ETHEREUM GUIDE

GITHUB SOURCE

Home

Felix Lange edited this page on 21 Dec 2017 · 65 revisions

- User documentation can be found at our Ethereum User Guide and reference manual.
- For the API reference and developer documentation head over to the auto generated GoDoc documentation.

This is the Wiki for the official Ethereum golang implementation. For generic Ethereum-related information (whitepaper, yellow paper, protocol and interface specs, APIs, DAPP development guides, etc) see the Ethereum main wiki.

Main entry points:

- Installation Instructions
- Developer/Management API
- Managing Accounts
- Command Line Options
- JavaScript Console
- Private Network
- Developers' Guide
- Whisper v5

Sidebar lists all pages.

Installation Instructions for Mac

ligi edited this page 6 days ago · 7 revisions

Installing with Homebrew

By far the easiest way to install go-ethereum is to use our Homebrew tap. If you don't have Homebrew, install it first.

Then run the following commands to add the tap and install `geth`:

```
brew tap ethereum/ethereum
brew install ethereum
```

You can install the develop branch by running `--devel`:

```
brew install ethereum --devel
```

After installing, run `geth account new` to create an account on your node.

You should now be able to run `geth` and connect to the network.

Make sure to check the different options and commands with `geth --help`

For options and patches, see: https://github.com/ethereum/homebrew-ethereum

Building from source

Building Geth (command line client)

Clone the repository to a directory of your choosing:

```
git clone https://github.com/ethereum/go-ethereum
```

Building `geth` requires the Go compiler:

```
brew install go
```

Finally, build the `geth` program using the following command.

```
cd go-ethereum
make geth
```

If you see some errors related to header files of Mac OS system library, install XCode Command Line Tools, and try again.

```
xcode-select --install
```

You can now run `build/bin/geth` to start your node.

Management APIs

Péter Szilágyi edited this page on 16 Nov 2017 · 38 revisions

Beside the official DApp APIs interface go-ethereum has support for additional management APIs. Similar to the DApp APIs, these are also provided using JSON-RPC and follow exactly the same conventions. Geth comes with a console client which has support for all additional APIs described here.

Enabling the management APIs

To offer these APIs over the Geth RPC endpoints, please specify them with the `--${interface}api` command line argument (where `${interface}` can be `rpc` for the HTTP endpoint, `ws` for the WebSocket endpoint and `ipc` for the unix socket (Unix) or named pipe (Windows) endpoint).

For example: `geth --ipcapi admin,eth,miner --rpcapi eth,web3 --rpc`

- Enables the admin, official DApp and miner API over the IPC interface
- Enables the official DApp and web3 API over the HTTP interface

The HTTP RPC interface must be explicitly enabled using the `--rpc` flag.

Please note, offering an API over the HTTP (`rpc`) or WebSocket (`ws`) interfaces will give everyone access to the APIs who can access this interface (DApps, browser tabs, etc). Be careful which APIs you enable. By default Geth enables all APIs over the IPC (`ipc`) interface and only the `db`, `eth`, `net` and `web3` APIs over the HTTP and WebSocket interfaces.

To determine which APIs an interface provides, the `modules` JSON-RPC method can be invoked. For example over an `ipc` interface on unix systems:

```
echo '{"jsonrpc":"2.0","method":"rpc_modules","params":[],"id":1}' | nc -U $datadir/geth.ipc
```

will give all enabled modules including the version number:

```
{
   "id":1,
   "jsonrpc":"2.0",
   "result":{
      "admin":"1.0",
```

```
      "db":"1.0",
      "debug":"1.0",
      "eth":"1.0",
      "miner":"1.0",
      "net":"1.0",
      "personal":"1.0",
      "shh":"1.0",
      "txpool":"1.0",
      "web3":"1.0"
   }
}
```

Consuming the management APIs

These additional APIs follow the same conventions as the official DApp APIs. Web3 can beextended and used to consume these additional APIs.

The different functions are split into multiple smaller logically grouped APIs. Examples are given for the JavaScript console but can easily be converted to an RPC request.

2 examples:

- Console: `miner.start()`
- IPC: `echo '{"jsonrpc":"2.0","method":"miner_start","params":[],"id":1}' | nc -U $datadir/geth.ipc`
- HTTP: `curl -X POST --data '{"jsonrpc":"2.0","method":"miner_start","params":[],"id":74}' localhost:8545`

With the number of THREADS as an arguments:

- Console: `miner.start(4)`
- IPC: `echo '{"jsonrpc":"2.0","method":"miner_start","params":[4],"id":1}' | nc -U $datadir/geth.ipc`
- HTTP: `curl -X POST --data '{"jsonrpc":"2.0","method":"miner_start","params":[4],"id":74}' localhost:8545`

List of management APIs

Beside the officially exposed DApp API namespaces (`eth`, `shh`, `web3`), Geth provides the following extra management API namespaces:

- `admin`: Geth node management
- `debug`: Geth node debugging
- `miner`: Miner and DAG management
- `personal`: Account management
- `txpool`: Transaction pool inspection

admin	**debug**	**miner**	**personal**	**txpool**
addPeer	backtraceAt	setExtra	ecRecover	content
datadir	blockProfile	setGasPrice	importRawKey	inspect
nodeInfo	cpuProfile	start	listAccounts	status
peers	dumpBlock	stop	lockAccount	
setSolc	gcStats	getHashrate	newAccount	
startRPC	getBlockRlp	setEtherbase	unlockAccount	
startWS	goTrace		sendTransaction	
stopRPC	memStats		sign	
stopWS	seedHashsign			
	setBlockProfileRate			

	setHead			
	stacks			
	startCPUProfile			
	startGoTrace			
	stopCPUProfile			
	stopGoTrace			
	traceBlock			
	traceBlockByNumber			
	traceBlockByHash			
	traceBlockFromFile			
	traceTransaction			
	verbosity			
	vmodule			
	writeBlockProfile			
	writeMemProfile			

Admin

The `admin` API gives you access to several non-standard RPC methods, which will allow you to have a fine grained control over your Geth instance, including but not limited to network peer and RPC endpoint management.

admin_addPeer

The `addPeer` administrative method requests adding a new remote node to the list of tracked static nodes. The node will try to maintain connectivity to these nodes at all times, reconnecting every once in a while if the remote connection goes down.

The method accepts a single argument, the `enode` URL of the remote peer to start tracking and returns a `BOOL` indicating whether the peer was accepted for tracking or some error occurred.

Client	Method invocation
Go	`admin.AddPeer(url string) (bool, error)`
Console	`admin.addPeer(url)`
RPC	`{"method": "admin_addPeer", "params": [url]}`

Example

```
> admin.addPeer("enode://a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69ad0dce72a4d8db5ebb4968de0e3bec910127f134779fbcb0cb6d3331163c@52.16.188.185:30303")
true
```

admin_datadir

The `datadir` administrative property can be queried for the absolute path the running Geth node currently uses to store all its databases.

Client	Method invocation
Go	`admin.Datadir() (string, error)`
Console	`admin.datadir`
RPC	`{"method": "admin_datadir"}`

Example

```
> admin.datadir
"/home/karalabe/.ethereum"
```

admin_nodeInfo

The `nodeInfo` administrative property can be queried for all the information known about the running Geth node at the networking granularity. These include general information about the node itself as a participant of the ÐΞVp2p P2P overlay protocol, as well as specialized information added by each of the running application protocols (e.g. `eth`, `les`, `shh`, `bzz`).

