Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication

Christian Berger
Universität Passau
Passau, Germany

Hans P. Reiser
Universität Passau
Passau, Germany

João Sousa
LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal

Alysson Bessani
LASIGE, Faculdade de Ciências,
Universidade de Lisboa, Portugal

***Abstract*—In geo-replicated systems, the heterogeneous latencies of connections between replicas limit the system's ability to achieve consensus fast. State machine replication (SMR) protocols can be refined for their deployment in wide-area networks by using a weighting scheme for active replication that employs additional replicas and assigns higher voting power to faster replicas. Utilizing more variability in quorum formation allows replicas to swiftly proceed to subsequent protocol stages, thus decreasing consensus latency. However, if network conditions vary during the system's lifespan or faults occur, the system needs a solution to autonomously adjust to new conditions.**

We incorporate the idea of self-optimization into geographically distributed, weighted replication by introducing AWARE, an automated and dynamic voting weight tuning and leader positioning scheme. AWARE measures replica-to-replica latencies and uses a prediction model, thriving to minimize the system's consensus latency. In experiments using different Amazon EC2 regions, AWARE dynamically optimizes consensus latency by self-reliantly finding a fast weight configuration yielding latency gains observed by clients located across the globe.

***Index Terms*—adaptation, weighted replication, consensus, geo-replication, Byzantine fault tolerance, self-optimization**

I. Introduction

State machine replication (SMR) is a classical approach for building resilient distributed systems. It achieves fault-tolerance by coordinating client interactions with independent server replicas [1]. SMR protocols typically use either some dynamically selected leader [2] or are fully decentralized [3]. In both cases, protocols usually require communication steps involving a major subset of all nodes.

With the emergence of novel Byzantine fault-tolerant (BFT) blockchain infrastructures, BFT SMR protocols have been increasingly catching academic attention over the last few years. For example, the BFT-SMaRt [4] library has been employed as an ordering service [5] for the Hyperledger Fabric [6] blockchain platform allowing for a high-performance, resilient service execution that achieves sub-second latencies in a geographically distributed environment using the WHEAT [7] optimization. In fact, BFT SMR protocols (typically called *BFT consensus*) are a key component of permissioned blockchains as they can be used to establish total order of transactions in a closed group of processes without relying on the expensive proof-of-work mechanism [8], achieving thus a much higher performance even with few tens of nodes, a significant consortium size for this type of blockchain [5], [6]. However, they might still be deployed in a wide-area network (WAN) for decentralization.

This work was supported by DFG through project OptSCORE2 (RE 3490/2-2) and by FCT through projects IRCoC (PTDC/EEISCR/6970/2014), Abyss (PTDC/EEI-SCR/1741/2041), and LASIGE Research Unit (UID/CEC/00408/2019).

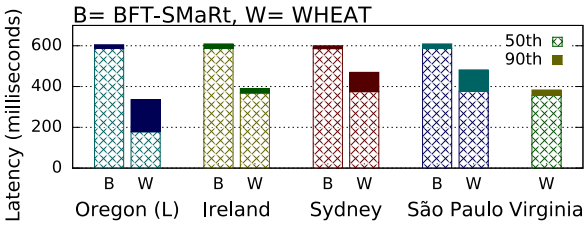


Figure 1: Latency gain using weighted replication in WHEAT and BFT-SMaRt [7], as measured by clients co-located with different replicas.

Weighted replication. While inter-node network latencies typically are similar between all pairs of nodes in a LAN environment, we can observe variations of latencies in WAN environments. The end-to-end latency of protocols waiting for a set of messages to be received is determined by the slowest replica in a subset which forms a quorum, i.e., contains enough replicas to convince a replica to advance to the next protocol stage. By introducing WHEAT [7] (**W**eig**H**t-**E**nabled, **A**ctive replica**T**ion), Sousa and Bessani have shown that using additional spare replicas and weighted replication allows the system to benefit from more variety in quorum formation. Thus, the system can make progress by accessing a proportionally smaller quorum of replicas and can obtain a significant latency decrease (see Figure 1).

Automation needed. However, this benefit holds only if one selects the optimal weight configuration. A manual selection is difficult in practice, as the decision of what is the best configuration for a given set of network characteristics is non-trivial. Further, network characteristics may be subject to run-time variations such as attacks and other problems, and thus the optimal configuration may also change at run-time. To make weighed replication practically usable, the SMR system needs a mechanism for automated and dynamic optimization. We present AWARE (**A**daptive **W**ide-**A**rea **RE**plication), a mechanism for allowing geo-replicated state machines to self-optimize at run-time: our method employs continuous self-measurements of the replicas' communication link latencies

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. This paper is accepted to be published in the proceedings of the 2019 38th IEEE International Symposium on Reliable Distributed Systems (SRDS 2019). For the final, published version refer to DOI 10.1109/SRDS.2019.00029

and analyzes the leader's expected consensus latency for all possible weight distributions and leaders. It reconfigures the system to minimize consensus latency, thus also leading to latency gains observed by clients distributed across the globe.

Challenges and contribution. We address practical challenges that arise when incorporating steady self-optimization into WHEAT by proposing AWARE, which allows the SMR system to dynamically find a fast configuration and to adapt to changing environmental conditions during the system's life span. Our contributions aim for making WHEAT more viable for practical deployment. Our paper addresses the following problems introduced by equipping a BFT system with such a self-optimizing mechanism:

- *Self-monitoring in an SMR system.* Defining suitable strategies for self-monitoring, including the question of how can the system cope with incomplete and possibly counterfeit measurement information?
- *Deciding an optimal configuration.* How can we compute the expected latency benefits for WHEAT configurations? We reason about consensus latency prediction and problems related to assessing configurations.
- *Safe and deterministic reconfiguration.* In the overall algorithm, all correct replicas must reach agreement on a self-optimization before triggering a reconfiguration.
- *Mitigating faulty replicas' influence.* Can we redistribute voting power of unavailable (e.g., crashed) replicas and which threats impose malicious replicas?

Outline. We start by explaining relevant preliminary work (§II). Then, we present AWARE's monitoring methodology (§III) and subsequently describe our algorithm for finding an optimal configuration and deciding on a reconfiguration (§IV). Moreover, we conduct an experimental evaluation of AWARE using different Amazon EC2 regions (§V). Finally, we discuss related research work (§VI) and draw conclusions (§VII).

II. BFT Replication

Castro and Liskov describe a practical replication algorithm for tolerating Byzantine faults, called Practical Byzantine Fault-Tolerance (PBFT) [2], that works in a partially asynchronous environment and incorporates optimization techniques to achieve throughput comparable to non-replicated services. Clients send requests to replicas (to the primary or upon timeout to all) that run consensus about the ordering of requests. PBFT tolerates up to f Byzantine servers, which may fail in an arbitrary way without compromising the service. The Byzantine fault model [9] usually requires a BFT protocol to use a total of $n = 3f + 1$ replicas to guarantee both liveness and safety under the partially synchronous system model.

