AWARE: Adaptive Wide-Area Replication for Fast and Resilient Byzantine Consensus

Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani

Abstract—With upcoming blockchain infrastructures, world-spanning Byzantine consensus is getting practical and necessary. In geographically distributed systems, the pace at which consensus is achieved is limited by the heterogeneous latencies of connections between replicas. If deployed on a wide-area network, consensus-based systems benefit from weighted replication, an approach that utilizes extra replicas and assigns higher voting weights to well-connected replicas. This approach enables more choice in quorum formation and replicas can leverage proportionally smaller quorums to advance, thus decreasing consensus latency. However, the system needs a solution to autonomously adjust to its environment if network conditions change or faults occur. We present Adaptive Wide-Area REplication (AWARE), a mechanism that improves the geographical scalability of consensus with nodes being widely spread across the world. Essentially, AWARE is an automated and dynamic voting-weight tuning and leader positioning scheme, which supports the emergence of fast quorums in the system. It employs a reliable self-monitoring process and provides a prediction model seeking to minimize the system's consensus latency. In experiments using several AWS EC2 regions, AWARE dynamically optimizes consensus latency by self-reliantly finding a fast configuration yielding latency gains observed by clients located across the globe.

Index Terms—adaptivness, weighted replication, consensus, geo-replication, Byzantine fault tolerance, self-optimization, blockchain

1 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]. Furthermore, SMR protocols typically use either some dynamically selected leader [2] or are fully decentralized [3]. In both cases, these 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 geographic decentralization.

- *C. Berger and H. P. Reiser are with Universität Passau, Germany. E-mail: {cb,hr}@sec.uni-passau.de*
- *J. Sousa and A. Bessani are with LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal. E-mail: {anbessani,jcsousa}@fc.ul.pt*

Manuscript received April 19, 2005; revised August 26, 2015. Corresponding author: Christian Berger. For information on obtaining reprints of this article, please send an e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. X

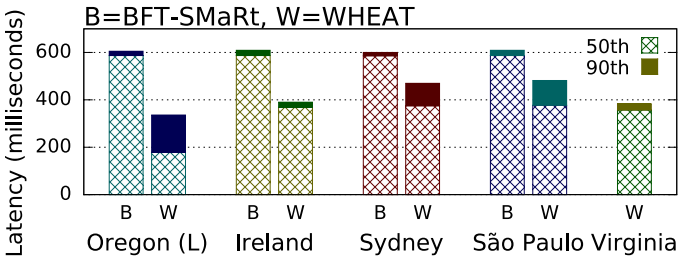


Figure 1: Latency gain using weighted replication in WHEAT and BFT-SMaRt [7], as measured by clients co-located with different replicas.

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 that forms a quorum, i.e., contains enough replicas to convince a replica to advance to the next protocol stage. By introducing WHEAT [7] (WeigHt-Enabled Active replicaTion), 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, and thus the optimal configuration may also change at run-time. To make weighted repli-

cation practically usable, the SMR system needs a mechanism for automated and dynamic optimization. We present AWARE (Adaptive Wide-Area REplication), a mechanism for allowing geo-replicated state machines to self-optimize at run-time (see Figure 2): our method employs continuous self-measurements of the replicas' communication link latencies and analyzes the leader's expected consensus latency for different 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.

Key constraints for SMR in blockchains. We want to tackle important challenges for SMR when used in blockchain systems, namely *scalability* with respect to the number of nodes [9], *geographical dispersion* as well as *adaptivity* towards the environment. This paper extends our conference paper [10] by exploring how the scalability of the AWARE approach can be improved for larger systems. It also shows how AWARE can work as a resilient and adaptive ordering service for Hyperledger Fabric.

1.1 Challenges and Contribution

We address practical challenges that arise when incorporating continuous 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. Equipping a BFT system with such a self-optimizing mechanism introduces a set of challenges and research questions. In particular, our paper addresses the following problems:

- *Self-monitoring in an SMR system.* Defining suitable strategies for self-monitoring, including the question of how the system can 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 high voting weights of unavailable (e.g., crashed) replicas and which threats impose malicious replicas?
- *Efficient self-adaption in larger systems.* How can we address a fast-growing configuration space (i.e., the problem of assigning weights) for larger systems?

1.2 Outline

We start by explaining relevant preliminary work (§2) and the system model we employ (§3). Then, we present AWARE's monitoring methodology (§4) and subsequently describe our algorithm for finding an optimal configuration and deciding on a reconfiguration (§5). Moreover, we conduct an experimental evaluation of AWARE using different Amazon EC2 regions (§6). This paper is the extended version of [10] and augments it by explaining how AWARE works in larger systems (§7) and how it is used as a self-adapting BFT ordering service for the Hyperledger blockchain platform (§8). Finally, we discuss related research work (§9) and draw conclusions (§10).

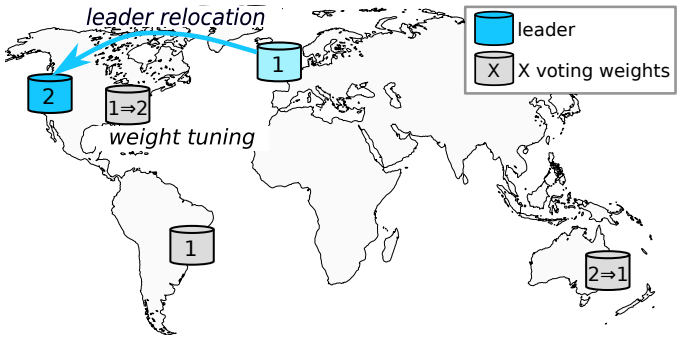


Figure 2: Optimizing a WHEAT configuration at run-time.

2 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 weakly synchronous 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 arbitrarily without compromising the service. The Byzantine fault model [11] usually requires a protocol to use a total of $n = 3f + 1$ replicas to guarantee both liveness and safety under the partially synchronous system model.

2.1 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 [12] 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 aware design and various optimization techniques it incorporates. Furthermore, it can be configured to run in crash fault tolerance (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).

The *programming model* of BFT-SMaRt assumes the following request-response interaction model: Clients call the *invokeOrdered(request)* interface to broadcast *requests* to all replicas. The replicated server implements the deterministic *executeOrdered(request)* interface, to which requests are delivered in *total order*, thus ensuring that correct replicas process requests in the same sequence, and therefore also apply the same sequence of state transitions. The client then waits for a specific quorum of matching replies. Replicas use a consensus algorithm to achieve the total order among all clients' requests: they decide the batch of requests to be executed next in every consensus instance. BFT-SMaRt uses

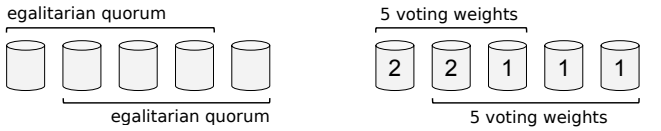
the Mod-SMaRt protocol [13], an algorithm for SMR that employs an underlying leader-driven consensus primitive (the consensus algorithm described by Cachin [14]).

This Byzantine consensus algorithm consists of three phases [14]: *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., $|Q_i \cap Q_j| \geq f + 1$.

