Enabling expiration of identities in Hyperledger Fabric

E. Androulaki, A. De Caro, J.Yellic, M. Vukolic

(Aims to address FAB-3698, FAB-5843)

This document aims to explore the support for identity validity w.r.t. the identity’s expiration time expressed by means of "real" time. It is important that we find a way around identity expiration checks as this would allow for reduced sizes of CRLs (as revoked certificates would not be tracked in CRLs after the certificates expire), various security-operations enablement such as key-rotation, etc.

Conclusions about an identity having expired or not in this case would need to be the same across all committing peers who process a block an identity is included in, regardless of each peer’s local state (e.g., the time in which the peer would validate the transaction, the drift in the peer’s local clock, or the time a peer would join the network).

Enabling identity expiration in Fabric would require changes to the existing MSP interface, policy evaluation interface, as well as to the security checks taking place on the client/peer/orderer side during a transaction’s lifecycle.

# 1. Changes to MSP interface

Changes to MSP interface to consider a timestamp at identity validation are explained in detail in [MSP refactoring document](https://docs.google.com/document/d/1FyvjMlBFasdFlOpjcaurNG8jYipHE8ENXfdbY122mdA/edit) .

# 

# 2. Changes to Policy evaluation mechanism

Changes to Policy interface to consider a timestamp at identity validation are explained in detail in the [MSP refactoring documen](https://docs.google.com/document/d/1FyvjMlBFasdFlOpjcaurNG8jYipHE8ENXfdbY122mdA/edit)t .

# 3. Added security checks in transaction lifecycle

To enhance transaction lifecycle with identity expiration ability, the following could take place:

1. **At endorsement request time**, client sends to endorser a proposal. Proposal already includes client certificate and a timestamp that is currently not checked. From now on  **proposal timestamp would constitute the client’s proposal for transaction’s timestamp**, e.g., denoted by **PROPOSAL\_TS** .  
   **Endorser** is here requested to
   1. check validity of its own signing identity
   2. check validity of proposal’s timestamp, w.r.t. its local time, e.g., that,  
         
      where **TWINDOW i**s set to the peer using some channel-scoped configuration.
   3. check validity of proposal access policies / client identity using **PROPOSAL\_TS**
2. **At endorsement response time**, **client** would need to check validity of the endorser identity using **PROPOSAL\_TS**, and only if this check passes to create a transaction including the received endorsement. Namely PROPOSAL\_TS would need to be properly signed by the endorsers in their endorsement. Client would construct a transaction with a timestamp TRANSACTION\_TS. Honest clients would set TRANSACTION\_TS to PROPOSAL\_TS.
3. **At transaction broadcast time**, orderers would need to check that received transactions have a timestamp (TRANSACTION\_TS**)** that does not exceed what is considered to be ***the blockchain time***. If the orderers’ clocks are synchronised this can be the orderer local time. Otherwise, we list in subsequent sections a couple of alternatives to define Blockchain time. More specifically, they would need to retrieve the timestamp the client inserted into the transaction, TRANSACTION\_TS, and ensure that
   1. TRANSACTION\_TS is an accepted timestamp for transactions, i.e., that:
   2. Client satisfies the channel’s access policy (WRITERS) assuming reference time TRANSACTION\_TS
4. At transaction validation time, a committing peer would consider PROPOSAL\_TS as the valid timestamp to perform endorser certificate expiration checks, i.e., it would need to
   1. check endorsement policy of the chaincodes whose state is being modified using PROPOSAL\_TS as reference timestamp
   2. ensure that PROPOSAL\_TS = TRANSACTION\_TS, i.e., the client timestamp is what is included in the endorsements of the transaction

# 4. Alternatives of what we consider Blockchain time

**Alternative 1: Leverage block transactions timestamp.** In this case the timestamp considered in a block would derive as a function of the timestamps of transactions that appear in the block and the timestamp computed for the previous block. The latter is needed as is important to ensure that Blockchain time seen as the series of block timestamps is non-decreasing.

For example, TS\_BLOCKi of the i^th block of a channel could be computed as follows

Clearly this would require that the orderers process a transaction multiple times; that is, they first filter transactions based on TS\_BLOCKi-1, they order transactions in a preliminary version of the ith block, compute TS\_BLOCK\_i and subsequently filter out transactions that have a timestamp preceding the Blockchain time beyond a time window. This can clearly trigger new rounds of computation of transaction timestamp, which may be costly.

An other variant of the above approach would be that we have orderers filter transactions based on the difference of TRANSACTION\_TS from the previous block’s timestamp. Transaction validation would consider TRANSACTION\_TS as reference timestamp as long as it is the same as PROPOSAL\_TS.

Such approaches would be vulnerable to attackers who attempt to delay the Blockchain clock by faking transactions with “slow” timestamps. Recall that orderers only check client signature to decide on whether they consider a timestamp in a block’s timestamp calculation.

Other than such attacks, this method could be viable as long as blocks are announced often enough, and the channel does not remain idle (i.e., without transactions) for a long period of time.

**Alternative 2: Leverage a configuration parameter on the channel level.** Configuration transactions could include at the top level of channel configuration (Channel configuration) a parameter called “BLOCKCHAIN\_TIME” that would represent the real-world. BLOCKCHAIN\_TIME parameter could be specified to have a modification policy equal to “Application.Admins”. This may increase the configuration transactions advertised in the system that would be ideal to be kept in low-frequency.

**Alternative 3: Leverage a configuration parameter on the MSP level.** MSP Configuration could include a real-time parameter, that would be considered for identity validation concerning that MSP. Updates to this parameter would be served again via config\_updates on the MSP parameter that would again trigger a new configuration update to take place. However in this case only admins of the MSP need to reach consensus for updating this parameter, and they would only need to do so as long as certificates they have issued are known to expire.

This method may seem easy to implement, as it does not require any changes to policy framework at all (only internal msp structures/configuration is required). However the timestamp generated in this way would only serve identity expiration checks, and cannot be used for other cases where connection to the real-world time is needed. Local MSP configuration would need to obtain the peer’s local time as timestamp of reference.

**Alternative 4: Leverage a Blockchain-wide timestamp orderers reach consensus upon.** That is, we could enhance ordering service (orderer protocols) to include inside BlockHeader a timestamp (a timestamp all orderers agree on) that accompanies block generation; this would require modifying the proto messages, and ensure that orderers reach consensus over the timestamp itself (despite clock drifts that may exist between orderers). For Kafka, the latter should be easier to be implemented, but for sBFT this may get more complex.

Assuming that timestamps in successive blocks will be non-decreasing, in this case, committing peers would need to assess access rights of clients/endorsers considering the previous block’s timestamp, which assuming we have frequent generation of blocks, this should not be an issue.

## 3. Q & A

**Including here popular questions on this matter.**

**Can’t we use peer’s local time if we assume that certificates are renewed long time before they are to expire?**

No. Even if we assume that certificates are renewed long before they expire, so that no chance to cause state-forks (at commiting peers who are part of the channel), we still have the issue that a peer that joins a channel much later than the cert's expiration time, cannot validate properly any of the endorsed transactions with expired endorser/client certificates. This issue was reported by [Angelo De Caro](https://jira.hyperledger.org/secure/ViewProfile.jspa?name=adc).