Membership Service Providers & Access Control in Hyperledger/Fabric

Authors: Elli Androulaki, Angelo De Caro, Binh Nguyen, Alessandro Sorniotti, Murali Srinivasan, Jason Yellick

1. Terminology

A ***Blockchain network*** consists of the following entities:

- Application(s) network (can include peers and clients or only clients)
- Network of peers (if not part of the application network)
- Ordering service (can be decentralized or centralized)
- Set of protocols to facilitate the communication between the clients, peers and the ordering service, enabling the application to create one or more chains throughout the operation of the system and submit transactions to it, as well as manage the access to these chains.

For some of these terms we refer the reader to Hyperledger Fabric architecture.

Ordering service is the component of a Blockchain network that offers atomic broadcast services. It can create one or more "atomic broadcast *channels*" upon authenticated request from appropriately authorized entities. The internal structure of the ordering service may differ from one implementation to the other, as it can be a service offered by one entity (centralized version) or more entities, e.g., running BFT or CFT protocols (decentralized version). In any case, ordering service should come with its client that should expose the following functionalities to ordering service users, i.e., entities that leverage atomic broadcast functionalities of the ordering service:

- Reconfigure permissions of members concerning the ordering service functionalities (e.g., channel creation)
- Create and submit channel creation requests
- Reconfigure channel permissions, e.g., update the channel access policies

The ordering client is also equipped with reconfiguration mechanisms that are triggered by properly authenticated transactions advertised within application or system channel. E.g., if the ordering service is decentralized, the ordering service client would need to know & understand the policies associated to updates of that ordering service's membership mechanism, and other parameters, e.g., batch size, etc.

Channel is an atomic broadcast channel managed by the ordering service. One or more channels may be created within the ordering service after request from the application network (see below for application network definition).

Chain is bound to a channel and is the data structure that maintains the history and state of transactions advertised within a channel. In particular, a chain consist of

- The total order of transactions as provided by the channel orderers
- The state associated to chaincodes defined within the channel (i.e., the set of transactions of the channel that are valid)
- Membership and access control information for this chain's operations (read-access to transactions, write-access to the state, etc).

Membership service provider (MSP): A set of cryptographic mechanisms and protocols for issuing and validating certificates and identities in the Blockchain network. Identities issued in the scope of a membership service provider can be evaluated within that membership service provider's rules.

Clients are defined by their membership service provider type and respective (public) configuration. The latter for the default membership service provider includes a list of root CAs and intermediate CAs, and a list of administrators.

Network of peers are defined either the same way as clients. A peer network, like the client network, comes along with a description of the membership service provider and its configuration.

Application network is a broader term to cover entities that could be part of the application infrastructure. This could include a set of clients, and potentially also sets of peers. To understand this better, let's take the example of an auctioning service that is implemented as a chaincode that runs on peers A, and B. Now, let's assume that for the purpose of this use-case, the application service has implemented client-application software that runs on the end-user machine, and application server software that runs on the application server. Both cases leverage the client-sdk and are perceived by fabric network as clients. In this scenario the application owns the application-server side, as well as peers A, and B, that would install and instantiate/simulate the chaincodes controlled and submitted by that application.

Interaction between the ordering service, and application entities (clients, and peers): This requires that the ordering service nodes have installed some piece of software to be able to minimally process messages coming from the application network. This includes membership-related functionalities, i.e., check whether a certain entity is authorized to do certain things or not, and membership-update functionalities. We will call this ***appshim***. At the same time, as mentioned before, application network should run the ordering service client that is able to process ordering service specific messages that are advertised through a channel. We will call this ***osshim***.

Application chaincodes: Applications can deploy and invoke application chaincodes. At deploy time the owner of a chaincode (application) should specify policies that will govern the impact of invocations of that chaincode in that chaincode's state. These policies are known as endorsement policies. These policies usually contain:

- an endorsement specific part, i.e., defining how an endorser should "endorse" such invocations (included in the endorsing peer code in the form of ESCC), and
- a validation part, that is used by the committing peers to assess if the endorsement policy is satisfied for a certain chaincode invocation transaction (included in the committing peer code in the form of the validation system chaincode (VSCC)).

Thus, at deploy time, the deployer needs to specify the id of a pair of ESCC, VSCC this chaincode should adhere to. Fabric equips the peers with default ESCC and VSCC implementations that cover certain types of policies. Blockchain network provider may develop additional ESCC and VSCC to support specific endorsements that are not covered by the default. What is important is that VSCC given a transaction concludes to a decision in a deterministic manner.

Organizations**:** Logical entities or corporations that constitute the stakeholders of a Blockchain network installation. Members of such organization could be authorized by that organization's membership service providers to submit transactions to certain chains.

Members: Represent the end-users of the Blockchain network. Each organization may have one or more members and acts as the root of trust (MSP) for its members. For the default MSP used by fabric applications, each member has one long-term identity and can use its long-term identity to generate one or more ephemeral identities.

Mapping between an organization and an MSP. This mapping is not enforced by the fabric configuration. It is up to the consortium of entities building their Blockchain network to decide how to leverage the modular nature of membership service providers, and the ability for many of them to co-exist in the network.

The simplest case would be that there is one to one mapping between an organization and a membership service provider. In this case, root certificates of the organization MSP could carry that organization's name, that can be used as the MSP's identifier within a chain. If an organization has more than one subdivisions, e.g., the ones that appear in the OU field of an X.509 based identity, then the identities of these divisions should be considered using the identity of the MSP/Organization as namespace for it.

In the last section we elaborate on best practices associated to membership service providers and their mapping to organizations.

2. Membership Service Providers (MSPs) in a Blockchain network

2.1 Definition of a Membership Service Provider

A Blockchain network may be governed by one or more MSPs. An MSP can be logically defined by the following components:

1. An identity format, also known as certificate, and optionally the algorithm to generate one identity
2. A signing algorithm that utilizes the secret associated to an identity, and a message to produce a byte array that is also bound to the identity
3. A signature verification algorithm that takes as input an identity, a message, and a signature (byte array), and outputs “accept” if the signature bytes correspond to a valid signature of the input message assuming the information in the input identity; otherwise, the algorithm outputs “reject”
4. The set of rules that need to be satisfied by an identity for the identity to be considered valid for this MSP
5. A set of administration identities, that are authorized to change configuration parameters that are MSP-specific

From an implementation perspective, many MSPs are similar in items (1) and (2), and (3) but differ in (4) and (5). For the purpose of this document we will overload the MSP notation to refer to a unique tuple of algorithms:

<MSP.id, MSP.sign, MSP.verify, MSP.validateid, MSP.admin>

2.2 Examples

Examples of MSPs used by Peers. Here we describe how the above MSP features are instantiated in the case of two popular MSP scenarios on the peer side. Notice that peers in the network are agnostic to the identity issuing process, as their role is restricted to the endorsement of client proposals, client identity validation and client identity signature validation.

Example 1: Classic MSP. Identities (i.e., MSP.id) in this case have the form of standard X.509 certificates, that are signed by exactly one root CA. The certificate of the root CA, that can also be a commercial CA, is part of this MSP description. Signing and signature verification algorithms (i.e., MSP.sign and MSP.verify) are ECDSA-based or RSA-based depending on the key-material in the certificate.

Validation of an identity (i.e,. MSP.validateid) in this case involves:

- Verifying the correctness of the signature (chain) included in the identity (X.509 certificate) assuming the trusted root CA,
- Confirming that the identity is not within the list of identities that have been revoked;
- depending on the MSP implementation this can be done either by means of "MSP identity revocation list" (IdRL), or of "MSP identity white-list" (IdWL), that are updated regularly. In the Blockchain setting the IdRL/IdWL is passed as parameter at MSP setup time, and is updated through (properly authenticated) reconfiguration messages advertized through the Blockchain.

Admin of such an MSP (i.e., MSP.admin) can be the X.509 certificate of the administrator of that MSP, i.e., the entity that can update the root CA certificate this MSP is governed by. By default, IdRLs or IdWLs are only updatable by the root CA itself, or the administrator.