2.2 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], [12], [13], [15], [16]. 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 [17]. To ensure quorum formation, BFT systems typically probe a Byzantine majority of replicas, as depicted in Figure 3a. 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 relying on weighted replication, as depicted in Figure 3b. In this case, by probing a majority of voting weights 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.



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

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 voting weights, 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].

2.3 Practical Implications

Proportionally smaller quorums in WHEAT. First, we would like to observe that the gains obtained by increasing n and not f in typical quorum-based protocols are not as good as in our approach. For example, the typical f-dissemination Byzantine quorum systems [17] employed in most BFT protocols require quorums of $q = \lceil \frac{n+f+1}{2} \rceil$ replicas (for any f and $n > 3f$), while in WHEAT/AWARE, the safe minimal quorum will always have $2f + 1$ replicas, which is strictly smaller than q for all $\Delta > 0$.

Resilience Performance Tradeoff. Second, the possibility to use additional replicas either to increase the f bound of the system or Δ, leads to a tradeoff situation between resilience and performance. We consider f a parameter of the system. Given a fixed value of n, e.g., the number of organizations running consensus nodes for a permissioned blockchain, we can trade off some resilience for smaller fast quorums, which may lead to better performance.

To illustrate this on an example, let's imagine a $n = 13$ system: we could use the most resilient ($\Delta = 0$, $f = 4$) or a slightly less resilient but possibly better performing

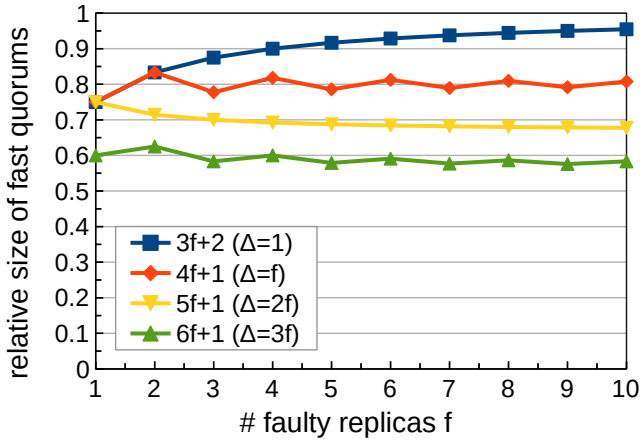


Figure 4: Relative size of fast quorums in proportion to the size of traditional Byzantine quorums for different system configurations.

($\Delta = 3$, $f = 3$) configuration. A configuration with higher Δ supports the emergence of fast quorums in the system.

One may decide to choose the value of Δ depending on f. E.g., for $\Delta = f$, the proportion of faulty replicas that can be tolerated changes from roughly $\frac{1}{3}$ to $\frac{1}{4}$ of n for larger n. An interesting aspect of this is to have the values of Δ and f be changed at run-time by an administrator, e.g., if changes in the threat level of the system are perceived. Figure 4 illustrates the proportion of the relative size of the small quorums consisting of the $2fV_{max}+1$ voting weights cast by $2f+1$ replicas to the traditional Byzantine quorums of size $\lceil \frac{n+f+1}{2} \rceil$ replicas for different system configurations.

3 System Model

We use the same system model as BFT-SMaRt and WHEAT, which we briefly summarize in this section.

Fault model. The system consists of a replica set I with a total of $n = 3f + 1 + \Delta$ replicas and an arbitrary number of clients. The number of faulty replicas is bounded by f, and we employ the Byzantine fault model for the behavior of faulty replicas. We guarantee that the safety of the system is never violated, while we require additional weak synchrony assumptions to guarantee liveness.

Communication model. To guarantee liveness, we employ the partially synchronous system model [18], which is also popular with many other consensus-based systems, e.g., *Steward* [16], *HotStuff* [19] (a variant of which, *LibraBFT* [20], is employed in Facebook's *Libra* blockchain) and *Stellar* [21]. The system can behave initially asynchronous, but we assume some upper bound δ on communication delay eventually exists after some unknown *global stabilization time* (GST). Once GST is reached, we can guarantee that the system makes progress. Note that we can always guarantee that the system stays safe – even during periods of asynchrony. Communication is point-to-point, authenticated and reliable. The implementation ensures this by using message authentication codes (MACs) over TCP/IP. The symmetric secrets for the replica-replica channels are established through *Signed Diffie-Hellman* using an RSA key pair (SK_i, PK_i) per replica.

Adversary model. The adversary has complete control over a set of replicas B with $|B| \leq f$, where the adversary can freely choose replicas from the replica set I. For each $b \in B$, we consider the behavior of b as *arbitrary*, but still bound by computational capabilities, e.g., the attacker cannot have a malicious replica break strong cryptographic primitives (such as to circumvent authentication). For instance, the attacker can let b disregard incoming messages or inject messages into the network.

In a worst-case scenario, an attacker chooses f replicas with V_{max} voting weights each, thus possessing in total fV_{max} voting power. Even then, because of WHEAT's *safe* voting scheme, there exists a fallback quorum of remaining $2f+1+\Delta$ replicas that have a majority of $fV_{max}+(f+1+\Delta)$ voting power. Under the assumption that replicas of the fallback quorum are correct, connected, and messages arrive within δ, the system makes progress. Regardless of the last assumption, the system always remains safe.

We further assume that the attacker has the power to control and possibly disrupt the whole network but not for an indefinitely long time span and, as soon as sufficiently enough synchrony is reached (this point in time is modeled by GST), the system is guaranteed to make progress.

4 Monitoring Strategy

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

4.1 Monitoring BFT Consensus

Problem. A naive approach for monitoring the consensus pattern would be to observe the sending and reception time of WRITE and ACCEPT messages for all replicas. Such *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 5. 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 weights.

WRITE and ACCEPT. To prevent this, we favor one-sided measurements, which require only *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*. Let T'_4 denote the arrival of the response message back at the measuring replica. Then, the measuring replica can use $(T'_4 - T_1)/2$ as

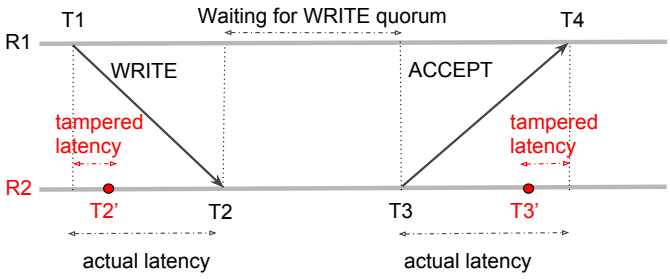


Figure 5: Problem with timestamps and Byzantine replicas.

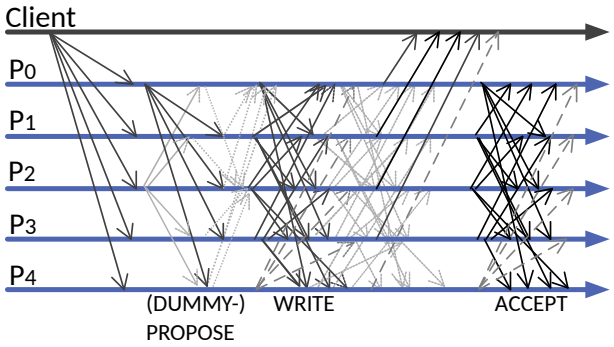


Figure 6: Message flow of AWARE ($f = 1; \Delta = 1$).

