Hyperledger Fabric (Multi-Cloud) Interoperability Working Group

Abstract

This document represents the current effort of the Hyperledger Fabric community to specify and standardize an interoperability workflow that allows organizations hosted on different cloud vendors to establish and join business networks independent from the infrastructure where the ordering services are hosted.
To archive this, this document lays out the necessary configuration artifacts that need to be provided by the joining organization, the high-level message flow and a chaincode based voting mechanism to approve or reject channel join requests.
Furthermore, this proposal should be fully compatible with Hyperledger Fabric 1.4 LTS.

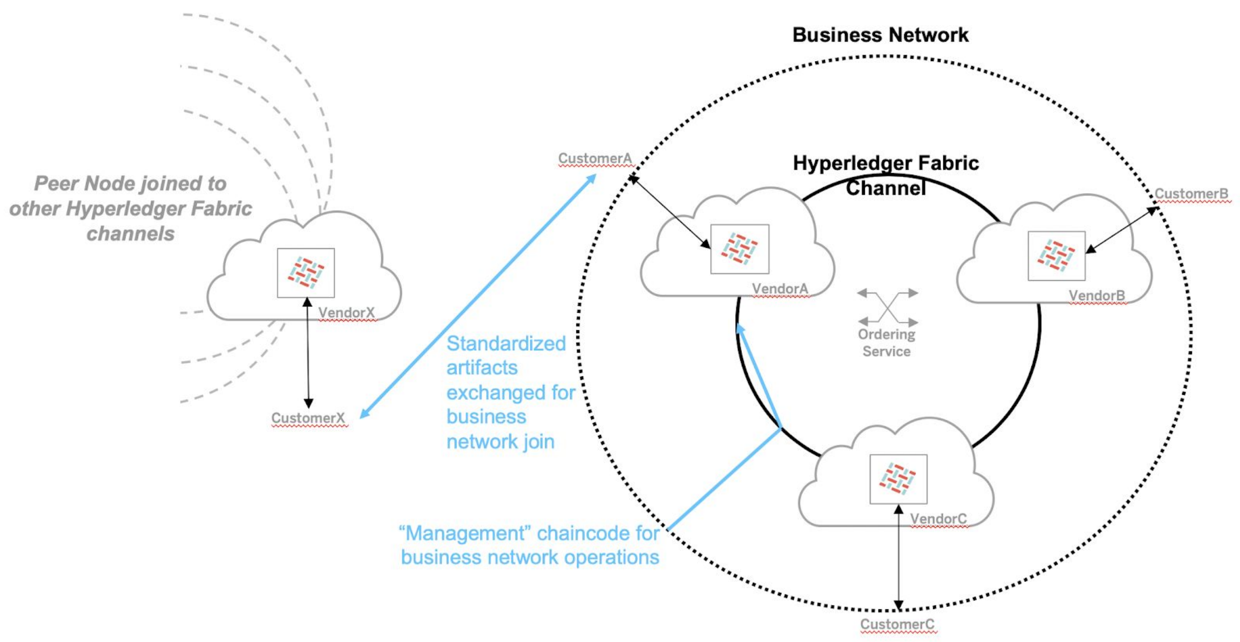
Next Action Items

Date	**Open Action Items**
17.01.2018	Specify the channel join process
24.01.2018	Incorporate feedback and verify that process works as required
31.01.2018	Schedule interoperability test in public domain

Scenario Description

The scope for the Fabric interoperability group is **initially limited** to a defined scenario.

In this scenario we have to distinguish between the technical interoperability (“joining a Fabric peer node to an existing Fabric channel”) and the business interoperability (“business network”). The interoperability workgroup will focus **only** on the technical interoperability.


Business Network
Hyperledger Fabric Channel
Peer Node joined to other Hyperledger Fabric channels
CustomerA
CustomerB
CustomerC
CustomerX
VendorA
VendorB
VendorC
VendorX
Ordering Service
Standardized artifacts exchanged for business network join
“Management” chaincode for business network operations

We assume that there already exists a Fabric channel relative to an ordering service. Attached to this channel can be any number of Fabric peer nodes. It is assumed for the discussion that each Fabric peer node has been provisioned by a (cloud) vendor in the name of the customer. In a simpler variation, any customer can also be their own “vendor”.

Important: in this scenario there exists two levels of interconnection. At the business level, we have typically a business network, which is agreed between any number of customers and governed by the business network rules and regulations. The second interconnected entity is the actual Fabric channel that spans the different Fabric nodes. It is NOT expected that any vendor will be part of the business network.

Integration story: CustomerX, having provisioned a Fabric peer node with its preferred vendor, wishes to join its Fabric peer node into the business network Fabric channel.

