely-coupled distributed systems. OSDI 2006.

[6] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to reliable and secure distributed programming.* Springer-Verlag New York Inc., 2011.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *TOCS*, 20(4):398–461, 2002.

[8] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: an Engineering Perspective. PODC 2007.

[9] N. Chatterjee, S.H. Pugsley, J. Spjut, and R. Balasubramonian. Optimizing a Multi-Core Processor for Message-Passing Workloads. UCAS-5 2009.

[10] J. A. Cowling, D. S. Myers, B. Liskov, R. Rodrigues, and L Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. OSDI 2006.

[11] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP 2013.

[12] D. Dobre, M. Majuntke, and N. Suri. CoReFP: Contention-Resistant Fast Paxos for WANs. Technical report, TU Darmstadt, 2006.

[13] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. NSDI 2014.

[14] M. D. Hill et al. 21st Century Computer Architecture: A community white paper. Technical report, 2012.

[15] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[16] R. Guerraoui, V. Quema, and M. Vukolic. The next 700 BFT protocols. EuroSys 2008.

[17] J. Handy. *The cache memory book.* Morgan Kaufmann, 1998.

[18] M. Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, January 1991.

[19] Jetlang. http://code.google.com/p/jetlang/.

[20] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing: The Internet as a Distributed System. NSDI 2008.

[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

[22] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), 2001.

[23] L. Lamport. Generalized consensus and Paxos. Technical report, MSR-TR-2005-33, Microsoft Research, 2005.

[24] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[25] L. Lamport, A. Hydrie, and D. Achlioptas. Multi-leader distributed system, November 21 2002. US Patent App. 10/302,572.

[26] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. PODC 2009.

[27] Leslie Lamport. The part-time parliament. *TOCS*,

16(2), 1998.

[28] B. Lampson and H. Sturgis. *Crash recovery in a distributed data storage system.* Xerox PARC, Palo Alto, California, 1979.

[29] libtask. http://swtch.com/libtask/.

[30] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. NSDI 2007.

[31] N.A. Lynch. *Distributed algorithms.* Morgan Kaufmann, 1996.

[32] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. OSDI 2008.

[33] M. Massa and L. Lamport. Cheap Paxos. DSN 2004.

[34] T.G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. SC 2008.

[35] S. Novakovic, B. Grot, A. Daglis, E. Bugnion, and B. Falsafi. Scale-Out NUMA. ASPLOS 2014.

[36] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *EDCC*, volume 2485, pages 44–61. Springer Berlin Heidelberg, 2002.

[37] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1):73–83, February 2011.

[38] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.C. Miao, J.F. Brown, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *Micro, IEEE*, 27(5):15–31, 2007.

```
Upon AcceptorFailure
  if (!$IamLeader$) return;
  ($P_i$,$instance$) = PaxosUtility.lastLeader();
  if ($P_i \neq me$) //somebody thought I am dead
    $A_a$ = $null$; $IamLeader$ = false;
    return;
  $A'_a$ = selectAcceptor();
  $proposals$ = uncommitedProposals();
  $success$ = PaxosUtility.propose($instance$,
            AcceptorChange($A'_a$,$proposals$));
  if (!$success$) return;
  $A_a$ = $A'_a$;
  $IamLeader$ = false;

Upon LeaderFailure
  propose();

proc propose()
  if ($IamLeader$)
    $in$ = next_uncommited_instance_number();
    $v$ = getAny($in$);
    sendto $A_a$ accept_request($in$, $pn$, $v$);
  else
    $YouMustBeFresh$ = true;
    $pn$ = new_pn();
    if ($A_a$ == $null$)
      ($A_a$,$instance$,$proposals$) =
        PaxosUtility.lastActiveAcceptor();
      $success$ = PaxosUtility.propose($instance$,
                    LeaderChange($me$, $A_a$));
      if (!$success$)
        $A_a$ = $null$; return;
      registerProposals($proposals$);
      $YouMustBeFresh$ = true;
    sendto $A_a$ prepare_request($in$, $pn$, $YouMustBeFresh$);

Upon Receive prepare_response($A_i$, $pn$, $ap$)
  if ($IamLeader$ || $A_i \neq A_a$) return;
  $IamLeader$ = true;
  registerProposals($ap$);
  $in$ = next_uncommited_instance_number();
  $v$ = getAny($in$);
  sendto $A_a$ accept_request($pn$, $v$);

Upon Receive prepare_request($P_i$, $pn$, $YouMustBeFresh$)
  if ($pn > hpn$)
    if ($IamFresh$ != $YouMustBeFresh$)
      return;
    $IamFresh$ = false;
    $hpn$ = $pn$;
    sendto $P_i$ prepare_response($pn$, $ap$);
  else sendto $P_i$ abandon($hpn$);

Upon Receive accept_request($P_i$, $in$, $pn$, $v$)
  if ($pn \neq hpn$)
    sendto $P_i$ abandon();
  else if ($ap[in] \neq null$)
    multicast $L$ learn($in$, $ap[in]$);
  else
    $ap[in]$ = ($pn$, $v$);
    multicast $L$ Learn($pn$, accepted);
```