Client	Method invocation
Go	`admin.NodeInfo() (*p2p.NodeInfo, error)`
Console	`admin.nodeInfo`
RPC	`{"method": "admin_nodeInfo"}`

Example

```
> admin.nodeInfo
{
  enode:
"enode://44826a5d6a55f88a18298bca4773fca5749cdc3a5c9f308aa7d810e9b31123f3e7c5fba0b
1d70aac5308426f47df2a128a6747040a3815cc7dd7167d03be320d@[::]:30303",
  id:
"44826a5d6a55f88a18298bca4773fca5749cdc3a5c9f308aa7d810e9b31123f3e7c5fba0b1d70aac5
308426f47df2a128a6747040a3815cc7dd7167d03be320d",
  ip: "::",
  listenAddr: "[::]:30303",
  name: "Geth/v1.5.0-unstable/linux/go1.6",
  ports: {
    discovery: 30303,
    listener: 30303
  },
  protocols: {
    eth: {
      difficulty: 17334254859343145000,
      genesis:
"0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3",
      head: "0xb83f73fbe6220c111136aefd27b160bf4a34085c65ba89f24246b3162257c36a",
      network: 1
    }
  }
}
```

admin_peers

The `peers` administrative property can be queried for all the information known about the connected remote nodes at the networking granularity. These include general information about the nodes themselves as participants of the ÐΞVp2p P2P overlay protocol, as well as specialized information added by each of the running application protocols (e.g. `eth`, `les`, `shh`, `bzz`).

Client	Method invocation
Go	`admin.Peers() ([]*p2p.PeerInfo, error)`
Console	`admin.peers`
RPC	`{"method": "admin_peers"}`

Example

```
> admin.peers
[{
    caps: ["eth/61", "eth/62", "eth/63"],
    id:
"08a6b39263470c78d3e4f58e3c997cd2e7af623afce64656cfc56480babcea7a9138f3d09d7b98793
44c2d2e457679e3655d4b56eaff5fd4fd7f147bdb045124",
    name: "Geth/v1.5.0-unstable/linux/go1.5.1",
    network: {
      localAddress: "192.168.0.104:51068",
      remoteAddress: "71.62.31.72:30303"
    },
    protocols: {
      eth: {
        difficulty: 17334052235346465000,
        head: "5794b768dae6c6ee5366e6ca7662bdff2882576e09609bf778633e470e0e7852",
        version: 63
      }
    }
}, /* ... */ {
    caps: ["eth/61", "eth/62", "eth/63"],
    id:
```

```
"fcad9f6d3faf89a0908a11ddae9d4be3a1039108263b06c96171eb3b0f3ba85a7095a03bb65198c35
a04829032d198759edfca9b63a8b69dc47a205d94fce7cc",
    name: "Geth/v1.3.5-506c9277/linux/go1.4.2",
    network: {
      localAddress: "192.168.0.104:55968",
      remoteAddress: "121.196.232.205:30303"
    },
    protocols: {
      eth: {
        difficulty: 17335165914080772000,
        head: "5794b768dae6c6ee5366e6ca7662bdff2882576e09609bf778633e470e0e7852",
        version: 63
      }
    }
}]
```


admin_setSolc

The `setSolc` administrative method sets the Solidity compiler path to be used by the node when invoking the `eth_compileSolidity` RPC method. The Solidity compiler path defaults to `/usr/bin/solc` if not set, so you only need to change it for using a non-standard compiler location.

The method accepts an absolute path to the Solidity compiler to use (specifying a relative path would depend on the current – to the user unknown – working directory of Geth), and returns the version string reported by `solc --version`.

Client	Method invocation
Go	`admin.SetSolc(path string) (string, error)`
Console	`admin.setSolc(path)`
RPC	`{"method": "admin_setSolc", "params": [path]}`

Example

```
> admin.setSolc("/usr/bin/solc")
"solc, the solidity compiler commandline interface\nVersion:
0.3.2-0/Release-Linux/g++/Interpreter\n\npath: /usr/bin/solc"
```

admin_startRPC

The `startRPC` administrative method starts an HTTP based JSON RPC API webserver to handle client requests. All the parameters are optional:

- `host`: network interface to open the listener socket on (defaults to `"localhost"`)
- `port`: network port to open the listener socket on (defaults to `8545`)
- `cors`: cross-origin resource sharing header to use (defaults to `""`)
- `apis`: API modules to offer over this interface (defaults to `"eth,net,web3"`)

The method returns a boolean flag specifying whether the HTTP RPC listener was opened or not. Please note, only one HTTP endpoint is allowed to be active at any time.

Client	Method invocation
Go	`admin.StartRPC(host *string, port *rpc.HexNumber, cors *string, apis *string) (bool, error)`
Console	`admin.startRPC(host, port, cors, apis)`
RPC	`{"method": "admin_startRPC", "params": [host, port, cors, apis]}`

Example

```
> admin.startRPC("127.0.0.1", 8545)
true
```

admin_startWS

The `startWS` administrative method starts an WebSocket based JSON RPC API webserver to handle client requests. All the parameters are optional:

- `host`: network interface to open the listener socket on (defaults to `"localhost"`)
- `port`: network port to open the listener socket on (defaults to `8546`)
- `cors`: cross-origin resource sharing header to use (defaults to `""`)
- `apis`: API modules to offer over this interface (defaults to `"eth,net,web3"`)

The method returns a boolean flag specifying whether the WebSocket RPC listener was opened or not. Please note, only one WebSocket endpoint is allowed to be active at any time.

Client	Method invocation
Go	`admin.StartWS(host *string, port *rpc.HexNumber, cors *string, apis *string) (bool, error)`
Console	`admin.startWS(host, port, cors, apis)`
RPC	`{"method": "admin_startWS", "params": [host, port, cors, apis]}`

Example

```
> admin.startWS("127.0.0.1", 8546)
true
```

admin_stopRPC

The `stopRPC` administrative method closes the currently open HTTP RPC endpoint. As the node can only have a single HTTP endpoint running, this method takes no parameters, returning a boolean whether the endpoint was closed or not.

Client	Method invocation
Go	`admin.StopRPC() (bool, error)`
Console	`admin.stopRPC()`
RPC	`{"method": "admin_stopRPC"`

Example

```
> admin.stopRPC()
true
```

admin_stopWS

The `stopWS` administrative method closes the currently open WebSocket RPC endpoint. As the node can only have a single WebSocket endpoint running, this method takes no parameters, returning a boolean whether the endpoint was closed or not.

Client	Method invocation
Go	`admin.StopWS() (bool, error)`
Console	`admin.stopWS()`
RPC	`{"method": "admin_stopWS"`

Example

```
> admin.stopWS()
true
```

Debug

The `debug` API gives you access to several non-standard RPC methods, which will allow you to inspect, debug and set certain debugging flags during runtime.

debug_backtraceAt

Sets the logging backtrace location. When a backtrace location is set and a log message is emitted at that location, the stack of the goroutine executing the log statement will be printed to stderr.

The location is specified as `<filename>:<line>`.

Client	Method invocation
Console	`debug.backtraceAt(string)`
RPC	`{"method": "debug_backtraceAt", "params": [string]}`

Example:

```
> debug.backtraceAt("server.go:443")
```

debug_blockProfile

Turns on block profiling for the given duration and writes profile data to disk. It uses a profile rate of 1 for most accurate information. If a different rate is desired, set the rate and write the profile manually using `debug_writeBlockProfile`.

