Interactive Consistency in practical, mostly-asynchronous systems

Panos Diamantopoulos, Stathis Maneas, Christos Patsonakis, Nikos Chondros and Mema Roussopoulos
Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
Athens, Greece
{panosd,smaneas,c.patswnakis,n.chondros,mema}@di.uoa.gr

***Abstract*—Interactive consistency is the problem in which n nodes, where up to t may be byzantine, each with its own private value, run an algorithm that allows all non-faulty nodes to infer the values of each other node. This problem is relevant to critical applications that rely on the combination of the opinions of multiple peers to provide a service. Examples include monitoring a content source to prevent equivocation or to track variability in the content provided, and resolving divergent state amongst the nodes of a distributed system.**

Previous works assume a fully synchronous system, where one can make strong assumptions such as negligible message delivery delays and/or detection of absent messages. However, practical, real-world systems are mostly asynchronous, i.e., they exhibit only some periods of synchrony during which message delivery is timely, thus requiring a different approach. In this paper, we present a thorough study on *practical interactive consistency*. We leverage the vast prior work on broadcast and byzantine consensus algorithms to design, implement and evaluate a set of algorithms, with varying timing assumptions and message complexity, that can be used to achieve interactive consistency in real-world distributed systems.

We provide a complete, open-source implementation of each proposed interactive consistency algorithm by building a multi-layered stack of protocols that include several broadcast protocols, as well as a binary and a multi-valued consensus protocol. Most of these protocols have never been implemented and evaluated in a real system before. We analyze the performance of our suite of algorithms experimentally by engaging in both single instance and multiple parallel instances of each alternative.

Keywords*—*Interactive consistency, Asynchronous, Consensus, Agreement, Byzantine

I. Introduction

Interactive consistency (IC) is defined in a system of n distinct nodes, each having its own private value, where up to t may be byzantine (faulty). The goal is for all non-faulty nodes to compute the same vector of values. For each non-faulty node, the corresponding slot in the vector should contain that node's private value.

To date, related work regarding interactive consistency has provided solutions tailored for synchronous systems ([1], [2], [3], [4], [5]). These algorithms deliver useful theoretical insight and may be suitable in cases (e.g., shared memory multi-processor systems) where one can make strong assumptions such as negligible message delivery delays ([6]) and/or detection of absent messages ([1], [2]). However, these assumptions are ill-suited for practical, real-world distributed systems which, in their vast majority are mostly asynchronous, i.e., they exhibit only some periods of synchrony during which message delivery is timely.

In a fully asynchronous environment, researchers have proposed a myriad of algorithms for closely related topics, such as byzantine agreement and byzantine consensus. Consequently, one might assume interactive consistency can be easily achieved in an asynchronous setting, by a simple synthesis of one or more steps of these algorithms. However, this is not the case, as it is impossible to detect process crashes in a completely asynchronous system, where messages can take arbitrarily long to be delivered. Additionally, in an asynchronous system, it is impossible to guarantee simultaneously, that all honest parties inputs' are included in the computation (in our case, in the resulting vector of values), and that all honest parties are guaranteed to terminate, as proved in [7]. These are the reasons *vector consensus* is considered the only achievable equivalent of interactive consistency for completely asynchronous systems ([8]). In vector consensus, the only guarantee is that the resulting vector contains at least $2t + 1$ values, of which at least $t+1$ were proposed by honest nodes.

Interactive consistency is required in a variety of real-world, critical applications. For example, a distributed fault-tolerant voting application runs a light-weight voting protocol during election hours [9]. At election end-time, each node's local view of the cast votes may be inconsistent with the views of the others. Interactive consistency can be used here once, upon election end-time, to derive a single set of votes and produce the result. As another example, an application may employ multiple peers to monitor the content delivered by a single source, as a means to verify its integrity. This prevents the source from equivocating, i.e., distributing different content to different peers, with the additional benefit of being able to prove reliably if such equivocation took place. Another closely related application is the recording of the variability in web content, as a means to track censorship ([10]), or other forms of personalization ([11]). Other applications include the ability of sensors to reliably compute complicated functions that depend on the combination of inputs from other sensors ([12]), system diagnosis, such as failure detection and group membership, cloud computing ([1]), and other problems requiring global knowledge.

In this paper, we present algorithms for solving interactive consistency in real-world distributed systems, with the minimal possible timing assumptions. We leverage prior work on broadcast and byzantine consensus protocols to design

our algorithms. We describe how we directly address, or circumvent, both theoretical and practical challenges that arise in solving IC. Examples of theoretical challenges are the well-known FLP impossibility result ([13]) and the impossibility of simultaneously achieving input completeness and guaranteed termination ([7]). Practical challenges, on the other hand, are a result of assumptions that several theoretical papers use to prove their algorithms, but that, unfortunately, do not hold in practice. Examples are unbounded memory at each node and a loss-less and/or corruption-free network ([5], [14]). Moreover, we have formally proved the correctness of our proposed algorithms.

To evaluate our algorithms, we first analytically compare their message complexity. We implement all of the proposed algorithms and present experimental measurements that consist of both single and multiple parallel instances. We compare the algorithms' performance in terms of throughput and latency and draw conclusions as to the appropriateness of using each algorithm in varied network environments.