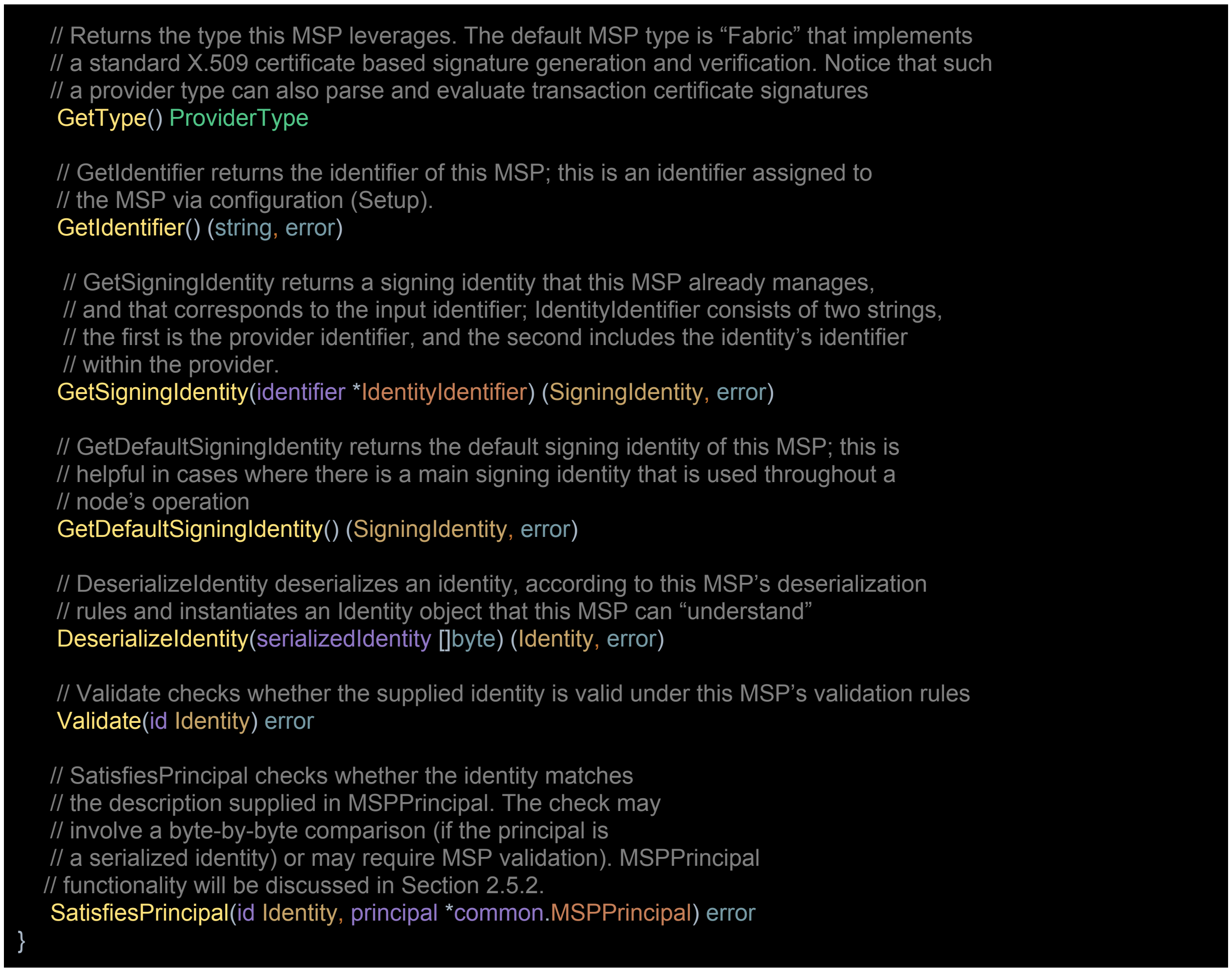
Example 2: MSP allowing for cross-signed certificates. As in the previous example, identities in this case also have the form of an X.509 certificates. However, in this case, the MSP client leverages standard X.509 certificate structure to accommodate within a single certificate signatures of one or more root CAs. In this case, the MSP is parameterized with a list of trusted CAs (by means of standard X.509 certificates) and the threshold number of these whose signature should appear in a valid identity. As before, validation of certificates, apart from the signature validation relates to IdRLs/IdWLs, which can be advertised by either the administrator, or the identity issuer CA.

Signing, and signature verification algorithm is an ECDSA based one or RSA based one depending on the key-material inside the certificate.

2.3 Generic interfaces for a fabric platform MSP

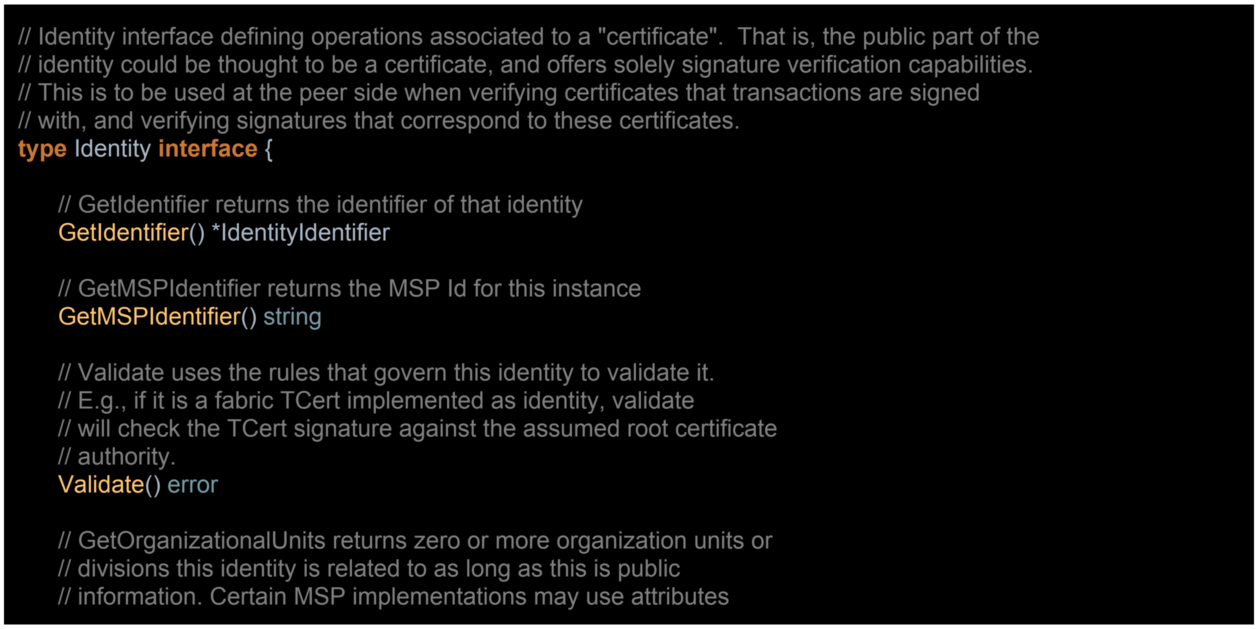
Reflecting the definition of an MSP from Section 2.1, we define generic interfaces for a membership service provider. These interfaces are shown in Figure1, and are strongly coupled with the notion of identity. Identity, that reflects the notion of publicly verifiable certificate is also defined in a generic way, through the interfaces described in Figure 2. Finally, Figure 3 depicts a SigningIdentity interface, i.e., an Identity with signing capabilities.

```
// MSP is the minimal Membership Service Provider Interface to reflect Membership
// Service Provider needs to be used on the peer and ordering node side. Notice that
// on these nodes, MSP is needed for verifying purposes.
type MSP interface {

    // Setup the MSP instance according to configuration information defined through
    // an MSPConfig data structure. This struct is generic enough so as to capture
    // any MSP configuration
    Setup(config *MSPConfig) error
```



```
    // Returns the type this MSP leverages. The default MSP type is “Fabric” that implements
    // a standard X.509 certificate based signature generation and verification. Notice that such
    // a provider type can also parse and evaluate transaction certificate signatures
     GetType() ProviderType

     // GetIdentifier returns the identifier of this MSP; this is an identifier assigned to
     // the MSP via configuration (Setup).
     GetIdentifier() (string, error)

      // GetSigningIdentity returns a signing identity that this MSP already manages,
      // and that corresponds to the input identifier; IdentityIdentifier consists of two strings,
      // the first is the provider identifier, and the second includes the identity’s identifier
      // within the provider.
     GetSigningIdentity(identifier *IdentityIdentifier) (SigningIdentity, error)

     // GetDefaultSigningIdentity returns the default signing identity of this MSP; this is
     // helpful in cases where there is a main signing identity that is used throughout a
     // node’s operation
     GetDefaultSigningIdentity() (SigningIdentity, error)

     // DeserializeIdentity deserializes an identity, according to this MSP’s deserialization
     // rules and instantiates an Identity object that this MSP can “understand”
     DeserializeIdentity(serializedIdentity []byte) (Identity, error)

     // Validate checks whether the supplied identity is valid under this MSP’s validation rules
     Validate(id Identity) error

    // SatisfiesPrincipal checks whether the identity matches
    // the description supplied in MSPPrincipal. The check may
    // involve a byte-by-byte comparison (if the principal is
    // a serialized identity) or may require MSP validation). MSPPrincipal
   // functionality will be discussed in Section 2.5.2.
    SatisfiesPrincipal(id Identity, principal *common.MSPPrincipal) error
}
```

Figure1. Description of the generic platform MSP interface.



```
// Identity interface defining operations associated to a "certificate".  That is, the public part of the
// identity could be thought to be a certificate, and offers solely signature verification capabilities.
// This is to be used at the peer side when verifying certificates that transactions are signed
// with, and verifying signatures that correspond to these certificates.
type Identity interface {

     // GetIdentifier returns the identifier of that identity
     GetIdentifier() *IdentityIdentifier

     // GetMSPIdentifier returns the MSP Id for this instance
     GetMSPIdentifier() string

     // Validate uses the rules that govern this identity to validate it.
     // E.g., if it is a fabric TCert implemented as identity, validate
     // will check the TCert signature against the assumed root certificate
     // authority.
     Validate() error

     // GetOrganizationalUnits returns zero or more organization units or
     // divisions this identity is related to as long as this is public
     // information. Certain MSP implementations may use attributes
```

```
    // that are publicly associated to this identity, or the identifier of
    // the root certificate authority that has provided signatures on this
    // certificate.
    // Examples:
    //  - if the identity is an x.509 certificate, this function returns one
    //    or more string which is encoded in the Subject's Distinguished Name
    //    of the type OU
    GetOrganizationalUnits() []string

    // Verify a signature over some message using this identity as reference
    Verify(msg []byte, sig []byte) error

    // Serialize converts an identity to bytes
    Serialize() ([]byte, error)

    // SatisfiesPrincipal checks whether this instance matches
    // the description supplied in MSPPrincipal. The check may
    // involve a byte-by-byte comparison (if the principal is
    // a serialized identity) or may require MSP validation
    SatisfiesPrincipal(principal *common.MSPPrincipal) error
}
```
Figure 2 Description of a generic Identity interface devised for Fabric platform needs.

