ARC

> Transaction processor for Bitcoin

Overview

ARC is a transaction processor for Bitcoin. It consists of four microservices: API, Metamorph, BlockTx and Callbacker, which are all described below.

All the microservices are written in Go, and use gRPC and Protobuf for internal communications.

All the microservices are designed to be horizontally scalable, and can be deployed on a single machine or on multiple machines. Each one has been programmed with a store interface and various databases can be used to store data. The default store is sqlite3, but any database that implements the store interface can be used.

Extended format

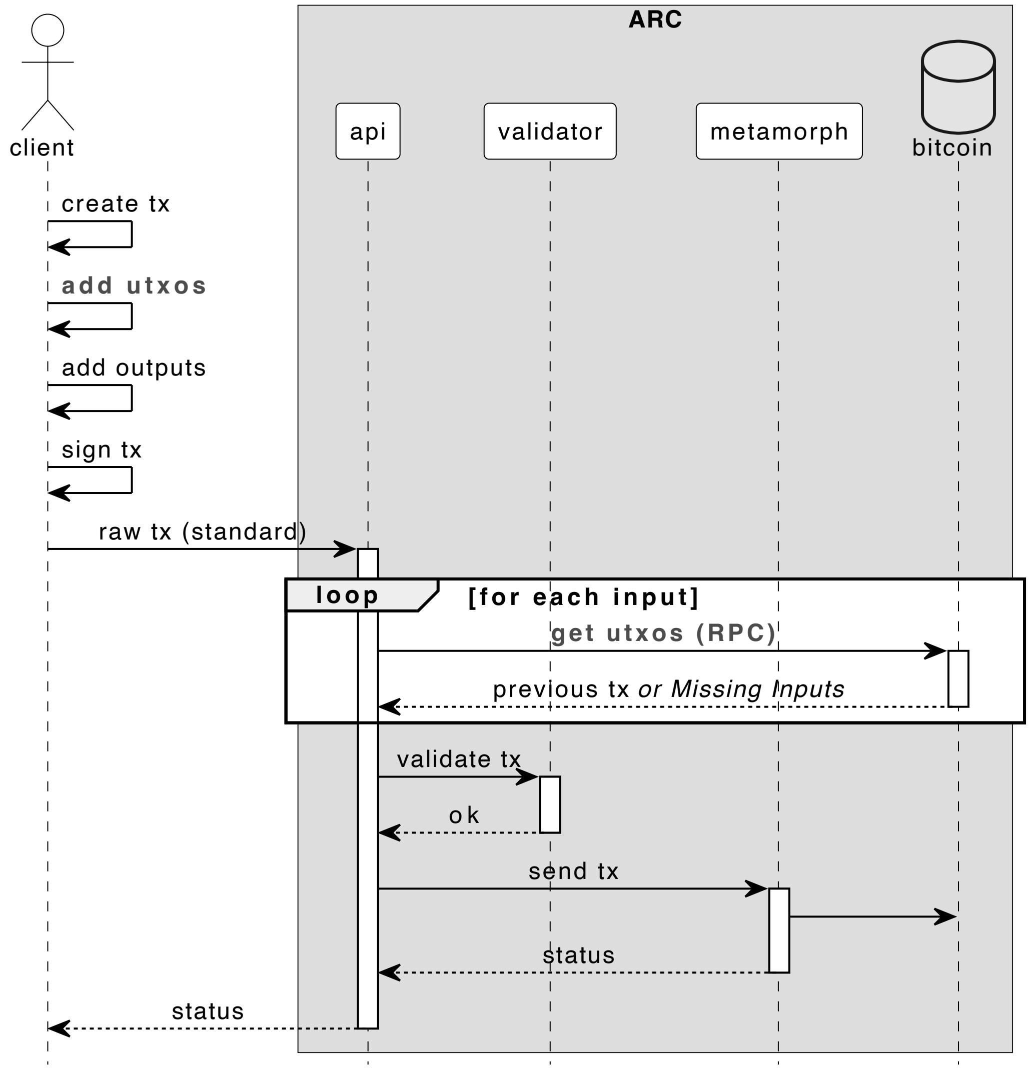
For optimal performance, ARC uses a custom format for transactions. This format is called the extended format, and is a superset of the raw transaction format. The extended format includes the satoshis and scriptPubKey for each input, which makes it possible for ARC to validate the transaction without having to download the parent transactions. In most cases the sender already has all the information from the parent transaction, as this is needed to sign the transaction.