Figure 12: 1Paxos Algorithm

# APPENDIX

## A. APPENDIX: 1Paxos

The pseudo code for our 1Paxos algorithm, explained in Section 5, is presented in Figure 12. If the proposer recognizes itself as the leader of the active acceptor, Variable *IamLeader* is set. The leader does not need to send a prepare_request message to the acceptor and starts directly with the accept_request message using the last promised proposal number. Variable $A_a$ refers to the active acceptor Id; *ap* is the map structure keeping the accepted proposals; $IamFresh$ indicates that the acceptor has adopted no leader yet. The highest proposal number is stored in Variable *hpn*. Initially the highest proposal number is equal to -$\infty$. Procedure *init*, presented in Figure 13, initializes the mentioned variables. As for the rest of the variables, *pn* is the proposal number, *in* is the Paxos instance number, $v$ is the value (proposed or accepted), $P$ is the list of the proposers, $L$ is the list of learners, *me* is the node Id, and *YouMustBeFresh* indicates that the proposer expects to be the first proposer that contacts the acceptor.

Upon a failure of the active acceptor, the leader first checks whether the others still believe him as the leader or not. If not, one other proposer has taken its position (probably because of a false leader failure alarm). In this case, it relinquishes the leadership position and return. Otherwise, it calls *selectAcceptor* function to select a new acceptor which is located on a separate node than the leader node. The leader then announces the change of the active acceptor, *AcceptorChange*, through PaxosUtility. It also attaches the uncommitted proposed values to the *AcceptorChange* entry. The failure of this step indicates that another item is chosen for the current instance of PaxosUtility. In this case, the leader returns from this procedure to try again later. In case of success, however, the leader resets Variable $IamLeader$ because it has to start from the first phase of Paxos with the new active acceptor.

Upon failure of the current leader, a proposer tries to take its position by calling Procedure *propose*. The procedure then obtains the active acceptor Id and sends a prepare_request message to it.

Procedure *propose* proposes a value for the next uncommitted instance number. If the node is already the leader, it directly sends an accept_request message to the active acceptor. Otherwise, it sends a prepare_request message to the active acceptor, in accordance with the first phase of the Paxos algorithm. If the active acceptor Id is unknown to the proposer, it must be obtained via PaxosUtility. The *lastActiveAcceptor* method checks the sequence of committed entries looking for the last *AcceptorChange* entry; this entry contains the active acceptor Id. Next, the proposer adds a *LeaderChange* entry via PaxosUtility. The failure of this step indicates that another item is chosen for the current instance of PaxosUtility. In this case, the procedure resets the value of $A_a$ and returns. We assume that the implementation retries the failed attempt via timers or some other mechanisms. In the case of success, before sending the prepare_request message, it first registers the proposed values which have been recorded with the last AcceptorChange entry. If the acceptor is supposed to be a fresh backup acceptor, it also sets Variable *YouMustBeFresh* which is sent by the message.