Identities equipped with the secret signing information that correspond to their public key, are called in our infrastructure ***SigningIdentities***. In the case of an X.509 based MSP, Identity would be instantiated as an X.509 certificate. ***SigningIdentity*** in this case, would also carry a reference to the signing key of the certificate's public key. SigningIdentity interface is described in Figure 3.

```
// SigningIdentity is an extension of Identity to cover signing capabilities. E.g., signing identity
// should be requested in the case of a peer who wishes to sign proposal responses. A form
// of signing identity is also used at the client side who would sign proposals and transactions.
type SigningIdentity interface {

    // Extends Identity
    Identity

    // Sign the input message using the singing identity's signing key
    Sign(msg []byte) ([]byte, error)

    // Removed SignOpts, GetAttributeProof for the same reasons as for VerifyOpts and VerifyAttributes

    // GetPublicVersion returns the public parts of this identity. In a signing identity corresponding
    // to X.509 certificates, GetPublicVersion would output the Identity object representing the
    // actual X.509 certificate
    GetPublicVersion() Identity

 }
```
Figure 3. Description of a generic interface for a signing identity.

2.4 Coupling node signing abilities with a (local) MSP

Orderers, and peers need to be equipped with signing abilities. To do so, the administrator of a node needs to specify at node setup time the configuration of the MSP that would carry the signing identity of the peer or orderer. As the MSP instance included here is created solely to instantiate the node's signing identity, we refer to this MSP by ***SignerMSP***. The latter is only possible to be updated manually by that node's administrator, and can naturally vary from node to node. For simplicity and for V.1, to setup SignerMSP, and assuming the default MSP type for fabric, the administrator is requested to copy to dedicated location in the node's file system four sets of files:

a. cacerts: PEM files containing the root authority certificates of the MSP
b. admincerts: PEM files containing the administrators' certificates of this MSP
c. keystore: PEM files containing the signing private key of the node
d. signcerts: PEM encoded certificate files corresponding to the identity of the node

In particular, the node admin is required to modify the setup .yaml file with the information depicted in the figure below. First of all, the BC crypto service provider of the node is configured, where it needs to be determined whether a software (SW) or HSM based CSP is used to store the key-material of the node. In the example below a SW provider is configured with the location of where the key material of the peer is to reside. In addition, for the default MSP case, the node is to retrieve from its .yaml file the location of the msp-related files (cacerts, admincerts, intermediatecas, and crls) are stored (parameter mspConfigPath), and identifier of the node's MSP (localMspId).

```yaml
# BCCSP (Blockchain crypto provider): Select which crypto implementation or
# library to use
BCCSP:
  Default: SW
  SW:
    # TODO: The default Hash and Security level needs refactoring to be
    # fully configurable. Changing these defaults requires coordination
    # SHA2 is hardcoded in several places, not only BCCSP
    Hash: SHA2
    Security: 256
    # Location of Key Store, can be subdirectory of SbftLocal.DataDir
    FileKeyStore:
      # If "", defaults to 'mspConfigPath'/keystore
      KeyStore:

# Path on the file system where peer will find MSP local configurations
mspConfigPath: msp/sampleconfig

# Identifier of the local MSP
# ----!!!!IMPORTANT!!!-!!!IMPORTANT!!!-!!!IMPORTANT!!!!----
# Deployers need to change the value of the localMspId string.
# In particular, the name of the local MSP ID of a peer needs
# to match the name of one of the MSPs in each of the channel
# that this peer is a member of. Otherwise this peer's messages
# will not be identified as valid by other nodes.
```

```
localMspId: DEFAULT
```

2.5 Coupling chain participation with MSPs

The genesis block of a chain must contain the specification (description) of the MSPs that govern the chain participants' identities. If an MSP covers multiple chains, it is important that we keep the state of that MSP on each chain. This is to avoid reconfiguration inconsistency attacks that can be caused by reconfiguration transactions of the organization's MSP arriving in each chain in a different order.

MSPs defined in the context of a chain or channel would enable the orderers and peers to authenticate chain transaction signers, endorsers, and/or creators of requests for chain/channel creation/termination, channel broadcast & delivery and others.

In particular, MSPs specified in the orderer system channel would allow the specification of the policies governing channel readers (to authenticate & validate channel delivery requests), writers (to authenticate & validate channel broadcast requests), chainCreators (to evaluate chain creation requests), and admins (to authenticate & validate channel reconfiguration requests). MSPs specified in an application chain or channel allow the specification of policies that govern chain readers, writers, admins, and chaincodeAdmins (to authenticate and validate chaincode instantiation requests). Evidently, MSPs within a chain have a **verifier role**, that comes in contrast to the **signer role** that local MSP(s) mean to offer**.** More specifically, peers and orderers are required to setup MSPs in the context of a channel (for orderer system channel) or chain to be able to authenticate transactions and configuration-related requests, and carry **no signature generation responsibility/ability**.

Clearly, peers and orderers need to be able to verify signatures that correspond to identities issued by multiple MSPs. To facilitate this, Hyperledger fabric introduces the concept of an MSP manager. In particular, an MSPManager interface is the fabric component that would instantiate one or more MSPs at chain setup time (which is also relevant for the orderer channel setup that takes place at orderer bootstrap), and use these to validate transaction signatures transparently to the rest of the code. MSPManager interface brings in two important advantages to the fabric.

- Pluggability of MSPs
- Support for multiple MSP providers simultaneously
- Hiding the complexity of internal policies of a single MSP and its architecture from the rest of the MSPs in the Blockchain network.

MSPManager uses the information from the configuration block of the chain (i.e., the Genesis block) to instantiate the MSPs, as shown in Figure 4.

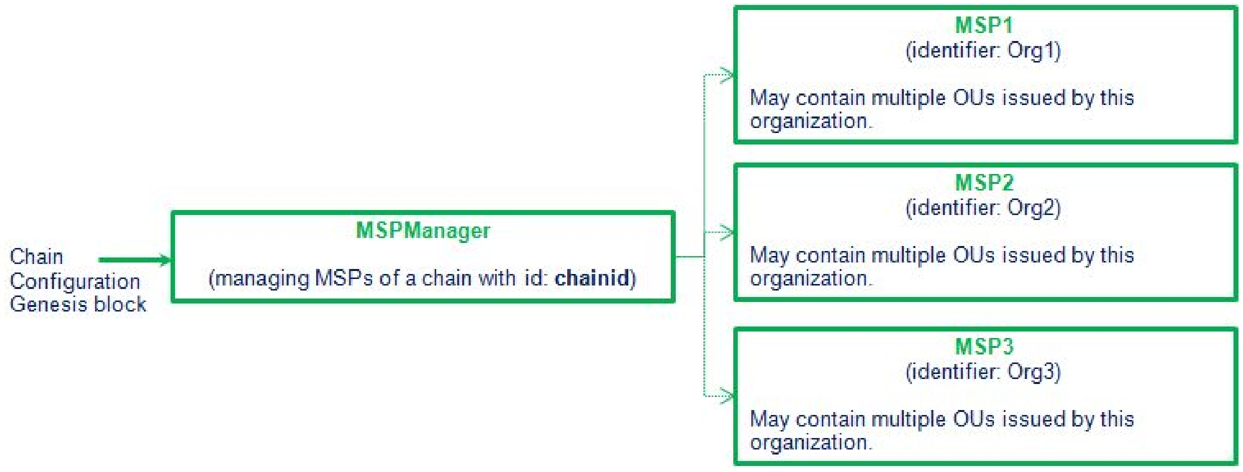

Chain Configuration Genesis block
MSPManager
(managing MSPs of a chain with id: chainid)
MSP1
(identifier: Org1)
May contain multiple OUs issued by this organization.
MSP2
(identifier: Org2)
May contain multiple OUs issued by this organization.
MSP3
(identifier: Org3)
May contain multiple OUs issued by this organization.