A. BFT-SMaRt

BFT-SMaRt [4] is an open-source library written in Java that implements robust and configurable BFT SMR. It has some important advantages compared to other SMR implementations – for example UpRight [10] or PBFT [2]: it employs a dynamically scalable replica set, provides a modular architecture using strictly separated and exchangeable components for different concerns, e.g., *state transfer*, *reconfiguration*, or *consensus*, and it also achieves high performance because of its multi-core awareness design and various optimization techniques it incorporates. Furthermore, it can be configured to run in CFT or BFT mode. For CFT, it requires fewer replicas ($n = 2f + 1$) and operates faster (2 protocol steps are then required for running consensus instead of 3).

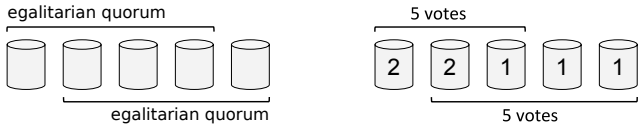
BFT-SMaRt uses the Mod-SMaRt protocol [11], an algorithm for SMR that employs an underlying leader-driven consensus primitive (the consensus algorithm described by Cachin [12]). A client broadcasts its request to all replicas and waits for a specific quorum of replies. Replicas use a consensus algorithm to achieve total order among all requests, thus deciding the batch of requests to be executed next in every consensus instance.

Byzantine consensus consists of three phases [12]: *PROPOSE*, *WRITE*, and *ACCEPT*. In the *PROPOSE* step, the leader broadcasts a message that contains a batch of requests that need to be decided to all other replicas. The following two communication steps, *WRITE* and *ACCEPT*, are all-to-all broadcasts used for commitment. In these steps, each replica i forms a quorum Q_i containing $\lceil \frac{n+f+1}{2} \rceil$ replicas to proceed (*Byzantine majority*). Two different replicas $i \neq j$ might use two different quorums to advance, but these quorums overlap in at least one correct replica, i.e., $|Q_i \cap Q_j| \geq f + 1$.

B. WHEAT

WHEAT [7] is a variant of BFT-SMaRt's state machine replication protocol that is optimized for geo-replicated environments. Its main innovation is the ability to decrease client latency by, counter-intuitively, *adding* more replicas to the system. The reason why this can result in a latency decrease instead of the opposite, is because quorums in WHEAT are not formed using a Byzantine majority of replicas, as it is done in the rest of the BFT literature [1], [2], [10], [11], [13], [14]. In WHEAT's case, the size of a particular quorum can actually be smaller than a Byzantine majority. Moreover, since WHEAT is expected to operate in wide-area networks, we can leverage the environment's heterogeneity to rely on the replicas that display the lowest end-to-end latency to be the ones forming these smaller quorums, and use the rest to form larger quorums that act as a fallback if the smaller ones are unavailable.

In order to understand how this works, let's consider a BFT system that, instead of comprising the usual number of four replicas (the theoretical limit to withstand a single Byzantine fault), actually comprises five (thus containing one extra replica). Let's consider the quorum formation for this setup. Recall that the definition of a BFT quorum system is a collection of subsets of replicas in which any two subsets intersect by $f + 1$ replicas [15]. To ensure quorum formation, BFT systems typically probe a Byzantine majority of replicas, as depicted in Figure 2a. As we can see, using a Byzantine majority, the extra replica makes the quorum size increase from 3/4 to 4/5 across all possible combinations. However, it is also possible to enforce quorum formation by relying on weighted replication, as depicted in Figure 2b. In this case, by



(a) Egalitarian: all quorums contain $\lceil \frac{n+f+1}{2} \rceil$ replicas.

(b) Weighted: a quorum contains at most $n-f$ and at least $2f+1$ replicas.

Figure 2: Possible quorums for $n = 5$, $f = 1$, $\Delta = 1$ (BFT).

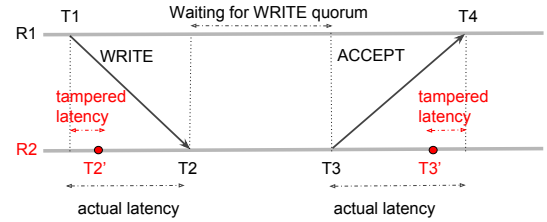


Figure 3: Problem with timestamps and Byzantine replicas.

probing a majority of votes rather than a majority of replicas, we can see that there exist combinations of replicas that still intersect by $f+1$ replicas, thus forming quorums in the size of 3/5 and others in the size of 4/5. Now imagine that this is a geo-replicated environment where the two best-connected replicas are assigned the highest weight with value 2. In spite of having five replicas in the system, progress is made by typically probing three replicas to form a smaller quorum. If for some reason any of these two fastest replicas is not available – either due to a period of asynchrony or a crash – progress can still be made by falling back to a quorum size of four replicas. Moreover, we can also re-distribute weights if any of the preferred replicas become slower. This approach is preferable to replacing replicas, since that would require new replicas to retrieve the state from others.

Further, for generalizing the above insight to any number of replicas, WHEAT employs the following safe weight distribution scheme [7]: let's assume a system of n replicas, tolerating f Byzantine faults and containing Δ additional replicas. The relation between these variables is as follows:

$$n = 3f + 1 + \Delta \tag{1}$$

Moreover, to account for weighted replication, WHEAT demands each replica to wait for Q_v votes, computed as follows:

$$Q_v = 2(f + \Delta) + 1 \tag{2}$$

In order to correctly form quorums, WHEAT adopts a binary weight distribution in which a replica can have a value of either V_{min} or V_{max}. These values are computed as follows:

$$V_{min} = 1 \tag{3}$$

$$V_{max} = 1 + \frac{\Delta}{f} \tag{4}$$

Finally, V_{max} is attributed to the $2f$ best-connected replicas in the system. All other replicas are attributed V_{min}. Using this distribution scheme, any quorum will contain between $2f+1$ replicas and $n-f$ replicas, instead of the fixed number of $\lceil \frac{n+f+1}{2} \rceil$ replicas as in traditional systems [2], [4].

III. Monitoring Strategy

AWARE's self-optimization approach relies on sound self-monitoring capabilities of the system, which in turn require reliable measurements.

A. Monitoring BFT Consensus

Problem. Quorum-based measurements (e.g., measuring the time between replica R_1 sending a *WRITE* to R_2 and receiving an *ACCEPT* back from R_2) do not allow reasoning about link latencies because both message types *do not casually depend on each other*. Replica R_2 might form a *WRITE* quorum without R_1 and the *WRITE* from R_1 might even arrive at R_2 after R_2's *ACCEPT* arrives at R_1. In a non-malicious setting, we could piggyback responses carrying timestamps T_2 and T_3 in *WRITE* messages of subsequent protocol runs and thus compute link latencies using timestamps generated by both parties, e.g., by approximating the latency using $((T_4 - T_1) - (T_3 - T_2))/2$ as shown in Figure 3. However, malicious replicas can attach corrupt timestamps, e.g., malicious replica R_2 might try to shift T_2 closer to T_1 and T_3 closer to T_4, while correct replica R_1 has no means to detect this lying behavior and thus attributes R_2 a better latency. Byzantine replicas could try to abuse such behavior to increase their voting power.