In summary, we make the following contributions:

- We present a study of interactive consistency in *practical, real-world systems*, illustrate the theoretical and practical challenges that need to be addressed, and propose algorithms for achieving interactive consistency in such environments.
- We provide an open-source implementation of each of the proposed alternatives. This required the development of a protocol stack that includes various asynchronous broadcast primitives, as well as a binary and a multi-valued byzantine consensus protocol. Some of these protocols (e.g., [8], [14]), have never been implemented and evaluated (to our knowledge) in a real system.
- We evaluate our algorithms experimentally by running both serial and parallel executions of each algorithm and compare their performance in terms of throughput and latency, in both LAN and WAN settings. We find that simple protocol variations that restrict the behavior of malicious nodes improve performance.

II. Background and Related Work

Our study has unveiled a large incoherence in the bibliography, regarding the terms "Byzantine Agreement" and "Byzantine Consensus". These are often used to refer to the same problem (e.g., [13] and [15]), while others, e.g., [16], use the terms interchangeably, even though these are two distinct problems. There is also inconsistent use of the term "Interactive Consistency", e.g., [17], [18]. To alleviate any confusion and for clarity, we start with some basic definitions.

Byzantine Agreement. Assume a system of n nodes, where a single source n_i has a private value v_i, and the following must be achieved:

- *Agreement:* All non-faulty nodes must agree on the same value.
- *Validity:* If n_i is non-faulty, then the agreed upon value by all non-faulty nodes is v_i.
- *Termination:* All non-faulty nodes must eventually decide on a value.

This problem, also known as the "Byzantine Generals Problem", was introduced by Lamport et al. [19]. Earlier work has proved there is no solution for the asynchronous case [14], when the source is faulty. Agreement algorithms that tolerate byzantine failures of (non-source) nodes in asynchronous systems are presented in [20] and [21].

Byzantine Consensus. Assume a system of n nodes, where each node n_i has a private value v_i, and the following must be achieved:

- *Agreement:* All non-faulty nodes must agree on the same value $v \in \{v_1, ..., v_n\}$.
- *Validity:* If all non-faulty nodes have the same initial value v, then the agreed upon value by all non-faulty nodes is v.
- *Termination:* All non-faulty nodes must eventually decide on a value.

The byzantine consensus problem is one of the most studied topics in distributed systems and the main topic of the well-known FLP impossibility result ([13]). There are several types of consensus protocols. The first distinction revolves around determinism (or non-determinism). In a deterministic consensus protocol, given the set of input values on all nodes, the message schedule and the failures that occur (if any), the result will always be the same. Deterministic consensus protocols require a synchronous system ([22]). In a purely asynchronous system, consensus can be achieved by randomization. FLP is circumvented by having nodes locally toss a coin to decide on their input values, in round $r + 1$, in cases where consensus cannot be achieved in round r. Thus, the result may be different across executions with the same inputs. Examples of randomized protocols that employ the local coin construct are introduced by Bracha [14], Bracha and Toueg [23] and Ben-Or [24]. These algorithms guarantee eventual termination after a probabilistic number of rounds. In [15], a trusted, non-faulty dealer is additionally employed to bound the number of rounds required to achieve consensus. In our work, we leverage the randomized approach by Bracha to ensure termination because we believe it is controversial to assume a trusted entity in an otherwise byzantine environment.

Other works ([25], [26]) leverage verifiable secret sharing techniques to implement a shared, or, common coin. These consensus algorithms are polynomially efficient and terminate in a constant number of rounds. Canetti et al. [25] present one of most well-established and signature-free common coin protocols. However, this protocol, although polynomial, is complex to implement and has very high bit complexity [27]. Mostéfaoui et al. [27] employ the common coin protocol that is presented in [28] which has guaranteed termination but requires a trusted dealer. We did not consider these algorithms as they are either inefficient or require a trusted dealer.

One last distinction, regarding consensus protocols, revolves around the agreed upon value. All of the aforementioned protocols are binary consensus protocols, i.e., the agreed upon value is $v \in \{0, 1\}$. In the multi-valued consensus protocol of Correia et al. [8], the set of values V is of arbitrary size. In our work, when needed, we achieve multi-valued consensus by using primitives such as reliable broadcast (described later) and binary consensus.

Failure Detectors. In [29], Chandra and Toueg proposed a solution for the consensus problem, in an asynchronous crash-fault environment, introducing a module called *failure*

detector (FD). There is extensive bibliography that expands the family of FDs to a number of applications ([30], [31], [32], [33]). In [34], Chandra and Toueg define the weakest FD capable of solving consensus in asynchronous crash fault systems. However, this failure detector requires known bounds on node processing speed and message delivery, that hold after a Global Stabilization Time ([35]). The same assumptions hold for the Byzantine Failure detector introduced in [33]. However, these assumptions are unlikely to hold in real-world distributed systems, rendering both FDs unimplementable ([36], [37]).

Broadcast Primitives. All asynchronous consensus algorithms employ some form of reliable broadcast protocol, where a source *broadcasts* a message m, and every correct node eventually *delivers* m (e.g., via an up-call to the application). Such a broadcast satisfies the following properties ([38]):

- *Validity*: If a non-faulty node broadcasts a message m, then it eventually delivers m.
- *Agreement*: If a non-faulty node delivers a message m, then all non-faulty nodes eventually deliver m.
- *Integrity*: For any message m, every non-faulty node delivers m at most once *iff* m was previously broadcast by $sender(m)$.

In [14], Bracha introduced a $\frac{n}{3}$-resilient reliable broadcast primitive (RBB, for Reliable Broadcast of Bracha) to solve the consensus problem. Another type of broadcast primitive, with lower message complexity, is *consistent broadcast* (CB). CB is designed to relax the *agreement* property of reliable broadcast, by allowing *some* non-faulty nodes to deliver m, while others may deliver nothing. The standard implementation of consistent broadcast is *Reiter's echo multicast* [39].

