As Secure as Possible Eventual Consistency

Work in Progress

Ali Shoker[*]
HASLab, INESC TEC & University of Minho,
Braga, Portugal

Houssam Yactine
HASLab, INESC TEC & University of Minho,
Braga, Portugal

Carlos Baquero[†]
HASLab, INESC TEC & University of Minho,
Braga, Portugal

ABSTRACT

Eventual consistency (EC) is a relaxed data consistency model that, driven by the CAP theorem, trades prompt consistency for high availability. Although, this model has shown to be promising and greatly adopted by industry, the state of the art only assumes that replicas can crash and recover. However, a Byzantine replica (i.e., arbitrary or malicious) can hamper the eventual convergence of replicas to a global consistent state, thus compromising the entire service. Classical BFT state machine replication protocols cannot solve this problem due to the blocking nature of consensus, something at odd with the availability via replica divergence in the EC model. In this work in progress paper, we introduce a new secure highly available protocol for the EC model that assumes a fraction of replicas and any client can be Byzantine. To respect the essence of EC, the protocol gives priority to high availability, and thus Byzantine detection is performed off the critical path on a consistent data offset. The paper concisely explains the protocol and discusses its feasibility. We aim at presenting a more comprehensive and empirical study in the future.

CCS CONCEPTS

• **Information systems → Data management systems**; *Information storage systems*; • **Security and privacy → Distributed systems security**; • **Computer systems organization → Reliability**; **Availability**;

KEYWORDS

Eventual consistency, Byzantine fault tolerance, security, availability, CRDT

ACM Reference format:
Ali Shoker, Houssam Yactine, and Carlos Baquero. 2017. As Secure as Possible Eventual Consistency. In *Proceedings of PaPoC'17,, Belgrade, Serbia, April 23 2017,* 5 pages.
DOI: http://dx.doi.org/10.1145/3064889.3064895

1 INTRODUCTION

Eventual Consistency (EC) [14] emerged as a relaxed trade-off model between strong consistency and availability, given that network partitions and high latency links cannot be avoided in geo-replicated and highly scalable systems [7]. Replicated services that are built through EC are highly available since client's requests are served via a local application server (or replica) without immediate synchronization with other servers; this step is however performed in the background to avoid blocking of client requests, but still ensure (eventual) data convergence. State-of-the-art research in EC assumes that replicas can crash and recover back to the last "healthy" state. Unfortunately, there is evidence that malicious and arbitrary (a.k.a., Byzantine [10]) faults are not rare even in leading Internet services [12, 13]. In the case of EC, a Byzantine server can apply operations in an incorrect way (deliberately or not) which hampers data convergence, and thus compromises the entire service. Consequently, secure EC solutions that are resilient to Byzantine faults, being the strongest fault model [4], are highly advocated when the deployment conditions of servers and clients creates risk for this class of faults.

Classical BFT protocols like state machine replication protocols [4, 9] cannot simply solve the EC problem due to two main reasons. The first is that such protocols are often blocking to the clients since total order coordination is required per operation. The second reason is that replicas are considered *correct* (i.e., not Byzantine) as long as all replies match; i.e., it requires that replies are exactly equivalent. In a recent work [5], the authors tried to solve the latter case by allowing a replica to immediately execute a request, without first establishing a total order, whereas Byzantine agreement between replicas is used, either periodically or on-demand, to establish a common state synchronization point as well as to identify the set of individual operations needed to resolve conflicts. Meanwhile, the client must wait for enough replies from a majority of replicas (after Byzantine agreement is achieved) to commit a reply, which is clearly blocking and impose high delays under network partitions or high latency. Another major issue is that servers may stop receiving new requests until Byzantine agreement among servers is achieved to withstand a Byzantine client. Indeed, we believe that this is impractical in scenarios where eventual consistency was selected to not forfeit availability. Another approach, followed in [12],

[*] The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505, LightKone project.
[†] Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020" is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

PaPoC'17,, Belgrade, Serbia
© 2017 ACM. 978-1-4503-4933-8/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064889.3064895

