Channel Configuration (configtx)

Shared configuration for a Hyperledger Fabric blockchain network is stored in a collection configuration transactions, one per channel. Each configuration transaction is usually referred to by the shorter name *configtx*.

Channel configuration has the following important properties:

1. **Versioned**: All elements of the configuration have an associated version which is advanced with every modification. Further, every committed configuration receives a sequence number.
2. **Permissioned**: Each element of the configuration has an associated policy which governs whether or not modification to that element is permitted. Anyone with a copy of the previous configtx (and no additional info) may verify the validity of a new config based on these policies.
3. **Hierarchical**: A root configuration group contains sub-groups, and each group of the hierarchy has associated values and policies. These policies can take advantage of the hierarchy to derive policies at one level from policies of lower levels.

Anatomy of a configuration

Configuration is stored as a transaction of type `HeaderType_CONFIG` in a block with no other transactions. These blocks are referred to as *Configuration Blocks*, the first of which is referred to as the *Genesis Block*.

The proto structures for configuration are stored in `fabric/protos/common/configtx.proto`. The Envelope of type `HeaderType_CONFIG` encodes a `ConfigEnvelope` message as the `Payload` `data` field. The proto for `ConfigEnvelope` is defined as follows:

```
message ConfigEnvelope {
    Config config = 1;
    Envelope last_update = 2;
}
```

The `last_update` field is defined below in the **Updates to configuration** section, but is only necessary when validating the configuration, not reading it. Instead, the currently committed configuration is stored in the `config` field, containing a `Config` message.

```
message Config {
    uint64 sequence = 1;
    ConfigGroup channel_group = 2;
}
```

The `sequence` number is incremented by one for each committed configuration. The `channel_group` field is the root group which contains the configuration. The `ConfigGroup` structure is recursively defined, and builds a tree of groups, each of which contains values and policies. It is defined as follows:

```
message ConfigGroup {
    uint64 version = 1;
    map<string,ConfigGroup> groups = 2;
    map<string,ConfigValue> values = 3;
    map<string,ConfigPolicy> policies = 4;
    string mod_policy = 5;
}
```

Because `ConfigGroup` is a recursive structure, it has hierarchical arrangement. The following example is expressed for clarity in golang notation.

```
// Assume the following groups are defined
var root, child1, child2, grandChild1, grandChild2, grandChild3 *ConfigGroup

// Set the following values
root.Groups["child1"] = child1
root.Groups["child2"] = child2
child1.Groups["grandChild1"] = grandChild1
child2.Groups["grandChild2"] = grandChild2
child2.Groups["grandChild3"] = grandChild3

// The resulting config structure of groups looks like:
// root:
//     child1:
//         grandChild1
//     child2:
//         grandChild2
//         grandChild3
```

Each group defines a level in the config hierarchy, and each group has an associated set of values (indexed by string key) and policies (also indexed by string key).

Values are defined by:

```
message ConfigValue {
    uint64 version = 1;
    bytes value = 2;
    string mod_policy = 3;
}
```

Policies are defined by:

```
message ConfigPolicy {
    uint64 version = 1;
    Policy policy = 2;
    string mod_policy = 3;
}
```

Note that Values, Policies, and Groups all have a `version` and a `mod_policy`. The `version` of an element is incremented each time that element is modified. The `mod_policy` is used to govern the required signatures to modify that element. For Groups, modification is adding or removing elements to the Values, Policies, or Groups maps (or changing the `mod_policy`). For Values and Policies, modification is changing the Value and Policy fields respectively (or changing the `mod_policy`). Each element's `mod_policy` is evaluated in the context of the current level of the config. Consider the following example mod policies defined at `Channel.Groups["Application"]` (Here, we use the golang map reference syntax, so `Channel.Groups["Application"].Policies["policy1"]` refers to the base `Channel` group's `Application` group's `Policies` map's `policy1` policy.)

- `policy1` maps to `Channel.Groups["Application"].Policies["policy1"]`
- `Org1/policy2` maps to `Channel.Groups["Application"].Groups["Org1"].Policies["policy2"]`
- `/Channel/policy3` maps to `Channel.Policies["policy3"]`

Note that if a `mod_policy` references a policy which does not exist, the item cannot be modified.

Configuration updates

Configuration updates are submitted as an `Envelope` message of type `HeaderType_CONFIG_UPDATE`. The `Payload` `data` of the transaction is a marshaled `ConfigUpdateEnvelope`. The `ConfigUpdateEnvelope` is defined as follows:

```
message ConfigUpdateEnvelope {
    bytes config_update = 1;
    repeated ConfigSignature signatures = 2;
}
```

The `signatures` field contains the set of signatures which authorizes the config update. Its message definition is:

```
message ConfigSignature {
    bytes signature_header = 1;
    bytes signature = 2;
}
```

The `signature_header` is as defined for standard transactions, while the signature is over the concatenation of the `signature_header` bytes and the `config_update` bytes from the `ConfigUpdateEnvelope` message.