WRITE and ACCEPT. To prevent this, we favor one-sided measurements, which only require *the measuring replica to be correct*. In this method, we employ additional response messages for monitoring the consensus pattern: replicas *immediately* respond to a protocol message by directly sending a *WRITE-RESPONSE* after receiving a *WRITE*. Thus, the measuring replica can use $(T_4 - T_1)/2$ as one-way link latency. However, this introduces monitoring overhead into the system. These latencies allow us to reason about times in which replicas form weighted quorums to proceed to subsequent protocol stages.

PROPOSE. PROPOSE messages are larger than other messages because they carry the actual consensus proposal (a batch of client requests) instead of just a cryptographic hash, thus they may have a higher latency. This is also relevant for predicting the consensus latency, because every replica can only start broadcasting its *WRITE* message after having received a *PROPOSE* first. Further, we also use a response message for the latency measurement of this phase.

The *PROPOSE* latency is also relevant for *automated leader location optimization*. Therefore, we measure the latencies of non-leaders proposing to other replicas in order to determine hypothetical latency gains for the system when using a different replica as the protocol leader.

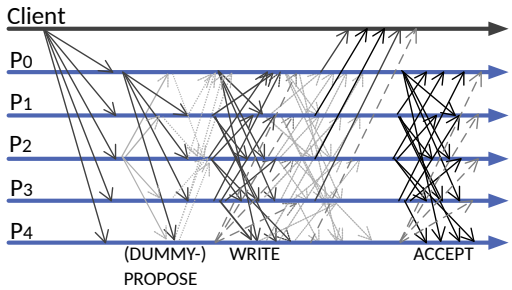


Figure 4: Message flow of BFT AWARE ($f = 1; \Delta = 1$).

B. AWARE Approach

Following a systematic approach, we develop customizable variants of AWARE. In the following, we give a brief summary of design decisions and configurable options.

Response to WRITE. In our approach, we expect each correct replica i to measure the latencies of its point-to-point links to every other replica and maintain a latency vector $L_i = \langle l_{i,0}, ..., l_{i,n-1} \rangle$. We use the *WRITE-RESPONSE*[1] messages to measure latencies between replicas. Further, the response message needs to include a challenge, e.g., a number which was beforehand randomly generated by the sender and attached to the original protocol message. This way we can guarantee that a replica has received the *WRITE* and that Byzantine replicas cannot send responses to messages before actually having received them.

Non-Leaders' PROPOSE. The *DUMMY-PROPOSE* allows measuring precisely the time non-leaders need to *PROPOSE* batches of possibly large size to the rest of the system, where we expect a difference in cases where the network becomes the bottleneck. Non-leaders do not simultaneously propose to avoid creating a high overhead to the system, degrading its performance and counter-acting our goal of improving the performance. We use a *rotation scheme* in which only one additional replica simultaneously broadcasts a *DUMMY-PROPOSE* along with the leader, proposing a dummy batch in the same way as the leader does, but without starting a new consensus instance and all replicas disregard the proposal. Replicas reply with a *PROPOSE-RESPONSE* and include the proposed batch in the response message. Using the *DUMMY-PROPOSE* is optional as it introduces overhead to the system (see §V-E) and it is also possible to approximate these latencies using the measurements of *WRITE-RESPONSE*.

Figure 4 shows the *message flow*[2] of AWARE utilizing all monitoring messages. This yields the variant of AWARE with the highest accuracy in leader selection. Furthermore, AWARE defines the number of recent monitoring messages to be used for computation of the latencies for each connected replica in a configurable *monitoring window*.

Moreover, in AWARE each correct replica i periodically reports its latency vector L_i to all other replicas. Replicas do this after some configurable *synchronization period* by disseminating these measurements with total order (thus running consensus on them) so that all replicas maintain the same latency matrix after some specific consensus instance. We employ a deterministic procedure for deciding a reconfiguration and use the same monitoring data in all correct replicas (while it would also be possible for replicas to have distinct views on the measurements and then run consensus on possible actions).

Once replicas have synchronized measurements after a given consensus instance, they employ the model we explain in §IV-C to predict the best weight distribution and leader. Replicas use a *calculation interval* defining the number of consensus instances after which a calculation and possibly a reconfiguration is being triggered.

Bounding monitoring overhead. We can arbitrarily decrease the monitoring overhead by specifying a parameter $\omega \in [0, 1]$ that determines the maximum overhead induced by the monitoring procedure. Implementation-level details such as the frequency of sending specific monitoring messages (e.g., *DUMMY-PROPOSE*) are automatically derived from ω. Frequent measurements provide more up-to-date monitoring data and allow for faster reaction to environmental changes but also negatively impact the maximum throughput (see §V-E).

C. Sanitization

All replicas maintain synchronized latency matrices M^P and M^W for keeping measurements of *PROPOSE* and *WRITE* latencies, both initially filled with entries

$$M[i,j] \leftarrow \begin{cases} +\infty, & \text{if } i \neq j \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

$M[i,j]$ expresses the latency of replica i to j as measured by i. Further, replica i can update a row of these matrices with its measurements L_i^P and L_i^W with total order by using the *invoke* interface of BFT-SMaRt:

$$\textit{invokeOrdered}(\mathsf{MEASURE}, L_i^P, L_i^W);$$

The updating process yields a matrix M, with $M[i,j] = L_i[j]$ if replica i sent its measurements within the last calculation interval c of measurement rounds, or a missing value ($+\infty$) if it did not send any measurements within the last c rounds. We sanitize both matrices immediately before the calculations happen to mitigate the influence of malicious replicas.

We do that by exploiting the symmetry characteristic of replica-to-replica latencies and let replicas have a pessimistic standpoint on measurements. They use the pairwise larger delay in calculations so that replicas cannot make themselves appear faster. This procedure yields

$$\hat{M}[i,j] = max(M[i,j], M[j,i]) \tag{6}$$

Figure 5 presents an example. This way, Byzantine replicas cannot fraudulently improve their link latency to any *correct* replica, and they also cannot blame (worsen a link latency to) a correct replica *without being contributed a bad latency themselves*. Still, in case of $f > 1$, Byzantine replicas may

[1]We do not need to use an *ACCEPT-RESPONSE* because the *ACCEPT* phase has the same message pattern as the *WRITE* phase.

[2]The message pattern of WHEAT/AWARE differs from BFT-SMaRt in the use of *tentative executions*, an optimization that was introduced in PBFT [2].

$$\begin{bmatrix} 0 & 68 & 69 & 93 & 40 \\ 67 & 0 & 133 & 92 & 35 \\ 69 & 132 & 0 & 157 & 99 \\ 92 & 92 & 156 & 0 & 69 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 68 & 69 & 93 & 40 \\ 68 & 0 & 133 & 92 & 35 \\ 69 & 133 & 0 & 157 & 99 \\ 93 & 92 & 157 & 0 & 70 \\ 40 & 35 & 99 & 70 & 0 \end{bmatrix}$$

(a) Before sanitization. (b) After sanitization.