Client	Method invocation
Console	`debug.blockProfile(file, seconds)`
RPC	`{"method": "debug_blockProfile", "params": [string, number]}`

debug_cpuProfile

Turns on CPU profiling for the given duration and writes profile data to disk.

Client	Method invocation
Console	`debug.cpuProfile(file, seconds)`
RPC	`{"method": "debug_cpuProfile", "params": [string, number]}`

debug_dumpBlock

Retrieves the state that corresponds to the block number and returns a list of accounts (including storage and code).

Client	Method invocation
Go	`debug.DumpBlock(number uint64) (state.World, error)`
Console	`debug.traceBlockByHash(number, [options])`
RPC	`{"method": "debug_dumpBlock", "params": [number]}`

Example

```
> debug.dumpBlock(10)
{
    fff7ac99c8e4feb60c9750054bdc14ce1857f181: {
      balance: "49358640978154672",
      code: "",
      codeHash:
"c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
      nonce: 2,
      root: "56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
      storage: {}
    },
    fffbca3a38c3c5fcb3adbb8e63c04c3e629aafce: {
      balance: "3460945928",
```

```
      code: "",
      codeHash:
"c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470",
      nonce: 657,
      root: "56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
      storage: {}
    }
  },
  root: "19f4ed94e188dd9c7eb04226bd240fa6b449401a6c656d6d2816a87ccaf206f1"
}
```

debug_gcStats

Returns GC statistics.

See https://golang.org/pkg/runtime/debug/#GCStats for information about the fields of the returned object.

Client	Method invocation
Console	`debug.gcStats()`
RPC	`{"method": "debug_gcStats", "params": []}`

debug_getBlockRlp

Retrieves and returns the RLP encoded block by number.

Client	Method invocation
Go	`debug.GetBlockRlp(number uint64) (string, error)`
Console	`debug.getBlockRlp(number, [options])`
RPC	`{"method": "debug_getBlockRlp", "params": [number]}`

References: RLP

debug_goTrace

Turns on Go runtime tracing for the given duration and writes trace data to disk.

Client	Method invocation
Console	`debug.goTrace(file, seconds)`
RPC	`{"method": "debug_goTrace", "params": [string, number]}`

debug_memStats

Returns detailed runtime memory statistics.

See https://golang.org/pkg/runtime/#MemStats for information about the fields of the returned object.

Client	Method invocation
Console	`debug.memStats()`
RPC	`{"method": "debug_memStats", "params": []}`

debug_seedHash

Fetches and retrieves the seed hash of the block by number

Client	Method invocation
Go	`debug.SeedHash(number uint64) (string, error)`
Console	`debug.seedHash(number, [options])`
RPC	`{"method": "debug_seedHash", "params": [number]}`

debug_setHead

Sets the current head of the local chain by block number. Note, this is a destructive action and may severely damage your chain. Use with *extreme* caution.

Client	Method invocation
Go	`debug.SetHead(number uint64)`
Console	`debug.setHead(number)`
RPC	`{"method": "debug_setHead", "params": [number]}`

References: Ethash

debug_setBlockProfileRate

Sets the rate (in samples/sec) of goroutine block profile data collection. A non-zero rate enables block profiling, setting it to zero stops the profile. Collected profile data can be written using `debug_writeBlockProfile`.

Client	Method invocation
Console	`debug.setBlockProfileRate(rate)`
RPC	`{"method": "debug_setBlockProfileRate", "params": [number]}`

debug_stacks

Returns a printed representation of the stacks of all goroutines. Note that the web3 wrapper for this method takes care of the printing and does not return the string.

Client	Method invocation
Console	`debug.stacks()`
RPC	`{"method": "debug_stacks", "params": []}`

debug_startCPUProfile

Turns on CPU profiling indefinitely, writing to the given file.

Client	Method invocation
Console	`debug.startCPUProfile(file)`
RPC	`{"method": "debug_startCPUProfile", "params": [string]}`

debug_startGoTrace

Starts writing a Go runtime trace to the given file.

Client	Method invocation
Console	`debug.startGoTrace(file)`
RPC	`{"method": "debug_startGoTrace", "params": [string]}`

debug_stopCPUProfile

Stops an ongoing CPU profile.

Client	Method invocation
Console	`debug.stopCPUProfile()`
RPC	`{"method": "debug_stopCPUProfile", "params": []}`

debug_stopGoTrace

Stops writing the Go runtime trace.

Client	Method invocation
Console	`debug.startGoTrace(file)`
RPC	`{"method": "debug_stopGoTrace", "params": []}`

debug_traceBlock

The `traceBlock` method will return a full stack trace of all invoked opcodes of all transaction that were included included in this block. Note, the parent of this block must be present or it will fail.

Client	Method invocation
Go	`debug.TraceBlock(blockRlp []byte, config. *vm.Config) BlockTraceResult`
Console	`debug.traceBlock(tblockRlp, [options])`
RPC	`{"method": "debug_traceBlock", "params": [blockRlp, {}]}`

References: RLP

Example

```
> debug.traceBlock("0xblock_rlp")
{
  gas: 85301,
  returnValue: "",
  structLogs: [{
      depth: 1,
      error: "",
      gas: 162106,
      gasCost: 3,
      memory: null,
      op: "PUSH1",
      pc: 0,
      stack: [],
      storage: {}
  },
    /* snip */
  {
      depth: 1,
      error: "",
      gas: 100000,
      gasCost: 0,
      memory: ["0000000000000000000000000000000000000000000000000000000000000006",
"0000000000000000000000000000000000000000000000000000000000000000",
"0000000000000000000000000000000000000000000000000000000000000060"],
      op: "STOP",
      pc: 120,
      stack: ["0000000000000000000000000000000000000000000000000d67cbec9"],
      storage: {
        0000000000000000000000000000000000000000000000000000000000000004:
"8241fa522772837f0d05511f20caa6da1d5a3209000000000000000400000001",
        0000000000000000000000000000000000000000000000000000000000000006:
"0000000000000000000000000000000000000000000000000000000000000001",
        f652222313e28459528d920b65115c16c04f3efc82aaedc97be59f3f377c0d3f:
"0000000000000000000000002e816afc1b5c0f39852131959d946eb3b07b5ad"
      }
  }]
```

debug_traceBlockByNumber

Similar to debug_traceBlock, `traceBlockByNumber` accepts a block number and will replay the block that is already present in the database.

Client	Method invocation
Go	`debug.TraceBlockByNumber(number uint64, config. *vm.Config) BlockTraceResult`
Console	`debug.traceBlockByNumber(number, [options])`
RPC	`{"method": "debug_traceBlockByNumber", "params": [number, {}]}`

References: RLP

debug_traceBlockByHash

Similar to debug_traceBlock, `traceBlockByHash` accepts a block hash and will replay the block that is already present in the database.

Client	Method invocation
Go	`debug.TraceBlockByHash(hash common.Hash, config. *vm.Config) BlockTraceResult`
Console	`debug.traceBlockByHash(hash, [options])`
RPC	`{"method": "debug_traceBlockByHash", "params": [hash {}]}`

References: RLP

debug_traceBlockFromFile

Similar to debug_traceBlock, `traceBlockFromFile` accepts a file containing the RLP of the block.

