Multichannel Consensus

Authors: Binh Nguyen, Christian Cachin, Jason Yellick, Elli Androulaki, Baohua Yang, Angelo De Caro, Kostas Christidis, Marko Vukolic

The current usage of DLT for confidentiality involves hashing the sensitive data from the transactions, and thus, requiring the application to maintain another network of database servers to safe-guard the sensitive data. The goal of multichannel consensus is to enable counterparties to form isolated channel to transact in complete private and confidential. The transactions in a channel is not visible to others outside of the channel, so all sensitive data may stay with the transactions without requiring a separate database. Of course, applications may choose to hash the data as Fabric protocol already supports transaction visibility.

Hyperledger Fabric architecture defines consensus as a 3-phase process: Endorsement, ordering, and validation. The ordering service provides messaging according to the publish-subscribe model using a topic-based model, where messages are communicated over logical channels (for example, a topic partition in Apache Kafka). The ordering service is performed by a network of entities called Orderers, whereas the endorsement and validation are done by Peers.

Each Peer connects to the ordering service on one or more channels like clients of a pub-sub communication system. Transactions broadcasted on a channel are ordered by consensus such that all subscribers (Peers) receive the same transactions in one channel and in the same order. The transactions are delivered to the subscribers (Peers) in cryptographically linked blocks. Each Peer validates the blocks and commits them to the ledger then provides other services to the applications to use the ledger. The ordering service treats messages on different channels independently, there are no guarantees across channels.

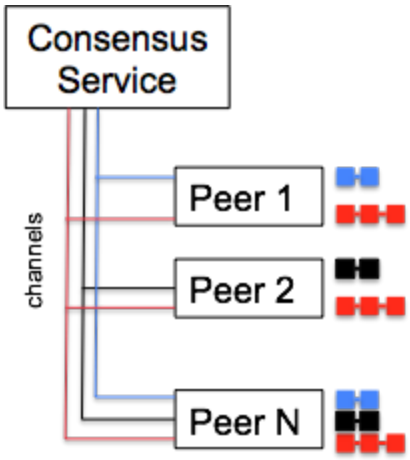
Terminologies

- Consensus: A process to execute and validate the outcome of a transaction on the network by all involved members.
- Network: A collection of Members and Channels for some purpose with an ordering service. Any number of Peers from the Members may join the network.
- Member: Member is a participant (like a company or organization) who might have multiple Peers, Orderers, and applications in the network. A member is identified by its CA certificate.
- Orderer: A network entity performs transaction ordering service, e.g., using crash or Byzantine fault tolerance.
- Peer: A network entity responsible for executing and validating transactions. It maintains 1 or more ledgers depending on the number of channels that it participates in. A peer identity is issued by a member of the network.

- Channel: Similar to pub-sub messaging queue, a channel is like a topic that authorized peers may subscribe to and become members of the channel. Only members of a channel may transact on that channel, and transactions on a channel is not visible on other channels.
- Ledger: Ledger keeps the log of transactions confirmed among the members of a channel. There is no 1 default global ledger, but depending on business needs, the application may set up such ledger.

Data Isolation for Confidentiality

A channel may be created for a subset of members on the network to transact, so these members make up the set of stakeholders of those transactions submitted to this channel, and only these members may receive the blocks containing the transactions, which are completely isolated from other transactions on other channels. A member who is not authorized on a channel may not join the channel and can't transact on that channel.



A member of a channel may have any number of Peers participating on the channel. The key goal is availability such that the member is always "present" on the channel. The Peers on a channel maintain a ledger containing the transaction log on that channel. So if a Peer participates in 2 channels, it will have 2 transaction logs or 2 ledgers as illustrated in the diagram on the left. Cross-channel data access is possible and discussed in the Implementation section below.

For example, as illustrated in the drawing on the above, Peer 1, 2, and N subscribe to the red channel and manage the red ledger among themselves; Peer 1 and N subscribe to the blue channel and manage the blue ledger (note that Peer 2 doesn't have the blue ledger since it is not on the blue channel); similarly, Peer 2 and Peer N are on the black channel and manage the black ledger.

The application decides based on business logic to send each transaction to 1 or more channels. There is no built-in restriction; the network has no knowledge and assumes no relationship between transactions on different channels. However, for CHAINCODE type transactions, since the chaincode on the ledger executes the transaction within the context of the ledger, which is associated with a channel, sending that same transaction to another channel will not work if the chaincode is not on that ledger.

Some blockchain networks may never need more than 1 channel if transactions are considered "public"; that is, all transactions are visible to all members, or application manages the

transaction visibility via hashing and broadcasts the hashed transactions to members. However, in networks where some transactions are considered private and confidential among a subset of members (eg bilateral transactions), a separate channel is the option to isolate the data to provide confidentiality.