Interactive Consistency. Assume a system of n nodes, where each node n_i has a private value v_i, and the following must be achieved:

- *Agreement:* All non-faulty nodes must agree on the same vector of values $V = [v_1, ..., v_n]$.
- *Validity:* If the private value of the non-faulty node n_i is v_i, then all non-faulty nodes agree on $V[i] = v_i$.
- *Termination:* All non-faulty nodes must eventually decide on a vector V.

Interactive consistency was first introduced and studied by Pease et al. [6], and has been the topic of several research papers ([1], [2], [3], [4], [5], [40]), focusing on synchronous systems. While these approaches might be feasible in environments such as shared memory multi-processors or digital flight control systems, we believe they are ill-suited for practical, real-world distributed systems. In [17] and [18], the authors provide solutions to various forms of consensus, despite their title references to IC.

A closely related problem to IC is vector consensus. These two problems differ only in terms of the *Validity* condition. Vector consensus delivers a vector with at least $2t + 1$ values, where at least $t+1$ values were proposed by non-faulty nodes. The reason for this difference is that in asynchronous systems, it is impossible to ensure that the resulting vector has the proposals of all non-faulty nodes ([7], [8]).

III. System Model

We assume a distributed system consisting of n nodes that are fully connected over a network. The network is mostly asynchronous, i.e., it exhibits one (or more, depending on the algorithm) period of synchrony, during which message delivery is timely. The network can drop, delay, duplicate, or deliver messages out of order. However, we assume that messages are eventually delivered, provided that the corresponding senders keep on retransmitting them. We assume authenticated channels, where the receiver of a message can always identify its sender. Each node has a public/private key pair and all nodes know the others' public keys. We use these keys to implement authenticated channels, and sign messages where needed.

We assume a Byzantine failure model where nodes may deviate arbitrarily from the protocol. We allow for a strong adversary that can coordinate faulty nodes. However, we assume he cannot delay the delivery of messages, or processing on correct nodes beyond the system's synchrony assumptions. The adversary is also assumed to be computationally bounded, meaning he cannot subvert common cryptographic techniques such as signatures and message authentication codes (MACs).

IV. Practical Interactive Consistency

A. Adapting approaches from synchronous systems

The original algorithm of Pease et al. [6] requires a total of $t+1$ rounds to achieve IC in a synchronous system, tolerating up to t faults, with a total message complexity of $(t + 1)n^2$. Our first approach is to adapt the same algorithm by simulating synchronous rounds with timeouts. Messages delivered after the time frame of each round, will be disregarded and counted towards the t system faults, according to the model presented in [41].

Two issues arise from the use of timeouts, as highlighted in [42]. The first one is efficiency. Assuming a timeout value of T_r for each round, the system will always require a constant amount of time, i.e., $(t+1)T_r$, to execute a request even in the presence of a single failure. The second is choosing a correct value for T_r. If we choose a conservative approach and set a large value for T_r, we could increase the execution time of the algorithm dramatically, thus, making it less practical. On the contrary, a small value might cause some slow nodes, who are otherwise correct, to be considered faulty. If this occurs multiple times, as is the case when one relies on multiple timeouts, it is possible that we will exceed the upper bound t of total failures in the system.

To avoid the issues associated with multiple timeouts, one might attempt to reduce IC to Byzantine Agreement (BA), by running n parallel instances of BA, as it was suggested for synchronous systems ([43]). In each instance, a node n_i would spread its private value v_i to the rest of the system. In a synchronous setting, this would result in all non-faulty nodes having the same vector of values. However, in a completely asynchronous environment, BA is impossible ([14]), as a crashed node may never even start its instance of BA, and nodes cannot distinguish between crashed nodes and slow nodes. Therefore, the non-faulty nodes need to decide, at a certain point, to exclude the suspected crashed nodes from IC and store a default (e.g., *null*) value at the slot corresponding

to each crashed node. Thus, they need a synchronization point, where they decide on the result vector; we call this point a *barrier*. This synchrony assumption allows for the circumvention of the impossibility of simultaneously achieving input completeness and guaranteed termination in a purely asynchronous system ([7]).

The introduction of the barrier introduces a new challenge as, at that point, a BA instance may have *delivered* the result in some nodes but not yet in others. This, for example, may be triggered by an adversary starting his own BA instance near the barrier. Thus, honest nodes will need to achieve consensus, for each individual slot of the result vector, on the value to be placed in that slot. We observe that the barrier splits the procedure in two phases. We call the first phase the *value dissemination phase*, where we assume the network delivers all messages of non-faulty nodes by the end of the phase. Recall that, as we stated in the first paragraph of this section, messages delivered after the time frame of the first phase will be counted towards the t system faults. We call the second phase, the *result consensus phase*, which can be completely asynchronous. Note that we have employed the costly BA approach for the first phase, but have shown that a consensus phase is still required.

B. Solution using Multi-Valued Consensus

With these observations, we seek less costly alternatives for the first phase, i.e., avoiding BA. Our first approach is to use a simple point-to-point message exchange, where each node announces its own private value to the rest of the system. As this exchange is unrestricted, it may result in each honest node receiving a different value from a malicious node. Thus, during the result consensus phase, nodes need to agree on the value to be placed in each slot of the result vector. We employ the multi-valued consensus (MC) algorithm from [8]; recall that this algorithm utilizes a binary consensus and a reliable broadcast primitive. We want to refrain from making any further synchrony assumptions, thus, making the result consensus phase completely asynchronous. In order to circumvent FLP, which states that achieving deterministic consensus is impossible in purely asynchronous systems, we employ a randomized consensus protocol. We use Bracha's binary consensus (BC) and reliable broadcast (RBB) primitives from [44], and we run n parallel instances of MC, one for each value of the vector.

