Adaptive Wide Area REplication

AWARE

geographically DS

WHEAT + BFT-SMaRt

↓ underlying weighting scheme

↓ replication protocol

resilient, adaptive Byzantine consensus

→ automated and dynamic voting-weight tuning and positioning scheme

↳ deterministic, self-optimisation algorithm

↳ dynamically optimise consensus latency by self-reliantly finding a fast configuration → latency gains

⇓

latency optimization scheme

1. BFT-SMaRt

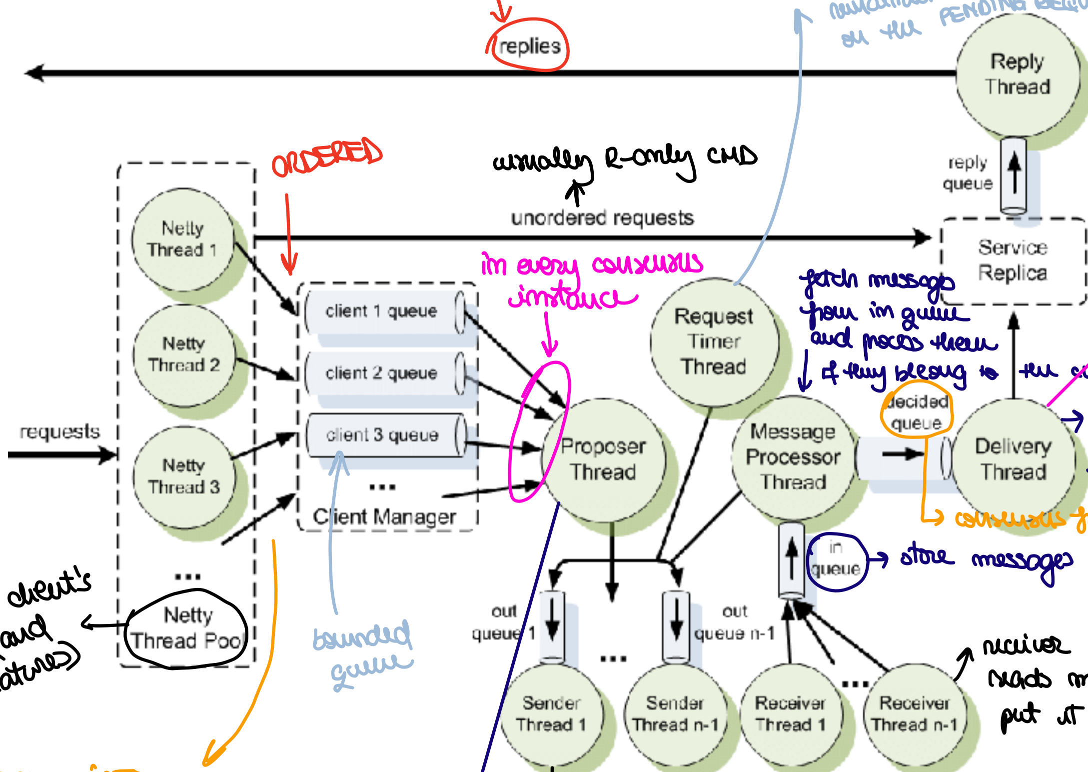
client waits for specific quorum of matching replies

periodically activated to verify if some request remained more than a pre-defined time on the PENDING REQUESTS QUEUE

ORDERED

usually R-only CMD

in every consensus instance



invoked to execute the requests + their replies

FINAL → mark current consensus

fetch message from in queue and process them if they belong to the current exec. consensus

→ get batches from queue, deserialize requests, remove them from queue

↳ consensus finished on replica ⇒ enqueue batch

→ store messages to be processed

verify client's MACs (and signatures)

bounded queue

receiver thread per sender reads m, authenticates it and put it "in queue"

↳ gets message m, serializes it, produce and attach MAC, send it using TCP sockets

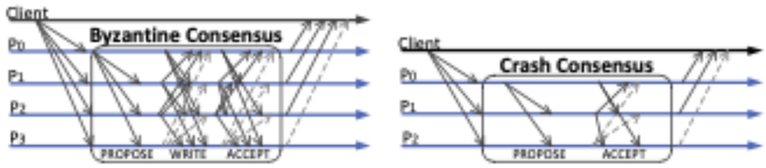
assemble a batch of REQUESTS and transmit PROPOSE message of the consensus protocol

INTERMEDIATE STEP

↳ client manager verifies integrity and adds REQ. to respective client queue

⇒ TOTAL-ORDER execution of requests

⇒ employs dynamically scalable replica set + provides modular architecture (use separated and exchangeable components)



(a) BFT message pattern. (b) CFT message pattern.

Byzantine consensus Algorithm

→ PROPOSE → leader broadcasts a message that contains a batch of requests that need to be decided to all other replicas

→ WRITE
→ ACCEPT

} ALL-TO-ALL broadcast used for commitment → each replica i waits for a quorum Q_i $\left\lceil \frac{n+f+1}{2} \right\rceil$ replicas to proceed

$i \neq j \Rightarrow |Q_i \cap Q_j| \geq f+1$

overlap with at least 1 CORRECT replica

2. WHEAT

↳ add more replicas to a system

the best connected replicas are assigned V_{max}

NOT AVAILABLE

FALLBACK to a quorum size of 4

(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.