Figure 4. Setup flow from chain configuration components, to MSPManager setup, and individual MSP setup.

The MSPManager exposes an interface to the rest of the fabric-code that is simple for components to integrate. This interface is in Figure 5.

```
// MSPManager is an interface defining a manager of one or more MSPs. This essentially acts
// as a mediator to MSP calls and routes MSP related calls to the appropriate MSP. This object
// is immutable, it is initialized once and never changed.
type MSPManager interface {

    // DeserializeIdentity deserializes an identity.
    // Deserialization will fail if the identity is associated to
    // an msp that is different from this one that is performing
    // the deserialization.
    DeserializeIdentity(serializedIdentity []byte) (Identity, error)


    // Setup the MSP manager instance according to configuration information
    Setup(msps []MSP) error

    // GetMSPs Provides a list of Membership Service providers
     GetMSPs() (map[string]MSP, error)
}
```

Figure 5. In this Figure we can see the definition of MSPManager interface. **Notice**, that in the code, "DeserializeIdentity" function is part of a "IdentityDeserializer" interface, that MSPManager extends. However, for simplicity of description we list directly the function inside MSPManager interface.

2.5.1 Setup of chain MSPs

An MSPManager instance is created for every new chain that is created through the "Setup" method, that takes as input a list of MSP configuration objects, i.e., "msp.MSPConfig". The proto message governing the structure of msp.MSPConfig is depicted in Figure 6.

```
// MSPConfig collects all the configuration information for
// an MSP. The Config field should be unmarshalled in a way
// that depends on the Type
message MSPConfig {
   // Type holds the type of the MSP; the default one would
   // be of type FABRIC implementing an X.509 based provider
   int32 Type = 1;

   // Config is MSP dependent configuration info
   bytes Config = 2;
}
```

Figure 6. Protocol message for generic configuration of an MSP.

Marshalling of field “Config” (and hence the way to be unmarshalled) is defined by the value of field “Type”. For the default case where the default MSP type (Farbric) is used and being configured, “Config” has the content shown in Figure 7.

```
// FabricMSPConfig collects all the configuration information for a Fabric MSP.
// Here we assume a default certificate validation policy, where any certificate
// signed by any of the listed rootCA certs would be considered as valid
// under this MSP. This MSP may or may not come with a signing identity. If
// it does, it can also issue signing identities. If it does not, it can only
// be used to validate and verify certificates.
message FabricMSPConfig {
   // Name holds the identifier of the MSP; MSP identifier is chosen by the
   // application that governs this MSP. For example, and assuming the default
   // implementation of MSP, that is X.509-based and considers a single Issuer,
   // this can refer to the Subject OU field or the Issuer OU field.
   string Name = 1;

   // List of root certificates associated
   repeated bytes RootCerts = 2;

   // Identity denoting the administrator of this MSP
   repeated bytes Admins = 3;

   // Identity revocation list
   repeated bytes RevocationList = 4;

   // SigningIdentity holds information on the signing identity
   // this peer is to use, and which is to be imported by the
   // MSP defined before
   SigningIdentityInfo SigningIdentity = 5;
}
```

Figure 7. Protocol message for configuration of the default MSP.

The configuration transaction that constitutes the genesis transaction of the chain, contains a list of one or more MSPs that would govern the chain. An example of the content related to MSPManager setup content with three MSPs of type “Fabric” is depicted in Figure 8.

```
{
 "MSPManager":
  [
  {
   "Type":0,
   "Config":{
     "Name":"org1",
     "RootCerts":["org1-identity1bytes","org1-identity2bytes","org1-identity3bytes"],
     "Admins":["adminidOrg1bytes"]
    }
  },
  {
   "Type":0,
   "Config":{
     "Name":"org2",
     "RootCerts":["org2-identity1bytes","org2-identity2bytes","org2-identity3bytes"],
     "Admins":["adminidOrg2bytes"]
    }
  },
  {
   "Type":0,
   "Config":{
     "Name":"org3",
     "RootCerts":["org3-identity1bytes","org3-identity2bytes","org3-identity3bytes"],
     "Admins":["adminidOrg3bytes"]
    }
  }
 ]
}
```

Figure 8. Example of configuration content included in a chain genesis block. We emphasize that in this Figure, we use json format for simplicity of presentation.

Configuration information of a (simple) MSP: To setup an MSP one would need the type of MSP be supported by that node's executable, i.e., the MSP type to be among the ones already defined and implemented in the node's peer or orderer executable. Each MSP is required to implement the interface presented in Figure1 and can be configured with instructions provided at chain genesis time. The information included in the genesis block of a chain for each MSP, is MSP-type-dependent. For the the default fabric MSP type, MSP configuration includes the following parts:

- A name to identify the MSP within the chain/Blockchain network; In the above example we use "org1", "org2", and "org3" as MSPIDs of the MSPS that govern the chain. This is because we follow the convention of Section 1. Terminology, where each MSP represents an organization.
- The type of MSP this provider uses; the provider mentioned in the example below is of type "Fabric", which aims to be the (default) MSP implementation in fabric. Alternative implementations may require cross-signed certificates from two or more root CAs etc, and would have a different type reference.
- A set of parameter values that include
    - a set of identities/certificates to constitute the root of trust, or CA server(s), e.g.,

      `"RootCerts":["org1-identity1bytes","org1-identity2bytes","org1-identity3bytes"],`

      Notice that all these identities in the default case have the form of plain X.509 certificates in PEM format.

- the admin authorized to perform updates to this MSP parameters/configuration (consisting primarily on the root of trust/ root CA certificates, and CRLs); for simplicity we show a single certificate/identity as admin:

```
"Admins":["adminidOrg1bytes"],
```

- the current list of revoked certificates in ***bytes***, which, if omitted, an empty list is implied. Notice that exact structure of revocation identities depends on the MSP type. However, one could represent it in the configuration file as an array of bytes.

This structure is depicted in the protocol message of Figure 7.

2.5.2 MSP principals

MSP principals constitute the building blocks of definition of access control policies within a chain or channel. In particular, they are used to describe one or more identities that share a common feature that a specific MSP manages.

In the simplest case, an MSPPrincipal can be the group of identities that are valid under a specific MSP's identity validation logic, or the administrator(s) of an MSP. In this case we say that the MSPPrincipal is defined using the role of the identity inside that MSP as a classification criteria.

Alternatively MSPPrincipal, can be tuned to define a specific (serialized) identity of an MSP configuration (MSPConfig object from above). In this case we say that the MSPPrincipal is defined using an identity based classification.

Finally, MSPPrincipal can be defined as a set of identities that are valid under a specific MSP's configuration, and belong to a certain organization unit. In this case we say that the MSPPrincipal is defined using an OrganizationUnit based classification.

In the future, MSPPrincipal, can be tuned to describe a set of identities that are valid under a specific MSP's configuration (MSPConfig object from above) and have a certain attribute in common. In this case we would say that the MSPPrincipal is defined using an attribute based classification.

MSPPrincipal structure is defined in Figure 9.

```
// MSPPrincipal aims to represent an MSP-centric set of identities.  In particular,
// this structure allows for definition of
//  - a group of identities that are member of the same MSP
//  - a group of identities that are member of the same organization unit
//    in the same MSP
//  - a group of identities that are administering a specific MSP
//  - a specific identity
// Expressing these groups is done given two fields of the fields below
//  - Classification, that defines the type of classification of identities
//    in an MSP this principal would be defined on; Classification can take
//    three values:
//     (i)  ByMSPRole: that represents a classification of identities within
//          MSP based on one of the two pre-defined MSP rules, "member" and "admin"
//     (ii) ByOrganizationUnit: that represents a classification of identities
//          within MSP based on the organization unit an identity belongs to
//     (iii)ByIdentity that denotes that MSPPrincipal is mapped to a single
//          identity/certificate; this would mean that the Principal bytes
//          message
//  -Principal that contains either a serialized MSPRole message, or a serialized
// MSPOrganizationUnit message, or a serialized SerializedIdentityMessage depending
// on the Classification value..

message MSPPrincipal {

   enum Classification {
       ByMSPRole = 0;  // Represents the one of the dedicated MSP roles, the
       // one of a member of MSP network, and the one of an
       // administrator of an MSP network
       ByOrganizationUnit = 1; // Denotes a finer grained (affiliation-based)
       // groupping of entities, per MSP affiliation
       // E.g., this can well be represented by an MSP's
       // Organization unit
       ByIdentity  = 2;    // Denotes a principal that consists of a single
       // identity
   }

   // Classification describes the way that one should process
   // Principal. An Classification value of "ByOrganizationUnit" reflects
   // that "Principal" contains the name of an organization this MSP
   // handles. A Classification value "ByIdentity" means that
   // "Principal" contains a specific identity. Default value
   // denotes that Principal contains one of the groups by
   // default supported by all MSPs ("admin" or "member").
   Classification PrincipalClassification = 1;

   // Principal completes the policy principal definition. For the default
   // principal types, Principal can be either "Admin" or "Member".
   // For the ByOrganizationUnit/ByIdentity values of Classification,
   // PolicyPrincipal acquires its value from an organization unit or
   // identity, respectively.
   bytes Principal = 2;
}


// OrganizationUnit governs the organization of the Principal
// field of a policy principal when a specific organization unity members
// are to be defined within a policy principal.
message OrganizationUnit {

   // MSPIdentifier represents the identifier of the MSP this organization unit
   // refers to
   string MSPIdentifier = 1;

   // OrganizationUnitIdentifier defines the organization unit under the
   // MSP identified with MSPIdentifier
   string OrganizationUnitIdentifier = 2;

}
```

```
// MSPRole governs the organization of the Principal
// field of an MSPPrincipal when it aims to define one of the
// two dedicated roles within an MSP: Admin and Members.
message MSPRole {

  // MSPIdentifier represents the identifier of the MSP this principal
  // refers to
  string MSPIdentifier = 1;

  enum MSPRoleType {
     Member = 0; // Represents an MSP Member
     Admin  = 1; // Represents an MSP Admin
  }

  // MSPRoleType defines which of the available, pre-defined MSP-roles
  // an identiy should posess inside the MSP with identifier MSPidentifier
  MSPRoleType Role = 2;

}
```
Figure 9. MSPPrincipal related protocol messages.