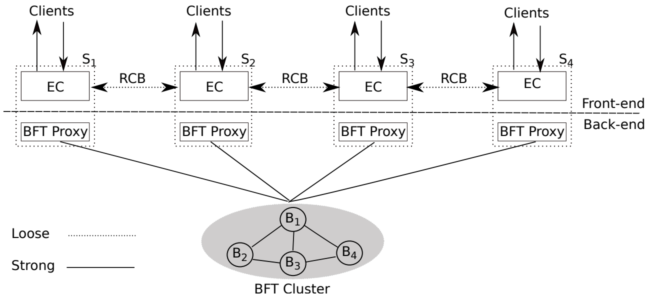


Figure 1: The system model showing how a consistent offset is always verified through the BFT cluster (back-end) without hindering clients access to application servers S_j (front-end) through eventual consistency. S_i are loosely coupled via a Reliable Causal Broadcast (RCB).

was to modify an existing protocol, i.e., Zyzzyva, to support the EC model. Unfortunately, this is impractical for two reasons: (1) it adds more complexity to Zyzzyva whose recovery phase is known to be very complex to implement and test [9], and (2) the industry is unlikely to completely replace a currently running middleware with a new (complex) one. Therefore, we believe that a layered approach that separates EC logic from Byzantine fault tolerance is more convenient as it does not require much changes in the running system.

In this paper, we introduce Byzec , a protocol that makes eventual consistency “as secure as possible”, without impact on system’s availability nor requiring a significant modification to an already deployed system. The protocol allows the service to run in an eventually consistent manner whereas Byzantine behaviors are detected off the critical path, in a back-end process, with the help of a black-box BFT cluster. In particular, and as described in Fig. 1, client’s requests are served by an associated application server as they arrive without immediate synchronization with other servers, which is done in the background and eventually leading to data convergence. Decoupled form this front-end logic, a server progressively sends consistent data offsets to the BFT cluster to be matched against similar versions of other servers, thus forming a “certificate”: a signed proof that up to this very offset, data is equivalent on an appropriate majority of non Byzantine application servers. The client progressively receives the most recent certificate along the replies of the associated server. This allows the client to verify the validity of the certificate; otherwise, it may switch to another server if it holds a proof (basically an invalid certificate) of detecting a Byzantine server, or if the certificate is not sufficiently up to date (which is verified through the other servers as well). This will ensure that Byzantine clients cannot send false acquisitions against correct application servers. Note that the application servers layer is important to prevent direct contact between the clients and the cluster which ensures convergence despite the presence of Byzantine clients, provided that a maximum of f application servers can be Byzantine.

One may argue that our solution is not sufficiently secure as clients can receive non certified data. While this is true, the client will be able to progressively detect any misbehaviors once the consistent data offset evolves. In our opinion, adopting more secure solutions like fault prevention or hiding will impose extra delays as it is done in the critical path, whereas our solution is accountable for Byzantine faults without impacting availability. We believe that in the same sense that the adopters of EC trade strong consistency — despite being a correctness property — for availability, they will likely be keen to trade high security in favor of high availability. What supports our argument is that current EC solutions in production still run in the wild without such Byzantine guarantees; and therefore, they may be less reluctant to adopt secure solutions like ours provided that availability is not compromised.

The solution we introduce is interesting for both: service and applications. On the service side, our solution is important as it guarantees convergence despite the presence of Byzantine servers or clients (see later), which is not possible in current EC systems. On the application side, it is interesting due to its flexibility through allowing a spectrum of options: A non sensitive client can proceed with operations without checking the certificate (i.e., as current systems do), whereas a very conservative client can only accept read operations from a certified consistent data (on the expense of stale data); a trade-off option is to accept a limited number of operations ahead the certified data as long as they will be verified in the future and can be rolled back.

We describe a short version of the protocol in the following sections, leaving the details to a comprehensive study in the future, accompanied with an empirical evaluation to assess the usefulness and feasibility of our approach.