Figure 5: Sanitized matrix reporting median *WRITE* latencies.

show intriguing behavior, e.g., by claiming to have tremendous connections to each other (see §V-F).

IV. Self-optimization

In our self-optimizing approach, replicas deterministically reconfigure to a new weight configuration and/or leader position. This requires all replicas to (1) agree on what the optimal configuration is and (2) decide whether they will adjust themselves accordingly by triggering a *view change*.

A. Optimizations

Overall, AWARE employs two dynamic optimizations:

Voting weights tuning. Adjusting voting weights leads to latency gains observed by clients across all sites [7]. AWARE searches for a weight distribution that optimizes the system's consensus latency.

Leader relocation. Relocating the leader in a well-connected site of the system reduces the request latency observed by clients [7], [16]. Hence, AWARE is capable of selecting the leader location as an optional optimization technique.

We follow the idea of obeying *leader selection constraints*. AWARE employs an abstraction, where the BFT protocol provides an interface for choosing a leader. In particular, we allow the protocol to provide a suitable set of *leader candidates* $\mathfrak{L}$ out of which AWARE chooses one.

In context of their decision, an *optimization goal* α defines the threshold (relative to the current configuration) by which a predicted consensus latency needs to be faster to trigger a reconfiguration. This prevents the system from *jumping back and forth* between configurations that are almost equally fast.

B. Consolidated Protocol

We ensure determinism by providing a deterministic consensus latency prediction function used for our calculation (see Algorithm 1). Further, we ensure that the measurement data is the same in all replicas after a specific consensus instance by synchronizing measured latency information with total order broadcast. We apply a *view-change* after a calculation performed in intervals specified by logical protocol rounds (epochs) rather than by time. This works as follows:

1) Each replica i collects its latency measurements (a moving median) in a vector $L_i = \langle l_{i,0}, ..., l_{i,n-1} \rangle$;
2) Periodically, each replica i disseminates its vectors for *PROPOSE* (L_i^P) and *WRITE* (L_i^W) with total order by calling $invokeOrdered(\mathsf{MEASURE}, L_i^P, L_i^W)$;
3) Once a replica i decides a batch, that batch may contain messages $\langle \mathsf{MEASURE}, L_j^P, L_j^W \rangle$ from some replica $j \in I$. It uses these vectors to update its synchronized matrices M_i^P and M_i^W, i.e., for replicas $k = 0, .., n-1$ assigning $M_i^W[j,k] = L_j^W[k]$ that is the information which i has about the latency between replica j and all other replicas measured by j. This applies to the maintained latency information for both *PROPOSE* (M_i^P) and *WRITE* (M_i^W).
4) When a defined number (specified by the calculation interval c) of consensus instances is reached, all replicas have the same matrices M^P and M^W, e.g., with $M^W[i,j] = L_i^W[j]$ if replica i sent its *WRITE* measurements within the last c consensus instances, or $M^W[i,j] = +\infty$ if i did not send its measurements. The same applies to M^P.
5) The next step is to deterministically sanitize the matrices to avoid the influence of malicious replicas (see §III-C), generating $\hat{M}^P$ and $\hat{M}^W$.
6) Now, every replica solves the following optimization problem, where $PredictLatency$ (Algorithm 1) is a function for predicting the latency of the consensus protocol using the latencies in $\hat{M}^P, \hat{M}^W$ and all possible weight distributions $W \in \mathfrak{W}$ and permitted leaders $l \in \mathfrak{L}$:

$$\langle \hat{l}, \hat{W} \rangle = \underset{W \in \mathfrak{W}, l \in \mathfrak{L}}{\arg\min}\ PredictLatency(l, W, \hat{M}^P, \hat{M}^W) \qquad (7)$$

In the end, the configuration $\langle \hat{l}, \hat{W} \rangle$ that provides optimal leader consensus latency is the one selected for the next reconfiguration if the predicted latency is better than the current configuration by the factor α (optimization goal). Note that since this procedure is deterministic, the $\langle \hat{l}, \hat{W} \rangle$ is the same in all replicas.
7) In case the replicas find a faster weight configuration, they update their view to respect the new voting weights for the following consensus instances. Optionally, if the system uses leader relocation, the replicas might also trigger a view change to elect a faster leader.

C. Latency Prediction

We predict the optimal configuration by simulating a protocol run for each configuration using the sanitized latency matrices $\hat{M}^P$ and $\hat{M}^W$ to compute the predicted consensus latency of the leader replica and subsequently select a configuration that minimizes this latency. We argue that a fast configuration is one that *yields a low consensus latency from the perspective of the leader.* Once having finished a consensus instance, a leader can prepare and propose the next batch [4], [11] while in the meantime, poorer connected replicas may still wait for messages to form their quorums. To make progress, a leader needs to fulfill its quorums of Q_v votes as well, so it needs to be well connected to replicas with preferably high voting power. The leader itself is always assigned V_{max} voting power in order to minimize its consensus latency.

Algorithm 1 is used to predict the leader's consensus latency for a given configuration by simulating the consensus protocol run and computing the times each replica can proceed to a subsequent protocol stage (illustrated in Figure 6), e.g.,

Algorithm 1: *PredictLatency* computes the consensus latency (amortized over multiple rounds) which requires forming weighted quorums of $Q_v = 2fV_{max}+1$

Data: replica set I, leader p, system sizes n, f, Δ, weight config. $W = \langle R_{max}, R_{min} \rangle$, latency matrices for *PROPOSE* $\hat{M}^P$ and *WRITE* $\hat{M}^W$, number of consensus rounds r
Result: consensus latency of the AWARE leader

1 $V_{max} \leftarrow 1 + \frac{\Delta}{f} \quad V_{min} \leftarrow 1 \quad V_i \leftarrow \begin{cases} V_{max}, & \text{if } i \in R_{max} \\ V_{min}, & \text{otherwise} \end{cases}$
2 **while** $r > 0$ **do**
3 **for** $i \in I$ **do**
4 $T_i^{PROPOSED} \leftarrow max(\hat{M}^P[p,i], oSet_i)$
5 $writes_i \leftarrow$ new PriorityQueue()
6 $accepts_i \leftarrow$ new PriorityQueue()
7 **for** $i \in I$ **do**
8 **for** $j \in I$ **do**
9 $writes_i.add(\langle T_j^{PROPOSED} + \hat{M}^W[j,i], V_j \rangle)$
10 **for** $i \in I$ **do**
11 $votes \leftarrow 0$
12 **while** $votes < Q_v$ **do**
13 $\langle T_{next}, V_{next} \rangle \leftarrow writes_i.poll()$
14 $votes \leftarrow votes + V_{next}$
15 $T_i^{WRITTEN} \leftarrow T_{next}$
16 **for** $i \in I$ **do**
17 **for** $j \in I$ **do**
18 $accepts_i.add(\langle T_j^{WRITTEN} + \hat{M}^W[j,i], V_j \rangle)$
19 **for** $i \in I$ **do**
20 $votes \leftarrow 0$
21 **while** $votes < Q_v$ **do**
22 $\langle T_{next}, V_{next} \rangle \leftarrow accepts_i.poll()$
23 $votes \leftarrow votes + V_{next}$
24 $T_i^{ACCEPTED} \leftarrow T_{next}$
25 **for** $i \in I$ **do**
26 $oSet_i \leftarrow T_i^{ACCEPTED} - T_p^{ACCEPTED}$
27 $consensusLatencies_r \leftarrow T_p^{ACCEPTED}$
28 $r \leftarrow r - 1$
29 **return** average of $consensusLatencies$