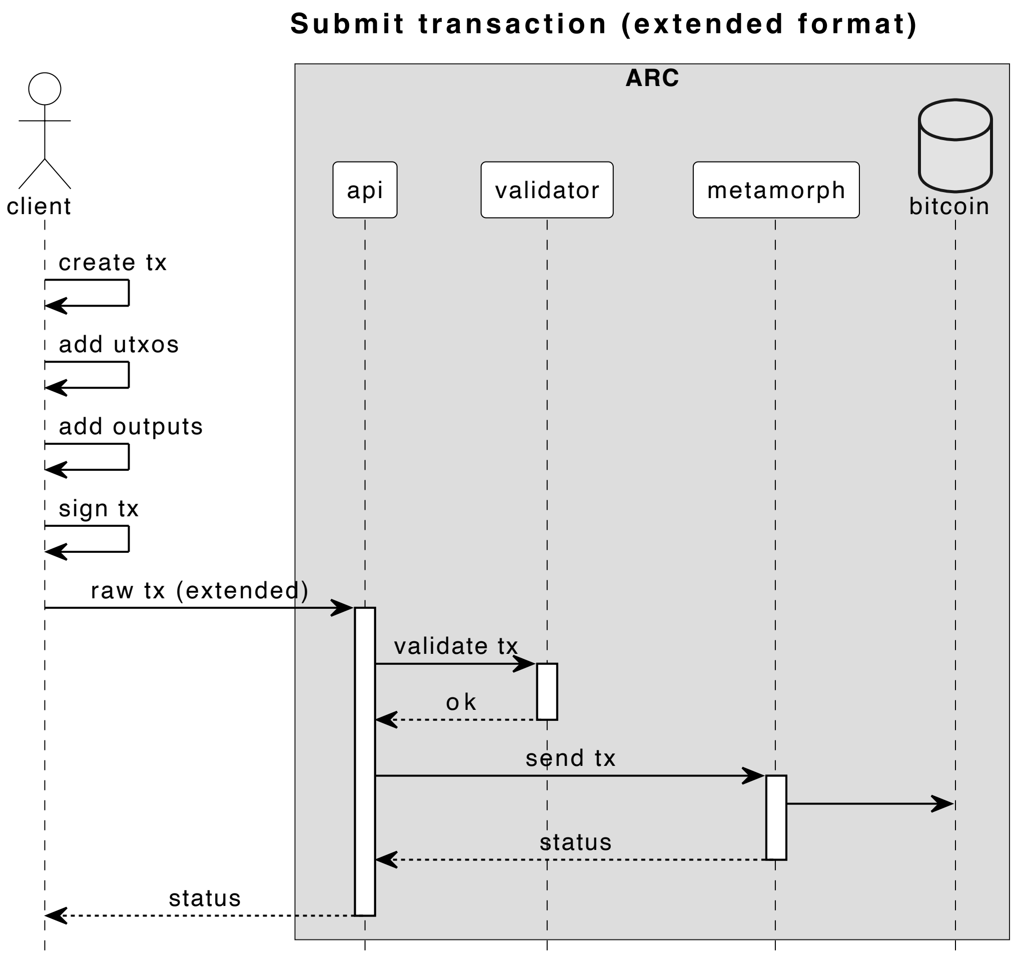
With the successful adoption of Bitcoin ARC, this format should establish itself as the new standard of interchange between wallets and non-mining nodes on the network.

The extended format has been described in detail in BIP-239.

The following diagrams show the difference between validating a transaction in the standard and extended format:

Submit transaction (standard format)





As you can see, the extended format is much more efficient, as it does not require any RPC calls to the Bitcoin node.

This validation takes place in the ARC API microservice. The actual utxos are left to be checked by the Bitcoin node itself, like it would do anyway, regardless of where the transactions is coming from. With this process flow we save the node from having to lookup and send the input utxos to the ARC API, which could be slow under heavy load.