Assumed:

- In general, all Fabric peers and orderers can be accessed using **public DNS** and **public IP addresses**.
- This proposal targets to be fully compatible with Hyperledger **Fabric 1.4 LTS** since that it is expected to be widely adopted over a long period of time.
- That customerX has an agreement with vendorX for the provisioning of a Fabric peer node. (Note that in all simplified scenarios customerX can be equal to vendorX and do their own provisioning.)
- Initially neither customerX, nor vendorX, has access to the existing Fabric channel.
- VendorX, not been part of the business network, does NOT have the mandate to interact with the business network (in the name of customerX). All interaction with the business network (the business entity) must be between customerX and any/all parties of the business network (here customers A,B&C). No vendor is involved in this business interaction.
- A business agreement between customerX and the business network is reached before a technical interoperability is established.

Out of Scope:

- Interaction model between any customer and vendor. It is accepted that each vendor will have its own (cloud based) Fabric offering and will provide its customers a rich UI and process steps to complete all relevant actions, or require that customers complete relevant actions via the Fabric SDK.
- The interaction between the customers within the business network. It is used some form of interaction model is defined, for example via emails, faxes, websites or predefined APIs. For customerX to join the business network, it will require some form of interaction paradigm with other business network members.

In Scope:

- **The artifacts** that customerX must exchange with any/all customers within the business network to enable the technical interoperability. This will include a definite description of all relevant artifacts that must be supplied (for example certificates, TCP/IP interconnectivity information, etc) and the format in which these artifacts must be encoded in for transport (JSON, protobuf, etc).
- Also, in scope is the definition of predefined **“management” chaincode** that can be used by business network members to facilitate the steps required by the business network to agree and execute the technical Fabric peer node join request. Note: this “management” chaincode is effectively business network “business” and could be

enhanced by the business network to contain additional business logic. The Fabric interoperability working group can define and deliver an example of such a "management" chaincode. What is NOT defined, is how the business network interacts with the management chaincode, whether this is managed and done by the different customers within the business network directly, or whether any vendor takes over part of this operation aspects.
Footnote: Optional, it should be considered to define (a subset of) chaincode functions which are standardized as mandatory, so as to enable a more homogenous management of business networks by either customers and/or vendors.

Approach:

- KISS: The goal of the interoperability working group is to attempt to define a possible solution that uses existing Fabric technology without any additional development as far as possible.

Expected Interaction Model:

- CustomerX interacts with the business network (any/all members) and reach a business agreement to join the business network.
- CustomerX interacts with VendorX to obtain the defined artifacts encoded in the defined format for the interoperability.
- CustomerX interacts with the business network (visualized above as the blue arrow between customerX and customerA) to transfer the artifacts to the business network (**"join request"**).
- The business network (customers A,B&C) interacts with their relevant vendors (A,B&C) and/or the "management" chaincode (visualized in blue above) to complete the relevant technical steps to enable the interoperability.
- The business network returns to customerX defined artifacts in the defined format (**"join response"**).
- CustomerX (possibly with the help of vendorX), based on the returned artifacts, joins the Fabric channel.

Channel Join Process

For the channel join request, the focus will be on the artifacts that needs to be exchanged between customerX and customerA (as proxy for the complete business network). Only the artifacts to be exchanged is defined, the exchange process, for example via email, is outside the scope of the definition.
Also not defined is the interaction between any customer and vendor. It is assumed that either the customer has a direct access to the peer node and can complete all steps required, or that the customer has access to a management UI of the vendor where the relevant join steps are encapsulated.

Join Request:

The join-request should be based on native Hyperledger Fabric protobuf definition (protos/common.ConfigGroup) since that includes all required information to represent an organization on a Hyperledger Fabric channel. Furthermore, it integrates well with the existing tooling.
The join-request can be generated with the existing Hyperledger Fabric tool set (configtxlator, peer CLI, configtxgen).

General Workflow:

1. VendorX/CustomerX creates common.ConfigGroup (e.g. using configtxgen, configtxlator). The result can be protobuf encoded or a JSON representation.
2. CustomerX sends the serialized protobuf or the JSON representation to customerA.
3. VendorA/CustomerA generates a channel update based on the current channel configuration (e.g. using configtxlator).
4. CustomerA proposes the update to the other channel members, possibly using the “management” chaincode.
5. CustomerB and customerC, if so defined or required by the business network rules, sign the update and send it to customerA (e.g. using peer CLI) possible with the help of a “management” chaincode. -- The “management” chaincode should support relevant rules and policies, such as which customers are required to sign the update, which kinds of conditions should apply, such as signing algorithm, consensus algorithm, etc. It is important to understand that the “management” chaincode is a technical implementation of the rules and regulations of this very specific business network (business entity) and that it can be conceived that the “management” chaincode will be different between different business networks.
6. CustomerA applies the Fabric channel update with the given signatures, signed by its own organization.