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. Proposals 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 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.

4.2 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 that 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 to ensure symmetric behavior on communication link latency. Using the *DUMMY-PROPOSE* is optional as it introduces overhead to the system (see §6.5) and it is also possible to approximate these latencies using the measurements of *WRITE-RESPONSE*.

Figure 6 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 (ordered together with the client requests) 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 §5.3 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 §6.5).

1. We do not need to use an additional *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].

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

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

Figure 7: Sanitized matrix of median *WRITE* latencies.

4.3 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

$$M^x[i,j] \leftarrow \begin{cases} +\infty, & \text{if } i \neq j \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

for $x \in \{P, W\}$. $M^x[i,j]$ expresses the latency of replica i to j as measured by i for message type x (either *RPOPOSE* or *WRITE*). Further, replica i can update a row of these matrices with its measurements L_i^P and L_i^W by using the *invoke* interface of BFT-SMaRt:

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

With this interface of BFT-SMaRt, we create a global total order on all client requests *and* measurement messages. The updating process yields a matrix M^x, with $M^x[i,j] = L_i^x[j]$ 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

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

for $x \in \{P, W\}$. $\hat{M}^P$ and $\hat{M}^W$ are then the sanitized latency matrices for *PROPOSE* and *WRITE* respectively. Figure 7 presents an example for sanitization. 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 show intriguing behavior, e.g., by claiming to have tremendous connections to each other (see §6.6).

5 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*.

5.1 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], [22]. 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.

Further, AWARE optimizations operate at the granularity of logical protocol rounds that successfully lead to an agreement, called *consensus instances*. Reconfigurations are evaluated at most once every *calculation interval*[3] c of consensus instances is reached. This is to ensure the stability of our approach by not performing reconfigurations too frequently.

5.2 Consolidated Protocol

We ensure determinism by providing a deterministic consensus latency prediction function used for our calculation (see Algorithm 2). 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. Replicas apply a change to the *current view* object to update the weight distribution and leader after a calculation that is performed in intervals specified by logical protocol rounds (consensus instances) 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 $\textit{invokeOrdered}(\mathsf{MEASURE}, L_i^P, L_i^W)$;
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 that 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

3. The calculation interval determines the frequency of optimizations and can be configured in our implementation. Our default value is 500.

$M^W[i,j] = L_i^W[j]$ 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 §4.3), generating $\hat{M}^P$ and $\hat{M}^W$.
6. Now, every replica solves the following optimization problem, where *PredictLatency* (Algorithm 2) is a function for predicting the latency of the consensus protocol using the latencies in $\hat{M}^P, \hat{M}^W$, and a set of weight distributions $W \in \mathfrak{W}$ and permitted leaders $l \in \mathfrak{L}$:

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

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.

5.3 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], [13], 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 voting weights as well, so it needs to be well connected to replicas with preferably high voting weights. The leader itself is always assigned V_{max} voting weights in order to minimize its consensus latency.

Algorithm 1 computes for each replica the time it can proceed to a subsequent protocol stage by forming a weighted quorum of $Q_v = 2V_{max} + 1$ voting weights given the time a replica starts in its current stage $T_{i \in I}^{current}$ and latency matrix $\hat{M}^W$. In particular, the algorithm computes the time a replica receives the *WRITE* messages from all others (lines 1–4) and then computes for every replica the time it has gathered enough voting weights Q_v to, e.g., proceed from *WRITE* to *ACCEPT* phase (lines 5–10). This works by letting each replica pull from a priority queue, in which incoming messages are sorted by ascending arrival time, and accumulate the voting weights until Q_v is reached. The arrival time of the last message, necessary to reach this quorum, determines the time a replica can proceed to the next protocol stage.

Algorithm 2 is used to predict the leader's consensus latency for a given configuration by simulating the consensus protocol run. It first computes the times each replica receives the leader's proposal (line 3–4), then it uses Algorithm 1 as building block to compute the times replicas complete the *WRITE* (line 5) and *ACCEPT* (line 6) stage, respectively. Further, this 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 8) 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 i is used in the quorum to reach Q_v voting weights, meaning the message of i is used as the *quorum formation speed determining* vote. We consider this in our calculations for a sequence of rounds by 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 mul-

Algorithm 1: *formQV* computes the times replicas form weighted quorums of $Q_v = 2fV_{max}+1$ voting weights.

Data: replica set I, latency matrix $\hat{M}^W$, times $(T_i^{current})_{i \in I}$ and voting weights $(V_i)_{i \in I}$
Result: times $(T_i^{next})_{i \in I}$ replicas can advance to the next protocol stage

1 **for** $i \in I$ **do**
2 $received_i \leftarrow$ new PriorityQueue()
3 **for** $j \in I$ **do**
4 $received_i.add(\langle T_j^{current} + \hat{M}^W[j,i], V_j \rangle)$
5 **for** $i \in I$ **do**
6 $weight \leftarrow 0$
7 **while** $weight < Q_v$ **do**
8 $\langle T_{next}, V_{next} \rangle \leftarrow received_i.dequeue()$
9 $weight \leftarrow weight + V_{next}$
10 $T_i^{next} \leftarrow T_{next}$
11 **return** $(T_i^{next})_{i \in I}$

Algorithm 2: *PredictLatency* computes the consensus latency (amortized over multiple rounds)

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* $\hat{M}^W$, consensus rounds r
Result: consensus latency of the AWARE leader

1 $V_{max} \leftarrow 1 + \frac{\Delta}{f}$ $V_{min} \leftarrow 1$ $V_i \leftarrow \begin{cases} V_{max}, & \text{if } i \in R_{max} \\ V_{min}, & \text{otherwise} \end{cases}$
2 $\forall i \in I: \ offset_i \leftarrow 0$
3 **while** $r > 0$ **do**
4 **for** $i \in I$ **do**
5 $T_i^{PROPOSED} \leftarrow max(\hat{M}^P[p,i], offset_i)$
6 $(T_i^{WRITTEN})_{i \in I} \leftarrow$ $formQV(I, \hat{M}^W, (T_i^{PROPOSED})_{i \in I}, (V_i)_{i \in I})$
7 $(T_i^{ACCEPTED})_{i \in I} \leftarrow$ $formQV(I, \hat{M}^W, (T_i^{WRITTEN})_{i \in I}, (V_i)_{i \in I})$
8 **for** $i \in I$ **do**
9 $offset_i \leftarrow T_i^{ACCEPTED} - T_p^{ACCEPTED}$
10 $consensusLatencies_r \leftarrow T_p^{ACCEPTED}$
11 $r \leftarrow r - 1$
12 **return** average of $consensusLatencies$