MSP interface is to be extended with the following function

```
SatisfiesPrincipal(id Identity, principal *MSPPrincipal) error
```

That would return true if the list of identities passed as parameters satisfy the provided MSPPrincipal.

Note: This function can be extended to include signatures if the only way to ensure that an identity is valid w.r.t. A certain MSP is through signatures. An example of such case is IBM Identity Mixer.

2.5.3 Reconfiguration of a channel MSP

A channel's MSPs are reconfigured through configuration blocks that are submitted to the channel and committed to that channel. More information on this is to be added soon.

As happens with the reconfiguration of parameters included in other configuration items, reconfiguration of an MSP includes the following steps:

1. Validate the authorization of the signers of configuration items included in the configuration transaction/block to reconfigure the signed configuration items using chain's ConfigManager (+jason.anonymous@gmail.com to add more information as needed here :)). For the specific case of MSPs we discuss later how this takes place.
2. Create a new chain object, and a fresh MSPManager using the MSPManager Setup function and the list of MSPConfig objects created after parsing the Configuration Block & transaction.
3. Do the same for the rest of components that configuration items define, and have the new chain object point to the most recent instances of the reconfigured chain's components.

4. After ensuring that the creation of each of the new components completes successfully, substitute chain pointers for MSPManager and other components to point to the freshly generated ones. Otherwise abandon the operation.
5. Destroy the obsolete chain and linked components.

More information on how a chain reconfiguration occurs is described in the last section.

3.Channel Access Control

In the genesis block of a channel, the following policies need to be defined:

- The ***readers*** of the chain or “*channelReaders”*, i.e., the policy to authenticate any request associated to read access to the transactions of a chain. For example this policy could define the identity or groups of identities (MSPPrincipals) that are allowed read-access to the chain; the same policy governs access to ordering service delivery requests and event access requests for that chain.
- The ***writers*** of the chain or “*channelWriters”*, i.e., the policy to authenticate any request associated to submitting transaction to a chain. For example, this policy may include the identities or groups of identities (MSPPrincipals) that should be allowed to submit transactions to the chain. In particular this policy governs the set of signatures that need to be acquired in a transaction for the latter to be allowed to be submitted to the chain. The same policy governs permissions of clients to submit proposals to endorsers concerning that chain.
- The ***admins*** of the channel or “*channelAdmins”*, i.e., the policy to authenticate any request associated to reconfiguration of specific channel parameters. Admins may determine the identities or groups of identities (MSPPrincipals, and the way to combine them) that have admin access to the chain configuration. Such policies specify the (combination of) MSPPrincipals that should sign chain-specific reconfigurations for the reconfiguration to be applied.
- The ***chaincodeAdmins*** of the chain, i.e., the policy to authenticate any request associated to chaincode deployment within a chain. In the default policy case, chainDeployers would determine the identities or groups of identities (MSPPrincipals, and the way to combine them) that have permission to create or upgrade chaincodes on that chain.

For the orderer system channel in particular the application needs to specify another policy called “**chainCreators**”, that would be used to evaluate chain creation requests by the orderers. Again chainCreators policy definition would be based on MSPPrincipals of MSPs that have been defined on the system channel.

Initially within fabric, we plan to provide default policies, that are defined with the use of the policy framework in coauthdsl package. That is,

- For read-permissions we allow that the chain content is readable within a selection of MSP networks that govern the chain. In the genesis block, this is done by:
    - Including in the genesis transaction a signed configuration item of type “Policy” whose value is the actual policy definition. In our example, we would use an OR (SignaturePolicy) type of policy of two MSP principals, one defining the members of org2, and one defining the members of org3. Let this configuration item be referred using key *channelReadersPolicy.* Its content is schematically described as follows:

      {“classification":"bymsprole","principal":{"mspid":"org2","msproletype": "member"}}
      OR
      {“classification":"bymsprole","principal":{"mspid":"org3","msproletype": "member"}}

    - Including a (signed) configuration item of type “Chain”, referenced by identifier “channelReaders” and takes as value the policy identifier channelReaderPolicy. Table 2 shows an example of how read policy of a chain/channel is defined in a genesis block.
- For write permissions we allow that the members of a selection of MSP networks that govern the chain are allowed to submit transactions to the chain. As in the case to channelReaders, this is reflected in the chain genesis transaction through the use of two configuration items:
    - A configuration item of type “Policy” whose value is the actual policy definition. In our example, we would define this policy as an OR (SignaturePolicy) type of policy of two MSP principals, one defining the members of org2, and one defining the members of org3. Let this configuration item be referred using key *channelWritersPolicy.* Its content is schematically described as follows:

      {“classification":"bymsprole","principal":{"mspid":"org2","msproletype": "member"}}
      OR
      {“classification":"bymsprole","principal":{"mspid":"org3","msproletype": "member"}}
    - Including a (signed) configuration item of type “Chain”, referenced by identifier “channelWriters” and takes as value the policy identifier channelWritersPolicy. Table 2 shows an example of how write policy of a chain/channel is defined in a genesis block.
- As admin policies we allow that the chain is re-configurable as long as reconfiguration requests on chain items are signed by the admins of all MSPs (via an external tool) that govern that chain. This is expressed in the genesis block in the same as the two previously described policies and using the following as content of the policy:

  {“classification":"bymsprole","principal":{"mspid":"org2","msproletype": "admin"}}
  AND
  {“classification":"bymsprole","principal":{"mspid":"org3","msproletype": "admin"}}

  For the purpose of this document we will refer to the related policy configuration item and chain parameter by channelAdminsPolicy, and channelAdmins respectively as reflected in Table 3.

As chaincodeAdmins policy of a chain we use the same methodology as the one listed before.

Definition of these policies is done through cauthdsl framework of fabric codebase that leverage MSPPrincipals, and satisfiesPrincipal(***id, principal***) function of MSP interface where ***principal*** denotes the MSPPrincipal of MSP's network,, and ***id*** is an identity.

For chain creation policies, that concern solely the system channel, the orderer defines chain creation policies within its orderer system chain. These policies may be arbitrary, and restrict the default policy further. For instance, the chain creation policy might require that a minimum of 4 parties are involved in a new chain, or that one particular party be involved in all new chains etc. The characteristics of this policy are negotiated with the orderer admins when an ordering service is configured to accept chain creation requests from a group of MSPs. The default policy in this case would require the signature of all application MSPs involved in the new chain.
An example of the structure of orderer system channel genesis is depicted in Table 2.

4. Using the default MSP: Best Practices

In this section we elaborate on best practices for membership service providers configuration for v1 in commonly met scenarios.

1. **Mapping between organizations/corporations and membership service providers**

We recommend that there is a one-to-one mapping between organizations and MSPs. If a different mapping ration is chosen the following needs to be to considered:

- One organization employing various MSPs. This corresponds to the case of an organization including a variety of divisions each represented by its membership service provider, either for management independence reasons, or for privacy reasons. One needs to know in this case that peers can only be owned by a single MSP, and will not recognize peers with identities from other MSPs as peers of the same organization. Implications of this is that peers may share organization-scoped data with a set of peers that are members of the same subdivision, and not with the full set of organizations.
- Multiple organizations using a single MSP. This corresponds to a case of an organization consortium whose membership architecture of individual organizations is compatible. One needs to know here that peers would propagate organization-scoped messages to the peers that have identity under the same MSP regardless of whether they belong to the same actual organization. This is a limitation of granularity of MSP definition, and/or of peer's configuration.