Join Response:

The join-response should include the channelID and orderer endpoints including TLS certificates so that the joining org (customerX) can download the channel genesis block via the SDK or using the peer CLI.
The response should look like this:

```
{
        “channelID”: “mybusinesschannel”,
        “ordererEndpoints”: [“orderer0.org1.com:7050”],
        “ordererTlsCertificates”: [“YmVpc3BpZWw….”]
}
```

Management Chaincode

The purpose of the management chaincode is to standardize the way how channel updates (new channel members, configuration parameters) are shared will all the participants and how the required signatures to authorize a channel update, are collected.

Assumptions:

- One instance of management chaincode should be installed/instantiated on each Hyperledger Fabric channel.
- Per default it should be installed on each joined peer of the channel members.
- The recommended endorsement policy should be **majority**.
  → The chaincode endorsement policy should reflect the majority of the channel members and might be upgraded periodically.

The minimum requirement to manage channel update proposals (upload new proposal, sign proposal, list proposals) would require the following interfaces:

- `proposeUpdate(update string) (proposalID string)`
  Channel members can propose a new channel update.
  - Update: the base64 encoded protos/common.ConfigUpdate
  - Returns: a new proposalID
- `addSignature(proposalID string, signature string)`
  Channel members can add their signature to a proposed channel update.
  - Signature: the base64 encoded protos/common.ConfigSignature
- `getProposals() ([]Proposal)`
  Channel members can view proposals to check the status, display them, or check if the channel policy is already met.

Note: That the channel policy has to be checked manually by the user and **not** by the chaincode.

    - Returns: the pending(/all) proposals with their update and signatures.

- `getProposal(proposalID string) (Proposal)`
  Channel members can request the current status of the proposal.

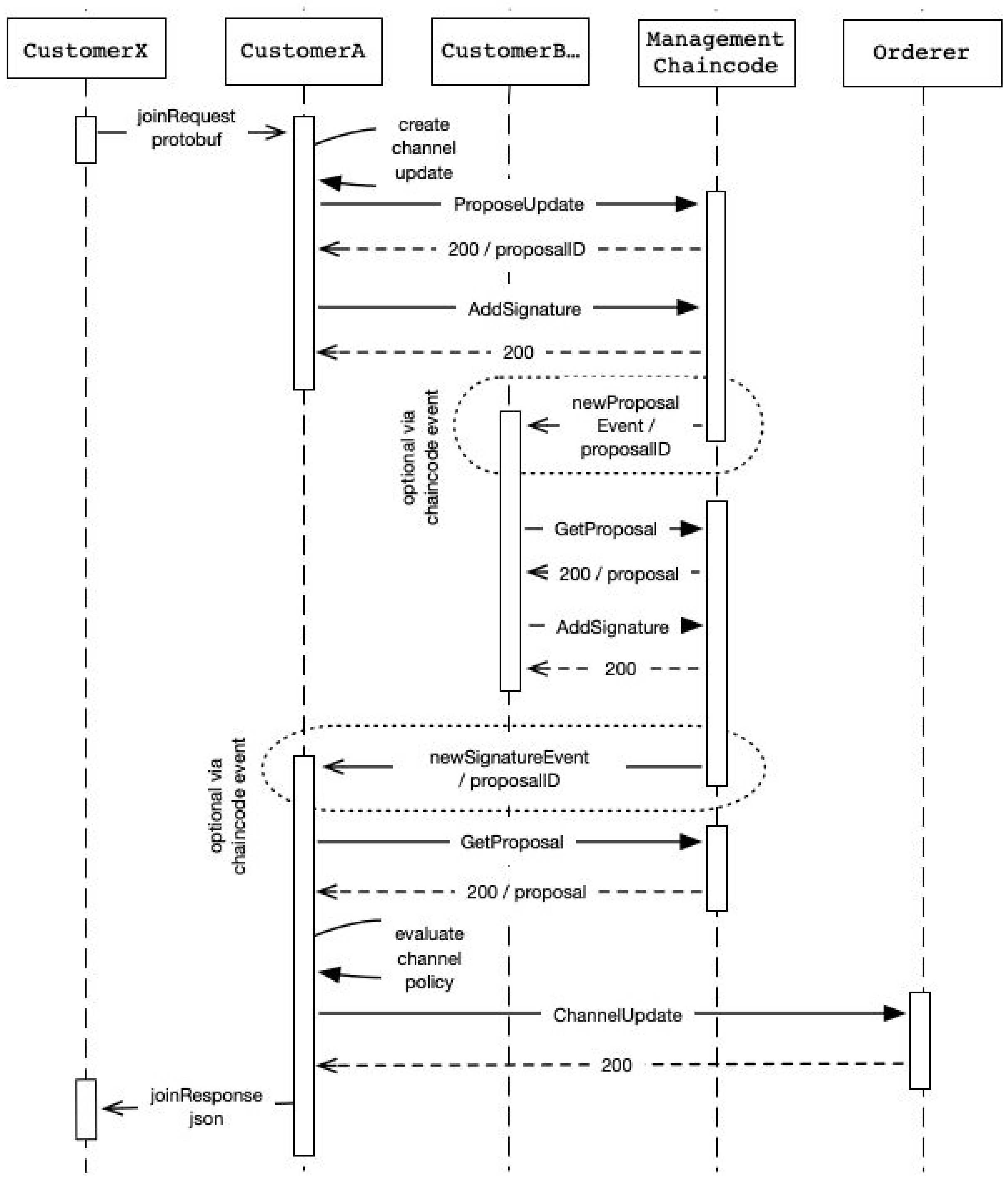
Using these interfaces, all members of a channel would be able to participate in the channel update process.