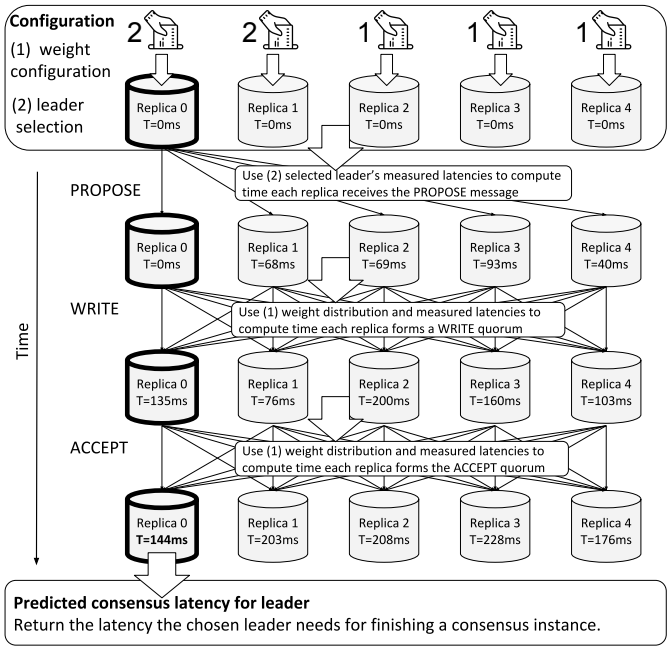


Figure 8: Computing the latency of a WHEAT consensus (here: $f = 1, \Delta = 1$) for a given configuration.

tiplied with the number of possible leaders (V_{max} replicas):

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

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* to determine the goodness of a found solution. Since we want AWARE to be practical in systems of larger sizes, we explain, implement, and validate such an approach in Section 7.

6 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 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 the proposal 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.

6.1 Margin of Latency Variations of 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 9 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 $V_{max} = 2$ voting weights. Each number corresponds to a region as explained in the setup and Figure 9. We notice a big difference between the configurations. The best configuration is $\langle 4, 0 \rangle$ showing a latency (avg. across all clients) of 360 ms, the left median configuration $\langle 3, 4 \rangle$ performs in 499 ms and the worst configuration $\langle 2, 1 \rangle$ requires 590 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, co-locating the leader with the client does not necessarily imply a Pareto-optimum: a client in Sydney observes 492 ms in $\langle 2, 1 \rangle$ (client co-located with the leader), while it achieves its best results (among all configurations) in $\langle 0, 4 \rangle$ with 403 ms.

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

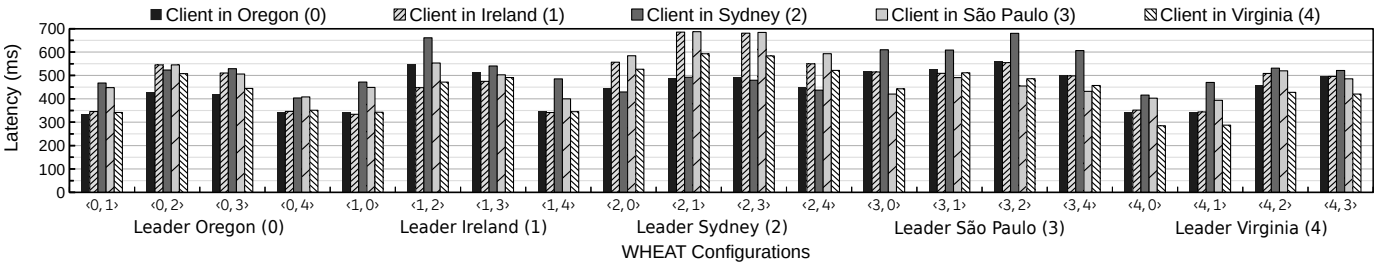


Figure 9: Measured average request latency of 11th to 90th percentile across clients in different regions.

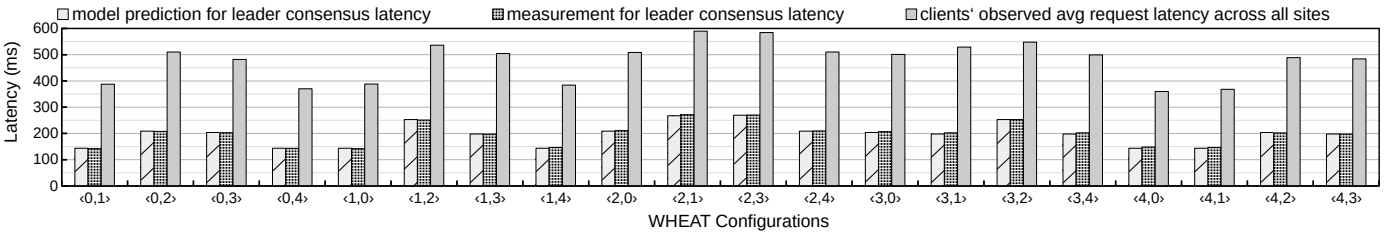


Figure 10: Comparison between predicted consensus latency, measured consensus latency and clients' request latency.

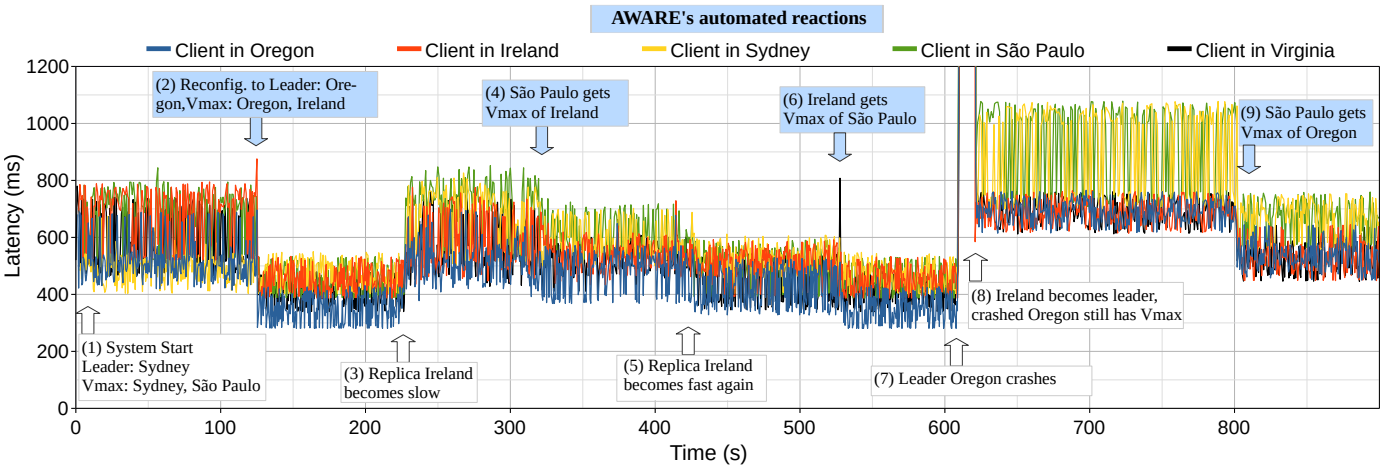


Figure 11: Runtime behavior of AWARE.

6.2 Accuracy of Consensus Latency Prediction

Our approach aims at finding a configuration with minimal leader consensus latency. Our prediction model (Algorithm 2) 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 10). 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 that are subject to smaller 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 143.5 ms amortized over 1000 consensus rounds. In our experiment, the measured latencies for these *optimal candidates* are between 141 ms $(\langle 1,0\rangle)$ and 148 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 ms $(\langle 3,4\rangle)$ and the worst is 271 ms $(\langle 2,1\rangle)$.

6.3 Clients' Observed Request Latency

Figure 10 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*.