This algorithm (*IC,MC-RBB*) achieves IC because, regardless of the unrestricted value dissemination phase, each instance of MC ensures that nodes agree on a single value for each slot of the result vector respectively. *(IC,MC-RBB)* uses only one synchrony barrier, as opposed to the adaptation of Pease's algorithm which needs $t + 1$. Its overall message complexity is $10n^4 + 5n^3 + n^2$. For full derivation details of all message complexities, and formal proofs of correctness of all IC algorithms, we refer the reader to the Appendix of the extended version of this paper ([45]).

C. Solution using Binary Consensus

Our next approach reduces the aforementioned message complexity. We observe multi-valued consensus uses one binary consensus and two reliable broadcast instances. We avoid the use of MC by changing the subject on which consensus is required. In the previous algorithm, the consensus question is "what is the actual value to be placed in the corresponding slot of the result vector?", because the first (value dissemination) phase is insecure. We make the first phase secure by using Consistent Broadcast (CB, [46]). Here, the source n_i first sends its value v_i to each node; then it collects signed endorsement responses. A recipient node endorses only the first value for each broadcast. Once $n - t$ such responses are accumulated, the sender forms a uniqueness certificate c_i that includes these endorsements, and sends $<n_i, c_i>$ to the rest of the nodes. CB *delivers* v_i *iff* c_i has at least $n - t$ valid signatures. Assuming signatures are unforgeable, it is impossible for a malicious node to construct two valid certificates for two different values. Thus, this protocol bounds the sender to either send a single value, or not send a value at all. As this value is guaranteed to be unique, we change the question of the result consensus phase to "is there a value to be placed in the corresponding slot of the result vector?". This question can now be answered by a binary consensus protocol, and we utilize Bracha's protocol ([44]) in our approach. Figure 1 depicts message exchanges for this protocol.

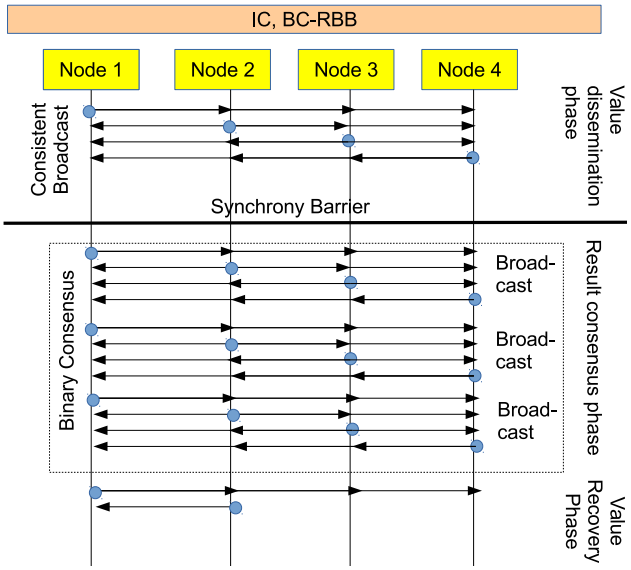

Fig. 1. Diagram of message exchanges for *(IC,BC-RBB)*, for a single value of the result vector (repeated n times to achieve IC).

An outcome of 0 from BC causes each node to place the *null* value in the corresponding slot of the result vector. Accordingly, a result of 1 from BC instructs each node to place the (unique) value v_i in the result vector. There are cases, however, where a consensus instance may produce a result different than the opinion with which an honest node entered BC. This can happen when the corresponding instance of CB *delivered* the value v_i at some nodes, but not at others (e.g., when a malicious CB source sends the value, along with the uniqueness certificate, only to some nodes). Thus, a node may possess a value for this slot, and the result of consensus may be 0, in which case it simply replaces the value with *null*. However, the contrary may also happen, where a node did not possess a value when it entered BC, but consensus resulted in 1. For this case, we add a final *recovery* phase, where a node

asks all other nodes for the correct value of the i^{th} position of the result vector. Any node that receives such a message replies with the $<v_i, c_i>$ tuple it possesses. At least one honest node is guaranteed to exist and submit such a reply; this is because, by definition of BC, if all honest nodes entered consensus with 0, the result would have been 0. As the result is 1, at least one honest node exists which has entered consensus with 1, thus possessing the correct value and uniqueness certificate for it.

To summarize, this IC algorithm *(IC,BC-RBB)* achieves IC because: a) during the value dissemination phase, an honest node either obtains a value guaranteed to be unique, or no value at all, b) during the result consensus phase, all nodes agree, for each slot of the result vector, whether to place a (guaranteed unique) value, or the *null* value, and c) during the recovery phase, any honest node is guaranteed to obtain missing values. The overall complexity of *(IC,BC-RBB)* is $6n^4 + 3n^3 + 3n^2$ messages and $n^3 + 2n^2$ signature operations. We formally prove the correctness of *(IC,BC-RBB)* in the Appendix of the extended version of this paper [45].