A rough workflow using this management chaincode could look like this:

1. CustomerA receives a channel join request from customerX e.g. via email.
2. CustomerA creates channel update and uses the `ProposeUpdate` interface of the management chaincode and receives a *proposalID.*
3. CustomerA creates a signature on the proposed channel update and publishes it via the `AddSignature` interface using the *proposalID*.
4. [optional] customerA informs other channel members about the new proposal. This should normally happen via a Channel Management App of the vendors.
5. CustomerB and customerC add their signatures via the `AddSignature` interface. (This call can be triggered via the custom UI implementation of a vendor).
6. CustomerA monitors the signing process and sends the channel update to the orderer when the channel update policy is fulfilled.
7. If the channel update has been accepted by the orderer, customerA compiles the channel join response message and sends it back to customerX.

The outline of the described workflow is also visualized in the Channel Join Workflow section. Furthermore, a high level overview of the proposed workflow from the view of different vendors can be found in the Appendix: Example of Vendor Supported UI-Flow.

Channel Join Workflow



To Be Considered Next

This section contains topics and items that should be considered / be worked on after the basic channel join scenario described above has been completed.

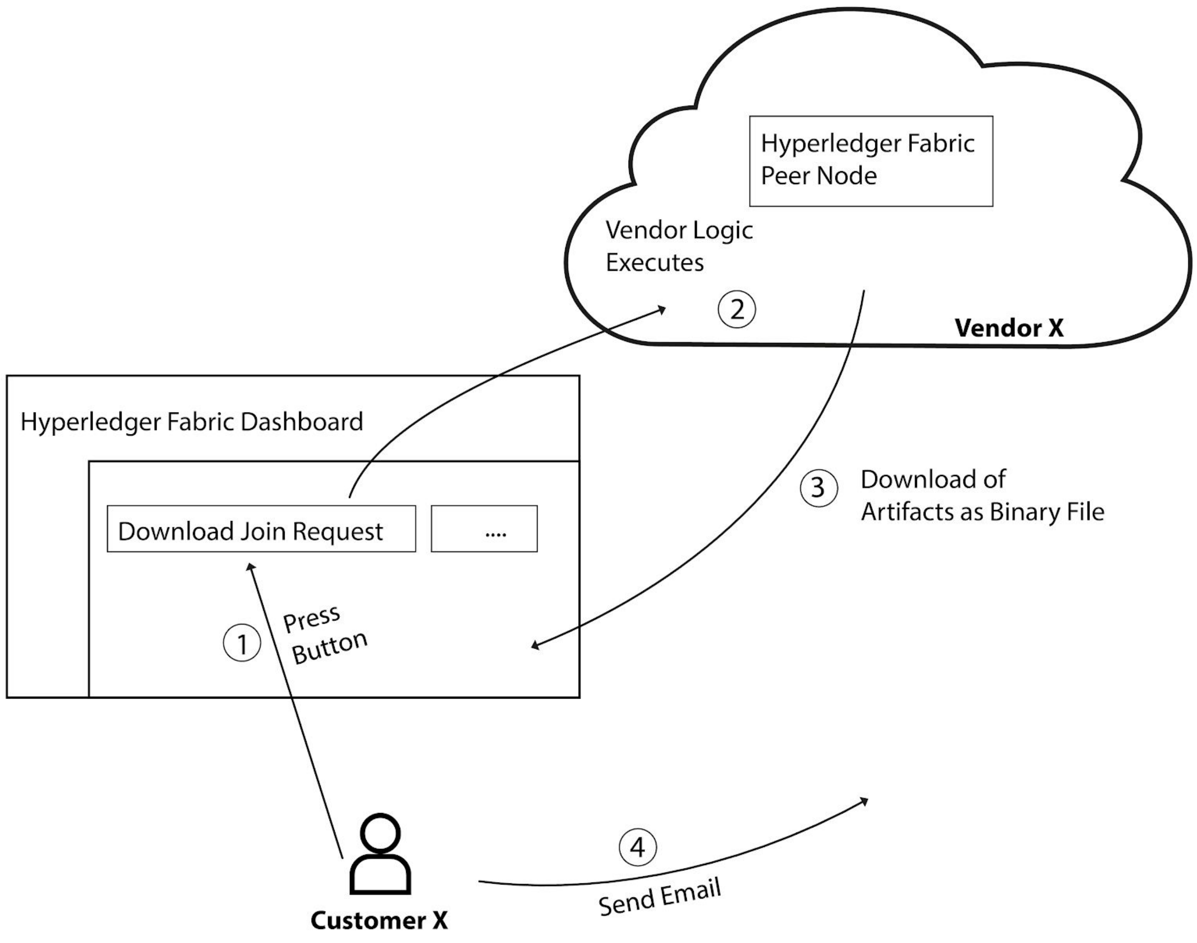
1	Handling of stale updates
2	Make sure that suggested process is fully compatible with HLF 2.0
3	Standardize the way how config artifacts are exchanged