+ REDISTRIBUTE weights if they become slower

SAFE WEIGHT DISTRIBUTION SCHEME

↳ n replicas tolerating f Byzantine faults and containing Δ additional replicas

$n = 3f + \Delta + 1$

$Q_v = 2(f + \Delta) + 1$ → wait for quorum formation

$V_{min} = 1$

$V_{max} = 1 + \frac{\Delta}{f}$ → attributed to $2f$ best-connected replicas

SELF-MONITORING

quorum-based measurements => DON'T ALLOW reasoning about link latencies

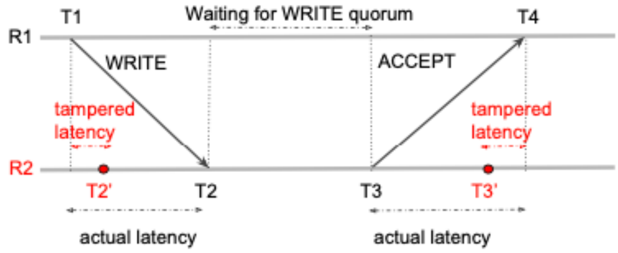


Figure 5: Problem with timestamps and Byzantine replicas.

NON MALICIOUS setting => piggyback responses carrying TIMESTAMPS

=> favor ONE-SIDED measurements → require only the measuring replica to be correct

WRITE AND ACCEPT → replicas immediately respond with WRITE-RESPONSE after receiving a WRITE

introduces monitoring overhead

AWARE

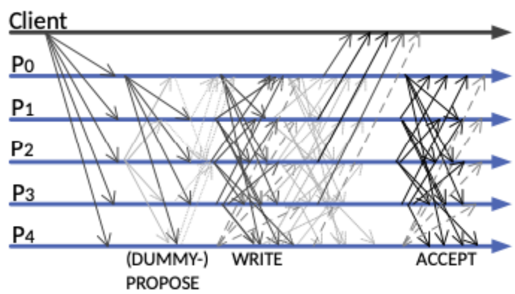


Figure 6: Message flow of AWARE ($f = 1; \Delta = 1$).

highest accuracy in leader selection

→ MONITORING WINDOW → # recent monitoring messages to be used for latency comp.

Response to WRITE → send WRITE-RESPONSE messages (which include a challenge)

→ $L_i = \langle l_{i,0}, \ldots, l_{i,n-1} \rangle$ latency vector of replica i → report after SYNCHRONIZATION PERIOD

DUMMY-PROPOSE → use ROTATION SCHEME s.t. one additional replica simultaneously sends a propose for a dummy batch, but without starting a new consensus instance and all replicas disregard the proposal

OPTIONAL

→ measure the time non-leaders need to PURPOSE

Bounding monitoring overhead → specify parameter $\omega \in [0,1]$ determines the MAXIMUM overhead induced by the monitoring procedure

CALCULATION INTERVAL c → # consensus instances after which a calculation and possibly a reconfiguration is triggered

All replicas maintain **SYNCHRONISED LATENCY MATRICES** M^P, M^W

1. **INITIALIZATION**

$$M^x[i,j] = \begin{cases} \infty, & i \neq j \\ 0, & \text{otherwise} \end{cases}$$

latency of replica i to j measured by i for message type x

2. **UPDATE** using BFT-SMaRt invoke

invokeOrdered(MEASURE, L_i^P, L_i^W)

⇒ global total-order on all client requests and measurement messages

3. **SANITIZE** matrices

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

SELF-OPTIMIZATION

↳ replicas deterministically reconfigure to a new weight configuration and/or leader position

1. VOTING WEIGHTS TUNING
   ↳ search for weight distribution that optimises the system's consensus latency
2. LEADER RELOCATION
   → allow protocol to provide a suitable set of d candidates → AWARE chooses one
   → optimization goal α → threshold by which a predicted consensus needs to be faster to trigger reconfiguration

evaluated at most once every calculation interval

MONITORING

1. Each replica i collects its latency measurements (a moving median) in a vector $L_i = \langle l_{i,0}, ..., l_{i,n-1} \rangle$;
2. Periodically, each replica i disseminates its vectors for *PROPOSE* (L_i^P) and *WRITE* (L_i^W) with total order by calling $invokeOrdered$(MEASURE, L_i^P, L_i^W);
3. Once a replica i decides a batch, that batch may contain messages $\langle$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 $M^W[i,j] = L_i^W[j]$ 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.

stopping condition

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.

OPTIMIZING

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

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

```
for i ∈ I do
    received_i ← new PriorityQueue()
    for j ∈ I do
        received_i.add(⟨T_j^current + M̂^W[j,i], V_j⟩)
for i ∈ I do
    weight ← 0
    while weight < Q_v do
        ⟨T_next, V_next⟩ ← received_i.dequeue()
        weight ← weight + V_next
        T_i^next ← T_next
return (T_i^next)_{i∈I}
```

→ compute the time the replica receives the WRITE from each others

→ for every replica

→ compute the time it had gathered enough voting weights

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

```
V_max ← 1 + Δ/f   V_min ← 1   V_i ← { V_max, if i ∈ R_max ; V_min, otherwise }
∀i ∈ I : offset_i ← 0
while r > 0 do
    for i ∈ I do
        T_i^PROPOSED ← max(M̂^P[p,i], offset_i)
    (T_i^WRITTEN)_{i∈I} ← formQV(I, M̂^W, (T_i^PROPOSED)_{i∈I}, (V_i)_{i∈I})
    (T_i^ACCEPTED)_{i∈I} ← formQV(I, M̂^W, (T_i^WRITTEN)_{i∈I}, (V_i)_{i∈I})
    for i ∈ I do
        offset_i ← T_i^ACCEPTED − T_p^ACCEPTED
    consensusLatencies_r ← T_p^ACCEPTED
    r ← r − 1
return average of consensusLatencies
```

→ the time each replica receives leader's proposal

leader

DETERMINISTIC LATENCY PREDICTION

→ FAST CONFIGURATION → the one that yields low consensus latency from the leader's perspective