D. Eventual Interactive Consistency

So far, we present solutions to IC using one or more synchrony barriers, as the problem is unsolvable in a completely asynchronous setting. There is, however, a weaker version of the problem, which can be solved without timing assumptions (synchrony barrier), which we introduce and briefly outline two solutions. We call this weaker version *Eventual Interactive Consistency* (EIC). In EIC, the *Agreement* part of the problem's definition is as follows:

- *Agreement:* All non-faulty nodes must **eventually agree on the same vector of values** $V = [v_1, ..., v_n]$**.**

In this scheme, a non-faulty node will eventually build the result vector, containing all private values from all non-faulty nodes. Until it does, however, it may have empty slots for values it does not yet know about. In practice, the result vector will be built slot by slot, and instead of a single up-call to deliver the complete vector, multiple up-calls will take place. Each up-call will inform the application the vector was augmented by one more value. This version of the problem is suitable, for example, for applications which gather opinions. The idea is, the application can serve the already gathered opinions to clients, with empty slots for the currently unknown ones, either immediately upon request, or when a system-defined threshold of entries has been filled in the vector. Eventually, when all non-faulty nodes provide their opinions, empty slots in the result vector will represent failed nodes. If the vector is used before it is completed, the only guarantee provided is that on a subsequent access, the previous entries in the vector will still be included, potentially along with newer ones.

One simple approach to achieve this is by using a version of Reliable Broadcast (RB). All nodes start one instance and, by definition of RB, eventually all correct nodes' broadcasts deliver the intended value. When one of these RB instances completes, the corresponding slot of the vector will be filled and a new notification will be sent to the upper level application. This approach leaves management of the result vector completely up to the application.

A more involved approach which, however, preserves and manages the result vector as well, is the use of a byzantine fault tolerant Replicated State Machine (RSM), such as [47], [48] enhanced to handle byzantine clients ([49], [50]). Each node of EIC becomes both a replica and a client of the same RSM. Each EIC node, as a client of the RSM, posts its private value to the RSM as a *write* operation. The application running on top of the RSM receives this *write* operation and accepts it only when no prior value is known for the sending node and this instance of EIC. Whenever an external client requests the EIC result vector, it is dynamically compiled from entries already posted from nodes. A malicious node cannot harm the system as long as the RSM's fault tolerance level (typically $t < \lceil \frac{n}{3} \rceil$) is not breached, while as a client, it is prohibited from posting more than one value by the aforementioned functionality running at the application layer behind the RSM.

V. Implementation

We developed an open-source protocol stack in Java to implement and evaluate our suite of algorithms. At the foundation lies an authenticated channels layer that uses SSL and manages message passing and timeout events; SSL provides for authentication and message integrity. We also provide alternatives for direct TCP/IP communication (without strong authentication), as well as, an intra-process communication infrastructure that allows us to run our unit tests and verify the correctness of our implementation. The remaining layers are agnostic of the network layout or communication means, as they simply register event handlers to process incoming messages. Finally, we simulate loss-less channels by creating one output queue for each node, where each queue is handled by a different thread. A message is deleted from a queue only when the sender receives an acknowledgment for that specific message by the destination.

On top of this foundation, we implement Consistent Broadcast, and the signature-free Reliable Broadcast primitive of Bracha (RBB). We then implement Bracha's binary consensus (BC) protocol ([14]), which uses RBB. Finally, we implement the multi-valued consensus protocol (MC) of Correia et al. [8], using BC and RBB.

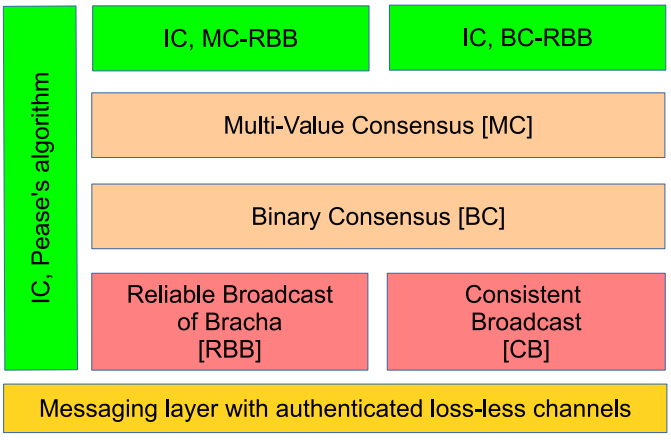

Fig. 2. The infrastructure and protocol stack of our implementation.

To reduce the overhead caused by signature operations used in Consistent Broadcast (CB), we use *authenticators* as

suggested by [47]. In this scheme, nodes exchange pair-wise messages to announce to the receiving party a symmetric secret key to use when sending messages to the sending party. This exchange is repeated often enough to make the symmetric key secure. When a node wishes to multicast a message to n nodes, it composes an *authenticator*, which is a vector of n HMACs, one for each receiving node, by using the corresponding key as input to the HMAC function. The receiving party uses its corresponding entry of the authenticator to verify both the integrity and the authenticity of the message, making this scheme analogous to digital signatures (for the closed world for which the authenticator provides HMACs). The performance improvement is vast as, in a simple evaluation on a contemporary desktop CPU, we can calculate approximately 300 SHA-1 based HMACs in the same amount of time required to produce a single digital signature using RSA with a 1024-bit key.