Appendix

Example of Vendor Supported UI-Flow

In following we present an example how the interoperability workflow (presented in Channel Join Workflow section) could be integrated into the vendor dashboards.

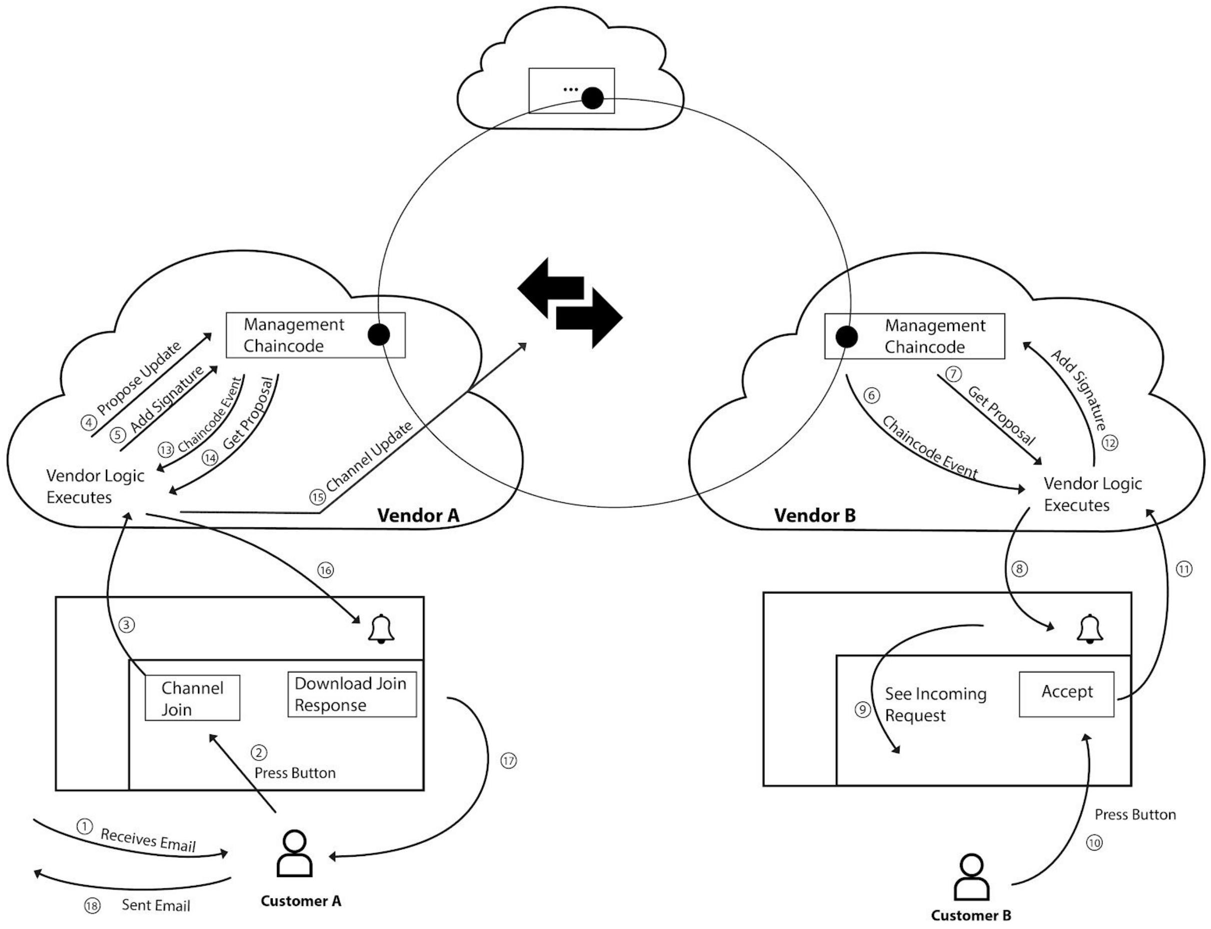
Step 1:
CustomerX downloads the JoinRequest (provisioned VendorX) and sends it via email to a business network member.


