As Secure as Possible Eventual Consistency

Work in Progress

Ali Shoker, Houssam Yactine, and 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 that is at odds 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.

1. INTRODUCTION

Eventual Consistency (EC) [13] 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 [6]. 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 [9]) faults are not rare even in leading Internet services [12, 11]. 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, 8] 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 [14], 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 [11], 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 [8], and (2) the industry is unlikely to completely replace a currently running middleware with a new (complex) one.

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 that front-end logic, a server progressively sends consistent data offsets to the

© 2017 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

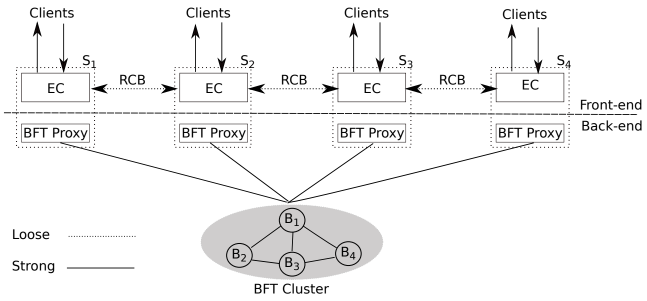


Figure 1: The system model showing how a consistent offset is always verified through the BFT cluster (back-end) without hindering clients access to application servers S_j (front-end) through eventual consistency. S_i are loosely coupled via a Reliable Causal Broadcast (RCB).

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).

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, 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 that asserts 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 well known approach is to use Conflict-free Replicated DataTypes [10] 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 [13, 10]. 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[1]. We also assume the presence of a BFT cluster that runs a classical BFT state-machine protocol like PBFT [4], Zyzzyva [8] or even an adaptive mix of these protocols as in Adapt [7]. 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.

2.2 An overview of Byzec

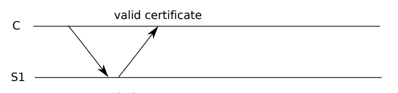
We present an overview of the protocol and we associate

[1]Note that $2f+1$ replicas are not sufficient as in the case of crash-stop fault models; 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 [9]

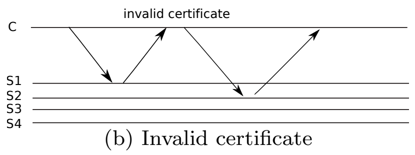
the corresponding pseudo-code in the Appendix 3 for convenience (given the page limits). The protocol works as follows: a client can access a single server, chosen through a load balancer, following the EC model. The normal case message pattern, depicted in Fig. 2a, is the regular case where no Byzantine 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. An 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.

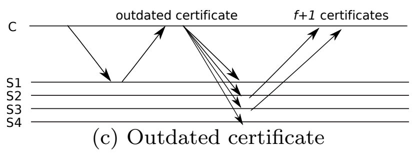
The remaining case is when the received certificate from a server is outdated – it has not been updated for a "long" time. This can be due to two reasons: either the server is Byzantine, or there is some network partitioning or delays preventing the certificates from being updated *on all servers* — 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.



(a) Normal case



(b) Invalid certificate



(c) Outdated certificate

Figure 2: Messaging patterns of the protocol.

3. CONCLUSION AND FUTURE WORK

Replication can improve systems performance and fault tolerance. Since communication and network-partition failures often occur in large scale systems, a strong consistency model would negatively impact availability. However, as giving up availability is normally not an option when timely responses are a goal, weaker consistency models such as eventual consistency (EC) [13, 5] are currently being used in many large scale deployments. A common consequence of an eventual consistency (EC) design is the need to address the reconciliation of replicas. Sound models for enabling deterministic reconciliation [10] are now commonplace. Nevertheless, these approaches only address the Crash-Recovery fault model, and adequate support for Byzantine models was found lacking.

Recent work 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 detection is performed in background, off the critical path on a consistent data offset. Given that design, the Byzec protocol can be used as an added value for practical EC based systems, increase security and fault-tolerance without affecting performance.