Note that the ordering service receives all transactions on all channels, so confidentiality is only pertaining to the Peers and not the Orderers. This is fine when the ordering service is made up of trusted parties and regulatory bodies in a permissioned network; that is, the transactions are visible to them as business requirements. On the other hand, if the members don't want other members' Orderers to see the content of the transactions, they should utilize other techniques to hide the sensitive data such as hash or encryption or setting up their own ordering service.

The ordering service acts as a trusted party. Specifically, if it is implemented with Byzantine fault-tolerant (BFT) consensus protocols (such as PBFT), then this prevents a threshold of untrusted Orderers from corrupting the consistency of the ledger and/or from impeding the availability. However, no such protocols offer confidentiality guarantees with this multi-channel design in the presence of untrusted Orderers.

Implementation

A channel is implicit in today's implementation of Fabric consensus service; when a Peer connects, it doesn't specify a channel. So to support multiple channels, both Orderer and Peer need to change. The Orderer must provide multichannel subscription and the Peer needs to know which channels to subscribe to. The following sections describe the technical details.

Bootstrapping

An ordering service consists of 1 or more Orderers. Each Orderer is configured with a matching genesis block, which is generated by a bootstrapping CLI command, which provides the necessary data, including a list of trusted roots, a list of Orderer certificates and IP addresses, a set of specific consensus algorithm properties, and access control policies (who may create channels). An Orderer boots up with the genesis configuration data.

A Peer bootstrap requires very minimum configuration: A supported certificate to join the member's network of peers, and a list of trusted roots that the peer may verify messages outside of channels.

To subscribe to a channel, a Peer's certificate must be signed by 1 of the members authorized to the channel. The members authorized to the channel are encoded in a configuration transaction on the channel, starting with the genesis block. That is, the genesis block of every channel contains a configuration transaction.

Either via CLI or application using SDK API, a Peer can be instructed to subscribe to existing channels. Access control on whom may join a channel is performed by the Orderers with the information given during channel creation or reconfiguration thereafter.

Creating a Channel

A channel may be created by sending Channel Configuration transaction to the ordering service with a list of participating members, represented by MSPs (Member Service Providers). Note that there are many potential configuration transactions, but we only address Channel configuration in this document.

The application decides on which members to create the channel for. The agreement of members to form a channel is done out of band as necessary for the application to create the Configuration transaction rather than using the CHAINCODE type transaction endorsement to avoid unnecessary complexity on the Orderer processing side.

We will introduce a new system chaincode, called Configuration system chaincode (CSCC), which will be responsible for processing all configuration related transactions on the Peer when it receives the configuration transaction. CSCC will provide functions to query various configuration data, including channels.

For example, suppose Peer A and B belonging to 2 different members Alice and Bob, who have ability to issue enrollment certificates for the computing nodes (Peers, Orderers) that they use on the network. Assuming that during bootstrap (or sometime later), we have configured Alice and Bob to be able to create channels. The following is a typical sequence (application uses SDK):

1. The application requests the endorsement of A and B for a CONFIG transaction to create channel “foo”, then sends the transaction (by invoking the Broadcast RPC) passing the endorsed CONFIG transaction to the ordering service.
2. The application then invokes the Deliver RPC on channel foo. This RPC will return an error until the channel is created by the ordering service.
3. When the channel is eventually created, the Deliver RPC returns the channel stream to the application. At this point, the channel foo should only have a genesis block containing the relevant orderers were bootstrapped with (or the most recent RECONFIG transaction), along with this CONFIG transaction.
4. The application invokes CSCC on the Peers A and B to join the channel by passing the genesis block of channel foo to them.
5. CSCC on A and B inspect the genesis block, including checking the endorsements from the CONFIG transaction in the block. If they approve what they see, they invoke the Deliver RPC on the channel to start receiving blocks.

On #3, the option was considered for the channel member Peers to wait and invoke the Deliver RPC; however, that would complicate the endorsement and put the Peer on a temporary "stateful" period: It has completed the endorsement, but a background process kept on running to check the channel was created successfully. This would go against our design goal to have a stateless peer.

If the channel already exists, the list of Participants is the replacement of the existing list. The Orderers automatically replace the subscribers and eventually deliver the configuration transaction to the new members on this channel. The new member peers will sync to have a complete ledger.

Terminating a Channel

Application may terminate a channel that it created by sending a CONFIGURATION transaction similar to creating a channel. It requires endorsements from channel participating members according to policy set by the application. Orderers will destroy the channel, and transactions on the channel will result in error. Peers will not automatically destroy the associated ledger, but the pruning process will take care of that in due time (based on regulations, policy, etc), which we will not discuss here.