In future versions of fabric this can change as we move towards (i) an identity channel that contains all membership related information of the network, (ii) peer notion of "trust-zone" being configurable, a peer's administrator specifying at peer setup time whose MSP members should be treated by peers as authorized to receive "organization"-scoped messages.

2. On organization has different divisions (say organizational units), to which it wants to grant access to different channels.

Two ways to deal with this:

1. Define one MSP to accommodate membership for all organization's members. Configuration of that MSP would consist of a list of root CAs, intermediate CAs and admin CAs, and membership identities would include the organizational unit (OU) the member belongs to. Policies can then be defined to capture members of a specific OU, and these policies can be read/write policies of a channel or even chaincode administrators. Limitation of this approach, is that gossip peers would still consider peers under their local MPS as members of the same organization, and therefore share state / ledger related information with these peers even though they belong to an OU that is forbidden access to a certain channel.
2. Defining one MSP to represent each division, i.e., specify for each division a set of certificates, for root CAs, intermediate CAs, and admin Certs, such that there is no common certification path across MSPs. Here the disadvantage is the management of more than one MSPs instead of one, but this circumvents the issue present in approach (1).
3. (available in the future) Define one MSP for each division by leveraging an OU extension of the MSP configuration.

3. Separating clients from peers of the same organization.

In many cases it is required that the "type" of an identity is retrievable from the identity itself,e.g., it may be needed that endorsements are guaranteed to have derived by peers, and not clients or nodes acting solely as orderers.

There is limited support for such requirements in v1.0. That is to allow for this separation currently, we would be required to create a separate intermediate CA, one for clients and one for peers, and configure two different MSPs one for clients, and one for peers. Channels this organization should be accessing, would need to include both MSPs, while endorsement policies will leveraging only the MSP that refers to the peers. This would ultimately result into the organization being mapped to two membership service provider instances, and would have certain consequences into the way peers and clients interact:

- Gossip would not be drastically impacted as all peers of the same organization would still belong to one MSP
- Peers allow the restrict the execution of certain system chaincodes to MSP-principals, e.g., local MSP based policies. For example, peers would only execute "JoinChannel" request if the request is signed by the admin of their local MSP who can only be a client (end-user should be sitting at the origin of that request). We can remedy this, if we exclude administrators of peer MSP to have a dual role.
- At the first phase peers would authorize event registration requests based on membership of request originator within their local MSP. Clearly if the originator of the request belongs to a different MSP, e.g., a client MSP, the peer would reject the request. This is to be changed soon, as registration requests are soon to be served using the check of whether identities are part of the readers of all the channels a peer has joined.

In the (near) future policy language is to be extended to support definition of organizational unit based policies. E.g., it would be possible to restrict endorsement policies to owners of identities that are members of a certain organizational unit of an MSP. This, would allow for another way of separating clients a peers of a given organization, e.g., by allocating identities of each type to different organizational unit.

Note: This section needs to be moved to a different document and be updated with the most recent re-configuration framework.

In this section we give an end-to-end example of how peers/orderers and chains MSPs are setup and what is the configuration information each of them uses to bootstrap its operation.

For the purpose of this example we assume that four organizations Org1, Org2, Org3, and Org4 have decided to deploy a Blockchain network using an ordering service that would consist of orderer nodes owned by Org1, and Org2. Notice that Org1 is only to contribute orderers in this network. Thus, for the purpose of our example:

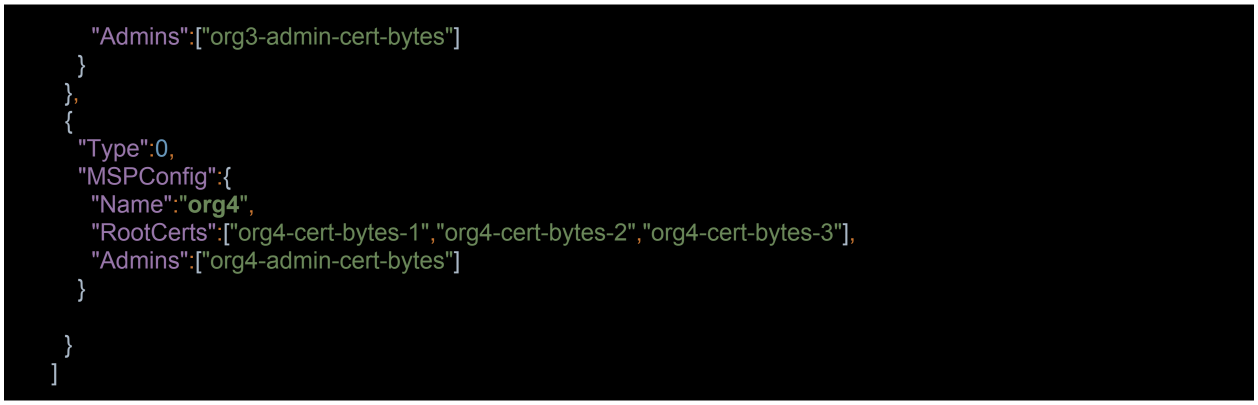
- Org1: orderers
- Org2: clients, peers, orderers
- Org3: clients, peers
- Org4: clients, peers

Step 1: The applications of each organization of the peer network (Org2, Org3, Org4) decide on the form of the Blockchain network genesis configuration from the application perspective, i.e., they decide on

- the MSP that represents each organization, each MSP's configuration, and each MSP identifier throughout the chain.
- the list of admins of the chain on behalf of the application

In particular, the applications of the three organizations, Org2, Org3, Org4, agree on the list of configuration components for each MSP of theirs, depicted in Figure 10 (in JSON for presentation simplicity).

```
"msplist":[
 {
   "Type":0,
   "MSPConfig":{
     "Name":"org2",
     "RootCerts":["org2-cert-bytes-1","org2-cert-bytes-2","org2-cert-bytes-3"],
     "Admins":["org2-admin-cert-bytes"]
   }
  },
  {
   "Type":0,
   "MSPConfig":{
     "Name":"org3",
     "RootCerts":["org3-cert-bytes-1","org3-cert-bytes-2","org3-cert-bytes-3"],
```



```
      "Admins":["org3-admin-cert-bytes"]
     }
   },
   {
     "Type":0,
     "MSPConfig":{
      "Name":"org4",
      "RootCerts":["org4-cert-bytes-1","org4-cert-bytes-2","org4-cert-bytes-3"],
      "Admins":["org4-admin-cert-bytes"]
     }

   }
 ]
```

Figure 10. Example of list of MSPs included in the Blockchain network description on application behalf in json. We emphasize that though json is not used in practice, we present this information here in json for simplicity of presentation of the configuration content.

Notice that for simplicity, the identifier of the MSP associated to organization Org1 (2, 3) has been chosen to be “org1” (org2, org3, respectively). This information is submitted to the orderers (agnostic to whether it is orderer administrator or kafka cluster admin or some system channel) by the application.

Disclaimer: We emphasize that for simplicity of presentation in the previous figure **we ignore the organization of MSP configuration** in configuration items inside a genesis block. It is assumed that each MSP’s configuration data, is marshalled into a separate ConfigurationItem that is signed by identities such that the defined modification policy is satisfied. (Specifically for genesis blocks we can simply ignore the validation of the signatures in ConfigurationItems against the modification policies of these items). As we will see later, checking the signature against established modification policies is imperative in re-configuration blocks that have the same structure as genesis blocks.

Step 2: The ordering service administrator configures the orderers. This configuration consists of two sets of data:

1. Local configuration of each orderer that includes setup of the crypto service provider, key-manager, and the node’s ***SignerMSP***, and any consensus related local information (e.g., where certain files are to be stored, etc).This is depicted in Figure 11***.*** The configured MSP is only possible to be updated manually by that orderer’s administrator, and can naturally vary from orderer to orderer. For simplicity and for V.1, to setup the local MSP, and assuming our default MSP type the administrator is requested to copy to dedicated location in orderer’s file system four sets of files:
    e. cacerts: PEM files containing the root authority certificates of the MSP
    f. admincerts: PEM files containing the administrators’ certificates of this MSP
    g. keystore: PEM files containing the signing key of the orderer
    h. signcerts: PEM files corresponding to the singing identity of the orderer

   Disclaimer: Currently SignerMSP is not used for Signature verification, and hence updating the cacerts or admincerts of this MSP is not of crucial importance. However this is to be revisited as gossip communication may require frequent updates of these values.

2. Configuration parameters that are to be in common by all orderers participating in the system, and that is imperative that are consistently updated across all orderers of the ordering service. These parameters are organized in a structure that would constitute the system / orderer channel genesis configuration, that is depicted in Figure 12 in json format, and include the following parameters:
    - The list of MSPs that are to be used throughout the Blockchain network (followed by the identifiers chosen by their owner organization), and reside below the "msp-manager" label. The MSP descriptions listed here aim to enable the orderers to validate peer/client signatures on system requests, e.g., chain creation requests. Hence MSPs in this case have more a “verifier” role, and will henceforth be referred to as ***VerifierMSP***. Notice, that there is one MSP listed for each of Org2, Org3, and Org4, an one MSP for Org1 as the latter contributes orderers.
    - The list of parameters associated to ordering client, and server. Ordering client includes parameters that anyone invoking broadcast and deliver request to the ordering service would need to know (e.g., peers) while the ordering server includes parameters that are in common across ordering nodes, and need to be consistently updated across the orderers.
    - The policies for readers, writers, and chain creators of the ordering channel that is to be created. that for now includes only the networks of orderers, i.e., members of Organization 1, and Organization 2.