This research work assumes a non-broken security level which will need special crypto functions, also it requires data storage that supports undo of yet non-certified executed operations, whose support still requires developing. In future work, we plan to implement the system, evaluate the obtained results in comparison with classical BFT models, and with existing optimistic BFT model.

4. REFERENCES

[1] C. Baquero, P. S. Almeida, and A. Shoker. Making operation-based crdts operation-based. In *Distributed Applications and Interoperable Systems - International Conference, DAIS 2014*, pages 126–140, 2014.

[2] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[3] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 568–590. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels.

Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[6] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

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

[8] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, Jan. 2010.

[9] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

[12] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. *SIGOPS Oper. Syst. Rev.*, 41(6):59–72, Oct. 2007.

[13] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[14] W. Zhao. Optimistic Byzantine fault tolerance. *International Journal of Parallel, Emergent and Distributed Systems*, 31(3):254–267, May 2016.

APPENDIX

A. THE PSEUDOCODE OF Byzec

We provide the pseudocode of Byzec for the client, server, and BFTServer in Algo 2, 3, and 4, respectively. The algorithms make use of some abstractions, defined in Def. 1, 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 a server s through load balancing. When a user invokes a new operation o, the client sends a REQUEST to the associated 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. The client first checks for the received message's validity (Authentication, Integrity and sequence number). If the received message is valid and contains a new valid certificate then the client processes the message and updates its local certificate; if the received message is valid with an outdated certificate (according to the client's policy) then the client sends a COMPLAIN message to all other servers. If the received message is not valid or the received certificate is not valid then the client, according to its policy, associates a new server, again through load balancing, undoes all non-certified operations submitted by the old server and resumes sending in the normal case.

When a client receives matching $f + 1$ valid blacklist messages about a given Byzantine server (Algorithm. 2, lines 27-36) then the client updates its blacklist, requests a new server (if its associated server is blacklisted) and resumes sending in the normal case.

A.2 The server protocol

On receiving a client's REQUEST (Algorithm. 3, lines 10-17), the associated server checks the received message's validity. If the received message and the client's policy are invalid, the server drops the message, otherwise, if only the client's policy is valid then the server ignores the received message and sends a RESPONSE asking the client to retransmit the operation. In case of receiving a valid message, the associated server processes the operation and sends a RESPONSE to the client with its last certificate.

When a server receives a COMPLAIN from a client (Algorithm. 3, lines 18-22) due to an old policy, the server checks again for message's validity. If the message is invalid, the server drops it, otherwise, the server updates the list of potential Byzantine servers BLACKLIST by adding the outdated received certificate to the BLACKLIST, broadcasts it to other servers and sends it back to the client.

At any time τ, when a server reaches stability, (Algorithm. 3, lines 24-25) it can generate a digest for a consistent offset including causally stable operations and sends it to the BFT cluster, asking for later state synchronization with other servers.