Microservices

To run all the microservices in one process (during development), use the `main.go` file in the root directory.

```shell
go run main.go
```

The `main.go` file accepts the following flags (`main.go --help`):

```shell
usage: main [options]
where options are:

    -api=<true|false>
          whether to start ARC api server (default=true)

    -metamorph=<true|false>
          whether to start metamorph (default=true)

    -blocktx=<true|false>
          whether to start block tx (default=true)

    -callbacker=<true|false>
          whether to start callbacker (default=true)

    -tracer=<true|false>
          whether to start the Jaeger tracer (default=false)
```

NOTE: If you start the `main.go` with a microservice set to true, it will not start the other services. For example, if you run `go run main.go -api=true` , it will only start the API server, and not the other services, although you can start multiple services by specifying them on the command line.

API

API is the REST API microservice for interacting with ARC. See the API documentation for more information.

The API takes care of authentication, validation, and sending transactions to Metamorph. The API talks to one or more Metamorph instances using client-based, round robin load balancing.

You can run the API like this:

```shell
go run cmd/api/main.go
```

or using the generic `main.go` :

```shell
go run main.go -api=true
```

The only difference between the two is that the generic `main.go` starts the Go profiler, while the specific `cmd/api/main.go` command does not.

Integration into an echo server

If you want to integrate the ARC API into an existing echo server, check out the `examples` folder in the GitHub repo.

A very simple example:

```go
package main

import (
    "fmt"

    "github.com/TAAL-GmbH/arc/api"
    apiHandler "github.com/TAAL-GmbH/arc/api/handler"
    "github.com/TAAL-GmbH/arc/api/transactionHandler"
    "github.com/labstack/echo/v4"
)

func main() {

    // Set up a basic Echo router
    e := echo.New()

    // add a single bitcoin node
    txHandler, err := transactionHandler.NewBitcoinNode("localho
    if err != nil {
        panic(err)
    }

    // initialise the arc default api handler, with our txHandle
    var handler api.HandlerInterface
    if handler, err = apiHandler.NewDefault(txHandler); err != n
        panic(err)
    }
    // Register the ARC API
    // the arc handler registers routes under /v1/...
    api.RegisterHandlers(e, handler)
    // or with a base url => /mySubDir/v1/...
    // arc.RegisterHandlersWithBaseURL(e. blocktx_api, "/arc")

    // Serve HTTP until the world ends.
    e.Logger.Fatal(e.Start(fmt.Sprintf("%s:%d", "0.0.0.0", 8080)
}
```

This will initialise the ARC API with a single Bitcoin node (not Metamorph), similar to how MAPI is run at the moment.

Metamorph

Metamorph is a microservice that is responsible for processing transactions sent by the API to the Bitcoin network. It takes care of re-sending transactions if they are not acknowledged by the network within a certain time period (60 seconds by default).

Metamorph is designed to be horizontally scalable, with each instance operating independently and having its own transaction store. As a result, they do not communicate with each other and remain unaware of each other's existence.

You can run metamorph like this:

```shell
go run cmd/metamorph/main.go
```

or using the generic `main.go` :

```shell
go run main.go -metamorph=true
```

The only difference between the two is that the generic `main.go` starts the Go profiler, while the specific `cmd/metamorph/main.go` command does not.

Metamorph transaction statuses

Metamorph keeps track of the lifecycle of a transaction, and assigns it a status. The following statuses are available:

Code	Status	Description
0	`UNKNOWN`	The transaction has been sent to metamorph, but no processing has taken place. This should never be the case, unless something goes wrong.
1	`QUEUED`	The transaction has been queued for processing.
2	`RECEIVED`	The transaction has been properly received by the metamorph processor.
3	`STORED`	The transaction has been stored in the metamorph store. This should ensure the transaction will be processed and retried if not picked up immediately by a mining node.
4	`ANNOUNCED_TO_NETWORK`	The transaction has been announced (INV message) to the Bitcoin network.
5	`REQUESTED_BY_NETWORK`	The transaction has been requested from metamorph by a Bitcoin node.
6	`SENT_TO_NETWORK`	The transaction has been sent to at least 1 Bitcoin node.
7	`ACCEPTED_BY_NETWORK`	The transaction has been accepted by a connected Bitcoin node on the ZMQ interface. If metamorph is not connected to ZQM, this status will never by set.
8	`SEEN_ON_NETWORK`	The transaction has been seen on the Bitcoin network and propagated to other nodes. This status is set when metamorph receives an INV message for the transaction from another node than it was sent to.
9	`MINED`	The transaction has been mined into a block by a mining node.
108	`CONFIRMED`	The transaction is marked as confirmed when it is in a block with 100 blocks built on top of that block.
109	`REJECTED`	The transaction has been rejected by the Bitcoin network.

