Hyperledger Fabric XML/CALL Adapter

Release 1.0

Blockchain Team/Luxoft LLC

Mar 15, 2018

USER MANUAL

1 Hyperledger Fabric 1
1.1 Network topology . . . 1
1.2 Logical topology . . . 1
1.3 Transaction processing . . . 2

2 XML Call service 3
2.1 Motivation . . . 3
2.2 Implementation information . . . 3
2.2.1 Self-descriptive invocation language . . . 3
2.2.2 JMS transport . . . 3
2.2.3 Protobuf descriptors . . . 4
2.2.4 Outline (or TL;DR!) . . . 4
2.3 Usage in Depth . . . 4
2.3.1 Root-level tag arguments . . . 6
2.3.2 XSD generation . . . 6
2.3.3 Start XML/Call adapter and configuration . . . 6
2.3.4 Logging . . . 7
2.3.5 Error Handling . . . 7

3 References 9

CHAPTER

ONE

HYPERLEDGER FABRIC

Hyperledger Fabric is a distributed ledger platform. It has no notion of cryptocurrency, assets, or any other financial concept, found in platforms like BitCoin, Ethereum, Ripple, and others. Instead it concentrates on being a framework to build distributed applications using distributed storage.

Hyperledger Fabric (Fabric for short) provides the same guaranties as you might expect from any other blockchain platform: it's a distributed, highly available, scalable, secure platform which allows no history modification. *Hyperledger Fabric* is oriented towards private networks, so it delivers a better level of privacy and confidentiality comparing to public-oriented platforms, like BitCoin or Ethereum.

From a developer point of view *Hyperledger Fabric* can be seen as follows:

Network topology

There is a *Fabric* network, which consists or nodes with different roles:

- **peer**: Peer is a node which hosts chaincode, and maintains state database.
- **orderer**: Orderer is a node, which responsible for total transactions ordering.

Logical topology

Fabric network is organized into non-overlapping *channels*. Each channel has its own ledger, set of *peers*, *orderers*, *users*, *chaincode* instances, and *state database*.

Peers and orderers described above. Note that both peer and orderer nodes can participate in several channels at the same time. Other than that channels are isolated from each other. Ledger delivered to the peers and orderers which participate in the channel only. And it is impossible for chaincode to access data from the other channel directly (by querying database or by examining disk content) even if located on the same peer. The only way is to perform regular chaincode member query.

Users is an entity, identified with the private key/certificate pair, which can send *transactions* to the channel. These transactions are stored on the ledger.

Chaincode is an service, created by application developer. Chaincode is instantiated in a channel and consists of a set of members. Each member has a name, set if parameters, and can return a value. *Hyperledger Fabric* supports several languages to create chaincode. Currently these are Go and JavaScript.

Transaction is a tuple *(chaincode-id, member-name, invocation arguments, invocation result, database-delta)*. Transactions are created by application via Fabric SDK, and supplied to the blockchain, as described in "*Transaction Processing*".

Ledger is a persistent sequence of transactions. It's a file, stored on the peers and orderers, which contains the ordered sequence of transactions.

State database is a database which is accessible by chaincode. State database is changed by the ledger transactions, using 'database-delta' field.

Transaction processing

Transactions in *Fabric* go through several stages:

1. Transaction proposal and endorsment. Application creates a transaction proposal which consists of the chaincode member name and supplied arguments, and sends the proposal to the peers for endorsment. Each peer executes the requested member, evaluates the result and database-delta, puts all of these together into transaction object, signs that object, and returns the endorsed transaction to the requesting application. Note that at this stage neither ledger nor state database is modified.
2. Application supplies all the endorsed proposals to the orderers. The orderers validate the endorser signatures, and come to agreement upon the order of transactions.

   Note: This step is done using some consensus algorithm. Currently *fabric* supports *Kafka*-based ordering and *solo* ordering. *Kafka* orderer, as the name implies, uses Kafka cluster to ensure the total transactions order. In this mode consensus is fault-tolerant. Another mode, which is used for development is *solo*. In that mode only a single orderer per channel exists and thus creates a single point of failure.

3. Orderers deliver new transaction in the agreed order to the peers.
4. Peers finally validate transaction *database-delta*, and if there is no conflict between state database as it used to be at the moment of proposal validation (see the step 1), the delta is applied to the database, thus bringing it to the new state.