Client	Method invocation
Go	`debug.TraceBlockFromFile(fileName string, config. *vm.Config) BlockTraceResult`
Console	`debug.traceBlockFromFile(fileName, [options])`
RPC	`{"method": "debug_traceBlockFromFile", "params": [fileName, {}]}`

References: RLP

debug_traceTransaction

The `traceTransaction` debugging method will attempt to run the transaction in the exact same manner as it was executed on the network. It will replay any transaction that may have been executed prior to this one before it will finally attempt to execute the transaction that corresponds to the given hash.

In addition to the hash of the transaction you may give it a secondary *optional* argument, which specifies the options for this specific call. The possible options are:

- `disableStorage`: `BOOL`. Setting this to true will disable storage capture (default = false).
- `disableMemory`: `BOOL`. Setting this to true will disable memory capture (default = false).
- `disableStack`: `BOOL`. Setting this to true will disable stack capture (default = false).
- `tracer`: `STRING`. Setting this will enable JavaScript-based transaction tracing, described below. If set, the previous four arguments will be ignored.
- `timeout`: `STRING`. Overrides the default timeout of 5 seconds for JavaScript-based tracing calls. Valid values are described here.

Client	Method invocation
Go	`debug.TraceTransaction(txHash common.Hash, logger *vm.LogConfig) (*ExecutionResurt, error)`
Console	`debug.traceTransaction(txHash, [options])`
RPC	`{"method": "debug_traceTransaction", "params": [txHash, {}]}`

Example

```
> debug.traceTransaction("0x2059dd53ecac9827faad14d364f9e04b1d5fe5b506e3acc886eff7a6f88a696a")
{
  gas: 85301,
  returnValue: "",
  structLogs: [{
      depth: 1,
      error: "",
      gas: 162106,
      gasCost: 3,
      memory: null,
      op: "PUSH1",
      pc: 0,
      stack: [],
      storage: {}
  },
    /* snip */
  {
      depth: 1,
      error: "",
      gas: 100000,
      gasCost: 0,
      memory: ["0000000000000000000000000000000000000000000000000000000000000006",
"0000000000000000000000000000000000000000000000000000000000000000",
"0000000000000000000000000000000000000000000000000000000000000060"],
      op: "STOP",
```

```
      pc: 120,
      stack: ["0000000000000000000000000000000000000000000000000000000d67cbec9"],
      storage: {
        0000000000000000000000000000000000000000000000000000000000000004:
"8241fa522772837f0d05511f20caa6da1d5a3209000000000000000400000001",
        0000000000000000000000000000000000000000000000000000000000000006:
"0000000000000000000000000000000000000000000000000000000000000001",
        f652222313e28459528d920b65115c16c04f3efc82aaedc97be59f3f377c0d3f:
"0000000000000000000000002e816afc1b5c0f39852131959d946eb3b07b5ad"
      }
  }]
```

JavaScript-based tracing

Specifying the `tracer` option in the second argument enables JavaScript-based tracing. In this mode, `tracer` is interpreted as a JavaScript expression that is expected to evaluate to an object with (at least) two methods, named `step` and `result`.

`step`is a function that takes two arguments, log and db, and is called for each step of the EVM, or when an error occurs, as the specified transaction is traced.

`log` has the following fields:

- `pc`: Number, the current program counter
- `op`: Object, an OpCode object representing the current opcode
- `gas`: Number, the amount of gas remaining
- `gasPrice`: Number, the cost in wei of each unit of gas
- `memory`: Object, a structure representing the contract's memory space
- `stack`: array[big.Int], the EVM execution stack
- `depth`: The execution depth
- `account`: The address of the account executing the current operation
- `err`: If an error occured, information about the error

If `err` is non-null, all other fields should be ignored.

For efficiency, the same `log` object is reused on each execution step, updated with current values; make sure to copy values you want to preserve beyond the current call. For instance, this step function will not work:

```
function(log) {
  this.logs.append(log);
```

```
}
```

But this step function will:

```
function(log) {
  this.logs.append({gas: log.gas, pc: log.pc, ...});
}
```

`log.op` has the following methods:

- `isPush()` - returns true iff the opcode is a PUSHn
- `toString()` - returns the string representation of the opcode
- `toNumber()` - returns the opcode's number

`log.memory` has the following methods:

- `slice(start, stop)` - returns the specified segment of memory as a byte slice
- `length()` - returns the length of the memory

`log.stack` has the following methods:

- `peek(idx)` - returns the idx-th element from the top of the stack (0 is the topmost element) as a big.Int
- `length()` - returns the number of elements in the stack

`db` has the following methods:

- `getBalance(address)` - returns a `big.Int` with the specified account's balance
- `getNonce(address)` - returns a Number with the specified account's nonce
- `getCode(address)` - returns a byte slice with the code for the specified account
- `getState(address, hash)` - returns the state value for the specified account and the specified hash
- `exists(address)` - returns true if the specified address exists

The second function, 'result', takes no arguments, and is expected to return a JSON-serializable value to return to the RPC caller.

If the step function throws an exception or executes an illegal operation at any point, it will not be called on any further VM steps, and the error will be returned to the caller.

Note that several values are Golang big.Int objects, not JavaScript numbers or JS bigints. As such, they have the same interface as described in the godocs. Their default serialization to JSON is as a Javascript number; to serialize large numbers accurately call `.String()` on them. For convenience, `big.NewInt(x)` is provided, and will convert a uint to a Go BigInt.

Usage example, returns the top element of the stack at each CALL opcode only:

```
debug.traceTransaction(txhash, {tracer: '{data: [], step: function(log) { if(log.op.toString() == "CALL") this.data.push(log.stack.peek(0)); }, result: function() { return this.data; }}'});
```

debug_verbosity

Sets the logging verbosity ceiling. Log messages with level up to and including the given level will be printed.

The verbosity of individual packages and source files can be raised using `debug_vmodule`.

Client	Method invocation
Console	`debug.verbosity(level)`
RPC	`{"method": "debug_vmodule", "params": [number]}`

debug_vmodule

Sets the logging verbosity pattern.

Client	Method invocation
Console	`debug.vmodule(string)`
RPC	`{"method": "debug_vmodule", "params": [string]}`

Examples

If you want to see messages from a particular Go package (directory) and all subdirectories, use:

```
> debug.vmodule("eth/*=6")
```

If you want to restrict messages to a particular package (e.g. p2p) but exclude subdirectories, use:

```
> debug.vmodule("p2p=6")
```

If you want to see log messages from a particular source file, use

```
> debug.vmodule("server.go=6")
```

You can compose these basic patterns. If you want to see all output from peer.go in a package below eth (eth/peer.go, eth/downloader/peer.go) as well as output from package p2p at level <= 5, use:

```
debug.vmodule("eth/*/peer.go=6,p2p=5")
```

debug_writeBlockProfile

Writes a goroutine blocking profile to the given file.

Client	Method invocation
Console	`debug.writeBlockProfile(file)`
RPC	`{"method": "debug_writeBlockProfile", "params": [string]}`

debug_writeMemProfile

Writes an allocation profile to the given file. Note that the profiling rate cannot be set through the API, it must be set on the command line using the `--memprofilerate` flag.

Client	Method invocation
Console	`debug.writeMemProfile(file string)`
RPC	`{"method": "debug_writeBlockProfile", "params": [string]}`

Miner

The `miner` API allows you to remote control the node's mining operation and set various mining specific settings.

miner_setExtra

Sets the extra data a miner can include when miner blocks. This is capped at 32 bytes.