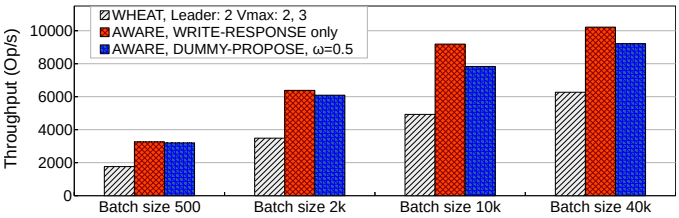


Figure 12: Maximum throughput comparison.

6.4 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. We induce events to evaluate AWARE's reactions (see Figure 11) to certain conditions:

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 weights.
② Reaction: After a calculation interval of $c = 500$ consensus instances, 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 at 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 weights become 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.

6.5 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 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 monitoring overhead with $\omega = 0.5$, hence, only every second consensus instance a replica, chosen by the rotation scheme, broadcasts a *DUMMY-PROPOSE* if it's not the leader).

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. 12) 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.

6.6 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 weights 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 weights, AWARE redistributes them to faster replicas (see Figure 11, 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 weights 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 weights and force correct replicas to use the only remaining fallback quorum of $fV_{max} + (f + 1 + \Delta)V_{min}$ voting weights 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.

6.7 Summary of Observations

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

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 relocating the leader.

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 servers located in poorer connected regions. AWARE dynamically adjusts to new conditions by shifting high voting weights to replicas that are the fastest in a recent timeframe.

Compensating for Faults. In the worst case, f replicas become unavailable. If they all have V_{max} voting weights, 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 weights in the system by redistributing high voting weights to the fastest of the remaining correct replicas.

7 AWARE in Larger Systems

In this section, we want to (1) explain how AWARE can be employed in larger system environments, which includes dealing with an exponentially growing configuration space, and (2) conduct experiments with several, widely-spread replicas to show that adaptive weighted replication is a suitable approach for geographically-scalable BFT consensus. In Section 5.3 we stated that the fast-growing configuration space of possible weight distributions quickly makes exhaustive search impossible. Exhaustive search predicts the latency for all possible configurations. The problem is that the number of weight configuration grows fast because these are essentially combinations for selecting $2f$ replicas with weight V_{max} out of a total of $3f + 1 + \Delta$ replicas in the system. Figure 13a shows the growing number of configurations for exemplary system sizes depicted in Table 1.

7.1 Simulated Annealing

To address this problem, we make use of a heuristic to efficiently traverse the search space. *Simulated annealing* [23] is a technique for finding an approximation of the global optimum of some function (see Algorithm 3). It is essentially a local search that aims for improving on a current solution c by randomly modifying it slightly (which is called neighbor picking), yielding a new configuration c'. If the new configuration c' improves the function output (in our case has a lower predicted latency), then the simulated annealing algorithm uses c' to proceed with its search. However, if c' worsens the function output, it can still be used to proceed with some acceptance probability which depends on the current *temperature* (a monotonously decreasing value) and the energy of the solution, which is essentially the difference between the predicted latency of c' and c. By accepting a temporary worsening, simulated annealing can *jump*, that is, escape from a local optimum to find the global optimum. However, these jumps become less frequently with decreasing temperature. The exit condition for this kind of search can be implemented, e.g., by using a temperature threshold. This guarantees termination if the temperature decreases at a fixed rate. Since simulated annealing is a probabilistic algorithm, it needs to generate a sequence of random numbers during its execution. Because we want to guarantee that replicas deterministically find a consistent solution, we need to employ a pseudo random number generator *PRNG(s)* that generates the same sequence of random numbers in all replicas given the same input seed s. We let replicas use the consensus id (replicas decide on a possible reconfiguration at specific consensus instances) as seed for generating these numbers. For simulated annealing, we use the following parameters: $t_0 = 120$, $\theta = 0.0055$, $threshold = 0.2$ (which were empirically chosen). With these parameters, the algorithm terminates after probing 1160 configurations.

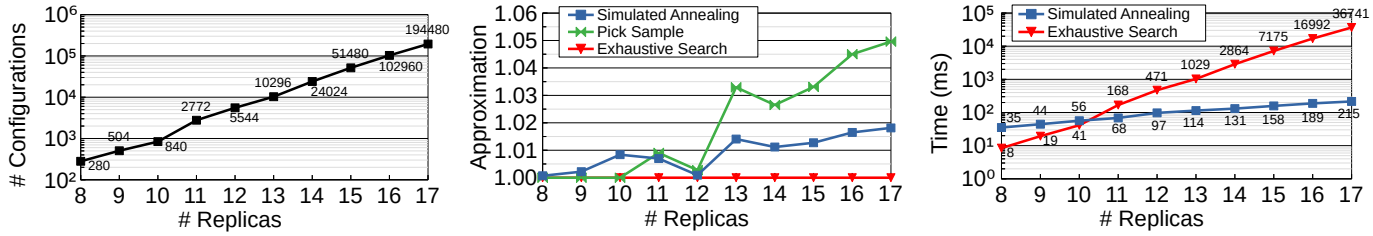
Algorithm 3: *SimulatedAnnealing* is a heuristic for efficiently traversing the search space of configs C

Data: replica set I, system sizes n, f, u, Δ, latency matrices for *PROPOSE* $\hat{M}^P$ and *WRITE* $\hat{M}^W$, consensus id cid, start temperature t_0, cooling rate θ, temperature $threshold$
Result: best (approx.) performing configuration found

$c \leftarrow$ some $c_0 \in C$
$c.prediction \leftarrow predictLatency(I, c, \hat{M}^P, \hat{M}^W, n, f, \Delta)$;
$c_{approx} \leftarrow c$
$temp \leftarrow t_0$
$random \leftarrow$ new Random(cid)
while $temp > threshold$ **do**
  /* Assign a V_{max} to another replica */
  $replicaFrom \leftarrow c.R_{max}[random.nextInt(u)]$
  $replicaTo \leftarrow c.R_{min}[random.nextInt(n - u)]$
  $c' \leftarrow c.swap(replicaFrom, replicaTo)$
  if *replicaFrom is leader* **then**
    $c'.setLeader(replicaTo)$
  $c'.prediction \leftarrow predictLatency(I, c', \hat{M}^P, \hat{M}^W, n, f, \Delta)$;
  /* If new solution is better, accept it */
  if *c'.prediction < c.prediction* **then**
    $c \leftarrow c'$
  else
    /* Compute an acceptance probability */
    $rand \leftarrow random.nextDouble()$
    if $exp(\frac{-(c'.prediction - c.prediction)}{temp}) > rand$ **then**
      $c \leftarrow c'$
  if $c'.prediction < c_{approx}.prediction$ **then**
    $c_{approx} \leftarrow c'$
  /* Cool down the system */
  $temp \leftarrow temp \cdot (1 - \theta)$
return c_{approx}

n	8	9	10	11	12	13	14	15	16	17
f	2	2	2	3	3	3	4	4	4	4
Δ	1	2	3	1	2	3	1	2	3	4

Table 1: Exemplary system sizes.