   Then application is notified whether transaction has finally been accepted or not.

Besides issuing transaction, application can query state database using one of read-only chaincode members. These queries do not change the state database and consists of a single peer request.

CHAPTER

TWO

XML CALL SERVICE

Motivation

Hyperledger Fabric provides two more or less mature SDKs currently:

- Node.js SDK
- Java SDK

Both SDKs implement the full transaction processing, as described in "*Transaction processing*", so they are pretty low-level, highly *Hyperledger Fabric* specific, and require good knowledge of *Hyperledger Fabric* transaction processing.

Hyperledger Fabric doesn't enforce any structure on the parameters. Each chaincode member accepts an array of byte strings, and is free to interpret these in any way an application developer feels appropriate.

xmlcall provides simple to use interface, with well-defined interface which relies on widely used and accepted techniques like JMS and XML.

Implementation information

xmlcall is developed on top of these three things:

1. Self-descriptive invocation language.
2. JMS transport
3. protobuf descriptors.

Self-descriptive invocation language

We used XML as a language of choice to accept requests and generate results. XML is a broadly used language in the Enterprise community with lots of tools to create, parse, validate, transform, and so on for almost each programming language.

JMS transport

We use JMS at transport layer. *xmlcall* has been developed and tested with Apache ActiveMQ and Spring framework.

To use the *xmlcall* service it is necessary to send plain JMS text message with XML document embedded with no other escapes to process. Results are returned in the same way.

Protobuf descriptors

xmlcall uses propobuf as its IPC description language. Protobuf was chosen due to some reasons, among then:

1. It's a feature-rich language, which allows us to describe almost all the aspects and details of invocation.
2. It has broad community and extensive support from both community and Google.

Outline (or TL;DR!)

As a general outline, the xmlcall adapter is used like that:

1. Interface to chaincode is defined using *protobuf* proto files. Chaincode is defined as `service`, and all the members are defined as `rpc` entries.
2. *.proto* files are compiled to descriptors using `protoc` compiler to build descriptor files.
3. Descriptor files can be used to generate XSD schema for the sake of application development.
4. XmlCall adapter starts with these descriptor files and accepts JMS requests as `<ServiceName.MethodName>` XML document.
5. Invocation or query request submitted to blockchain.
6. When result is ready, reply is send to application via `<TypeName>` XML document.
7. If call fails, result is delivered as `<ChaincodeFault>` XML document.

Usage in Depth

In order to use *xmlcall* an blockchain service should be described as a protobuf service.

Note: *xmlcall* itself has no relation to gRPC, it only uses the augmented gRPC descriptors.

Imagine we have an `Counter` chaincode, exposing following members with obvious semantics:

- `addAndGet(integer) -> integer`
- `getValue() -> integer`

Start with describing the necessary types in protobuf:

Listing 2.1: counter.proto

```
syntax = "proto3";
package counter;

// a type to be used as an argument and result
message Value
{
    int32 value = 1;
}
```

Now, `Value` message could be passed to the chaincode members. Let's sketch it now (use Java as a prototype language - implementation is not important to us):

Listing 2.2: Counter service mock

```
class Counter {
    int current = 0;

    public Value addAndGet(Value value) {
        current = value.getValue();
    }

    public Value getValue() {
        return Value.newBuilder().setValue(current).build();
    }
}
```

Having `Value` message defined, add a service information to the counter.proto:

Listing 2.3: counter.proto modified version

```
// counter.proto
syntax = "proto3";
package counter;

//include the necessary xmlcall definitions
import "xmlcall.proto";

// include Empty message to follow protobuf's conventions
import "google/protobuf/empty.proto";

// a type to be used as an argument and result
message Value
{
    int32 value = 1;
}

service Counter {
   rpc addAndGet(Value) returns (Value) {
       option(xmlcall.exec_type) = INVOKE;
   }
   rpc getValue(google.protobuf.Empty) returns (Value) {
       option(xmlcall.exec_type) = QUERY;
   }
}
```

So, our `Counter` service contains two members defined: `addAndGet` and `getValue`. Note that `getValue` member follows gRPC's convention: each service member accepts exactly one argument and returns one argument.