Client	Method invocation
Go	`miner.setExtra(extra string) (bool, error)`
Console	`miner.setExtra(string)`
RPC	`{"method": "miner_setExtra", "params": [string]}`

miner_setGasPrice

Sets the minimal accepted gas price when mining transactions. Any transactions that are below this limit are excluded from the mining process.

Client	Method invocation
Go	`miner.setGasPrice(number *rpc.HexNumber) bool`
Console	`miner.setGasPrice(number)`
RPC	`{"method": "miner_setGasPrice", "params": [number]}`

miner_start

Start the CPU mining process with the given number of threads and generate a new DAG if need be.

Client	Method invocation
Go	`miner.Start(threads *rpc.HexNumber) (bool, error)`
Console	`miner.start(number)`
RPC	`{"method": "miner_start", "params": [number]}`

miner_stop

Stop the CPU mining operation.

Client	Method invocation
Go	`miner.Stop() bool`
Console	`miner.stop()`
RPC	`{"method": "miner_stop", "params": []}`

miner_setEtherBase

Sets the etherbase, where mining rewards will go.

Client	Method invocation
Go	`miner.SetEtherbase(common.Address) bool`
Console	`miner.setEtherbase(address)`
RPC	`{"method": "miner_setEtherbase", "params": [address]}`

Personal

The personal API manages private keys in the key store.

personal_importRawKey

Imports the given unencrypted private key (hex string) into the key store, encrypting it with the passphrase.

Returns the address of the new account.

Client	Method invocation
Console	`personal.importRawKey(keydata, passphrase)`
RPC	`{"method": "personal_importRawKey", "params": [string, string]}`

personal_listAccounts

Returns all the Ethereum account addresses of all keys in the key store.

Client	Method invocation
Console	`personal.listAccounts`
RPC	`{"method": "personal_listAccounts", "params": []}`

Example

```
> personal.listAccounts
["0x5e97870f263700f46aa00d967821199b9bc5a120",
"0x3d80b31a78c30fc628f20b2c89d7ddbf6e53cedc"]
```

personal_lockAccount

Removes the private key with given address from memory. The account can no longer be used to send transactions.

Client	Method invocation
Console	`personal.lockAccount(address)`
RPC	`{"method": "personal_lockAccount", "params": [string]}`

personal_newAccount

Generates a new private key and stores it in the key store directory. The key file is encrypted with the given passphrase. Returns the address of the new account.

At the geth console, `newAccount` will prompt for a passphrase when it is not supplied as the argument.

Client	Method invocation
Console	`personal.newAccount()`
RPC	`{"method": "personal_newAccount", "params": [string]}`

Example

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0x5e97870f263700f46aa00d967821199b9bc5a120"
```

The passphrase can also be supplied as a string.

```
> personal.newAccount("h4ck3r")
"0x3d80b31a78c30fc628f20b2c89d7ddbf6e53cedc"
```

personal_unlockAccount

Decrypts the key with the given address from the key store.

Both passphrase and unlock duration are optional when using the JavaScript console. If the passphrase is not supplied as an argument, the console will prompt for the passphrase interactively.

The unencrypted key will be held in memory until the unlock duration expires. If the unlock duration defaults to 300 seconds. An explicit duration of zero seconds unlocks the key until geth exits.

The account can be used with `eth_sign` and `eth_sendTransaction` while it is unlocked.

Client	Method invocation
Console	`personal.unlockAccount(address, passphrase, duration)`
RPC	`{"method": "personal_unlockAccount", "params": [string, string, number]}`

Examples

```
> personal.unlockAccount("0x5e97870f263700f46aa00d967821199b9bc5a120")
Unlock account 0x5e97870f263700f46aa00d967821199b9bc5a120
Passphrase:
true
```

Supplying the passphrase and unlock duration as arguments:

```
> personal.unlockAccount("0x5e97870f263700f46aa00d967821199b9bc5a120", "foo", 30)
true
```

If you want to type in the passphrase and stil override the default unlock duration, pass `null`as the passphrase.

```
> personal.unlockAccount("0x5e97870f263700f46aa00d967821199b9bc5a120", null, 30)
Unlock account 0x5e97870f263700f46aa00d967821199b9bc5a120
Passphrase:
true
```

personal_sendTransaction

Validate the given passphrase and submit transaction.

The transaction is the same argument as for `eth_sendTransaction` and contains the `from`address. If the passphrase can be used to decrypt the private key belogging to `tx.from` the transaction is verified, signed and send onto the network. The account is not unlocked globally in the node and cannot be used in other RPC calls.

Client	Method invocation
Console	`personal.sendTransaction(tx, passphrase)`
RPC	`{"method": "personal_sendTransaction", "params": [tx, string]}`

Note, prior to Geth 1.5, please use `personal_signAndSendTransaction` as that was the original introductory name and only later renamed to the current final version.

Examples

```
> var tx = {from: "0x391694e7e0b0cce554cb130d723a9d27458f9298", to:
"0xafa3f8684e54059998bc3a7b0d2b0da075154d66", value: web3.toWei(1.23, "ether")}
undefined
> personal.sendTransaction(tx, "passphrase")
0x8474441674cdd47b35b875fd1a530b800b51a5264b9975fb21129eeb8c18582f
```

personal_sign

The sign method calculates an Ethereum specific signature with:`sign(keccack256("\x19Ethereum Signed Message:\n" + len(message) + message))`.

By adding a prefix to the message makes the calculated signature recognisable as an Ethereum specific signature. This prevents misuse where a malicious DApp can sign arbitrary data (e.g. transaction) and use the signature to impersonate the victim.

See ecRecover to verify the signature.

Client	Method invocation
Console	`personal.sign(message, account, [password])`
RPC	`{"method": "personal_sign", "params": [message, account, password]}`

Examples

```
> personal.sign("0xdeadbeaf", "0x9b2055d370f73ec7d8a03e965129118dc8f5bf83", "")
"0xa3f20717a250c2b0b729b7e5becbff67fdaef7e0699da4de7ca5895b02a170a12d887fd3b17bfdce3481f10bea41f45ba9f709d39ce8325427b57afcfc994cee1b"
```

personal_ecRecover

`ecRecover` returns the address associated with the private key that was used to calculate the signature in `personal_sign`.

Client	Method invocation
Console	`personal.ecRecover(message, signature)`
RPC	`{"method": "personal_ecRecover", "params": [message, signature]}`

Examples

```
> personal.sign("0xdeadbeaf", "0x9b2055d370f73ec7d8a03e965129118dc8f5bf83", "")
"0xa3f20717a250c2b0b729b7e5becbff67fdaef7e0699da4de7ca5895b02a170a12d887fd3b17bfdc
e3481f10bea41f45ba9f709d39ce8325427b57afcfc994cee1b"
> personal.ecRecover("0xdeadbeaf",
"0xa3f20717a250c2b0b729b7e5becbff67fdaef7e0699da4de7ca5895b02a170a12d887fd3b17bfdc
e3481f10bea41f45ba9f709d39ce8325427b57afcfc994cee1b")
"0x9b2055d370f73ec7d8a03e965129118dc8f5bf83"
```

Txpool

The `txpool` API gives you access to several non-standard RPC methods to inspect the contents of the transaction pool containing all the currently pending transactions as well as the ones queued for future processing.

txpool_content

The `content` inspection property can be queried to list the exact details of all the transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.

The result is an object with two fields `pending` and `queued`. Each of these fields are associative arrays, in which each entry maps an origin-address to a batch of scheduled transactions. These batches themselves are maps associating nonces with actual transactions.

Please note, there may be multiple transactions associated with the same account and nonce. This can happen if the user broadcast mutliple ones with varying gas allowances (or even complerely different transactions).

Client	Method invocation
Go	`txpool.Content() (map[string]map[string]map[string][]*RPCTransaction)`
Console	`txpool.content`
RPC	`{"method": "txpool_content"}`

Example