receiving the *PROPOSE* (line 4), forming the *WRITE* quorum (lines 10-15) and deciding the consensus value (lines 19-24). To be precise, the time a replica fulfills a quorum is computed by continuously polling messages from a priority queue (sorted by ascending arrival time of messages) and adding the voting weights up until the sum of gathered votes reaches the necessary quorum Q_v. The arrival time of the last message, necessary to reach this quorum, determines the time a replica can proceed to the next protocol stage. Further, the algorithm computes the amortized leader consensus latency over multiple rounds r. Note that replicas achieve consensus at different times (as can be seen in Figure 6) and if the difference between the time leader p decides and the time replica i decides is greater than the propose latency $\hat{M}^P[p,i]$, then replica i might receive a *PROPOSE* for the next consensus instance but wait for its last consensus to finish before broadcasting its *WRITE*. This might throttle the leader but only if he uses i in its quorum as the Q_vth *quorum formation speed determining* vote. We consider this in our calculations for a sequence of rounds by

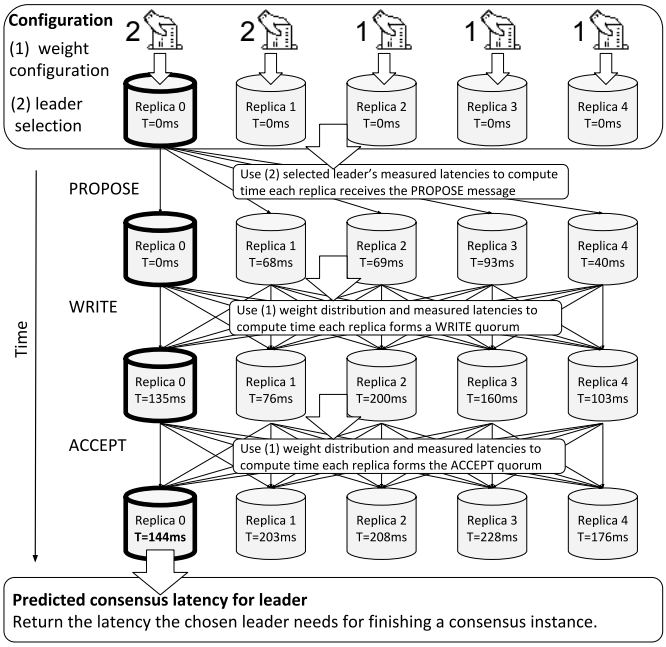


Figure 6: Computing the latency of a WHEAT consensus (here: BFT mode, $f = 1, \Delta = 1$) for a given configuration.

computing *offsets* (additive times other replicas need to finish their consensus relative to the leader).

An AWARE configuration defines the weight configuration and selects a leader. Hence, the number of possible configurations is the number of weight configurations multiplied with the number of possible leaders (V_{max} replicas):

$$\binom{3f+1+\Delta}{2f} \cdot 2f = \frac{\prod_{i=2f}^{3f+1+\Delta} i}{(f+1+\Delta)!} \tag{8}$$

This yields 20 possibilities for a $n = 5, f = 1, \Delta = 1$ system and 504 possibilities for a $n = 9, f = 2, \Delta = 2$ system. Traversing the entire search space of possible configurations becomes unfeasible for large f. However, (1) BFT systems typically run with tens of nodes and (2) if larger systems are needed, we can employ heuristics for approximating the optimum (e.g., *simulated annealing*) using *PredictLatency* as fitness function.

V. Evaluation

Throughout this section, we (1) experimentally quantify the margin of latency variations among different WHEAT configurations, (2) compare our model prediction for consensus latency with real-world measurements in terms of accuracy, (3) determine the correlation between consensus latency and measured request latency observed by clients across multiple regions, (4) evaluate the run-time behavior of AWARE when carrying out self-optimizations, (5) evaluate the maximum throughput of AWARE and investigate the monitoring overhead induced by the *DUMMY-PROPOSE*, and (6) reason about the system behavior in the presence of faulty replicas.

Setup. Unless stated otherwise, we use the Amazon AWS cloud to place our EC2 instances in specific regions. Since we

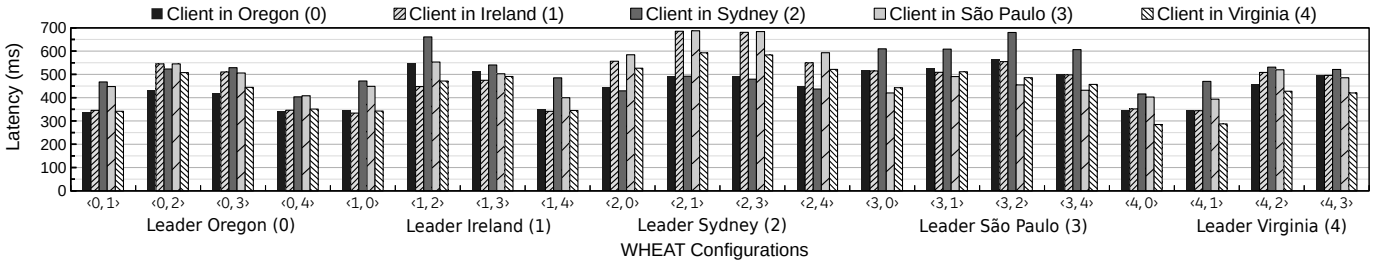


Figure 7: Measured average request latency of 11th to 90th percentile across clients in different regions.