The `ConfigUpdateEnvelope` `config_update` bytes are a marshaled `ConfigUpdate` message which is defined as follows:

```
message ConfigUpdate {
    string channel_id = 1;
    ConfigGroup read_set = 2;
    ConfigGroup write_set = 3;
}
```

The `channel_id` is the channel ID the update is bound for, this is necessary to scope the signatures which support this reconfiguration.

The `read_set` specifies a subset of the existing configuration, specified sparsely where only the `version` field is set and no other fields must be populated. The particular `ConfigValue` `value` or `ConfigPolicy` `policy` fields should never be set in the `read_set`. The `ConfigGroup` may have a subset of its map fields populated, so as to reference an element deeper in the config tree. For instance, to include the `Application` group in the `read_set`, its parent (the `Channel` group) must also be included in the read set, but, the `Channel` group does not need to populate all of the keys, such as the `Orderer` `group` key, or any of the `values` or `policies` keys.

The `write_set` specifies the pieces of configuration which are modified. Because of the hierarchical nature of the configuration, a write to an element deep in the hierarchy must contain the higher level elements in its `write_set` as well. However, for any element in the `write_set` which is also specified in the `read_set` at the same version, the element should be specified sparsely, just as in the `read_set`.

For example, given the configuration:

```
Channel: (version 0)
    Orderer (version 0)
    Application (version 3)
       Org1 (version 2)
```

To submit a configuration update which modifies `Org1`, the `read_set` would be:

```
Channel: (version 0)
    Application: (version 3)
```

and the `write_set` would be

```
Channel: (version 0)
    Application: (version 3)
        Org1 (version 3)
```

When the `CONFIG_UPDATE` is received, the orderer computes the resulting `CONFIG` by doing the following:

1. Verifies the `channel_id` and `read_set`. All elements in the `read_set` must exist at the given versions.
2. Computes the update set by collecting all elements in the `write_set` which do not appear at the same version in the `read_set`.
3. Verifies that each element in the update set increments the version number of the element update by exactly 1.
4. Verifies that the signature set attached to the `ConfigUpdateEnvelope` satisfies the `mod_policy` for each element in the update set.
5. Computes a new complete version of the config by applying the update set to the current config.
6. Writes the new config into a `ConfigEnvelope` which includes the `CONFIG_UPDATE` as the `last_update` field and the new config encoded in the `config` field, along with the incremented `sequence` value.
7. Writes the new `ConfigEnvelope` into a `Envelope` of type `CONFIG`, and ultimately writes this as the sole transaction in a new configuration block.

When the peer (or any other receiver for `Deliver`) receives this configuration block, it should verify that the config was appropriately validated by applying the `last_update` message to the current config and verifying that the orderer-computed `config` field contains the correct new configuration.

Permitted configuration groups and values

Any valid configuration is a subset of the following configuration. Here we use the notation `peer.<MSG>` to define a `ConfigValue` whose `value` field is a marshaled proto message of name `<MSG>` defined in `fabric/protos/peer/configuration.proto`. The notations `common.<MSG>`, `msp.<MSG>`, and `orderer.<MSG>` correspond similarly, but with their messages defined in `fabric/protos/common/configuration.proto`, `fabric/protos/msp/mspconfig.proto`, and `fabric/protos/orderer/configuration.proto` respectively.

Note, that the keys `{{org_name}}` and `{{consortium_name}}` represent arbitrary names, and indicate an element which may be repeated with different names.

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Application":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                        "AnchorPeers":peer.AnchorPeers,
                    },
                },
            },
        },
        "Orderer":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                    },
                },
            },

            Values:map<string, *ConfigValue> {
                "ConsensusType":orderer.ConsensusType,
                "BatchSize":orderer.BatchSize,
                "BatchTimeout":orderer.BatchTimeout,
                "KafkaBrokers":orderer.KafkaBrokers,
            },
        },
        "Consortiums":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{consortium_name}}:&ConfigGroup{
                    Groups:map<string, *ConfigGroup> {
                        {{org_name}}:&ConfigGroup{
                            Values:map<string, *ConfigValue>{
                                "MSP":msp.MSPConfig,
                            },
                        },
                    },
                    Values:map<string, *ConfigValue> {
                        "ChannelCreationPolicy":common.Policy,
                    }
                },
            },
        },
    },