```
> txpool.content
{
  pending: {
    0x0216d5032f356960cd3749c31ab34eeff21b3395: {
      806: [{
        blockHash:
"0x0000000000000000000000000000000000000000000000000000000000000000",
        blockNumber: null,
        from: "0x0216d5032f356960cd3749c31ab34eeff21b3395",
        gas: "0x5208",
        gasPrice: "0xba43b7400",
        hash:
"0xaf953a2d01f55cfe080c0c94150a60105e8ac3d51153058a1f03dd239dd08586",
        input: "0x",
        nonce: "0x326",
        to: "0x7f69a91a3cf4be60020fb58b893b7cbb65376db8",
        transactionIndex: null,
        value: "0x19a99f0cf456000"
      }]
    },
    0x24d407e5a0b506e1cb2fae163100b5de01f5193c: {
      34: [{
        blockHash:
"0x0000000000000000000000000000000000000000000000000000000000000000",
        blockNumber: null,
        from: "0x24d407e5a0b506e1cb2fae163100b5de01f5193c",
        gas: "0x44c72",
        gasPrice: "0x4a817c800",
        hash:
"0xb5b8b853af32226755a65ba0602f7ed0e8be2211516153b75e9ed640a7d359fe",
        input:
"0xb61d27f60000000000000000000000024d407e5a0b506e1cb2fae163100b5de01f5193c0000000
0000000000000000000000000000000000000053444835ec580000000000000000000000000000
0000000000000000000000000000000000006000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000
000",
        nonce: "0x22",
        to: "0x7320785200f74861b69c49e4ab32399a71b34f1a",
```

```
        transactionIndex: null,
        value: "0x0"
      }]
    }
  },
  queued: {
    0x976a3fc5d6f7d259ebfb4cc2ae75115475e9867c: {
      3: [{
        blockHash:
"0x0000000000000000000000000000000000000000000000000000000000000000",
        blockNumber: null,
        from: "0x976a3fc5d6f7d259ebfb4cc2ae75115475e9867c",
        gas: "0x15f90",
        gasPrice: "0x4a817c800",
        hash:
"0x57b30c59fc39a50e1cba90e3099286dfa5aaf60294a629240b5bbec6e2e66576",
        input: "0x",
        nonce: "0x3",
        to: "0x346fb27de7e7370008f5da379f74dd49f5f2f80f",
        transactionIndex: null,
        value: "0x1f161421c8e0000"
      }]
    },
    0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a: {
      2: [{
        blockHash:
"0x0000000000000000000000000000000000000000000000000000000000000000",
        blockNumber: null,
        from: "0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a",
        gas: "0x15f90",
        gasPrice: "0xba43b7400",
        hash:
"0x3a3c0698552eec2455ed3190eac3996feccc806970a4a056106deaf6ceb1e5e3",
        input: "0x",
        nonce: "0x2",
        to: "0x24a461f25ee6a318bdef7f33de634a67bb67ac9d",
        transactionIndex: null,
        value: "0xebec21ee1da40000"
      }],
      6: [{
```

```
        blockHash:
"0x0000000000000000000000000000000000000000000000000000000000000000",
        blockNumber: null,
        from: "0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a",
        gas: "0x15f90",
        gasPrice: "0x4a817c800",
        hash:
"0xbbcd1e45eae3b859203a04be7d6e1d7b03b222ec1d66dfcc8011dd39794b147e",
        input: "0x",
        nonce: "0x6",
        to: "0x6368f3f8c2b42435d6c136757382e4a59436a681",
        transactionIndex: null,
        value: "0xf9a951af55470000"
      }, {
        blockHash:
"0x0000000000000000000000000000000000000000000000000000000000000000",
        blockNumber: null,
        from: "0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a",
        gas: "0x15f90",
        gasPrice: "0x4a817c800",
        hash:
"0x60803251d43f072904dc3a2d6a084701cd35b4985790baaf8a8f76696041b272",
        input: "0x",
        nonce: "0x6",
        to: "0x8db7b4e0ecb095fbd01dffa62010801296a9ac78",
        transactionIndex: null,
        value: "0xebe866f5f0a06000"
      }],
    }
  }
}
```

txpool_inspect

The `inspect` inspection property can be queried to list a textual summary of all the transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only. This is a method specifically tailored to developers to quickly see the transactions in the pool and find any potential issues.

The result is an object with two fields `pending` and `queued`. Each of these fields are associative arrays, in which each entry maps an origin-address to a batch of scheduled transactions. These batches themselves are maps associating nonces with transactions summary strings.

Please note, there may be multiple transactions associated with the same account and nonce. This can happen if the user broadcast mutliple ones with varying gas allowances (or even complerely different transactions).

Client	Method invocation
Go	`txpool.Inspect() (map[string]map[string]map[string][]string)`
Console	`txpool.inspect`
RPC	`{"method": "txpool_inspect"}`

Example

```
> txpool.inspect
{
  pending: {
    0x26588a9301b0428d95e6fc3a5024fce8bec12d51: {
      31813: ["0x3375ee30428b2a71c428afa5e89e427905f95f7e: 0 wei + 500000 × 20000000000 gas"]
    },
    0x2a65aca4d5fc5b5c859090a6c34d164135398226: {
      563662: ["0x958c1fa64b34db746925c6f8a3dd81128e40355e: 1051546810000000000 wei + 90000 × 20000000000 gas"],
      563663: ["0x77517b1491a0299a44d668473411676f94e97e34: 1051190740000000000 wei + 90000 × 20000000000 gas"],
      563664: ["0x3e2a7fe169c8f8eee251bb00d9fb6d304ce07d3a: 1050828950000000000
```

```
wei + 90000 × 20000000000 gas"],
      563665: ["0xaf6c4695da477f8c663ea2d8b768ad82cb6a8522: 1050544770000000000
wei + 90000 × 20000000000 gas"],
      563666: ["0x139b148094c50f4d20b01caf21b85edb711574db: 1048598530000000000
wei + 90000 × 20000000000 gas"],
      563667: ["0x48b3bd66770b0d1eecefce090dafee36257538ae: 1048367260000000000
wei + 90000 × 20000000000 gas"],
      563668: ["0x468569500925d53e06dd0993014ad166fd7dd381: 1048126690000000000
wei + 90000 × 20000000000 gas"],
      563669: ["0x3dcb4c90477a4b8ff7190b79b524773cbe3be661: 1047965690000000000
wei + 90000 × 20000000000 gas"],
      563670: ["0x6dfef5bc94b031407ffe71ae8076ca0fbf190963: 1047859050000000000
wei + 90000 × 20000000000 gas"]
    },
    0x9174e688d7de157c5c0583df424eaab2676ac162: {
      3: ["0xbb9bc244d798123fde783fcc1c72d3bb8c189413: 3000000000000000000 wei +
85000 × 21000000000 gas"]
    },
    0xb18f9d01323e150096650ab989cfecd39d757aec: {
      777: ["0xcd79c72690750f079ae6ab6ccd7e7aedc03c7720: 0 wei + 1000000 ×
20000000000 gas"]
    },
    0xb2916c870cf66967b6510b76c07e9d13a5d23514: {
      2: ["0x576f25199d60982a8f31a8dff4da8acb982e6aba: 2600000000000000000 wei +
90000 × 20000000000 gas"]
    },
    0xbc0ca4f217e052753614d6b019948824d0d8688b: {
      0: ["0x2910543af39aba0cd09dbb2d50200b3e800a63d2: 1000000000000000000 wei +
50000 × 1171602790622 gas"]
    },
    0xea674fdde714fd979de3edf0f56aa9716b898ec8: {
      70148: ["0xe39c55ead9f997f7fa20ebe40fb4649943d7db66: 1000767667434026200 wei
+ 90000 × 20000000000 gas"]
    }
  },
  queued: {
    0x0f6000de1578619320aba5e392706b131fb1de6f: {
      6: ["0x8383534d0bcd0186d326c993031311c0ac0d9b2d: 9000000000000000000 wei +
21000 × 20000000000 gas"]
    },
```