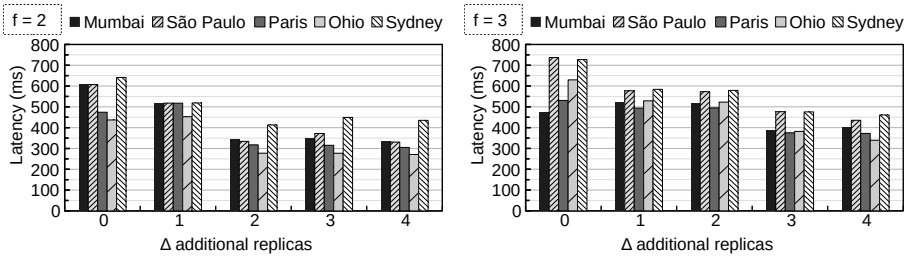


(a) # Configurations. (b) Approximation quality. (c) Computation time.

Figure 13: A heuristic can help to efficiently traverse the configuration space to find a good solution.

	initial replica set	Δ-replicas
$f = 2$	Mumbai, São Paulo, Paris, Ohio, Sydney, Tokio, Canada	Virginia, Ireland, Stockholm, Frankfurt
$f = 3$	Mumbai, São Paulo, Paris, Ohio, Sydney, Tokio, Canada, Bahrain, Stockholm, Seoul	Frankfurt, Virginia, Ireland, Oregon

(a) Test environments.



(b) $f = 2$ configuration. (c) $f = 3$ configuration.

Figure 14: Latency results of AWARE experimentally tested in larger-scale setups.

7.2 Approximation Quality

We validate the quality of our heuristic *Simulated Annealing* (SA) (using above parameters) by comparing the predicted latency of the approximate solution it finds to the optimum found by *Exhaustive Search* (ES) and a third strategy, *Pick-Sample*, which samples through the configuration space in equal steps and probes the same number of configurations as SA does. We compute the average deviation from the optimum by simulating a set of 1000 randomly generated setups for each system size n. Figure 13b shows that the predicted latencies of solutions found by SA are on average less than 1.02 higher than the optimum found by ES. It can be observed that SA performs better than naive sampling as solutions found by *PickSample* display a higher derivation from the optimum (up to 1.05).

7.3 Computation Time

Further, we evaluate the average time a replica needs to find a solution on an Intel i7-4790 @3.60 GHz processor using 1000 randomly generated setups for simulation for each system size n. Figure 13c compares the time needed to find a solution between two strategies, SA (with the above specified parameters) and ES for different system sizes (see Table 1). As expected the time needed by ES grows exponentially for an increasing n. SA only computes through a constant number of configurations, however, the time still increases as the *PredictLatency* function needs more time to predict the latency of larger systems, since it essentially is a simulation of the consensus protocol run. To be precise, the time complexity of *PredictLatency* is $\mathcal{O}(n^2 log(n))$ because every replica maintains a priority queue of messages (ordered by ascending arrival time) as a min heap, which takes $\mathcal{O}(nlog(n))$ time to construct, and we need to consider each of the n replicas during the simulation. The constant number of configurations that SA examines depends on the chosen parameters for start temperature, temperature threshold, and the cooling schedule. In total, the time complexity of SA is in $\mathcal{O}(n^2 log(n))$.

7.4 Experiments on AWS

In the next step, we conduct experiments on the Amazon AWS cloud infrastructure to evaluate how adaptive weighted replication behaves for systems with larger replica sizes than the $f = 1, n = 5$ setup, which we evaluated in Section 6. In particular, we want to investigate if for a fixed size of f, adding additional spare replicas (called Δ-replicas) to an initial replica set can lead to latency gains observed by clients spread across the world.

7.4.1 Setup

We use the Amazon AWS infrastructure and the same overall setup as in Section 6 to do our measurements but add more regions to place our replicas (see Table 14a). Here, for both $f = 2$ and $f = 3$ we start with an initial replica set and then repeat the experiment with increased system sizes where the additionally added spare replicas serve as Δ-replicas used to improve the consensus latency of the system. These replicas are added to the system configuration in the order in which they appear in Table 14a. Replicas use AWARE to self-optimize. Even for a $\Delta = 0$ configuration, where all replicas have equal weight, AWARE still automates the selection of the leader position, hence choosing the best leader for the given environment. AWARE is configured to self-optimize after every 500 consensus instances. Moreover, we locate 5 clients in different regions across the globe, namely in *Mumbai*, *São Paulo*, *Paris*, *Ohio*, and *Sydney*. Each system environment is tested by clients first sending 500 requests for a warm-up phase to make sure AWARE has adapted to a fast-performing configuration.

Then, clients measure request latency as the median of 1000 observed requests which they simultaneously send.

7.4.2 Observations

Figure 14b and Figure 14c show the results for the $f = 2$ experiments with system sizes between 7 and 11 replicas and the $f = 3$ experiments with system sizes between 10 and 14 replicas, respectively. For the $f = 2$ experiments we observe that after adding *Virgina* as additional replica, the average of request latencies over all clients improves from $553\ ms$ to $505\ ms$, and further adding *Ireland* to the system results in an average of $337\ ms$. Adding even more replicas ($\Delta = 3$ or $\Delta = 4$) does not yield further substantial gains. Moreover, for the $f = 3$ experiments we observe that after adding *Frankfurt*, the average latency of all clients improves from $619\ ms$ to $541\ ms$. Interestingly, the latency that the client in Mumbai observes slightly increases. This is due to the fact that Mumbai is leader in the $\Delta = 0$ configuration, but another replica becomes leader in the $\Delta = 1$ configuration and co-residency with the leader is beneficial for a client. As we add more replicas, the average latency across clients improves further, in particular $537\ ms$ (adding *Virginia*), $419\ ms$ (adding *Ireland*) and $402\ ms$ (adding *Frankfurt*). We conclude that adaptive weighted replication can improve the latency of geographically-scalable BFT consensus, as assigning high voting weights to fast replicas supports the emergence of fast quorums in the system. Fast consensus generally is beneficial for clients, however other factors also exist, e.g., being located near the protocol leader.

8 Integration of AWARE into the Hyperledger Fabric Blockchain Platform

To illustrate a practical use case for the AWARE protocol, we show that it can be employed as a building block for distributed ledger infrastructures. As an example, we implant AWARE into the Hyperledger Fabric (HLF) [6] blockchain platform as a self-adapting ordering service. Specifically, its task is to repetitively achieve agreement on which block is appended next to the blockchain. Our protocol is particularly tailored as consensus substrate for blockchain infrastructures that (1) are geographically decentralized with ordering nodes being spread across different regions in the wide-area network, (2) adopt Byzantine fault-tolerance, and (3) want to achieve adaptiveness to their environments.