Hyperledger Fabric Peer Node
Vendor Logic Executes
2
Vendor X
Hyperledger Fabric Dashboard
3
Download of Artifacts as Binary File
Download Join Request
....
1
Press Button
4
Send Email
Customer X

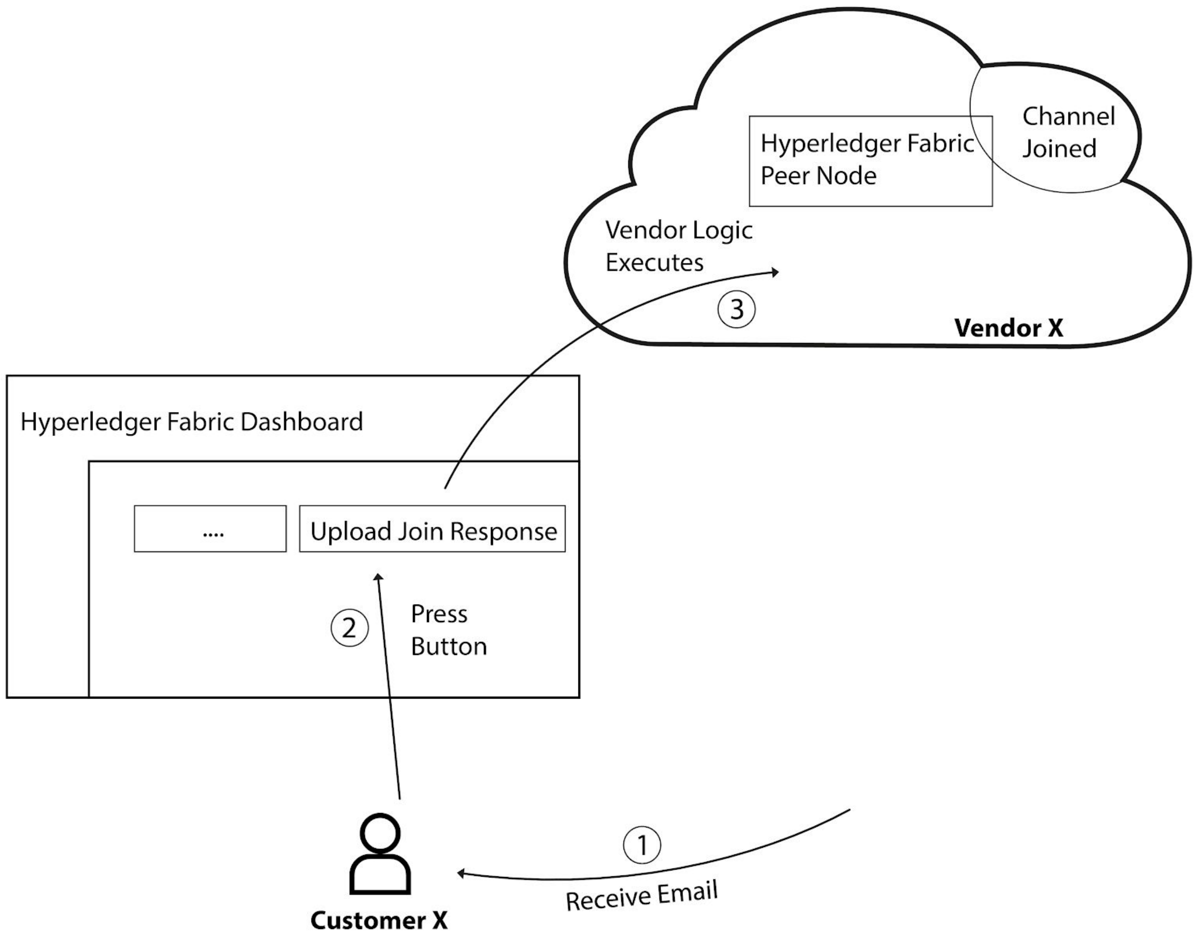
Step 2:
CustomerA receives the joinRequest sent by customerX and uploads it via the UI offered by VendorA. In the background, the UI proposes the channelUpdate via the standardized interfaces of the management chaincode.

CustomerB gets a notification in his UI based on the emitted *NewProposalEvent* chaincode event. CustomerB will be presented all the details about the new channel join request in his dashboard. Via his dashboard, customerB can then approve the request. If he approves, his signature will be added to the channelUpdateProposal.

CustomerA will receive a notification in his UI about the newly added signature via the *NewSignatureEvent*. If the number of signatures is sufficient, customerA can now trigger the channel update process. If the update was successful CustomerA can download the channel join response and send it to customerX via email.



Step 3:
CustomerX receives the channelJoinResponse via email and uploads it via his Hyperledger Fabric dashboard offered by vendorX. In the background the orderer endpoints, orderer certificates and channel information are propagated to databases etc. so that the customer can access the channel.



Example Bash Scripts

Create common.ConfigGroup (customerX)

Based on the `configtx.yaml` and the msp directories, the ConfigGroup can be created using the configtxgen printOrg function.

```
configtxgen -printOrg Org1 > configGroup.json
```

Create common.ConfigUpdate (customerA)

In this section the ConfigUpdate is created by using fabric tools.