2 PROTOCOL

2.1 Background, system model and fault model

We address a system model where application servers are geo-replicated and (fully) share data structures. A client is directed, via a load balancer, to a given application server. A client can change the associated application server through providing an “acceptable” argument to the load balancer (e.g., the old server is Byzantine). This is described in the front-end in Fig. 1. To ensure high availability in face of network partitions, the front-end components follow the eventual consistency data model: operations of clients are served by the associated server without prompt synchronization with other servers, and they are background propagated to other servers via Reliable Causal Broadcast [2]. Since operations can be applied in different orders on different servers, a conflict resolution method must be used. Without loss of generality, a recently well known approach is to use Conflict-free Replicated DataTypes [11] that encapsulate conflict resolution through mathematically sound policies. At any time, a server can have a different data version provided that all replicas will eventually converge to the same state. Obviously, since a running system is very unlikely to be idle, convergence will not be observed immediately; however, a consistent offset of the data must be ensured once the same set of operations are executed on all replicas and provided that no concurrent operations are expected. This notion is similar to causal stability used in [1] and background global sequence formation in [3], both for non Byzantine settings.

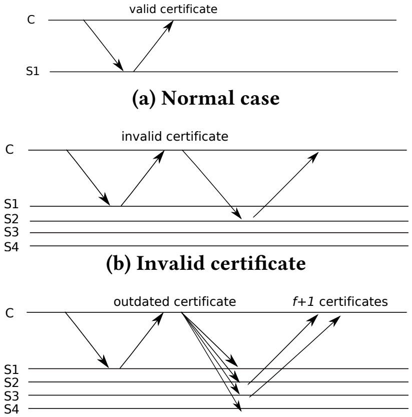
Currently, data convergence is guaranteed as long as replicas execute the operations correctly, and assuming that a crashed sever can recover to the recent correct state [11, 14]. A single Byzantine server can however prevent convergence since the wrong execution of a single operation on the Byzantine server may lead to an inconsistent data state. In this paper, we assume that f application servers out of $3f + 1$ can be Byzantine, and that any client can be Byzantine. Note that $2f + 1$ application servers are not sufficient as in the case of crash-stop fault models[1]; to achieve liveness in the Byzantine model, additional f replicas are required since it is impossible to distinguish a Byzantine node from another one that is just slow [10]. We also assume the presence of a BFT cluster that runs a classical BFT state-machine protocol like PBFT [4], Zyzzyva [9] or even an adaptive mix of these protocols as in Adapt [8]. The purpose is to use this cluster to achieve agreement using strong consistency methods. The fact that this cluster uses consensus will have no impact on the availability of the service once used in the background, as shown in Fig. 1. In particular, we assume that application servers can send (through a BFT proxy process) data offsets to the BFT cluster, which ensures the agreement of at least $2f + 1$ application servers on the common offset.

Finally, we assume that clients and servers (secretly) exchange cryptographic keys that cannot be broken. We don't address flooding attacks, we rather assume the existence of another security layer to guard against them. In addition, we require that clients (that can be end users, proxies, or third party servers), have a method to rollback data changes that have been recently made. We do not address Denial of Service (DoS) attacks, and we treat requests issued by Byzantine clients as normal requests as long as they are legitimate (i.e., following the algorithm).

2.2 An overview of Byzec

We present an overview of the protocol and we associate the corresponding pseudo-code in the Appendix 3 for convenience (given the page limits). The protocol works as follows: a client can access a single server, chosen through a load balancer, following the EC model. The normal case message pattern, depicted in Fig. 2a, is the regular case where no Byzantine behaviors are present. Clients follow this case as long as they receive valid *certificates*. A certificate is a hash digest of an incrementally consistent data offset that is signed by at least $f + 1$ application servers which guarantees its correctness and integrity. A non Byzantine application server initiates the preparation of a new certificate once it has a new *causally stable* operation: an operation for which concurrent operations are no longer delivered through the RCB [2]. (This is usually known once a newer operation, in the causal future, is received from each server.). Since a stable operation has already been executed on all servers, the data offset corresponding to all stable operations must be a consistent offset. This is ensured through preparing a corresponding certificate by the application servers, off the critical path, with the help of a BFT cluster.