Configuration of the non-local part of orderers can have the form of a genesis transaction a diagram of which is depicted in Table 2, since it is anyway the genesis block of system channel. For simplicity of presentation we make the same convention as before and ignore mapping of MSP configuration to configuration items.

Figure 11. Local orderer .yaml configuration file

```
General:

   # Local MSP config file
   LocalMSP:  file location of root folder of cacerts, admincerts, signcert, and keystore.

   # Orderer Type: The orderer implementation to start
   # Available types are "solo" and "kafka"
   OrdererType: solo

   # Ledger Type: The ledger type to provide to the orderer (if needed)
   # Available types are "ram", "file". When "kafka" is chosen as the
   # OrdererType, this option is ignored.
   LedgerType: ram
```

```
# Batch Timeout: The amount of time to wait before creating a batch
BatchTimeout: 10s

# possibly other local orderer config
```

Figure 12. This can be used to generate the genesis block which is needed for orderer bootstrap.

```
{
 "description":"Orderer channel genesis (Orgs: 01, 02)",
 "chain-id":"OrdererChannel-01-02",
 "msp-manager":[
     {
       "msp-type":0,
       "msp-config":{
         "msp-identifier":"org1",
         "rootca-identities":["org1-cert-bytes-1","org1-cert-bytes-2","org1-cert-bytes-3"],
         "admins":["org1-admin-cert-bytes"]
       }
     },
     {
       "msp-type":0,
       "msp-config":{
         "msp-identifier":"org2",
         "rootca-identities":["org2-cert-bytes-1","org2-cert-bytes-2","org2-cert-bytes-3"],
         "admins":["org2-admin-cert-bytes"]
       }
     },
     {
       "msp-type":0,
       "msp-config":{
         "msp-identifier":"org3",
         "rootca-identities":["org3-cert-bytes-1","org3-cert-bytes-2","org3-cert-bytes-3"],
         "admins":["org3-admin-cert-bytes"]
       }
     },
     {
       "msp-type":0,
       "msp-config":{
         "msp-identifier":"org4",
         "rootca-identities":["org4-cert-bytes-1","org4-cert-bytes-2","org4-cert-bytes-3"],
         "admins":["org4-admin-cert-bytes"]
       }
     }
   ]
 },
 "readers":[
   {"msp-identifier":"org1","group":"member"},
   {"msp-identifier":"org2","group":"member"}
 ],
 "writers":[
   {"msp-identifier":"org1","group":"member"},
   {"msp-identifier":"org2","group":"member"}
 ],
 "admins":[
   {"msp-identifier":"org1","group":"admin"},
   {"msp-identifier":"org2","group":"admin"}
 ]
}
```

Step 3: The orderers run and can identify now members of all participant organizations.

Type	Key	Value	Modification Policy
Chain	HashingAlgorithm	SHAKE256	OrdererAdminPolicy
Chain	BlockDataHashStructure	Merkle tree width 10	OrdererAdminPolicy
Chain	OrdererAddresses	[addr1, addr2, …]	OrdererAdminPolicy
Orderer	ChainCreationPolicyNames	ChainCreationPolicy1, ChainCreationPolicy2	OrdererAdminPolicy
Orderer	BatchSize	100	OrdererAdminPolicy
Orderer	BatchTimeout	10s	OrdererAdminPolicy
Orderer	IngressPolicyNames	OrdererWriterPolicy, PeerWriterPolicy	OrdererAdminPolicy
Orderer	EgressPolicyNames	OrdererReaderPolicy, PeerReaderPolicy	OrdererAdminPolicy
Orderer	ConsensusType	Kafka	OrdererAdminPolicy
Orderer	KafkaBrokersKey	[addr1, addr2, …]	OrdererAdminPolicy
MSP	Org1ID	*Org1 MSPConf*	MSPInternal
MSP	Org2ID	*Org2 MSPConf*	MSPInternal
MSP	Org3ID	*Org3 MSPConf*	MSPInternal
MSP	Org4ID	*Org4 MSPConf*	MSPInternal
Policy	OrdererReaderPolicy	*Org1.User or Org2.User*	OrdererAdminPolicy
Policy	OrdererWriterPolicy	*Org1.User or Org2.User*	OrdererAdminPolicy
Policy	OrdererAdminPolicy	*Org1.admin AND Org2.admin*	OrdererAdminPolicy

Policy	BlockValidationPolicy	*Org1.Cert or Org2.Cert*	OrdererAdminPolicy
Policy	SignedByOrdererPolicy	*Org1.Cert or Org2.Cert*	OrdererAdminPolicy
Policy	ChainCreationPolicy1	*Org3.Admin and (Org2.admin or Org4.admin)*	OrdererAdminPolicy
Policy	ChainCreationPolicy2	*Org4.admin*	OrdererAdminPolicy
Policy	NewConfigItemCreationPolicy	*Org1.admin AND Org2.admin*	OrdererAdminPolicy
Policy	MSPInternal	*empty*	RejectAlwaysPolicy
Policy	RejectAlwaysPolicy	*1 out of 0*	RejectAlwaysPolicy

Table 2. Example of list of configuration items in orderer channel genesis transaction. Each row in the table corresponds to a (signed) configuration item of the genesis block.

Step 4: For their initialization, peers need similar information as the orderers to setup the (local) MSP the peer belongs to, and signing identity/key information the peer would use to sign messages to the rest of Blockchain participants. Finally, the peer initialization information provides the reference to the signing identity the peer is to use when asked to create endorsements. As in the case of orderers, we instantiate SignerMSP for peers by having the administrator fill up three folders with the appropriate certificates/key-material:

a. cacerts: PEM files containing the root authority certificates of the MSP
b. admincerts: PEM files containing the administrators' certificates of this MSP
c. keystore: PEM files containing the signing key of the peer
d. signcerts: PEM files corresponding to the singing identity of the peer

Disclaimer: Currently SignerMSP is not used for signature verification, and hence updating the cacerts or admincerts of this MSP is not of crucial importance. However this is to be revisited as GOSSIP communication may require frequent updates of these values.

Step 5. Now let's assume that Org2, and Org3 want to create a chain for their bilateral transactions. Applications of Org2, and Org3 agree **off-band** on certain configuration aspects of the chain, i.e., the configuration of the MSP contributed by each organization, the readers and the writers of the new chain, as well as the admins of the resulting chain. This information the application of Org2, and Org3 combines with the ordering information, to result to a configuration transaction that includes the information included in Table 3. Notice that Org4, does not appear anywhere in this genesis transaction.

In the example of Table 3 below, Organization 1 and 2 are the only ones contributing orderers, while Organization 2 and 3 are the ones deploying and invoking application chaincodes.

- Readers of the chain have been set to all listed MSP's groups.

- Writers of the chain have been set to all application MSP's groups, i.e., Organization 2 and 3 members.
- Chaincode deployer access is given to the admins of the two application MSPs, i.e., Organization 2 and 3.
- Finally, re-configuration of the chain requires approval from admins of the MSPs of Organization 2 and Organization 3.

In the table below, we assumed the following mapping between parts of configuration and (signed) configuration items.

- Two configuration items per MSPConfig, of type "MSP", whose Key is the organization name and whose value is a marshaled MSPConfig proto and whose modification policy is [organization name + "Internal"], and another configuration item of type Policy, of Key of [organization name + "Internal"] of Policy Type MSP, and nil value with modification policy of itself
- One configuration item named "ChannelAdmins" of type "Policy" and Policy.Type "SignaturePolicy" whose value is a SignaturePolicy containing the conditions for administrative action. For example "2 out of the following three MSP principals must sign (each an admin). The "ChannelAdmins", has a modification policy "ChannelAdmins" (itself)
- One configuration item of type "Policy" and Policy.Type MSP for "chain-readers", and modification policy "chain-admins"
- One configuration item named "ChannelReaders" of type "Policy" and Policy.Type "SignaturePolicy" whose value is a SignaturePolicy containing the conditions for reading the chain. Since this applies to Deliver calls, it should be "1 out of the following n MSP principals must sign (each a reader)". The "ChannelReaders", has a modification policy "ChannelAdmins"
- One configuration item named "ChannelWriters" of type "Policy" and Policy.Type "SignaturePolicy" whose value is a SignaturePolicy containing the conditions for reading the chain. Since this applies to Broadcast calls, it should be "1 out of the following n MSP principals must sign (each a writer)". The "ChannelWriters", has a modification policy "ChannelAdmins"
- One configuration item named "ChaincodeLifecycleAdmins" of type "Policy" and Policy.Type "SignaturePolicy" whose value is a SignaturePolicy containing the conditions for deploying a chaincode to the chain. This might be something like "3 out of the following 4 MSP principals must sign (each a developer)". The "ChaincodeLifecycleAdmins", has a modification policy "ChannelAdmins"

The convention for items of type Orderer is to have the Key <KeyString> have a value of a marshaled <KeyString> message from the protos/common/orderer/configuration.proto. The same is true for items of type Chain, and Peer, but corresponding to protos/common/configuration.proto and protos/peer/configuration.proto. For type Policy, the value is always a marshaled protos/peer/configuration.proto.Policy message, and for type MSP the value is always a marshaled MSPConf from msp/protos (this should probably be moved to protos/msp.