Download the current `CONFIG` block of the channel and save it as `channel_block.pb`. Then, decode the block to JSON to be able to extract the config and update it later.

```
configtxlator proto_decode --type common.Block --input channel_block.pb --output
channel_block.json
```

Extract the channel config (assuming the `CONFIG` transaction is the first transaction).

```
cat channel_block.json | jq '.data.data[0].payload.data.config' >
channel_config.json
```

Extract the MSPID and the config of the new organization.

```
MSPID=$(cat configGroup.json | jq -r '.values.MSP.value.config.name')
CONFIG=$(cat configGroup.json)
```

Insert the new config of the organization into the channel config.

```
cat channel_config.json | jq ".channel_group.groups.Application.groups.${MSPID} =
${CONFIG}" > updated_config.json
```

Encode the original and updated config block to proto to calculate the diff.

```
configtxlator proto_encode --type common.Config --input channel_config.json
--output channel_config.pb
configtxlator proto_encode --type common.Config --input updated_config.json
--output updated_config.pb
```

Calculate the channel update based on the created artifacts.

```
configtxlator compute_update --original channel_config.pb --updated
updated_config.pb --channel_id=my-channel --output channel_update.pb
```

Sign the Update (customerA…)

To be able to check the update content, it can be decoded by the following command.

```
configtxlator proto_decode --type common.ConfigUpdate --input channel_update.pb
--output channel_update.json
```

Create the signature to approve the update. This can be done using any SDK.
In case you want to use the peer CLI, the `channel_update.pb` has to be wrapped multiple times into a common.Envelope. An official tutorial can be found here. Finally, the resulting `update_in_envelope.pb` can be signed using the peer signconfig command.

```
peer channel signconfigtx -f update_in_envelope.pb
```