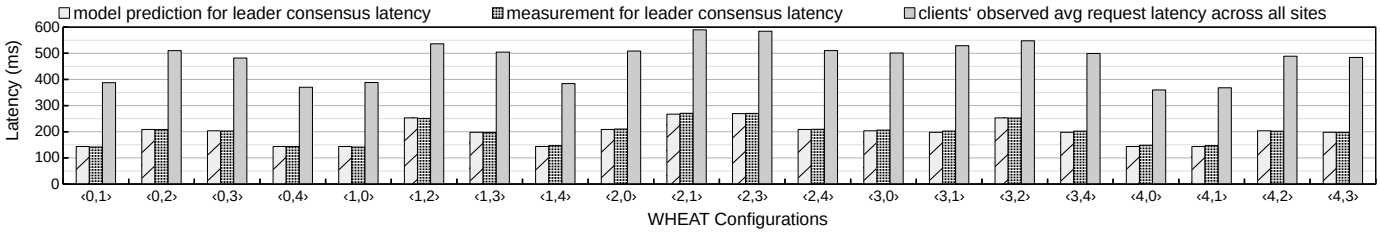


Figure 8: Comparison between predicted consensus latency, measured consensus latency and clients' observed request latency.

do not have high hardware requirements for our latency experiments, we use the *t2.micro* instance type, which is equipped with 1 vCPU, 1 GB of RAM and 8 GB standard SSD volume (gp2). We use WHEAT in the Byzantine fault model with $f = 1$ and $\Delta = 1$ additional spare replicas. Further, we select the (numbered) regions *(0) Oregon*, *(1) Ireland*, *(2) Sydney, (3) São Paulo* and *(4) Virginia*. In each region we start one virtual machine (VM) to construct our world-spanning replicated system. Every VM carries a replica and a client which conduct latency measurements. *Consensus latency* defines the time between a leader sending a proposal and it being decided. *Request latency* is the time between a client sending a request and receiving enough replicas' responses to accept the result. Replicas measure the average consensus latency of a 1000 consensus instances sample. Clients simultaneously send at least 1000 requests each and continue sending requests until each client has finished its measurements. A client request arriving at the leader replica may wait for some time until it gets included in a batch when there is currently a consensus instance running. We use synchronous clients that wait for the result and send the next request after waiting for a random time interval between 0 and 150 ms. Further, clients compute the average latency from the 11th to 90th percentile (to mitigate the influence of outliers) of perceived request latencies.

A. Margin of Latency Variations of Different Configurations

We start by justifying the question whether a dynamic approach to self-reliantly finding a well-performing configuration is needed by showing the gap between different WHEAT configurations. Figure 7 illustrates the observed client latencies for different regions. Each configuration is represented by a tuple $\langle L, R_{Vmax} \rangle$ where L is the leader and R_{Vmax} is the other replica (besides the leader) that has a voting weight of $V_{max} = 2$. Each number corresponds to a region as explained in the setup and Figure 7. We notice a big difference between the configurations. The best configuration is $\langle 4, 0 \rangle$ showing a latency (avg. across all clients) of 359.78 ms, the left median configuration $\langle 3, 4 \rangle$ performs in 499.06 ms and the worst configuration $\langle 2, 1 \rangle$ requires $589,56$ ms. The best WHEAT configuration is 38.7% faster than the median and 63.9% faster than the worst configuration. Further, we make four important observations:

(1) *Tuning voting weights can reduce latency:* the adjustment of weights is a promising optimization to reduce the latency even if the leader is fixed (see different configurations with the same leader).

(2) *Leader selection may be necessary for optimal latency:* a leader in *Sydney* or *São Paulo* is not well connected enough to the rest of the system. Relocation can improve the latency observed by all clients.

(3) *Co-located clients achieve slightly better latency:* a client co-located with the leader tends to observe lower request latencies than other clients within a specific configuration. Still, this does not necessarily imply a pareto-optimum: a client in Sydney observes 492.19 ms in $\langle 2, 1 \rangle$ (co-located with the leader) while it achieves its best results (among all configurations) in $\langle 0, 4 \rangle$ with 403.46 ms.

(4) *A global optimum does not exist* but a few pareto-optimal configurations dominate poorer performing configurations.

B. Accuracy of Consensus Latency Prediction

Our approach aims at finding a configuration with minimal leader consensus latency. Our prediction model (Algorithm 1) lets us compute these latencies for all configurations.

We compare our model prediction with the actual consensus latency of the leader that we measured for every configuration during our experiment (see Figure 8). For these configurations, our predictions are off by 1.08% on average. The highest prediction error is for $\langle 4, 0 \rangle$ (3.22%). Since our prediction relies on latency measurements which are subject to smaller

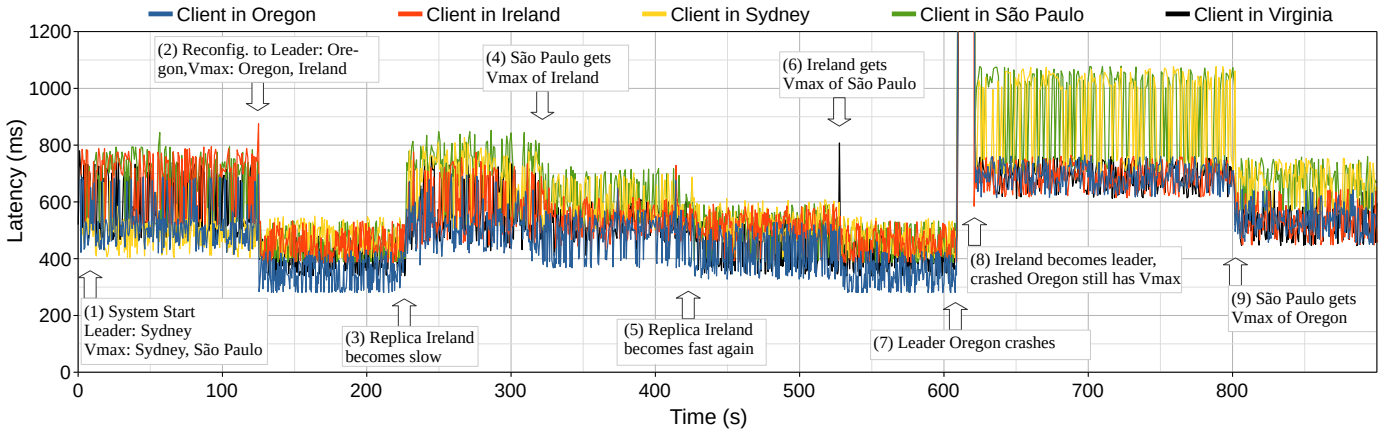


Figure 9: Runtime behavior of AWARE.

variations, we argue that these results are *reasonable for choosing a well-performing configuration* – however, AWARE might not always choose the actual best configuration but decide for some configuration that is close to the optimum.

In our example, AWARE will pick any configuration of $\langle 0,1\rangle$, $\langle 0,4\rangle$, $\langle 1,0\rangle$, $\langle 1,4\rangle$, $\langle 4,0\rangle$ or $\langle 4,1\rangle$ for which it predicts a leader consensus latency of $143.52\ ms$ amortized over 1000 consensus rounds. In our experiment, the measured latencies for these *optimal candidates* are between $141.30\ ms$ ($\langle 1,0\rangle$) and $148.31\ ms$ ($\langle 4,0\rangle$). If there is an optimal configuration containing the current leader, AWARE preferably chooses it over configurations where a leader change is necessary. On a side note, the median predicted leader's consensus latency is $202.26\ ms$ ($\langle 3,4\rangle$) and the worst is $270.50\ ms$ ($\langle 2,1\rangle$).

C. Clients' Observed Request Latency

Figure 8 also shows the clients' observed request latency (average across all sites) for all configurations and compares them with both model predictions and measurements for consensus latency.