On receiving a certificate σ' from BFT cluster (Algorithm. 3, lines 26-31), the server checks if the received message is invalid, possibly dropping it. Otherwise, the server checks if the received certificate is valid according to its sequence number and timestamp, then the server 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 server (Algorithm. 4, lines 1-7), the BFT cluster checks the received message's validity, drops the message if it is invalid. Otherwise, it tries to match at least $2f + 1$ 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 before changing a server.
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.

```
1  init:
2      $s$ := loadBalance($S$)
3      $\mathsf{N_{Req}} := 0$
4      $\mathsf{N_{Complain}} := 0$
5      lastReq := $\phi$
6      BList := $\phi$
7      Buffered := $\phi$
8      $\sigma := \phi$
9      data := $\phi$

10 on invoked$_i$(OPERATION, $o$):
11     $\mathsf{N_{Req}} := \mathsf{N_{Req}} + 1$
12     lastReq := ($\mathsf{N_{Req}}, o, c_i$)
13     send(REQUEST, $\langle$lastReq$\rangle^{\alpha}_{c_i}$, $s$)

14 on receive$_i$(RESPONSE, $in = \langle m, \sigma'\rangle^{\alpha}_{s}$):
15     if $\neg$validMsg($in, s$) $\vee$
16        $\neg$validCertificate($\sigma'$) then
17         if $\neg$validPolicy() then
18             $s$ := loadBalance($S$)
19             data := rollback(data, $\sigma$);
20         send(REQUEST, $\langle$lastReq$\rangle^{\alpha}_{c_i}$, $s$)
21     else
22         if outdatedCertificate($\sigma'$) then
23             $\mathsf{N_{Complain}} := \mathsf{N_{Complain}} + 1$
24             send(COMPLAIN, $\langle \mathsf{N_{Complain}}, in\rangle^{\alpha}_{c_i}$, $S$)
25         else
26             process($m$)

27 on receive$_i$(BLACKLIST, $in = \langle bList, \sigma\rangle^{\alpha}_{s_j}$):
28     if $\neg$validMsg($in, s_j$) then
29         dropMsg($\langle bList, \sigma\rangle^{\alpha}_{s_j}$)
30     else
31         add(Buffered, $bList$)
32         if matching(Buffered) $> f$ then
33             BList := $bList$
34             if $s \in$ BList then
35                 $s$ := loadBalance($S$)
36             send(REQUEST, $\langle$lastReq$\rangle^{\alpha}_{c_i}$, $s$)
```

Algorithm 3: The server protocol.

```
1  Init:
2      data := $\phi$
3      $\forall i \in I$, LastRes[i] = 0
4      $\forall i \in I$, $\mathsf{N_{Req}}$[i] = 0
5      $\sigma := \phi$
6      BList := $\phi$
7      timestamp := $(0, 0 \ldots)$
8      seq := 0
9  With Clients:
10 on receive$_j$(REQUEST, $in = \langle m\rangle^{\alpha}_{c_i}$):
11     if $\neg$validMsg($in, c_i$) then
12         if validPolicy() then
13             send(RESPONSE, $\langle$LastRes[i], $\sigma\rangle^{\alpha}_{s_j}$, $c_i$)
14         else
15             dropMsg($in$)
16     else
17         LastRes[i] := process($m$, data, timestamp)
           $\mathsf{N_{Req}}$[i] := $\mathsf{N_{Req}}$[i] + 1
           send(RESPONSE, $\langle$LastRes[, ]$\sigma\rangle^{\alpha}_{s_j}$, $c_i$)

18 on receive$_j$(COMPLAIN, $in = \langle\langle m, \sigma\rangle^{\alpha}_{s}\rangle^{\alpha}_{c_i}$):
19     if $\neg$validMsg($in, c_i$) then
20         dropMsg($in$)
21     else
22         add(BList, outdatedCertificate($\langle m, \sigma\rangle^{\alpha}_{s}$, $s$))
           send(BLACKLIST, $\langle$BList, $\sigma\rangle^{\alpha}_{s_j}$, $c_i$)

23 With BFT Cluster:
24 on stable$_j$($\tau$):
25     consistentOffset := stableOffset(data, $\tau$)
       $D$ := Digest(consistentOffset)
       send(STABLE, $\langle D, \tau$, seq$\rangle^{\alpha}_{s_j}$, BFTCluster)

26 on receive$_j$(CERTIFICATE, $in = \langle \sigma', seq\rangle^{\alpha}_{b}$):
27     if $\neg$validMsg($in, b$) then
28         dropMsg($in$)
29     else
30         if validCertificate($\sigma', seq$, seq) then
31             $\sigma := \sigma'$
```

Algorithm 4: The BFT cluster protocol.

```
1  With BFT Cluster:
2  on receive$_j$(STABLE, $in = \langle D, stableTS, seq\rangle^{\alpha}_{s_j}$):
3      if $\neg$validMsg($in, s_j$) $\vee$ then
4          dropMsg($in$)
5      else
6          $(cert, Servers)$ := bftAgree($D, stableTS, seq$)
7          if $|\ Servers\ | \geq 2f+1$ then
8              send(CERTIFICATE, $\langle cert, seq\rangle^{\alpha}_{b}$, $Servers$)
```