This status is returned in the `txStatus` field whenever the transaction is queried.

Metamorph stores

The metamorph store has been implemented for multiple databases, depending on your needs. In high-volume environments, you may want to use a database that is optimized for high throughput, such as Badger.

The following databases have been implemented:

- Sqlite3 (`sqlite` or `sqlite_memory` for in-memory)
- Postgres (`postgres`)
- Badger (`badger`)
- BadgerHold (`badgerhold`)

You can select the store to use by setting the `metamorph_dbMode` in the settings file or adding `metamorph_dbMode` as an environment variable.

Connections to Bitcoin nodes

Metamorph can connect to multiple Bitcoin nodes, and will use a subset of the nodes to send transactions to. The other nodes will be used to listen for transaction INV message, which will trigger the SEEN_ON_NETWORK status of a transaction.

The Bitcoin nodes can be configured in the settings file.

Whitelisting

Metamorph is talking to the Bitcoin nodes over the p2p network. If metamorph sends invalid transactions to the Bitcoin node, it will be **banned** by that node. Either make sure not to send invalid or double spend transactions through metamorph, or make sure that all metamorph servers are **whitelisted** on the Bitcoin nodes they are connecting to.

ZMQ

Although not required, zmq can be used to listen for transaction messages (`hashtx` , `invalidtx` , `discardedfrommempool`). This is especially useful if you are not connecting to multiple Bitcoin nodes, and therefore are not receiving INV messages for your transactions.

If you want to use zmq, you can set the `peer_%d_zmq` settings to the full url (example `zmq://bitcoin:bitcoin@localhost:18332`) of the zmq endpoints in the settings file.

ZMQ does seem to be a bit faster than the p2p network, so it is recommended to turn it on, if available.

BlockTx

BlockTx is a microservice that is responsible for processing blocks mined on the Bitcoin network, and for propagating the status of transactions to each Metamorph that has subscribed to this service.

The main purpose of BlockTx is to de-duplicate processing of (large) blocks. As an incoming block is processed by BlockTx, each Metamorph is notified of transactions that they have registered an interest in. BlockTx does not store the transaction data, but instead stores only the transaction IDs and the block height in which they were mined. Metamorph is responsible for storing the transaction data.

You can run BlockTx like this:

```shell
go run cmd/blocktx/main.go
```

or using the generic `main.go` :

```shell
go run main.go -blocktx=true
```

The only difference between the two is that the generic `main.go` starts the Go profiler, while the specific `cmd/blocktx/main.go` command does not.

BlockTx stores

The BlockTx store has been implemented for multiple databases, depending on your needs. In high-volume environments, you may want to use a database that is optimized for high throughput, such as Postgres.

The following databases have been implemented:

- Sqlite3 (`sqlite` or `sqlite_memory` for in-memory)
- Postgres (`postgres`)

You can select the store to use by setting the `blocktx_dbMode` in the settings file or adding `blocktx_dbMode` as an environment variable.

Please note that if you are running multiple instances of BlockTX for resilience, each BlockTx can be configured to use a shared database and in this case, Postgres is probably a sensible choice.

Callbacker

Callbacker is a very simple microservice that is responsible for sending callbacks to clients when a transaction has been accepted by the Bitcoin network. To register a callback, the client must add the `X-Callback-Url` header to the request. The callbacker will then send a POST request to the URL specified in the header, with the transaction ID in the body. See the API documentation for more information.