Application may continue to read the data from a terminated ledger as long as it hasn't been pruned, but since the channel has been destroyed, no further transactions are possible.

Querying a Channel

A channel can only be queried by members of the channel; that is, transactors whose signatures can be verified by the member CA certificates stored on the channel ledger configuration block. This is done by sending a query transaction to the CSCC specifying the chain ID to query its channel. The result is the configuration block, which contains member certificates among other configuration data.

Transactions on Channel

A transaction must include a channel ID to target. A channel ID indicates which channel to place the transaction on, which members to send transaction to, and which ledger to store the transaction log.

The consensus service will place the transaction on the specified channel, identified by the channel ID, and to be ordered within that channel, independent of transactions on other

channels. Eventually a block is created containing transactions on the channel and delivered to those Peers subscribed to the channel.

Note that each channel operates independently and concurrently, so a Peer may receive and process blocks on different channels simultaneously.

A CHAINCODE transaction can only manipulate state variables within the specified channel (similar to functions operate on instance variables of an object). See the next section Chaincodes on Channel for more details.

Chaincodes on Channel

A chaincode is deployed on a channel and operates within that channel. Note that the chaincode names are managed by the life-cycle system chaincode to enforce uniqueness. Application might want to use namespace (eg acme.com) for clarity.

There are use-cases where we want to call chaincode from another channel to transition state on either the caller's channel or the channel that the chaincode is deployed on. For example, suppose we had a Weather system chaincode which records input events, and ABC was a newly created channel. We might want to call Weather from ABC but returning the value from the system ledger, so sending transaction [ABC, Weather.worldMap()] would return the current weather map on the system ledger. But we might also want Weather to operates on local variables, so sending [ABC, Weather.temperature()] would return the current temperature on ABC channel, not on the system channel.

The example is contrived, but it illustrates that we need ability to call chaincodes cross channels either from a transaction or from another chaincode. This is especially useful for those chaincodes that we would want to deploy once and invoke them from any channel.

We need a model that allows programming to express the intended context where a called chaincode should operate. An obvious choice is to include a context which encapsulates the ledger (data object) that the chaincode would run against. However, a chaincode might have some initialization (init function) on the channel where it is deployed, so we need the ability to initialize the chaincode on every new channel that we want the chaincode to be able to operate locally. This should be done as a transaction on the new channel so that every peer can have the right state to operate the chaincode. We can implement this behavior with a "use" transaction (instead of deploy transaction) to say that we want to use the chaincode already deployed on the other channel rather than redeploying it.

At this point, we should set some convention to make the programming model clear:

1. Calling chaincode from a transaction always operates on the channel that the transaction is sent.

2. Calling a chaincode from another chaincode operates on the specified context. Default is the context of the caller (the context of the transaction), and in which case, the called chaincode operates like a local function invocation.
3. To simplify for 1.0 release, calling chaincode from another channel than the transaction channel is read-only (ie query values but not modify them), and the read-set won't include the values read from the called chaincode.

Restriction number 3 is important to simplify the model at this point for our current release development effort. The complexity would be cross-channel state transition without a transaction context. Had we allowed to modify state on system chain with a transaction on a private chain, then we would have to automatically add another transaction on the system channel to record the modification, and we would end up with 2 transactions on 2 separate channels that would require to commit atomically.

API

As described in the implementation section above, we will have a new gRPC API on the Peer and a new top level transaction type.

The API allows the App/SDK to notify the Peer to join the channel, which has been successfully created. The input to join channel API is the configuration block returned by the consensus service on the newly created channel. The Peer uses this block to set up the ledger associated with the channel.

The new transaction type is called CONFIGURATION. This type of transactions may be interpreted by both Orderer and Peer. Configuration transactions contain the entirety of the chain configuration, and a single configuration always stands on its own. This simple encoding will allow easy pruning in the future and simplify consumption of the configuration transaction. New configuration transactions must contain all previous configuration entries, and all new/modified configuration entries must be labeled with the sequence number and chain ID of their containing configuration envelope. Each configuration entry has an enumerated type, and unique (scoped by type) ID as well as a modification policy referenced by name. The ordering service will verify any configuration transaction against the existing configuration policies and reject it if any of the modification policies are not satisfied. Note that configuration policies are themselves expressed as configuration entries (of enumerated type 'Policy') and are therefore managed by the same mechanisms. See https://gerrit.hyperledger.org/r/#/c/1955/ for proto definitions of the configuration transaction payload (note, this is currently being reworked on top of the fabric next protos).

The SDK may provide further abstraction to the API. For example, it could provide 1 API, CreateChannel(listOfMemberCerts), which would carry out all 5 steps discussed in Create

Channel section. At the end, the SDK would invoke a callback on the application to signal the status of the channel creation.