Bracha's binary consensus algorithm operates in a probabilistic number of phases. It requires nodes to buffer all messages, even the ones referring to future phases to guarantee termination. This, in conjuction with the fact that the number of rounds is not bounded, may require nodes to buffer an arbitrary number of messages. In a practical system, it is unrealistic to assume nodes with unbounded memory. Thus, malicious nodes could bombard non-faulty nodes with spurious messages which, since they are required to buffer them, would result in a state-explosion attack. Our approach on this matter is twofold. First, we identify that each phase of Bracha's consensus protocol is independent from any previous phases. This means that once a node enters phase $i + 1$, it can safely discard any buffered messages from phase i since they are no longer needed. Second, to defend against the state-explosion attack, nodes buffer messages whose current phase number i is a total of H phases ahead. However, this requires a recovery protocol for slow nodes that have fallen behind and are unable to progress due to the fact that the others have reached a phase $j > i + H$, which we leave as future work.

Finally, we leverage this protocol stack to provide the following interactive consistency algorithm suite:

1) Our adaption of Pease's algorithm *(IC,PEASE)*.
2) Consistent Broadcast for the value dissemination phase and Bracha's binary consensus, in conjunction with the reliable broadcast of Bracha for the result consensus phase *(IC,BC-RBB)*.
3) Multicast for the value dissemination phase and multi-valued consensus, in conjunction with the reliable broadcast of Bracha for the result consensus phase *(IC,MC-RBB)*.

VI. Evaluation

In this section, we experimentally evaluate the performance of the algorithms presented above, under various system settings. We conduct our experiments using a dedicated cluster with eight identical nodes, directly connected via an isolated 1Gbps Ethernet switch. Each node is configured with 4GB of RAM and dual Intel(R) Xeon(TM) CPUs running at 2.80GHz. We utilize up to 16 logical nodes by placing at most 2 nodes per physical machine. Measurements remain accurate, as cross-machine communication is always required for the algorithms

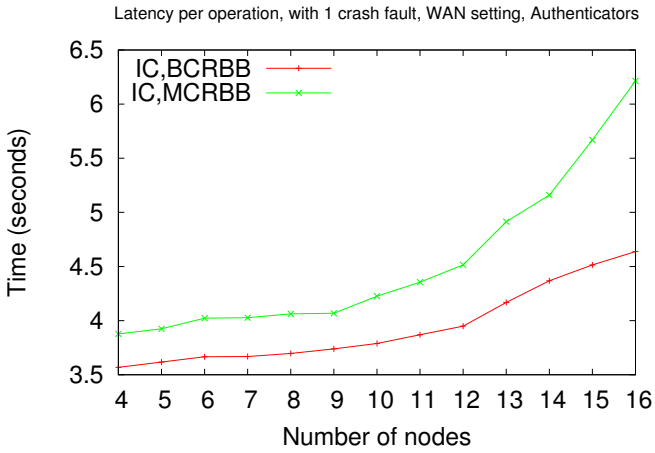

Fig. 3. Latency of *IC,BC-RBB* and *IC-MC-RBB*, with faults in WAN setting.

to progress and overshadows any communication benefits from co-located nodes. To emulate a WAN environment we utilize *netem* [51], a network emulator for Linux, to inject a uniform latency of 50ms.

In Figure 4a, we illustrate the total time required to complete one instance of each algorithm, without faults, in a local area network (LAN) setting. All algorithms are optimized to progress without waiting for timeouts, if all expected replies have been received. Our adaptation of Pease's algorithm exhibits the best performance due to its significantly lower message complexity and the fact that no timeouts are triggered. From the remaining alternatives, the one relying on multi-valued consensus for the result consensus phase performs worse than its binary consensus counterpart. This illustrates that the *(IC,BC-RBB)* approach, even though it has a more costly first phase, outperforms *(IC,MC-RBB)* because of its efficiency in the second phase. We repeat this experiment in the WAN setting (Figure 4d) and find that the same trend applies.

We now examine the effect of faults on the performance of each algorithm. We inject a single fault in the system, which is enough to trigger timeouts and reveal the cost of timing dependencies. In Figure 4b, we illustrate the performance of each algorithm in a LAN setting, with a modest timeout value of three seconds. Results illustrate the inefficiency of employing multiple timeouts in such an environment. Our adaptation of Pease's algorithm, which is the best alternative in the fault-free case, actually exhibits the worst performance in the presence of a single fault. We repeat this experiment in a WAN environment. Our results are depicted in Figure 4e and illustrate that the same trend applies. However, we note that we use the same timeout value in both LAN and WAN settings, i.e., three seconds, to provide for an even comparison. In a real deployment, one would employ a much larger timeout value for the WAN case, resulting in a more significant impact on the algorithms' performance. In Figure 3 we plot the same data as that shown in Figure 4e, but without Pease's algorithm. This allows us to see more clearly the differences between the other two algorithms. We find that *(IC,BC-RBB)* consistently provides latency improvements over *(IC,MC-RBB)*, ranging from 10% to 30%.