8.1 Hyplerledger Fabric

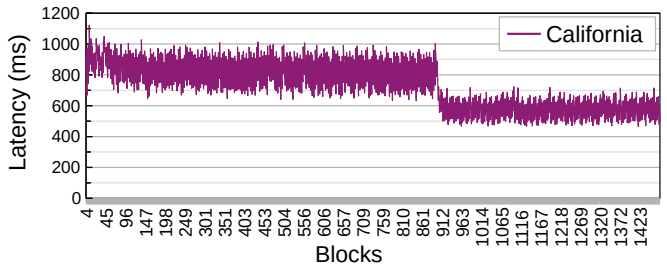
Hyperledger Fabric [6] is a modular and extensible open-source blockchain platform that assumes the permissioned blockchain model. Its core innovation is to provide an abstraction and separation of different concerns, which manifest in distinct building blocks. The *ledger* is a totally ordered, append-only blockchain maintained by the *endorsing peers* that execute transactions, thus generating *endorsements* (result of an execution against the current state, specifically the read and write sets). Clients verify and assemble endorsements into signed *envelopes*, which contain the endorsing peers' read and write sets, and submit these envelopes to a separate ordering service. The stateless *ordering service* is used to atomically broadcast blocks to endorsement peers. This service creates a total order of transactions by running consensus among the *ordering nodes*, which create a sequence of blocks of ordered envelopes. Each newly created block is then disseminated to the receiving endorsing peers, which append it to the ledger. In HLF, the notion of membership is flexible: a *membership service provider* manages the mapping of node identities to their public keys. Fabric introduces the *execute-order-validate* paradigm, which separates the transaction flow into (1) confirming the correctness of a transaction, executing it, and outputing an endorsement, (2) using a consensus protocol to order transactions by reaching agreement on the succession of the blocks, and (3) validating a transaction, e.g., against application specific trust assumptions.

8.2 Integration of AWARE into Hyperledger Fabric

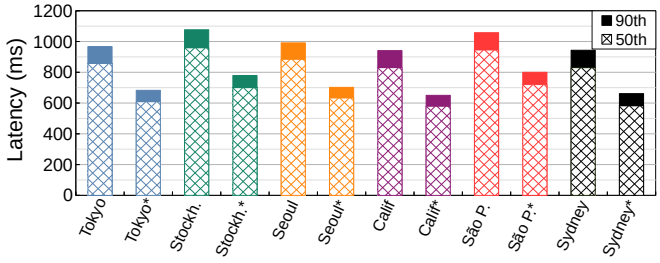
Sousa and Bessani designed and implemented a BFT ordering service for HLF on top of BFT-SMaRt [5]. We use the same service to integrate AWARE, which uses interfaces identical to that of BFT-SMaRt. HLF provides a client interface to submit envelopes to an external ordering service (HLF *consenter*). These are submitted to a Java *frontend*, which consists of a client thread pool, receiver thread, and an asynchronous *BFT Proxy* (which is part of the client-side BFT-SMaRt library), which invokes these envelopes: they are being broadcasted to the ordering nodes and subsequently ordered as they pass through the *total order multicast layer* of BFT-SMaRt. The ordering nodes extend BFT-SMaRt's *ServiceReplica* class. They receive a stream of totally ordered envelopes and use a *blockcutter* to aggregate them into blocks. Created and signed blocks are distributed back to the *BFT Proxy* of all receiving (listening) frontends by utilizing the *Replier* interface of BFT-SMaRt. Once the BFT Proxy has gathered sufficiently many equal and verified messages for a block, the block can be passed to HLF to be appended to the ledger.

8.3 Experimental Setup

In our experiments, we want to evaluate the AWARE ordering service for HLF, in particular, the latencies for ordering envelopes, generating blocks, and receiving them back as observed by frontends located in different AWS regions. We use a BFT $f = 1$ and $\Delta = 1$ configuration and choose the regions *Sydney (leader, V=2)*, *São Paulo (V=2)*, *California (V=1)*, *Tokio (V=1)* and *Stockholm (V=1)* to place a *t2.micro* instance. Each instance runs both an ordering node and a frontend. An extra frontend runs in *Seoul*. Frontends send sufficiently many envelopes (with transaction message payload of size 100 bytes) to satisfy a throughput of at least 100 envelopes/s in the system. The ordering service is configured to create blocks containing 10 envelopes and distribute created blocks to all frontends. Note that a consensus instance may contain multiple blocks. The AWARE ordering nodes use a calculation interval of $c = 250$ after which they may change the configuration to optimize the consensus latency.



(a) Frontend in California.



(b) Latency across clients before and after* optimization.

Figure 15: AWARE as self-adapting ordering service for Hyperledger Fabric leads to latency gains observed by frontends.

8.4 Observations

We show the envelope latencies as observed by all of the frontends in Figure 15b. Roughly at around block #900[4], AWARE reconfigures the system, shifting the maximum voting weights V_{max} from *São Paulo* to *California*. The runtime behavior from *California's* perspective is shown in Figure 15a. This optimization results in a latency improvement observed by all frontends (see Fig. 15b for median and 90th percentile request latencies), e.g., the median latency observed by *Sydney* (where the replica stays leader) decreases from 830 ms to 579 ms, which corresponds to a speedup of 1.43. AWARE predicts the improvement of the consensus latency from 323 ms to 180 ms after the reconfiguration. Note that latency gains that frontends observe are higher than this 143 ms consensus latency decrease. This is because an envelope received by the ordering service might not immediately be proposed by the leader, since the currently running consensus instance needs to be finished first. This results in a random waiting time of the envelope at the leader, whereby the expected waiting time is roughly half of the consensus latency.

9 Related Work

A variety of research touches the fields of SMR optimizations in WAN environments [15], [16], [24], [25], [26], [27] and dynamic approaches for latency awareness [22], [28], [29], [30]. Further, recent research efforts develop and refine BFT consensus protocols for blockchains [19], [21], [31], [32], [33], [34], [35], [36], [37] or researches on their integration into an existing blockchain platform such as Hyperledger Fabric [5], [6], [38].

4. Note that blocks and consensus decisions are different things. The reconfiguration around block 900 is due to the the reconfiguration triggered after 250 consensus instances.

9.1 WAN Optimizations for SMR

Steward [16] 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). *Stewards*'s idea of a hierarchical architecture is also used in the recent *Fireplug* [39] for efficient geo-replication of graph databases by compositing multiple BFT-SMaRt groups. *Mencius* [24] 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 [25], 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* [15] 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 communication 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* [26] 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* [40] 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. Another recent crash-fault tolerant wide-area replication optimization is *Weave* [41], which can give fast execution *guarantees* for client requests by avoiding that clients need to wait for wide-area communication steps. it employs local groups (of size $f + 1$ replicas) that assign sequence numbers to the write requests of their corresponding local clients. Further, it is also possible to assist system integrators with geographic SMR deployments: a recently proposed method [42] ranks possible SMR replica deployments by calculating expected performance from known round trip times between replicas.

9.2 Dynamic Approaches

The protocols *Droopy* and *Dripple* [22] 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 [30]. A protocol for latency-aware leader selection in geo-replicated systems is *ARCHER* [28], 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 [29] by the implementation of switching algorithms in the BFT-SMaRt library and evaluation of different techniques for a multi-datacenter (WAN) setting.