Example Management Chaincode

```go
package main

import (
  "encoding/base64"
  "encoding/json"
  "fmt"
  "github.com/golang/protobuf/proto"
  "github.com/hyperledger/fabric/core/chaincode/shim"
  "github.com/hyperledger/fabric/protos/common"
  "github.com/hyperledger/fabric/protos/msp"
  pb "github.com/hyperledger/fabric/protos/peer"
  "math/rand"
  "time"
)

// ManagementChaincode example
type ManagementChaincode struct {
}

type Proposal struct {
  // Description describes the proposal.
  Description string `json:"description,omitempty"`

  // ConfigUpdate contains the base64 string representation of the common.ConfigUpdate.
  ConfigUpdate string `json:"config_update,omitempty"`

  // Signatures contains a map of signatures: mspID -> base64 string representation of
common.ConfigSignature
  Signatures map[string]string `json:"signatures,omitempty"`
}

const (
  ProposalIDLength  = 12
  NewProposalEvent  = "newProposalEvent"
  NewSignatureEvent = "newSignatureEvent"
  alphanum          = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
)

func (mcc *ManagementChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
  return shim.Success(nil)
}

func main() {
  rand.Seed(time.Now().UTC().UnixNano())
  err := shim.Start(new(ManagementChaincode))
  if err != nil {
     fmt.Printf("Error starting management chaincode: %s", err)
  }
}

func (mcc *ManagementChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
  function, args := stub.GetFunctionAndParameters()
  switch function {
  case "proposeUpdate":
     return mcc.proposeUpdate(stub, args)

  case "addSignature":
     return mcc.addSignature(stub, args)

  case "getProposals":
     return mcc.getProposals(stub, args)

  case "getProposal":
     return mcc.getProposal(stub, args)

  default:
     return shim.Error("Invalid invoke function name. Expecting \"proposeUpdate\"
\"addSignature\" \"getProposals\" \"getProposal\" ")
  }
}

func (mcc *ManagementChaincode) proposeUpdate(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
  if len(args) != 2 {
     return shim.Error("incorrect number of arguments - expecting 2: configUpdate,
description")
  }

  // check if the configUpdate is in the correct format: base64 encoded
proto/common.ConfigUpdate
  update, err := base64.StdEncoding.DecodeString(args[0])
  if err != nil {
     return shim.Error(fmt.Sprintf("error happened decoding the configUpdate base64 string:
%v", err))
  }
  if err := proto.Unmarshal(update, &common.ConfigUpdate{}); err != nil {
     return shim.Error(fmt.Sprintf("error happened decoding common.ConfigUpdate: %v", err))
  }

  // create and store the proposal
  proposalID := getProposalID()
  proposal := Proposal{
     ConfigUpdate: args[0],
     Description:  args[1],
  }
  propsosalJSON, err := json.Marshal(proposal)
```

```
  if err != nil {
     return shim.Error("error happened marshalling the new proposal: " + err.Error())
  }
  if err := stub.PutState(string(proposalID), propsosalJSON); err != nil {
     return shim.Error("error happened persisting the new proposal on the ledger: " +
err.Error())
  }
  if err = stub.SetEvent(NewProposalEvent, []byte(proposalID)); err != nil {
     return shim.Error("error happened emitting event: " + err.Error())
  }
  return shim.Success([]byte(fmt.Sprintf("{\"proposal_id\":\"%v\"}", proposalID)))
}

func (mcc *ManagementChaincode) addSignature(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
  if len(args) != 2 {
     return shim.Error("incorrect number of arguments - expecting 2: proposalID, signature")
  }
  proposalID := args[0]
  signature := args[1]

  // check if the signature is in the correct format: base64 encoded
proto/common.ConfigSignature
  sig, err := base64.StdEncoding.DecodeString(signature)
  if err != nil {
     return shim.Error(fmt.Sprintf("error happened decoding the signature base64 string:
%v", err))
  }
  if err := proto.Unmarshal(sig, &common.ConfigSignature{}); err != nil {
     return shim.Error(fmt.Sprintf("error happened decoding common.ConfigSignature: %v",
err))
  }

  creator, err := stub.GetCreator()
  if err != nil {
     return shim.Error("error happened reading the transaction creator: " + err.Error())
  }
  mspID, err := getMSPID(creator)
  if err != nil {
     return shim.Error(err.Error())
  }

  // fetch and update the state of the proposal
  proposalJSON, err := stub.GetState(proposalID)
  if err != nil {
     return shim.Error(fmt.Sprintf("error happened reading proposal with id (%s) to update
signature: %v", proposalID, err))
  }
  if len(proposalJSON) == 0 {
     return shim.Error(fmt.Sprintf("proposal with id (%s) not found", proposalID))
```

```go
  }
  proposal := &Proposal{}
  if err := json.Unmarshal(proposalJSON, proposal); err != nil {
     return shim.Error("error happened unmarshalling the proposal JSON representation to
struct: " + err.Error())
  }
  if proposal.Signatures == nil {
     proposal.Signatures = make(map[string]string)
  }
  proposal.Signatures[mspID] = signature

  // store the updated proposal
  proposalJSONUpdated, err := json.Marshal(proposal)
  if err != nil {
     return shim.Error("error happened marshalling the updated proposal: " + err.Error())
  }
  if err := stub.PutState(proposalID, proposalJSONUpdated); err != nil {
     return shim.Error("error happened persisting the updated proposal on the ledger: " +
err.Error())
  }
  if err = stub.SetEvent(NewSignatureEvent, []byte(proposalID)); err != nil {
     return shim.Error("error happened emitting event: " + err.Error())
  }
  return shim.Success(nil)
}

func (mcc *ManagementChaincode) getProposals(stub shim.ChaincodeStubInterface, args
[]string) pb.Response {
  if len(args) != 0 {
     return shim.Error("incorrect number of arguments - expecting 0")
  }
  proposals := make(map[string]*Proposal)
  proposalIterator, err := stub.GetStateByRange("", "")
  if err != nil {
     return shim.Error("error happened reading keys from ledger: " + err.Error())
  }
  defer proposalIterator.Close()

  for proposalIterator.HasNext() {
     proposalJSON, err := proposalIterator.Next()
     if err != nil {
        return shim.Error("error happened iterating over available proposals: " +
err.Error())
     }
     proposal := &Proposal{}
     if err = json.Unmarshal(proposalJSON.Value, proposal); err != nil {
        return shim.Error("error happened unmarshalling a proposal JSON representation to
struct: " + err.Error())
     }
     proposals[proposalJSON.Key] = proposal
  }

  proposalsJSON, err := json.Marshal(proposals)
  if err != nil {
     return shim.Error("error happened marshalling the update proposals: " + err.Error())
  }
  return shim.Success(proposalsJSON)
}

func (mcc *ManagementChaincode) getProposal(stub shim.ChaincodeStubInterface, args []string)
pb.Response {
  if len(args) != 1 {
     return shim.Error("incorrect number of arguments - expecting 1: proposalID")
  }
  proposalID := args[0]
  proposalJSON, err := stub.GetState(proposalID)
  if err != nil {
     return shim.Error(fmt.Sprintf("error happened reading proposal with id (%v): %v",
proposalID, err))
  }
  if len(proposalJSON) == 0 {
     return shim.Error(fmt.Sprintf("proposal with id (%s) not found", proposalID))
  }
  return shim.Success(proposalJSON)
}

func getProposalID() string {
  bytes := make([]byte, ProposalIDLength)
  for i := range bytes {
     bytes[i] = alphanum[rand.Int()%len(alphanum)]
  }
  return string(bytes)
}

func getMSPID(creator []byte) (string, error) {
  identity := &msp.SerializedIdentity{}
  if err := proto.Unmarshal(creator, identity); err != nil {
     return "", fmt.Errorf("error happened unmarshalling the creator: %v", err)
  }
  return identity.Mspid, nil
}
```