As expected, consensus speed contributes to total latency. We notice a positive correlation $\rho(L^{MP}, L^{CR}) = 0.961$ between our series (over all configurations) of model predictions for leader consensus latency L^{MP} and the measurement series of average clients' request latency L^{CR}, indicating that *faster consensus is beneficial for geographically distributed clients*.

D. Runtime Behavior of AWARE

We deploy AWARE in our usual setting and observe its behavior during the system's lifespan. Overall, the clients' request latencies show high variations which is caused by a waiting time of a request at the leader: since all clients simultaneously send requests and the leader batches these, a client request may wait until the current consensus finishes to get into the next batch, which takes a varying time depending on how shortly the request arrived before the next consensus can be started[3]. We induce events to evaluate AWARE's reactions (see Figure 9) to certain conditions, in particular:

[3]This is the reason the clients' observed average request latencies are a little more than twice as high as the consensus latency in Figure 8.

1) Action: We start AWARE in a low-performance configuration $\langle 2,3\rangle$ with *Sydney* being the leader and *Sydney* and *São Paulo* having maximum voting power.
2) Reaction: After a calculation interval of $c = 500$, AWARE decides that *Oregon* and *Ireland* are faster and changes its configuration to $\langle 0,1\rangle$ leading to *latency gains observed by all clients across all sites*.
3) Action: We create network perturbations, in particular we add an outgoing delay of $120\ ms$ and $20\ ms$ jitter to the *Ireland* replica, thus making it slower (the client and replica of Ireland are not co-located on the same VM).
4) Reaction: AWARE attributes one of the V_{max} to *São Paulo* while *Ireland's* weight is reduced to V_{min}. Clients observe a small improvement in request latencies.
5) Action: We end the network delay for *Ireland*, thus the network stabilizes and the communication links of *Ireland* become just as fast as in the beginning of our experiment.
6) Reaction: AWARE notices this improvement and assigns the V_{max} of *São Paulo* back to *Ireland* since it predicts latency gains for this configuration. After the reconfiguration, clients observe faster request latencies identical to what happened after the first reconfiguration (Event 2).
7) Action: We crash the leader *Oregon* (which has V_{max}).
8) Reaction: Replicas' request timers expire and BFT-SMaRt triggers the leader change protocol: *Ireland* becomes the next leader. Since fV_{max} voting power becomes unavailable, all *remaining correct replicas are forced to use the same quorum* Q_v (all 3 V_{min} replicas and the leader). Accordingly, clients observe higher request latencies.
9) Reaction: AWARE redistributes the V_{max} to a former V_{min} replica, *São Paulo*, hence *restoring some degree of variability in quorum formation*. Replicas now can form smaller quorums. This leads to clients observing latency improvements across all regions.

E. Maximum Throughput

For measuring *maximum throughput*, we change the instance types to *c5.xlarge* (4 vCPU, 8 GB of RAM, 8 GB SSD) and use 5 VMs in our usual regions to place replicas, and

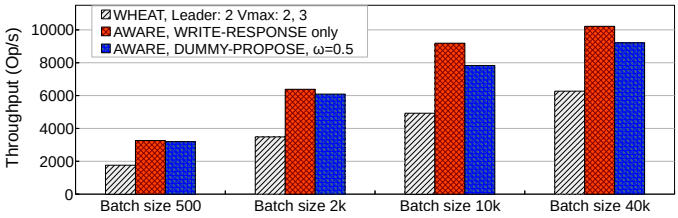


Figure 10: Maximum throughput comparison.

5 other VMs to launch as many clients as necessary to saturate the system. Asynchronous clients send requests of size 1 kB after randomly waiting between 0 ms and 10 ms and replicas' responses also have a size of 1 kB. We compare 3 different variants: (1) WHEAT in a bad configuration ($\langle 2, 3\rangle$), (2) AWARE, after having adjusted to $\langle 0, 4\rangle$ using *WRITE-RESPONSE*, and (3) AWARE, in configuration $\langle 0, 4\rangle$ and with enabled *DUMMY-PROPOSE* (we bound overhead with $\omega = 0.5$, hence only for every second consensus, one replica broadcasts a *DUMMY-PROPOSE*).

In principle, faster completing consensus rounds and increasing the batch size (number of requests decided per consensus) can both raise the throughput. However, larger batch sizes also lead to higher *PROPOSE* latencies and thus can slow down consensus. In our experiments, we observe (see Fig. 10), that (1) low consensus latency indeed has positive effects on throughput for different batch sizes and (2) the monitoring overhead induced by enabling the *DUMMY-PROPOSE* is noticeable, but still passable, given that AWARE is mainly thought of as a latency optimization technique.

F. Effect of Faulty Replicas

AWARE deals with both crash and malicious faults as long as they are limited to f faulty replicas. In WHEAT, if replicas become unavailable for quorum formation, then their voting weight cannot be accessed until they get repaired. In AWARE, we need to distinguish two cases: first, replicas may crash, so their weight cannot be accessed for quorum formation anymore. Second, malicious replicas may adapt their behavior to prevent AWARE from redistributing weights.

Crash faults. In the first case, AWARE detects the unavailability as crashed replicas do not disseminate measurements nor do they respond to protocol messages. Thus $+\infty$ latencies are fed into AWARE's prediction model. In case crashed replicas have V_{max} voting power, AWARE redistributes it to faster replicas (see Figure 9, Event 9) and hence restores some variability in quorum formation. If the former fast but crashed replicas are repaired and reintegrate in the system, then they might re-obtain high voting power as well.

Malicious faults. In the second case, malicious replicas might not participate in the achieving of consensus (e.g., do not send *WRITE* messages), but still send response messages to other replicas and disseminate latency vectors that attribute them very low latencies, e.g., if $f > 1$, then pairs of malicious replicas could assert that their pairwise communication links have a latency close to 0. Since malicious replicas might participate in the monitoring process, their unavailability cannot be easily detected by AWARE and thus restricts AWARE's ability to redistribute voting weights only between correct replicas. If we assume the worst case, then f malicious replicas might in total possess up to fV_{max} voting power and force correct replicas to use the only remaining fallback quorum of $fV_{max} + (f+1+\Delta)V_{min}$ votes cast by all $2f+1+\Delta$ correct replicas to make progress. Still, we can always guarantee the availability of the system. Restricting the quorum formation variability can – in the presence of faults – generally happen in consensus protocols that make use of quorum systems, regardless if they use egalitarian or weighted quorums.

G. Summary of Observations

We conclude that AWARE's approach refines the practical utilization of WHEAT in several ways:

1) Ease of Deployment: For deployment, it is irrelevant to choose a good starting configuration because AWARE provides the needed automation for finding an optimal configuration by tuning voting weights and/or relocating the leader.

2) Adjusting to Varying Network Conditions: The quality of communication links may vary for different reasons, e.g., bad routing, overloads or DDoS attacks. This might be less a problem for Amazon's data centers but can occur for server located in poorer connected regions. AWARE dynamically adjusts to new conditions by shifting high voting power to replicas that are the fastest in a recent timeframe.