A valid certificate informs the client that the data received corresponding to that certified data offset is fault-free; however, no security guarantees are promised for the operations corresponding to the non-certified data, in favor of high availability. If the certificate is invalid (Fig. 2b) the client will change the current application server showing a proof of mis-behavior of the previous server. Once the client updates its state after communicating with the new server, it rolls back the non-certified operations and returns to the normal case.



(a) Normal case

(b) Invalid certificate

(c) Outdated certificate

Figure 2: Messaging patterns of the protocol.

The remaining case is when the received certificate from a server is outdated — it has not been updated for a "long" time. This can be due to two reasons: either the server is Byzantine, or there is some network partitioning or delays preventing the certificates from being updated *on all servers* — because causal stability does not take place on these conditions and the consistent offset stays the same. In the latter case, it is enough for the client to receive $f+1$ matching responses showing that the certificate is up to date; consequently, the client will have no advantage of changing a server. Notice that $f + 1$ matching replies are enough to rule out Byzantine faults and at the same time tolerate network partitions. To the contrary, if the certificate is too old according to $f + 1$ servers which hold a more up-to-date certificates, these servers add the Byzantine server to the blacklist and reply back to the client. Once $f + 1$ matching replies are received by the client, it becomes eligible to as for changing the server and continue through the normal case again.

3 CONCLUSION AND FUTURE WORK

Eventual consistency (EC) is a weak data consistency model that favors availability over consistency. Although EC is becoming prominent in large deployments [6, 14], data convergence approaches only address the Crash-Recovery fault model, and thus adequate support for stronger fault models (like the Byzantine model) is still missing. Recent works (like [5, 12]) tried to solve the problem, but the result was at the expense of the most important criteria in such system which is availability, as it required blocking all clients during synchronization between replicas.

Byzec , outlined here, is a new protocol that handles this issue by respecting the essence of EC: the protocol gives priority to high availability, and thus Byzantine faults detection is performed in background, off the critical path on a consistent data offset. Given that design, Byzec can be used as an added value for practical EC based systems, increase security and fault-tolerance without affecting performance. We are currently implementing the protocol to

[1] Read and Write quorums must interest in at least one non faulty server even under network partitions.

assess its behavior and performance. We plan to drive an empirical study to compare Byzec with classical BFT models as well as existing optimistic BFT models.

REFERENCES

[1] Carlos Baquero, Paulo S Almeida, and Ali Shoker. 2014. Making Operation-Based CRDTs Operation-Based. In *Distributed Applications and Interoperable Systems - International Conference, DAIS 2014*. 126–140.

[2] Kenneth Birman, André Schiper, and Pat Stephenson. 1991. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.* 9, 3 (Aug. 1991), 272–314. DOI: https://doi.org/10.1145/128738.128742

[3] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 568–590. DOI: https://doi.org/10.4230/LIPIcs.ECOOP.2015.568

[4] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461. DOI: https://doi.org/10.1145/571637.571640

[5] Hua Chai and Wenbing Zhao. 2014. Byzantine Fault Tolerance for Services with Commutative Operations. In *Proceedings of the 2014 IEEE International Conference on Services Computing (SCC '14)*. IEEE Computer Society, Washington, DC, USA, 219–226. DOI: https://doi.org/10.1109/SCC.2014.37

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. DOI: https://doi.org/10.1145/1323293.1294281

[7] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. DOI: https://doi.org/10.1145/564585.564601

[8] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. 2015. Making BFT Protocols Really Adaptive. In *In the Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*. IEEE-CS.

[9] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.* 27, 4, Article 7 (Jan. 2010), 39 pages. DOI: https://doi.org/10.1145/1658357.1658358

[10] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. DOI: https://doi.org/10.1145/357172.357176

[11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642

[12] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. 2009. Zeno: Eventually Consistent Byzantine-fault Tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 169–184. http://dl.acm.org/citation.cfm?id=1558977.1558989

[13] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. 2007. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 59–72. DOI: https://doi.org/10.1145/1323293.1294268

[14] Werner Vogels. 2009. Eventually Consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. DOI: https://doi.org/10.1145/1435417.1435432

A THE PSEUDOCODE OF BYZEC

We provide the pseudocode of Byzec for the client, application server, and BFT cluster in Algorithm 2, 31, and 4, respectively. The algorithms make use of some abstractions, defined in Def. 10, which we avoid to include in the pseudocode as they are self-explanatory, and to keep the description concise and clear.