You can run the Callbacker service like this:

```shell
go run cmd/callbacker/main.go
```

or using the generic `main.go` :

```shell
go run main.go -callbacker=true
```

Settings

The settings available for running ARC are managed by gocore. The settings are defined in 2 files `settings_local.conf` , in the root directory, and `settings.conf` , which lives a directory level deeper. The settings in `settings_local.conf` override the settings in `settings.conf` and any settings that are specified as environment variables, will superceed them. The environment variable name is the same as the setting name. For example, the setting `metamorph_dbMode` can be overridden by setting the environment variable `metamorph_dbMode=... go run main.go` .

For more detailed information about the settings, please see the gocore documentation.

ARC stats

`gocore` keeps real-time stats about the metamorph servers, which can be viewed at `/stats` (e.g. `http://localhost:8011/stats`). These stats show aggregated information about a metamorph server, such as the number of transactions processed, the number of transactions sent to the Bitcoin network, etc. It also shows the average time it takes for each step in the process.

More detailed statistics are available at `/pstats` (e.g. `http://localhost:8011/pstats`). These stats show information about the internal metamorph processor. The processor stats also allows you to see details for a single transaction. If a transaction has already been mined, and evicted from the processor memory, you can still see the stored stats retrieved from the data store, and potentially the timing stats, if they are found in the log file.

ARC can also expose a Prometheus endpoint that can be used to monitor the metamorph servers. Set the `prometheusEndpoint` setting in the settings file to activate prometheus. Normally you would want to set this to `/metrics`.

Client Libraries

Go

WIP

Javascript

A typescript library is available in the js-arc-client repository.

Example usage:

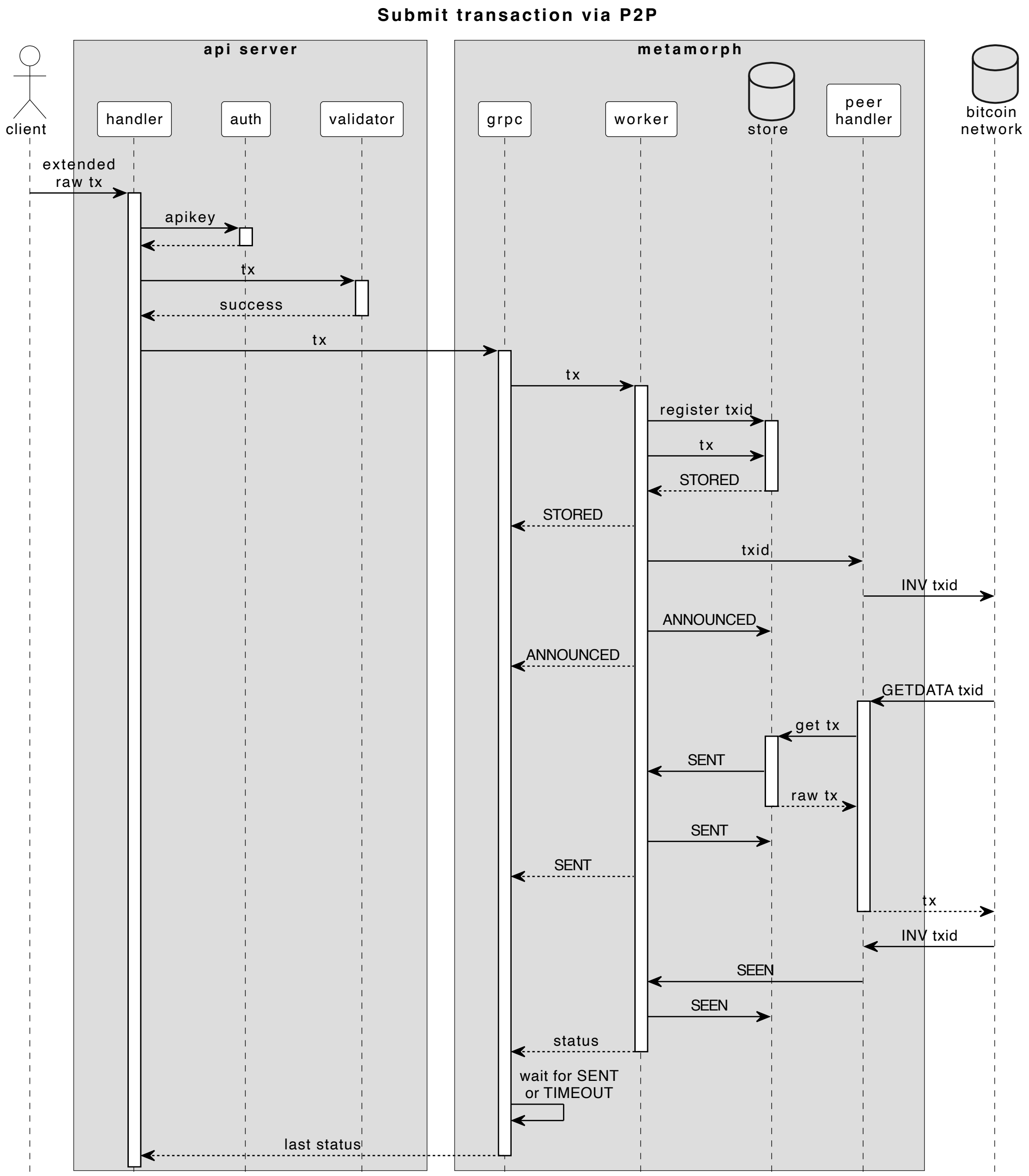
```javascript
import { ArcClient } from 'js-arc-client';

const arcClient = new ArcClient({
  host: 'localhost',
  port: 8080,
  apiKey: '<api-key>'
});
```