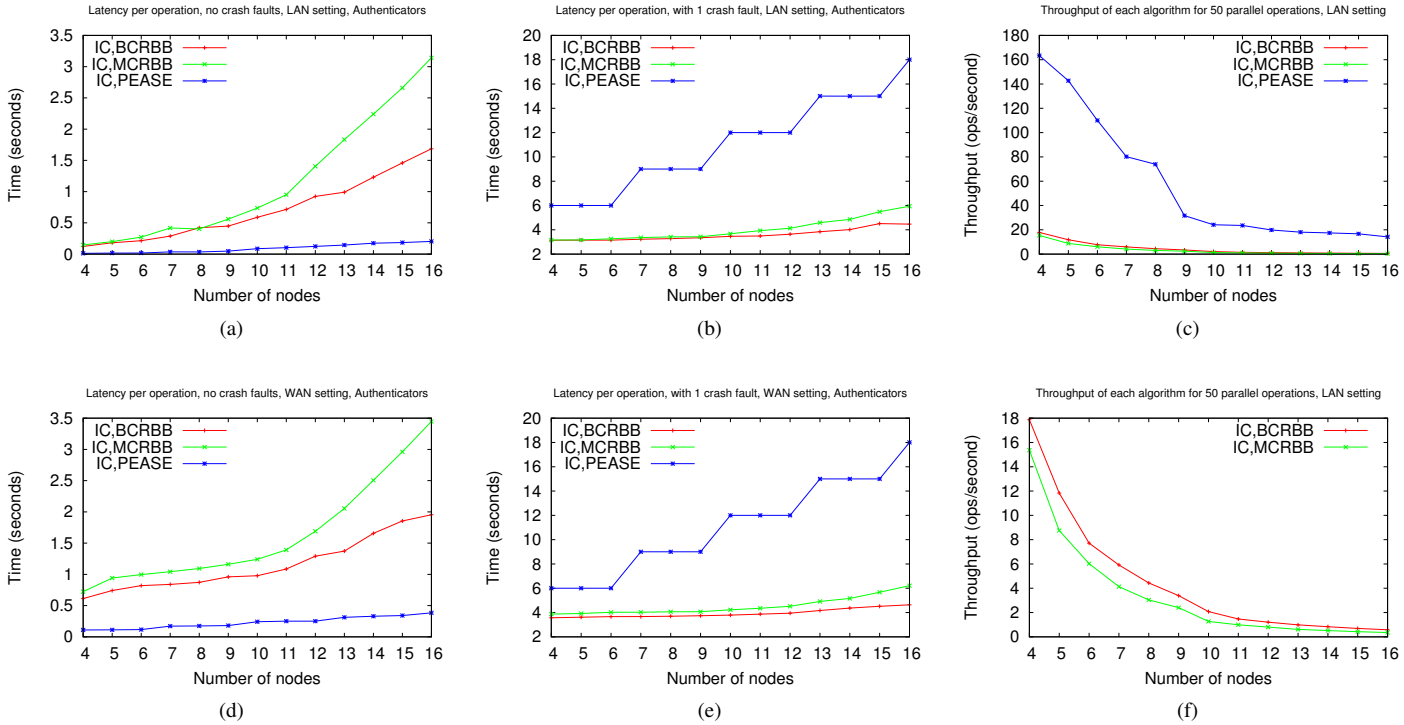

Fig. 4. Performance of the algorithms under LAN (4a - 4c, 4f) and WAN (4d - 4e) settings. The first two columns depict the latency of each algorithm under fault-free (4a, 4d) and faulty scenarios (4b, 4e). The third column depicts the throughput of each algorithm while executing 50 parallel instances with and without Pease's variant.

Lastly, we evaluate the throughput of the algorithms by running 50 parallel instances of each algorithm, in the LAN setting. We present our results in Figure 4c (all algorithms) and 4f (without Pease's variant). Since it is a fault-free case, Pease's variant exhibits the best performance. Between *(IC,BC-RBB)* and *(IC,MC-RBB)*, the former consistently provides higher throughput ranging from 16% to 63% improvement. *(IC,BC-RBB)*'s improved performance results from the reduced complexity of the result consensus phase, which is achieved by the use of consistent broadcast in the value dissemination phase. We repeated the same experiments in the WAN setting and observe similar trends, which we omit due to lack of space.

To summarize, our adaptation of Pease's variant is fastest in fault-free settings, due to its reduced message complexity and its lack of signature operations. However, once faults are introduced, *(IC,BC-RBB)* performs the best. Furthermore, *(IC,BC-RBB)* is among the top-two for all evaluated scenarios and requires only one synchronization point, in contrast with Pease's variant, which requires multiple. The reader will notice that absolute performance numbers are low. However, we are not proposing IC as a means to implement high-throughput applications. Instead, we have found IC to be useful in applications, such as a distributed e-voting system we built. We use it there, only once per election, to resolve diverged views of multiple vote collectors, after the election has completed [9].

VII. Conclusion

In this paper, we tackle the problem of Interactive Consistency (IC) in practical, real-world systems. This problem has received little attention in this setting so far, and we present a suite of algorithms and their implementations which can be used to solve this problem. These range from porting Pease's synchronous algorithm and making multiple timing assumptions, to composing more sophisticated algorithms based on existing broadcast and consensus primitives with a single timing assumption. We also define a more relaxed version of the problem, which we call *Eventual Interactive Consistency* (EIC), that is suitable for some applications, and we describe possible approaches for solving the problem without any timing assumptions.

Most of the algorithms in the protocol stack we built have been proposed but never been implemented and evaluated (to our knowledge) in a real system before. We have experimentally compared the performance of all algorithms and highlighted trade-offs that arise in different system settings. We find that one size does not fit all; for example, our adaptation of Pease's algorithm appears to be more appropriate for settings where node failures are rare, but once faults do occur, its performance degrades more than the remaining approaches. With this work, we hope to provide a framework with which system designers can reason about the appropriate IC approach to use. Our open-source software can be found in [52].

Acknowledgment

We thank the anonymous reviewers for their constructive comments that helped us improve our presentation. This work has been partially supported by ERC Starting Grant # 279237 and by the FINER project funded by the Greek Secretariat of Research and Technology under action "ARISTEIA 1".

REFERENCES

[1] S.-S. Wang, K.-Q. Yan, and S.-C. Wang, "Achieving efficient agreement within a dual-failure cloud-computing environment," *Expert Syst. Appl.*, 2011.