The `xmlcall.exec_type` option is mandatory and declared how corresponding method should be executed - as a transaction invocation or as a query.

Next step is to generate protobuf descriptors out of these:

```
$ protoc --descriptor_set_out=counter.desc  --include_imports \
         counter.proto
```

This command generates protobuf's descriptor file, which contains all the information from compiled files - all the types, service, et cetera.

Note: Both Gradle or Maven support options to generate descriptor file. Refer respective plugin documentation for more info.

xmlcall would read this file and marshal requests using these types.

So now it would accept following XML request:

Listing 2.4: addAndGet request content

```
<Counter.addAndGet
     in.channel="counter-channel"
     in.chaincodeId="counter-chaincode-id">
   <value>10</value>
</Counter.addAndGet>
```

And, assuming current counter state is 1, would reply with following XML document:

Listing 2.5: addAndGet request reply

```
<main.Value
     out.txid="<some transaction id string>">
   <value>11</value>
</main.Value>
```

Root-level tag arguments

XML/Call expects root tag to contain some attribtes, which identify call info.

- Input attributes are:
  - ***in.chaincode*: chaincode to call. Use exactly the name, the** chaincode was deployed with.
  - *in.channel*: Channel to call the chaincode in.
- Output attributes are:
  - ***out.txid*: If a transactionrequest was supplied, this includes** the transaction id (the hex string). For query this field is meaningless.

XSD generation

Sometimes it might be useful to convert protobuf descriptor into XML schema (XSD).

xmlcall provides utility which can do it: `com.luxoft.xmlcall.wsdl.proto2wsdl.Main`

`proto2wsdl` might be used to generate single XSD file, which contains all the necessary definitions:

```
$ java -jar xmlcall.jar com.luxoft.xmlcall.wsdl.proto2wsdl.Main \
       -schema \
       -output <target-file> \
       counter.desc
```

`target-file` specifies the output file name.

If it is necessary to have separate files, `proto2wsdl` might be used to generate single xsd per service member:

```
$ java -jar xmlcall.jar com.luxoft.xmlcall.wsdl.proto2wsdl.Main \
       -schema-set \
       -output <target-dir> \
       counter.desc
```

Start XML/Call adapter and configuration

As mentioned above, xmlcall adapter created using Spring and thus configured using spring properties:

Note: More information on configuring Spring applications can be found in official documentation.

- *descriptorFileName*: compiled descriptor file name. Must be specified.
- *xmlCallJmsDestination*: JMS topic name to listen on. Default value is 'blockchain-xmlcall'.
- *connectorClass*: java class to connect to blockchain. Default value is "XmlCallFabricConnector", which implements connection to *Hyperledger Fabric* using *fabric-utils* semantics (refer *Fabric Utils* for details).

  Otherwise it should be a full class name.
- *connectorArg*: connector-specific argument. for *XmlCallFabricConnector* this is a path to *config.yaml* (refer *Fabric Utils* for details).
- *spring.activemq.broker-url*: is a ActiveMQ address, set to "tcp://localhost:61616" by default

> **Note:** *xmlcall* uses *Apache ActiveMQ* broker as JMS service, look for documentation for configuration details. '

Full example looks like that:

Listing 2.6: application.yml

```
descriptorFileName: data/proto/services.desc
xmlCallJmsDestination: blockchain-xmlcall
spring.activemq.broker-url: tcp://localhost:61616
connectorClass: XmlCallFabricConnector
connectorArg: config.yaml
```

Logging

xmlcall compiled with slf4j logger, backed by logback. Refer respective documentation for configuration details.

Error Handling

If something went wrong with chaincode invocation, an error is described in *xmlcall* logs (including stack trace), and for the client application an XML document generated:

Listing 2.7: Fault reply

```
<ChaincodeFault>
   <message>...</message>
</ChaincodeFault>
```

CHAPTER

THREE

REFERENCES

- Fabric Utils: Java library, developed by Luxoft's blockchain team to access *Hyperledger Fabric*. Details can be found in *"Fabric Utils" User Manual*.
- Protobuf language reference: https://developers.google.com/protocol-buffers/docs/proto
- Hyperledger Fabric: https://www.hyperledger.org/projects/fabric Blockchain implmentation drived by IBM under Linux Foundation.