```
    0x5b30608c678e1ac464a8994c3b33e5cdf3497112: {
      6: ["0x9773547e27f8303c87089dc42d9288aa2b9d8f06: 50000000000000000000 wei +
90000 × 50000000000 gas"]
    },
    0x976a3fc5d6f7d259ebfb4cc2ae75115475e9867c: {
      3: ["0x346fb27de7e7370008f5da379f74dd49f5f2f80f: 140000000000000000 wei +
90000 × 20000000000 gas"]
    },
    0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a: {
      2: ["0x24a461f25ee6a318bdef7f33de634a67bb67ac9d: 17000000000000000000 wei +
90000 × 50000000000 gas"],
      6: ["0x6368f3f8c2b42435d6c136757382e4a59436a681: 17990000000000000000 wei +
90000 × 20000000000 gas", "0x8db7b4e0ecb095fbd01dffa62010801296a9ac78:
16998950000000000000 wei + 90000 × 20000000000 gas"],
      7: ["0x6368f3f8c2b42435d6c136757382e4a59436a681: 17900000000000000000 wei +
90000 × 20000000000 gas"]
    }
  }
}
```

txpool_status

The `status` inspection property can be queried for the number of transactions currently pending for inclusion in the next block(s), as well as the ones that are being scheduled for future execution only.

The result is an object with two fields `pending` and `queued`, each of which is a counter representing the number of transactions in that particular state.

Client	Method invocation
Go	`txpool.Status() (map[string]*rpc.HexNumber)`
Console	`txpool.status`
RPC	`{"method": "txpool_status"}`

Example

```
> txpool.status
{
  pending: 10,
  queued: 7
}
```

Managing your accounts

Felix Lange edited this page on 21 Dec 2017 · 32 revisions

WARNING Remember your password.

If you lose the password you use to encrypt your account, you will not be able to access that account. Repeat: It is NOT possible to access your account without a password and there is no *forgot my password* option here. Do not forget it.

The ethereum CLI `geth` provides account management via the `account` command:

```
$ geth account <command> [options...] [arguments...]
```

Manage accounts lets you create new accounts, list all existing accounts, import a private key into a new account, migrate to newest key format and change your password.

It supports interactive mode, when you are prompted for password as well as non-interactive mode where passwords are supplied via a given password file. Non-interactive mode is only meant for scripted use on test networks or known safe environments.

Make sure you remember the password you gave when creating a new account (with new, update or import). Without it you are not able to unlock your account.

Note that exporting your key in unencrypted format is NOT supported.

Keys are stored under `<DATADIR>/keystore`. Make sure you backup your keys regularly! SeeDATADIR backup & restore for more information. If a custom datadir and keystore option are given the keystore option takes preference over the datadir option.

The newest format of the keyfiles is: `UTC--<created_at UTC ISO8601>-<address hex>`. The order of accounts when listing, is lexicographic, but as a consequence of the timespamp format, it is actually order of creation

It is safe to transfer the entire directory or the individual keys therein between ethereum nodes. Note that in case you are adding keys to your node from a different node, the order of accounts may change. So make sure you do not rely or change the index in your scripts or code snippets.

And again. DO NOT FORGET YOUR PASSWORD

```
COMMANDS:
     list    Print summary of existing accounts
     new     Create a new account
     update  Update an existing account
     import  Import a private key into a new account
```

You can get info about subcommands by `geth account <command> --help`.

```
$ geth account list --help
list [command options] [arguments...]

Print a short summary of all accounts

OPTIONS:
  --datadir "/home/bas/.ethereum"  Data directory for the databases and keystore
  --keystore                       Directory for the keystore (default = inside
the datadir)
```

Accounts can also be managed via the [Javascript Console]

Examples

Interactive use

creating an account

```
$ geth account new
Your new account is locked with a password. Please give a password. Do not forget
this password.
Passphrase:
Repeat Passphrase:
Address: {168bc315a2ee09042d83d7c5811b533620531f67}
```

Listing accounts in a custom keystore directory

```
$ geth account list --keystore /tmp/mykeystore/
Account #0: {5afdd78bdacb56ab1dad28741ea2a0e47fe41331}
keystore:///tmp/mykeystore/UTC--2017-04-28T08-46-27.437847599Z--5afdd78bdacb56ab1d
ad28741ea2a0e47fe41331
Account #1: {9acb9ff906641a434803efb474c96a837756287f}
keystore:///tmp/mykeystore/UTC--2017-04-28T08-46-52.180688336Z--9acb9ff906641a4348
03efb474c96a837756287f
```

Import private key into a node with a custom datadir

```
$ geth account import --datadir /someOtherEthDataDir ./key.prv
The new account will be encrypted with a passphrase.
Please enter a passphrase now.
Passphrase:
Repeat Passphrase:
Address: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

Account update

```
$ geth account update a94f5374fce5edbc8e2a8697c15331677e6ebf0b
Unlocking account a94f5374fce5edbc8e2a8697c15331677e6ebf0b | Attempt 1/3
Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
Account 'a94f5374fce5edbc8e2a8697c15331677e6ebf0b' unlocked.
Please give a new password. Do not forget this password.
Passphrase:
Repeat Passphrase:
0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
```

Non-interactive use

You supply a plaintext password file as argument to the `--password` flag. The data in the file consists of the raw characters of the password, followed by a single newline.

Note: Supplying the password directly as part of the command line is not recommended, but you can always use shell trickery to get round this restriction.

```
$ geth account new --password /path/to/password
```

```
$ geth account import  --datadir /someOtherEthDataDir --password
/path/to/anotherpassword ./key.prv
```

Creating accounts

Creating a new account

```
$ geth account new
$ geth account new --password /path/to/passwdfile
$ geth account new --password <(echo $mypassword)
```

Creates a new account and prints the address.

On the console, use:

```
> personal.NewAccount()
... you will be prompted for a password ...
```

```
or
```

```
> personal.newAccount("passphrase")
```

The account is saved in encrypted format. You must remember this passphrase to unlock your account in the future.

For non-interactive use the passphrase can be specified with the `--password` flag:

```
geth account new --password <passwordfile>
```

Note, this is meant to be used for testing only, it is a bad idea to save your password to file or expose in any other way.

Creating an account by importing a private key

```
   geth account import <keyfile>
```

Imports an unencrypted private key from `<keyfile>` and creates a new account and prints the address.