3) Compensating for Faults: In the worst case f replicas become unavailable. If they all have V_{max} voting power, then all correct replicas need to access the same quorum without any variability. For non-malicious behavior, AWARE detects this and restores the availability of up to $f(V_{max} - V_{min})$ voting power in the system by redistributing high voting weights to the fastest of the remaining correct replicas.

VI. Related Work

A variety of research touches the fields of SMR optimizations in WAN environments [13], [14], [17]–[19] and dynamic approaches for latency awareness [16], [20]–[22].

WAN optimizations. Steward [14] is a hierarchical architecture for scaling BFT replication in WANs. Multiple replication groups (each group is located at a site and runs Byzantine agreement) are geographically distributed and groups are connected over a CFT protocol. Like our approach, it uses additional spare replicas but with much higher replication costs (4 replicas are needed at each site). *Mencius* [17] is an approach for building efficient SMR for WANs by employing a rotating coordinator scheme where clients choose their geographically closest replica as the coordinator. However, *Mencius* only supports the CFT model because of its skipping technique. The idea of a rotating leader in *Mencius* is later enhanced in the *RAM* protocol [18] which additionally employs attested append-only memory and assumes *mutually suspicious domains* to achieve low latency SMR for uncivil WANs. In our work, we assume the BFT model where clients do not trust their local leaders. *EBAWA* [13] is a protocol

that improves SMR in WANs under a hybrid fault model. A trusted component on the replicas allows reducing the number of replicas in the system to $2f + 1$ and the communications steps needed for agreement to 2. It also uses a rotating leader technique where clients send their requests to their local server. In contrast, *Egalitarian Paxos* [19] allows all replicas to propose and employs a mechanism for solving conflicts if operations interfere. Clients choose a well-connected replica to propose their operations. *GeoPaxos* [23] decouples order from execution, utilizes partial order instead of total order and exploits geographic locality to achieve fast geographic SMR, but only tolerates crash faults just like Egalitarian Paxos.

Dynamic approaches. The protocols *Droopy* and *Dripple* [16] follow a dynamic approach to reduce latency for geo-replicated state machines under imbalanced localized workloads. The authors suggest that the choice (and the number) of leaders depends on both replica configuration and workload, thus is subject to variations over time. *Droopy* dynamically reconfigures the leader set of each partition while *Dripple* coordinates state partitions. Further research work shows that clients can also dynamically react to changing workloads by efficiently changing their quorum selections to achieve good performance [22]. A protocol for latency-aware leader selection in geo-replicated systems is *ARCHER* [20], which uses clients' observed end-to-end response latencies to select the optimal leader and hence can dynamically adjust to varying workloads. In contrast, AWARE measures replica-to-replica latencies and uses weight tuning additional to leader selection. We follow the empirical observations of WHEAT, in which a mechanism that makes clients closer to their leaders (or using a leader in the same region) gives less latency gains than just using the fastest replica as the leader [7]. Dynamic adaptation of consensus algorithms for BFT systems is being studied in latest research work [21] by the implementation of switching algorithms in the BFT-SMaRt library and evaluation of different techniques for a multi-datacenter (WAN) setting.

VII. Conclusion

World-spanning Byzantine consensus systems can benefit from dynamic self-optimizing approaches. We showed how to construct such a dynamic approach using WHEAT as underlying weighting scheme and BFT-SMaRt as replication protocol by letting the system perform continuous measurements and decide on an optimal configuration. Further, we described a deterministic, self-optimization algorithm that enables the system to minimize its consensus latency and thus to faster respond to clients.

AWARE enriches the idea of weighted replication by providing the needed automation to adapt to changing environmental conditions. Our method implements resilient, *adaptive* Byzantine consensus, in particular it automates voting weight tuning and leader positioning, hence thriving for latency gains at run-time by selecting a fast performing WHEAT configuration. Evaluation results show that the potential for latency and throughput gains is substantial. Specifically, the best configuration performs about 38.7% faster on average in terms of observed latency across clients' sites than the median.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[2] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *OSDI*, 1999, pp. 173–186.

[3] H. Moniz, N. F. Neves, M. Correia, and P. Verissimo, "RITAS: services for randomized intrusion tolerance," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 1, pp. 122–136, 2011.

[4] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *44th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), 2014*, 2014, pp. 355–362.

[5] J. Sousa, A. Bessani, and M. Vukolic, "A Byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," in *48th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 51–58.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger Fabric: a distributed operating system for permissioned blockchains," in *Proc. of the 13th EuroSys Conf.* ACM, 2018, p. 30.

[7] J. Sousa and A. Bessani, "Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines," in *34th IEEE Symp. on Reliable Distributed Systems (SRDS)*. IEEE, 2015, pp. 146–155.

[8] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[9] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM TOPLAS*, vol. 4, no. 3, pp. 382–401, 1982.

[10] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proc.of the ACM SIGOPS 22nd Symp. on Operating Systems Principles*. ACM, 2009, pp. 277–290.

[11] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *9th European Dependable Computing Conf. (EDCC), 2012*. IEEE, 2012, pp. 37–48.

[12] C. Cachin, "Yet another visit to Paxos," *IBM Research, Zurich, Switzerland, Tech. Rep. RZ3754*, 2009.

[13] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "EBAWA: Efficient Byzantine agreement for wide-area networks," in *12th IEEE Int. Symp. on High Assurance Syst. Eng.*, Nov 2010, pp. 10–19.

[14] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling Byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, Jan 2010.

[15] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed computing*, vol. 11, no. 4, pp. 203–213, 1998.

[16] S. Liu and M. Vukolić, "Leader set selection for low-latency geo-replicated state machine," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1933–1946, 2017.

[17] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation*, 2008, pp. 369–384.

[18] Y. Mao, F. Junqueira, and K. Marzullo, "Towards low latency state machine replication for uncivil wide-area networks," in *Workshop on Hot Topics in System Dependability*, 2009.

[19] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of the Twenty-Fourth ACM Symp. on Operating Systems Principles*, 2013, pp. 358–372.

[20] M. Eischer and T. Distler, "Latency-aware leader selection for geo-replicated Byzantine fault-tolerant systems," in *1st Workshop on Byzantine Consensus and Resilient Blockchains (BCRB '18)*, 2018, pp. 140–145.

[21] C. Carvalho, D. Porto, L. Rodrigues, M. Bravo, and A. Bessani, "Dynamic adaptation of Byzantine consensus protocols," in *Proc. of the 33rd Annu. ACM Symp. on Applied Computing*, 2018, pp. 411–418.

[22] M. G. Merideth, F. Oprea, and M. K. Reiter, "When and how to change quorums on wide area networks," in *28th IEEE Int. Symp. on Reliable Distributed Systems*, Sep. 2009, pp. 12–21.

[23] P. Coelho and F. Pedone, "Geographic state machine replication," in *37th IEEE Symp. on Reliable Distributed Systems (SRDS)*, Oct 2018, pp. 221–230.