    Values: map<string, *ConfigValue> {
        "HashingAlgorithm":common.HashingAlgorithm,
        "BlockHashingDataStructure":common.BlockDataHashingStructure,
        "Consortium":common.Consortium,
        "OrdererAddresses":common.OrdererAddresses,
    },
}
```

Orderer system channel configuration

The ordering system channel needs to define ordering parameters, and consortiums for creating channels. There must be exactly one ordering system channel for an ordering service, and it is the first channel to be created (or more accurately bootstrapped). It is recommended never to define an Application section inside of the ordering system channel genesis configuration, but may be done for testing. Note that any member with read access to the ordering system channel may see all channel creations, so this channel's access should be restricted.

The ordering parameters are defined as the following subset of config:

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Orderer":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                    },
                },
            },

            Values:map<string, *ConfigValue> {
                "ConsensusType":orderer.ConsensusType,
                "BatchSize":orderer.BatchSize,
                "BatchTimeout":orderer.BatchTimeout,
                "KafkaBrokers":orderer.KafkaBrokers,
            },
        },
    },
```

Each organization participating in ordering has a group element under the `Orderer` group. This group defines a single parameter `MSP` which contains the cryptographic identity information for that organization. The `Values` of the `Orderer` group determine how the ordering nodes function. They exist per channel, so `orderer.BatchTimeout` for instance may be specified differently on one channel than another.

At startup, the orderer is faced with a filesystem which contains information for many channels. The orderer identifies the system channel by identifying the channel with the consortiums group defined. The consortiums group has the following structure.

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Consortiums":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{consortium_name}}:&ConfigGroup{
                    Groups:map<string, *ConfigGroup> {
                        {{org_name}}:&ConfigGroup{
                            Values:map<string, *ConfigValue>{
                                "MSP":msp.MSPConfig,
                            },
                        },
                    },
                    Values:map<string, *ConfigValue> {
                        "ChannelCreationPolicy":common.Policy,
                    }
                },
            },
        },
    },
},
```

Note that each consortium defines a set of members, just like the organizational members for the ordering orgs. Each consortium also defines a `ChannelCreationPolicy`. This is a policy which is applied to authorize channel creation requests. Typically, this value will be set to an `ImplicitMetaPolicy` requiring that the new members of the channel sign to authorize the channel creation. More details about channel creation follow later in this document.

Application channel configuration

Application configuration is for channels which are designed for application type transactions. It is defined as follows:

```
&ConfigGroup{
    Groups: map<string, *ConfigGroup> {
        "Application":&ConfigGroup{
            Groups:map<String, *ConfigGroup> {
                {{org_name}}:&ConfigGroup{
                    Values:map<string, *ConfigValue>{
                        "MSP":msp.MSPConfig,
                        "AnchorPeers":peer.AnchorPeers,
                    },
                },
            },
        },
    },
}
```

Just like with the `Orderer` section, each organization is encoded as a group. However, instead of only encoding the `MSP` identity information, each org additionally encodes a list of `AnchorPeers`. This list allows the peers of different organizations to contact each other for peer gossip networking.

The application channel encodes a copy of the orderer orgs and consensus options to allow for deterministic updating of these parameters, so the same `Orderer` section from the orderer system channel configuration is included. However from an application perspective this may be largely ignored.

Channel creation

When the orderer receives a `CONFIG_UPDATE` for a channel which does not exist, the orderer assumes that this must be a channel creation request and performs the following.

1. The orderer identifies the consortium which the channel creation request is to be performed for. It does this by looking at the `Consortium` value of the top level group.
2. The orderer verifies that the organizations included in the `Application` group are a subset of the organizations included in the corresponding consortium and that the `ApplicationGroup` is set to `version` `1`.
3. The orderer verifies that if the consortium has members, that the new channel also has application members (creation consortiums and channels with no members is useful for testing only).
4. The orderer creates a template configuration by taking the `Orderer` group from the ordering system channel, and creating an `Application` group with the newly specified members and specifying its `mod_policy` to be the `ChannelCreationPolicy` as specified in the consortium config. Note that the policy is evaluated in the context of the new

configuration, so a policy requiring `ALL` members, would require signatures from all the new channel members, not all the members of the consortium.

5. The orderer then applies the `CONFIG_UPDATE` as an update to this template configuration. Because the `CONFIG_UPDATE` applies modifications to the `Application` group (its `version` is `1`), the config code validates these updates against the `ChannelCreationPolicy`. If the channel creation contains any other modifications, such as to an individual org's anchor peers, the corresponding mod policy for the element will be invoked.
6. The new `CONFIG` transaction with the new channel config is wrapped and sent for ordering on the ordering system channel. After ordering, the channel is created.