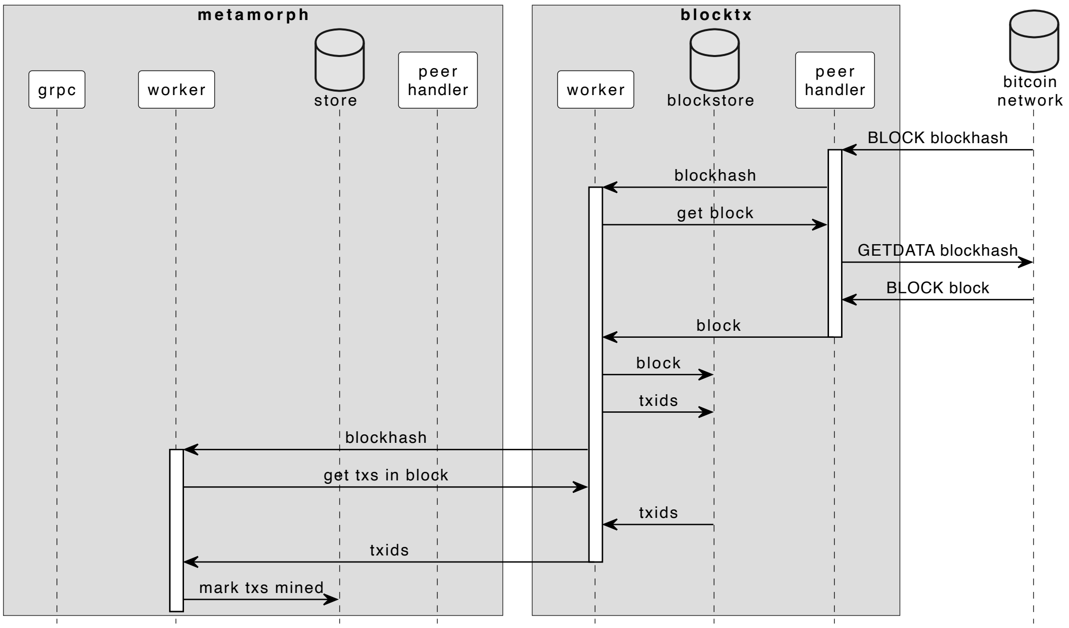
```
const txid = 'd4b0e1b0c0b0c0b0c0b0c0b0c0b0c0b0c0b0c0b0c0b0c0b0c0
const result = await arcClient.getTransactionStatus(txid);
```

See the repository for more information.

Process flow diagrams



Process block via P2P


metamorph
blocktx
grpc
worker
store
peer handler
worker
blockstore
peer handler
bitcoin network
BLOCK blockhash
blockhash
get block
GETDATA blockhash
BLOCK block
block
block
txids
blockhash
get txs in block
txids
txids
mark txs mined