A.1 The client protocol

On start, a client chooses an application server s through any load balancing approach. When a user invokes a new operation o, the client sends a REQUEST to the associated application server (Algorithm. 2, lines 10-13), where N_{Req} is the last sequential client's request number, c_i is the client identifier and $\langle\rangle_{c_i}^{\alpha}$ is the encrypted security token (e.g. digital signature and hash digest) signed with the private key α if the client.

When a client receives a RESPONSE from the associated server (Algorithm. 2, lines 14-26), where m is the received message and σ' is the last received certificate, it checks its validity (i.e., authentication, integrity, and sequence number): if it received a number of invalid messages (defined in a policy), it simply changes the application server through the load balancer (Algorithm .2, lines 15-20). However, if the message holds an invalid certificate, the client sends a COMPLAIN message to all other application servers. Otherwise, the client processes the received message.

When a client receives $f + 1$ matching and valid BLACKLIST messages as a response to its COMPLAIN request (Algorithm. 2, lines 27-36), it updates its blacklist accordingly, requests a new server (if its associated server is blacklisted), and resumes sending REQUEST in the normal case.

A.2 The server protocol

On receiving a valid client's REQUEST (Algorithm. 3, lines 10-17), the associated application server checks the received message's validity. According to the retransmission policy, the server resends a stored RESPONSE message if it was executed earlier; otherwise, it processes the new operation and sends a RESPONSE to the client with its last certificate. On the other hand, when the server receives a COMPLAIN from a client (Algorithm. 3, lines 18-22) due to detecting an old certificate, the server checks for message's validity (mainly the signatures in the old certificate and the correspoding server) to make sure that the client is not lying. If the message is invalid, the server drops it; otherwise, it updates the list of Byzantine servers by adding the outdated received certificate to the BLACKLIST, broadcasts it to other servers, and sends it back to the client.

When an application server reaches stability at time τ (Algorithm. 3, lines 24-25), it generates a digest for a consistent offset including causally stable operations and sends a STABLE message to the BFT cluster, asking for new certificate synchronization with other servers. Once the new certificate σ' is received (Algorithm. 3, lines 26-31), the application server checks if the received certificate is valid and updates its old certificate σ with a new one σ'.

A.3 The BFT cluster protocol

When the BFT cluster receives a digest of a consistent offset STABLE from a (non Byzantine) server (Algorithm. 4, lines 1-7), the BFT cluster checks the received message's validity and drops the message if it is invalid. Otherwise, it tries to match at least $2f + 1$ messages with same new updated certificate to broadcast it to all servers. The BFT cluster algorithm we provide excludes the encapsulated BFT state-machine protocol that is used as a black box.

Definitions 1: Auxiliary abstractions.

1 loadBalance() : chooses a server through load balancing.

2 validMsg(): encapsulates authentication, integrity, and sequence nb of a messages.

3 validCertificate(): checks if certificate is signed by $f + 1$ servers.

4 validPolicy(): a retransmission policy followed by clients and servers.

5 outdatedCertificate(): a certificcate has not been updated for a long time according to a certain policy.

6 rollback(): rolls back requests issued after last correct certificate.

7 matching() : checks if messages are matching.

8 stable(): returns a timestamp that has become causally stable.

9 stableOffset() : returns a datatype offset corresponding to a stable timestamp.

10 bftAgree(): returns a certificate signed by all BFT agreed servers on a stable timestamp and corresponding digest.