Upon receipt of the prepare_request message from Proposer $P_i$, the acceptor verifies the highest proposal number *hpn* to be less than the requested proposal number, *pn*. Otherwise, it sends an abandon message back to Proposer $P_i$. If Variable *IamFresh* is set but Variable *YouMustBeFresh* is not, it indicates that the proposer expected the acceptor to be already adopted by the last leader. However, due to the acceptor reset, the acceptor has lost its data, including *hpn* and *ap*. This check avoids the cases where the active acceptor silently reboots before the leader switch. In this case, the last leader should switch the rebooted acceptor.

```
proc init()
  $IamLeader$ = false; $A_a$ = $null$;
  $ap$ = emptyMap(); $hpn = -\infty$
  $IamFresh$ = true;

proc getAny($in$, $ap$)
  $v$ = $proposed[in]$;
  if ($v \neq null$) return $v$;
  $v$ = nextClientRequest();
  $proposed[in]$ = $v$;
  return $v$;

proc registerProposals($proposals$)
  foreach p in $ap$
    $proposed[p.in]$ = $p.v$;
```

Figure 13: The implementation of Procedures init, getAny, and registerProposals, in 1Paxos Algorithm

Upon receipt of the prepare_response message from the active acceptor, the proposer claims the leadership position by setting Variable $IamLeader$. The *getAny* method, presented in Figure 13, picks a value to be accepted for the instance *in*. The picked value can be any given value, unless there is already a proposed but uncommitted value for the instance *in*. This case can occur in change of the active acceptor, when some proposed values are not committed yet by the previous active acceptor. If any proposal matches the instance number *in*, to avoid inconsistency, the proposer picks the same previously proposed value. It then sends an accept_request message to the active acceptor.

Upon receipt of the accept_request message from the leader, the acceptor first checks for the proposal number. Also, it checks that there is no proposal accepted corresponding to the instance number, i.e., $ap[in]$. Otherwise, it broadcasts the learn message of the accepted proposal again to cover the cases that the lost learn message has motivated the proposer to retry. It then stores the proposal in the accepted proposal map, $ap[in]$. Afterwards, the accepted proposal is broadcasted to all the learners accordingly.

# B. APPENDIX: PROOF OF CORRECTNESS

Here, we prove the correctness of the algorithm presented in Appendix A. We first prove some properties for the entries in PaxosUtility, which we then use to prove that no two different values would be accepted for the same instance number. The proof for the simple case where there is no change in the active acceptor nor the leader node, is trivial and similar to the proofs of Paxos. Here, we focus on the complex cases where the algorithm switches the leader and the active acceptor.

PaxosUtility contains entries for changing the active acceptor, i.e. AcceptorChange, and entries for changing the leader, i.e. LeaderChange. We define the *Global leader* and

*Global acceptor* as follows:

*definition*: In the sequence of PaxosUtility entries, the node which has inserted the last LeaderChange entry is the *Global leader*. Similarly, the active acceptor announced by the last AcceptorChange message, represents the *Global acceptor*. We use $GL_i$ to represent the $i$th Global leader and $GA_i$ to represent the $i$th Global acceptor.

*Lemma 1*: An AcceptorChange entry is inserted only by the Global leader.

Lemma 1 is guaranteed by lines 3..13 of Figure 12. In Line 4 the leader verifies that it is still the Global leader. It also keeps the index of the last empty instance number, *instance*. Later in Line 10, it proposes the AcceptorChange message for that instance number. The failure of this phase implies that another node has inserted something in the meanwhile. In this case, the handler returns to retry the procedure later from scratch. Therefore, the AcceptorChange message is inserted only by the Global leader.

According to Lemma 1, the Global acceptor represents the active acceptor which the Global leader is working with.

Now we prove by induction that the same value will always be accepted for a particular instance number. The first step is to show that a Global leader does not propose two different values for the same instance number when it switches between the acceptors. Hereafter, we use the pair ($v$,$i$) to represent the value $v$ and instance number $i$ of a given accept_request messages.