Type	Key	Value	Modification Policy
Chain	HashingAlgorithm	SHAKE256	OrdererAdminPolicy
Chain	BlockDataHashStructure	Merkle tree width 10	OrdererAdminPolicy
Chain	OrdererAddresses	[addr1, addr2, …]	OrdererAdminPolicy
Orderer	BatchSize	100	OrdererAdminPolicy
Orderer	BatchTimeout	10s	OrdererAdminPolicy
Orderer	IngressPolicyNames	OrdererWriterPolicy, PeerWriterPolicy	OrdererAdminPolicy
Orderer	EgressPolicyNames	OrdererReaderPolicy, PeerReaderPolicy	OrdererAdminPolicy
Orderer	ConsensusType	Kafka	OrdererAdminPolicy
Orderer	KafkaBrokersKey	[addr1, addr2, …]	OrdererAdminPolicy
MSP	Org1ID	*Org1 MSPConf*	MSPInternal
MSP	Org2ID	*Org2 MSPConf*	MSPInternal
MSP	Org3ID	*Org3 MSPConf*	MSPInternal
MSP	Org4ID	*Org4 MSPConf*	MSPInternal
Peer	ReaderPolicyNames	*PeerReaderPolicy*	PeerAdminPolicy
Peer	WriterPolicyNames	*PeerWriterPolicy*	PeerAdminPolicy
Peer	AdminPolicyNames	*PeerAdminPolicy*	PeerAdminPolicy
Peer	ChaincodeLifecycleAdmins	*ChaincodeLifecyclePolicy*	AdminsPolicy
Policy	PeerReaderPolicy	*org1.members OR org2.members OR org3.members*	PeerAdminPolicy
Policy	PeerWriterPolicy	*org1.members OR org3.members OR org4.members*	PeerAdminPolicy
Policy	PeerAdminPolicy	*org2.admin AND org3.admin*	PeerAdminPolicy
Policy	ChaincodeLifecyclePolicy	*org2.admin OR org3.admin*	PeerAdminPolicy

Policy	OrdererReaderPolicy		OrdererAdminPolicy
Policy	OrdererWriterPolicy		OrdererAdminPolicy
Policy	OrdererAdminPolicy		OrdererAdminPolicy
Policy	BlockValidationPolicy		OrdererAdminPolicy
Policy	SignedByOrdererPolicy		OrdererAdminPolicy
Policy	NewConfigItemCreationPolicy		OrdererAdminPolicy
Policy	MSPInternal	*empty*	RejectAlwaysPolicy
Policy	RejectAlwaysPolicy	*1 out of 0*	RejectAlwaysPolicy

Table 3. Example of organization of chain genesis transaction in configuration items. Each row in the table corresponds to a (signed) configuration item of the genesis block.

Step 6. Now from the config file, Org2, and Org3 construct a configuration transaction based on the resulting config file, that would constitute the only transaction included in the new channel's genesis block. Although we are agnostic to the way the application constructs this configuration transaction, it is the application's responsibility to submit the constructed genesis block (carrying a chain identity that has not yet been used) to the ordering service via a broadcast request.

Step 7. Now each orderer that receives this genesis block for a channel that does not exist assumes this to be a chain creation request. Assuming that the orderer approves of the chain creation request (assuming that is requested by properly authorized requestors), it embeds the (verbatim) configuration transaction as the contents of the genesis block for the new chain. If chain deployers policy exists in orderers' system channel, then the orderers would need to check that the signatures in the configuration block received match the chain deployers' policy. Finally, they create a new channel and use the constructed genesis block as the first block of the new channel.

Step 8. Application of the two organizations calls deliver on the new channel and obtains the genesis block. It checks the validity of the configuration parameters in there to ensure that it is the same parameters Org2, and Org3, had agreed on (included in the chain creation request configuration).

Step 9. Application of Org2 send the obtained genesis block to its peers within a JoinChannel request and asks these peers to join the channel. The peer uses the local MSP to authenticate the JoinChannel request.

Step 10. Upon joining a channel, and processing the genesis block included in the JoinChannel request the peers retrieve the list of anchor peers. The gossip layer of the peer notifies the gossip leader of the organization the peer belong to, that it has joined the channel. That leader peer can then connect to the ordering service on the peer's behalf. The peer receives the channel related ledger blocks by connecting directly to the ordering service itself (leader peer) or from the leader peer. As a result, the peer receives the new channel's transactions (normally it will be only the genesis block) and will have to check that the genesis block it received matches the one the application provided to it with the JoinChannel request. After that, it parses the genesis block and instantiates the corresponding chain instance, i.e., MSPManager, Ledger, and cache.

Step 11. After Initializing the channel's MSP, the peers that joined the channel disseminate via the gossip layer the channel's MSP to all peers of their organization, in order to ensure that peers that belong to different organizations but their organizations share a channel, can communicate with each other.

6. Orderer Chain Creation Implementation Details

6.1 Orderer System Chain

The orderer network is bootstrapped with a genesis block which contains the **orderer system chain** id and the initial set of orderer network governance policies, including, the orderer MSPs, the orderer consensus protocol configuration, and the chain creation policies. Although application network MSPs may also be defined here (to be referenced in the chain creation policies), they are not strictly required at bootstrap.

Only the ordering organizations have permission to read or write on this chain, so from a peer consumer perspective, the existence of this chain is irrelevant.

The **chain creators** are enumerated in a configuration item of type Orderer, key ChainCreators, and value common.ChainCreators. The common.ChainCreators is a repeated list of strings, each corresponding to a defined configuration item of type Policy. Note that this is a repeated list, rather than a single policy, to accommodate the possibility of multi-tenancy. Different consortiums may desire different chain creation policies, and although it might be possible to construct one large policy with many 'ORs' to accommodate this, it is more natural to specify and manage the policies on a per consortium basis.

An example of ordering channel genesis organization in configuration items is depicted in Table 2.

6.2 Chain Creation Request (Configuration Transaction)

When a consortium wishes to use an ordering service, the ordering service creates a chain creation policy for that consortium, typically requiring the signatures of any two participants in the consortium, but as it uses the underlying policy framework, the signature requirements may be arbitrarily complex (for instance, requiring one authoritative member, and arbitrary two others). This chain creation policy is simply a named string, and is added internally to the **chain creators** of the ordering system chain. The consortium will also be told by the ordering service admins the set of configuration items which must be present in any chain creation request (such as the ordering system MSP definitions and block validation policies).

When members of the consortium wish to create a new chain, they simply create the set of configuration items which defines their new chain, concatenate the marshaled bytes, and compute the hash of this data. They then create a special configuration item of type Orderer, key **CreationPolicy**, and type orderer.CreationPolicy, with the digest field set to the hash of the chain configuration, and the policy set to the named **chain creators** policy provided by the ordering service. This ConfigurationItem is then inserted into a SignedConfigurationItem and is appropriately signed as required by the chain creation policy. This is all wrapped into a ConfigurationEnvelope and packaged into a signed Envelope and submitted for ordering.

Note then, that the signed Envelope message is exactly the contents of the new genesis block. All of the requesting parties have their signature encoded into this genesis block, and so the only validation required by the application is to ensure that the signature on the **CreationPolicy** configuration item is valid, and that the digest encoded corresponds to the remaining configuration.

An example of the structure of a chain genesis transaction appears in Table 3.

When the ordering service receives a chain configuration transaction, it first checks to see if the chain ID already exists. If it does, then it is treated as a reconfiguration transaction and accepted/rejected accordingly. In the event that the chain does not exist, the orderer then validates that it is a well formed and currently valid chain creation request, it then wraps this request inside of an Envelope of type ORDERER_TRANSACTION bound for the ordering system chain, and submits it for consensus. Eventually, once the transaction has been ordered, it is unwrapped, and inspected a second time for validity now that ordering has occurred. Only in cases where the ordering system chain configuration changed or a chain creation request for the identical chain ID was submitted concurrently can this validation fail, in which case the request is logged and discarded.

Finally, after consensus is achieved and the still valid configuration transaction is processed, new ledger resources are allocated and the configuration transaction is embedded into the genesis block for the new chain. The created can poll for this creation via a Deliver request.