Algorithm 2: The client protocol.

```
init:
    $s$ := loadBalance($S$)
    $N_{Req}$ := 0
    $N_{Complain}$ := 0
    lastReq := $\phi$
    BList := $\phi$
    Buffered := $\phi$
    $\sigma$ := $\phi$
    data := $\phi$
on invoked$_i$(OPERATION, $o$):
    $N_{Req}$ := $N_{Req}$ + 1
    lastReq := ($N_{Req}$, $o$, $c_i$)
    send(REQUEST, $\langle \text{lastReq} \rangle^{\alpha}_{c_i}$, $s$)
on receive$_i$(RESPONSE, $in = \langle m, \sigma' \rangle^{\alpha}_{s}$):
    if ¬ validMsg($in$, $s$) ∨
      ¬ validCertificate($\sigma'$) then
        if ¬ validPolicy() then
            $s$ := loadBalance($S$)
            data := rollback(data, $\sigma$);
        send(REQUEST, $\langle \text{lastReq} \rangle^{\alpha}_{c_i}$, $s$)
    else
        if outdatedCertificate($\sigma'$) then
            $N_{Complain}$ := $N_{Complain}$ + 1
            send(COMPLAIN, $\langle N_{Complain}, in \rangle^{\alpha}_{c_i}$, $S$)
        else
            process($m$)
on receive$_i$(BLACKLIST, $in = \langle bList, \sigma \rangle^{\alpha}_{s_j}$):
    if ¬ validMsg($in$, $s_j$) then
        dropMsg($\langle bList, \sigma \rangle^{\alpha}_{s_j}$)
    else
        add(Buffered, $bList$)
        if matching(Buffered) > $f$ then
            BList := $bList$
            if $s$ ∈ BList then
                $s$ := loadBalance($S$)
            send(REQUEST, $\langle \text{lastReq} \rangle^{\alpha}_{c_i}$, $s$)
```

Algorithm 3: The server protocol.

```
Init:
    data := $\phi$
    $\forall i \in I$, LastRes[i] = 0
    $\forall i \in I$, $N_{Req}$[i] = 0
    $\sigma$ := $\phi$
    BList := $\phi$
    timestamp := (0, 0 . . . )
    seq := 0
With Clients:
on receive$_j$(REQUEST, $in = \langle m \rangle^{\alpha}_{c_i}$):
    if ¬ validMsg($in$, $c_i$) then
        if validPolicy() then
            send(RESPONSE, $\langle \text{LastRes[i]}, \sigma \rangle^{\alpha}_{s_j}$, $c_i$)
        else
            dropMsg($in$)
    else
        LastRes[i] := process($m$, data, timestamp)
          $N_{Req}$[i] := $N_{Req}$[i] + 1
          send(RESPONSE, $\langle \text{LastRes[, ]}\sigma \rangle^{\alpha}_{s_j}$, $c_i$)
on receive$_j$(COMPLAIN, $in = \langle \langle m, \sigma \rangle^{\alpha}_{s} \rangle^{\alpha}_{c_i}$):
    if ¬ validMsg($in$, $c_i$) then
        dropMsg($in$)
    else
        add(BList, outdatedCertificate($\langle m, \sigma \rangle^{\alpha}_{s}$, $s$))
          send(BLACKLIST, $\langle \text{BList}, \sigma \rangle^{\alpha}_{s_j}$, $c_i$)
With BFT Cluster:
on stable$_j$($\tau$):
    consistentOffset := stableOffset(data, $\tau$)
      $D$ := Digest(consistentOffset)
      send(STABLE, $\langle D, \tau, \text{seq} \rangle^{\alpha}_{s_j}$, BFTCluster)
on receive$_j$(CERTIFICATE, $in = \langle \sigma', seq \rangle^{\alpha}_{b}$):
    if ¬ validMsg($in$, $b$) then
        dropMsg($in$)
    else
        if validCertificate($\sigma'$, $seq$, seq) then
            $\sigma$ := $\sigma'$
```

Algorithm 4: The BFT cluster protocol.

```
With BFT Cluster:
on receive$_j$(STABLE, $in = \langle D, stableTS, seq \rangle^{\alpha}_{s_j}$):
    if ¬ validMsg($in$, $s_j$) ∨ then
        dropMsg($in$)
    else
        ($cert$, $Servers$) := bftAgree($D$, $stableTS$, $seq$)
        if | $Servers$ |≥ $2f + 1$ then
            send(CERTIFICATE, $\langle cert, seq \rangle^{\alpha}_{b}$, $Servers$)
```