9.3 BFT Consensus in Blockchains

A number of BFT consensus protocols has been developed for blockchains because a *"one size fits all solution"* may be impossible to design [43], [44], [45] which leads to BFT consensus protocols being tailored to specific purposes, as they differ in their ambitions and assumptions [38], [46], [47], [48]. For instance, *Algorand* [33] scales for a large magnitude of nodes and was evaluated with up to 500k nodes while *FastBFT* [36] employs trusted hardware components to achieve low latencies. *HoneyBadgerBFT* [31] tolerates arbitrary network failures, as it adopts asynchronous atomic broadcast. *Gosig* [35] is able to withstand adversarial network conditions on a wide area network. This variety is respected by *Hyperledger Fabric*'s modularity [6] as it encapsulates consensus as distinct building block called the *ordering service*, ideally allowing operators to choose the consensus that fits best for their application. However, in practice, integrating a BFT consensus protocol still requires substantial changes (e.g. for majority voting) thus breaking Fabric's modularity. *Bloxy* [38] tackles this problem: it is a blockchain-aware trusted proxy that encapsulates the client functionality of BFT consensus and thus simplifies and accelerates the integration of BFT consensus protocols into HLF. Furthermore, *Fastfabric* [49] presents a number of design changes for HLF to increase throughput from 3k to 20k transactions per second.

10 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[5] 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 weights tuning and leader positioning, hence thriving for latency gains at run-time by selecting a fast performing WHEAT configuration. Evaluation results with several AWS EC2 regions show that the potential for latency and throughput gains is substantial. For many blockchains seeking a high degree of decentralization, geographical scalability (where many nodes are widely spread across the globe) becomes an important limitation. We demonstrated that AWARE also works with larger system sizes and can be used as a self-adapting consensus substrate for blockchain infrastructures, such as the Hyperledger blockchain platform.

5. The open-source implementation of AWARE is available at https://github.com/bergerch/aware.

Acknowledgments

This work was supported by DFG through project OptSCORE2 (RE 3490/2-2) and by FCT through projects IRCoC (PTDC/EEI-SCR/6970/2014), ThreatAdapt (FCT-FNR/0002/2018), and the LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

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] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *International Workshop on Open Problems in Network Security*. Springer, 2015, pp. 112–125.
[10] C. Berger, H. P. Reiser, J. Sousa, and A. Bessani, "Resilient wide-area Byzantine consensus using adaptive weighted replication," in *Proc. of the 38th IEEE Symp. on Reliable Distributed Systems*, 2019.
[11] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM TOPLAS*, vol. 4, no. 3, pp. 382–401, 1982.
[12] 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.
[13] J. Sousa and A. Bessani, "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation," in *Proc. of the 9th European Dependable Computing Conf. (EDCC)*. IEEE, 2012, pp. 37–48.
[14] C. Cachin, "Yet another visit to Paxos," *IBM Research, Zurich, Switzerland, Tech. Rep. RZ3754*, 2009.
[15] 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.
[16] 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.
[17] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed computing*, vol. 11, no. 4, pp. 203–213, 1998.

[18] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[19] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *Proc. of the 2019 ACM Symp. on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: ACM, 2019, pp. 347–356.

[20] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, "State machine replication in the Libra blockchain," Technical Report. Calibra. [Online]. Available: https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain/2019-09-26.pdf, 2019.

[21] G. Losa, E. Gafni, and D. Mazières, "Stellar consensus by instantiation," in *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[22] 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.

[23] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: https://science.sciencemag.org/content/220/4598/671

[24] 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.

[25] 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.

[26] 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.

[27] W. Wei, H. T. Gao, F. Xu, and Q. Li, "Fast mencius: Mencius with low commit latency," in *Proceedings IEEE INFOCOM*. IEEE, 2013, pp. 881–889.

[28] 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.

[29] 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.

[30] 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.

[31] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 31–42.

[32] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 279–296.

[33] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 51–68.

[34] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 583–598.

[35] P. Li, G. Wang, X. Chen, and W. Xu, "Gosig: Scalable Byzantine consensus on adversarial wide area network for blockchains," *arXiv preprint arXiv:1802.01315*, 2018.

[36] J. Liu, W. Li, G. Karame, and N. Asokan, "Scalable Byzantine consensus via hardware-assisted secret sharing," *IEEE Transactions on Computers*, 2018.

[37] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From Byzantine replication to blockchain: Consensus is only the beginning," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020.

[38] S. Rüsch, K. Bleeke, and R. Kapitza, "Bloxy: Providing transparent and generic BFT-based ordering services for blockchains," in *Proc. of the 38th IEEE Symp. on Reliable Distributed Systems*, 2019.

[39] R. Neiheiser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia, "Fireplug: Efficient and robust geo-replication of graph databases," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1942–1953, 2020.

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

[41] M. Eischer, B. Straßner, and T. Distler, "Low-latency geo-replicated state machines with guaranteed writes," in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–9.

[42] S. Numakura, J. Nakamura, and R. Ohmura, "Evaluation and ranking of replica deployments in geographic state machine replication," in *38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE, 2019, pp. 37–42.

[43] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT protocols under fire." in *NSDI*, vol. 8, 2008, pp. 189–204.

[44] M. Vukolić, "Rethinking permissioned blockchains," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. ACM, 2017, pp. 3–7.

[45] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous BFT made practical," in *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2028–2041.

[46] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," *arXiv preprint arXiv:1707.01873*, 2017.

[47] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," *arXiv preprint arXiv:1711.03936*, 2017.

[48] C. Berger and H. P. Reiser, "Scaling Byzantine consensus: A broad analysis," in *Proc. of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2018.

[49] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2019, pp. 455–463.

Christian Berger received the BSc and MSc degrees in computer science from the University of Passau, Germany, in 2014 and 2017, respectively. In 2018, he started working as a research associate and PhD candidate at the assistant professorship of security in information systems. His research interests include Byzantine fault tolerance, resilient distributed systems as well as blockchain and distributed ledger technology.

Hans P. Reiser is an assistant professor at the University of Passau, Germany, associated with the Institute of IT Security and Security Law (ISL). He graduated in computer science at University of Erlangen and he holds a PhD (Dr. rer. nat.) in the area of middleware for fault-tolerant systems from Ulm University. He spent one semester at the Carnegie-Mellon-University, Pittsburgh, USA as a visiting professor, and one semester at Karlsruhe Institue of Technology as visiting full professor. Hans P. Reiser's research focus is on technical aspects of reliability and security in distributed systems. More information about him can be found at https://www.fim.uni-passau.de/sis/.

João Sousa obtained his graduation at Faculdade de Ciências, Universidade de Lisboa (FCUL). In 2009 he joined LASIGE as a junior researcher and became the leader programmer for the BFT-SMaRt replication library. After finishing his masters degree in 2010, João started his PhD in 2011 (also at LASIGE/FCUL). He delivered his PhD thesis in 2017, and defended it in 2018. In 2019, he was awarded the William C. Carter PhD Dissertation Award in Dependability. Currently, he remains in LASIGE as a post-doctorate, collaborating with NEC in the design and implementation of a consortium blockchain system.

Alysson Bessani is an Associate Professor of the Department of Informatics of the University of Lisboa Faculty of Sciences (Portugal), and a member of LASIGE research unit. He received his B.S. degree in Computer Science from UEM (Brazil) in 2001, the MSE and PhD in Electrical Engineering from UFSC (Brazil) in 2002 and 2006, respectively. He spent some time as a visiting professor in Carnegie Mellon University (2010) and as a visiting researcher in Microsoft Research Cambridge (2014). Alysson participated in more than ten international projects and co-authored more than 100 peer-reviewed publications on dependability, security, and distributed systems. Additional information about him can be found at http://www.di.fc.ul.pt/~bessani.