[2] P. M. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failure modes," in *SRDS*, 1988.

[3] P. Lincoln and J. Rushby, "Formal verification of an interactive consistency algorithm for the draper FTP architecture under a hybrid fault model," in *Compass*, 1994.

[4] A. Gascón and A. Tiwari, "A synthesized algorithm for interactive consistency," in *NFM*, 2014.

[5] J. H. Lala, "A byzantine resilient fault tolerant computer for nuclear power plant applications," in *FTCS*, 1986.

[6] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, 1980.

[7] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, "Universally composable synchronous computation," in *TCC*, 2013.

[8] M. Correia, N. F. Neves, and P. Veríssimo, "From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures," *Comput. J.*, 2006.

[9] N. Chondros, B. Zhang, T. Zacharias, P. Diamantopoulos, S. Maneas, C. Patsonakis, A. Delis, A. Kiayias, and M. Roussopoulos, "A distributed, end-to-end verifiable, internet voting system," http://arxiv.org/pdf/1507.06812v1.pdf.

[10] E. Athanasopoulos, S. Ioannidis, and A. Sfakianakis, "Censmon: A web censorship monitor," in *USENIX FOCI*, 2011.

[11] G. Goodell, M. Roussopoulos, and S. Bradner, "A directory service for perspective access networks," *IEEE/ACM Trans. Netw.*, 2009.

[12] S. Wang, K. Yan, C. Ho, and S. Wang, "The optimal generalized byzantine agreement in cluster-based wireless sensor networks," *Computer Standards & Interfaces*, 2014.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*.

[14] G. Bracha, "Asynchronous byzantine agreement protocols," *Inf. Comput.*, 1987.

[15] S. Toueg, "Randomized byzantine agreements," in *PODC*, 1984.

[16] A. Patra and C. P. Rangan, "Communication optimal multi-valued asynchronous byzantine agreement with optimal resilience," in *ICITS*, 2011.

[17] Z. Milosevic, M. Hutle, and A. Schiper, "Unifying byzantine consensus algorithms with weak interactive consistency," in *OPODIS*, 2009.

[18] A. Postma and T. Krol, "Interactive consistency in quasi-asynchronous systems." in *ICECCS*, 1996.

[19] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, 1982.

[20] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, 1985.

[21] R. Canetti and T. Rabin, "Fast asynchronous byzantine agreement with optimal resilience," in *STOC*, 1993.

[22] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, 1987.

[23] G. Bracha and S. Toueg, "Resilient consensus protocols," in *PODC*, 1983.

[24] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," in *PODC*, 1983.

[25] R. Canetti and T. Rabin, "Fast asynchronous byzantine agreement with optimal resilience," 1998.

[26] A. Patra, A. Choudhury, and C. P. Rangan, "Asynchronous byzantine agreement with optimal resilience," *Distributed Computing*, 2014.

[27] A. Mostefaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous byzantine consensus with $t < n/3$ and o(n^2) messages," in *PODC*, 2014.

[28] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," in *PODC*, 2000.

[29] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, 1996.

[30] A. Doudou and A. Schiper, "Muteness detectors for consensus with byzantine processes," in *PODC*, 1997.

[31] R. Guerraoui and P. Kouznetsov, "On the weakest failure detector for non-blocking atomic commit," in *IFIP TCS*, 2002.

[32] J.-M. Hélary, M. Hurfin, A. Mostefaoui, M. Raynal, and F. Tronel, "Computing global functions in asynchronous distributed systems with perfect failure detectors," *TPDS*, 2000.

[33] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Byzantine fault detectors for solving consensus," *The Computer Journal*, 2003.

[34] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, 1996.

[35] M. Larrea, A. Fernández, and S. Arévalo, "Optimal implementation of the weakest failure detector for solving consensus," in *SRDS*, 2000.

[36] V. K. Garg and J. R. Mitchell, "Implementable failure detectors in asynchronous systems," in *FSTTCS*, 1998.

[37] M. Correia, G. S. Veronese, N. F. Neves, and P. Veríssimo, "Byzantine consensus in asynchronous message-passing systems: a survey," *IJCCBS*, 2011.

[38] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," 1994.

[39] M. K. Reiter, "Secure agreement protocols: Reliable and atomic group multicast in rampart," in *CCS*, 1994.

[40] M. Ben-Or and R. El-Yaniv, "Resilient-optimal interactive consistency in constant time," *Distributed Computing*, 2003.

[41] D. Dolev and R. Strong, *Distributed Commit with Bounded Waiting*, 1982.

[42] R. Guerraoui and A. Schiper, "Consensus: the big misunderstanding," in *FTDCS*, 1997.

[43] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *FCT*, 1983.

[44] G. Bracha, "An asynchronous $[(n-1)/3]$-resilient consensus protocol," in *PODC*, 1984.

[45] P. Diamantopoulos, S. Maneas, C. Patsonakis, N. Chondros, and M. Roussopoulos, "Interactive consistency, in practical, mostly asynchronous systems," http://arxiv.org/pdf/1410.7256.pdf.

[46] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Advances in Cryptology, Crypto 2001*.

[47] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[48] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *DSN*, 2014.

[49] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *NSDI*, 2009.

[50] B. Liskov and R. Rodrigues, "Byzantine clients rendered harmless," in *Distributed Computing*, 2005.

[51] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*.

[52] "Interactive consistency repo," http://ds.di.uoa.gr/software.html.