*Lemma 2a*: Suppose that $GL_l$ has issued two accept_request messages, ($v_a$,$i_a$) and ($v_{a+1}$,$i_{a+1}$), to two consecutive Global acceptors $GA_a$ and $GA_{a+1}$, respectively. If $i_a = i_{a+1}$, then $v_a = v_{a+1}$.

Lemma 2a is directly followed by the implementation of the Procedure *getAny* in Figure 13. There, the leader first checks the history of the proposed values. If any value has already been proposed for the requested instance number, then the procedure returns the same value. Hence, as long as the Global leader is not changed, the proposed value for a particular instance number will be always the same.

The next step is to show that an acceptor accepts the same proposals from two consecutive Global leaders.

*Lemma 2b*: Suppose that the active acceptor $GA_a$ accepts two accept_request messages, ($v_l$,$i_l$) and ($v_{l+1}$,$i_{l+1}$), from two consecutive Global leaders $GL_l$ and $GL_{l+1}$, respectively. If $i_l = i_{l+1}$, then $v_l = v_{l+1}$.

Node $GL_{l+1}$ becomes the Global leader only after successfully inserting a LeaderChange entry via PaxosUtility. In the algorithm presented in Figure 12, this happens only at Line 30 inside the Procedure *propose*. It also implies that the value of Variable *Iamleader* is false (Line 25). $GL_{l+1}$ will not start proposing values unless the value of Variable *Iamleader* changes to true (Line 21). Line 41 is the only location where the value of this variable is changed to true upon receipt of a prepare_response message. It indicates that the active acceptor has received the prepare_request message, approved the proposal number, and responded by the prepare_response message which is also piggybacked by all the previous accepted proposals, *ap*. The received accepted proposals are registered by the leader (Line 42). The registered values will be later used for all the next proposals in Procedure *getAny*. In other words, the $GL_{l+1}$ will propose the same values which acceptor $GA_a$ has already accepted.

Similar to Basic-Paxos, $GA_a$ will reject all the other potential issued accept_request messages by $GL_l$ after sending the prepare_response message to $GL_{l+1}$. On the other hand, as we showed above, if $GA_a$ has accepted any value from $GL_l$ for a particular instance number, it will not receive any different value from $GL_{l+1}$ for that sequence number. Consequently, $GA_a$ always accept the same values from two consecutive Global leaders.

Having Lemma 2a and Lemma 2b, now we present the correctness proof of the algorithm.

(*) Suppose that two acceptors $GA_a$ and $GA_{a'}$ accept two accept_request messages, ($v_a$,$i_a$) and ($v_{a'}$,$i_{a'}$), received from the Global leaders $GL_l$ and $GL_{l'}$, respectively, where $l' \geq l$ and $a' \geq a$. If $i_a = i_{a'}$, then $v_a = v_{a'}$.

The proof is by induction on the size of sequence of entries in the PaxosUtility utility. Assume that property (*) holds when PaxosUtility has $k$ entries. We prove that it still holds when PaxosUtility has $k + 1$ entries.

Recall that the entries in the PaxosUtility utility are either AcceptorChange or LeaderChange. If the $k + 1$th entry is AcceptorChange, based on Lemma 1 it is inserted by the last Global leader. Thus, the $GL$ is the same and the $GA$ changes. This is the case in Lemma 2a for which we proved that no two values will be proposed for the same instance number. If the $k+1$th entry is LeaderChange, we can assume that $GA$ is the same during this change. This is provided by the Lines 29..30 in Figure 12, where the new leader takes the same active acceptor as was taken by the last leader. This case is covered by Lemma 2b for which we proved that no two values will be accepted for the same instance number. Consequently, if we assume that no two values are accepted for the same instance number in the first $k$ entries of PaxosUtility utility, this also holds for the first $k + 1$ entries.

Now, to complete the proof, we need to show that the theory holds for $k = 2$. We can make it hold by an initialization process. At the start up, the node with the smallest Id can insert two entries for LeaderChange and AcceptorChange to announce itself as the Global leader and its active acceptor as the Global acceptor. Because, no change in the roles happens in the initial case, neither for the leader nor for the active acceptor, then the theory directly holds for this case.