The keyfile is assumed to contain an unencrypted private key as canonical EC raw bytes encoded into hex.

The account is saved in encrypted format, you are prompted for a passphrase.

You must remember this passphrase to unlock your account in the future.

For non-interactive use the passphrase can be specified with the `--password` flag:

```
geth account import --password <passwordfile> <keyfile>
```

Note: Since you can directly copy your encrypted accounts to another ethereum instance, this import/export mechanism is not needed when you transfer an account between nodes.

Warning: when you copy keys into an existing node's keystore, the order of accounts you are used to may change. Therefore you make sure you either do not rely on the account order or doublecheck and update the indexes used in your scripts.

Warning: If you use the password flag with a password file, best to make sure the file is not readable or even listable for anyone but you. You achieve this with:

```
touch /path/to/password
chmod 700 /path/to/password
cat > /path/to/password
>I type my pass here^D
```

Updating an existing account

You can update an existing account on the command line with the `update` subcommand with the account address or index as parameter. You can specify multiple accounts at once.

```
geth account update 5afdd78bdacb56ab1dad28741ea2a0e47fe41331
9acb9ff906641a434803efb474c96a837756287f
geth account update 0 1 2
```

The account is saved in the newest version in encrypted format, you are prompted for a passphrase to unlock the account and another to save the updated file.

This same command can therefore be used to migrate an account of a deprecated format to the newest format or change the password for an account.

After a successful update, all previous formats/versions of that same key are removed!

Importing your presale wallet

Importing your presale wallet is very easy. If you remember your password that is:

```
geth wallet import /path/to/my/presale.wallet
```

will prompt for your password and imports your ether presale account. It can be used non-interactively with the --password option taking a passwordfile as argument containing the wallet password in cleartext.

Listing accounts and checking balances

Listing your current accounts

From the command line, call the CLI with:

```
$ geth account list
Account #0: {5afdd78bdacb56ab1dad28741ea2a0e47fe41331}
keystore:///tmp/mykeystore/UTC--2017-04-28T08-46-27.437847599Z--5afdd78bdacb56ab1dad28741ea2a0e47fe41331
Account #1: {9acb9ff906641a434803efb474c96a837756287f}
keystore:///tmp/mykeystore/UTC--2017-04-28T08-46-52.180688336Z--9acb9ff906641a434803efb474c96a837756287f
```

to list your accounts in order of creation.

Note: This order can change if you copy keyfiles from other nodes, so make sure you either do not rely on indexes or make sure if you copy keys you check and update your account indexes in your scripts.

When using the console:

```
> eth.accounts
["0x5afdd78bdacb56ab1dad28741ea2a0e47fe41331",
```

```
"0x9acb9ff906641a434803efb474c96a837756287f"]
```

or via RPC:

```
# Request
$ curl -X POST --data
'{"jsonrpc":"2.0","method":"eth_accounts","params":[],"id":1}
http://127.0.0.1:8545'

# Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": ["0x5afdd78bdacb56ab1dad28741ea2a0e47fe41331",
"0x9acb9ff906641a434803efb474c96a837756287f"]
}
```

If you want to use an account non-interactively, you need to unlock it. You can do this on the command line with the `--unlock` option which takes a comma separated list of accounts (in hex or index) as argument so you can unlock the accounts programmatically for one session. This is useful if you want to use your account from Dapps via RPC. `--unlock` will unlock the first account. This is useful when you created your account programmatically, you do not need to know the actual account to unlock it.

Create account and start node with account unlocked:

```
geth account new --password <(echo this is not secret!)
geth --password <(echo this is not secret!) --unlock primary --rpccorsdomain
localhost --verbosity 6 2>> geth.log
```

Instead of the account address, you can use integer indexes which refers to the address position in the account listing (and corresponds to order of creation)

The command line allows you to unlock multiple accounts. In this case the argument to unlock is a comma delimited list of accounts addresses or indexes.

```
geth --unlock
"0x407d73d8a49eeb85d32cf465507dd71d507100c1,0,5,e470b1a7d2c9c5c6f03bbaa8fa20db6d40
4a0c32"
```

If this construction is used non-interactively, your password file will need to contain the respective passwords for the accounts in question, one per line.

On the console you can also unlock accounts (one at a time) for a duration (in seconds).

```
personal.unlockAccount(address, "password", 300)
```

Note that we do NOT recommend using the password argument here, since the console history is logged, so you may compromise your account. You have been warned.

Checking account balances

To check your the etherbase account balance:

```
> web3.fromWei(eth.getBalance(eth.coinbase), "ether")
6.5
```

Print all balances with a JavaScript function:

```
function checkAllBalances() {
    var totalBal = 0;
    for (var acctNum in eth.accounts) {
        var acct = eth.accounts[acctNum];
        var acctBal = web3.fromWei(eth.getBalance(acct), "ether");
        totalBal += parseFloat(acctBal);
        console.log("  eth.accounts[" + acctNum + "]: \t" + acct + " \tbalance: "
+ acctBal + " ether");
    }
    console.log("  Total balance: " + totalBal + " ether");
};
```

That can then be executed with:

```
> checkAllBalances();
  eth.accounts[0]: 0xd1ade25ccd3d550a7eb532ac759cac7be09c2719    balance:
63.11848 ether
  eth.accounts[1]: 0xda65665fc30803cb1fb7e6d86691e20b1826dee0    balance: 0 ether
  eth.accounts[2]: 0xe470b1a7d2c9c5c6f03bbaa8fa20db6d404a0c32    balance: 1 ether
  eth.accounts[3]: 0xf4dd5c3794f1fd0cdc0327a83aa472609c806e99    balance: 6 ether
```

Since this function will disappear after restarting geth, it can be helpful to store commonly used functions to be recalled later. The loadScript function makes this very easy.

First, save the `checkAllBalances()` function definition to a file on your computer. For example, `/Users/username/gethload.js`. Then load the file from the interactive console:

```
> loadScript("/Users/username/gethload.js")
true
```

The file will modify your JavaScript environment as if you has typed the commands manually. Feel free to experiment!

Command Line Options

Péter Szilágyi edited this page on 21 Nov 2017 · 39 revisions

```
$ geth help
NAME:
   geth - the go-ethereum command line interface

   Copyright 2013-2017 The go-ethereum Authors

USAGE:
   geth [options] command [command options] [arguments...]

VERSION:
   1.7.3-stable

COMMANDS:
   account     Manage accounts
   attach      Start an interactive JavaScript environment (connect to node)
   bug         opens a window to report a bug on the geth repo
   console     Start an interactive JavaScript environment
   copydb      Create a local chain from a target chaindata folder
   dump        Dump a specific block from storage
   dumpconfig  Show configuration values
   export      Export blockchain into file
   import      Import a blockchain file
   init        Bootstrap and initialize a new genesis block
   js          Execute the specified JavaScript files
   license     Display license information
   makecache   Generate ethash verification cache (for testing)
   makedag     Generate ethash mining DAG (for testing)
   monitor     Monitor and visualize node metrics
   removedb    Remove blockchain and state databases
   version     Print version numbers
   wallet      Manage Ethereum presale wallets
   help, h     Shows a list of commands or help for one command
```

```
ETHEREUM OPTIONS:
  --config value                           TOML configuration file
  --datadir "/home/karalabe/.ethereum"  Data directory for the databases and keystore
  --keystore                               Directory for the keystore (default = inside the
datadir)
  --nousb                                  Disables monitoring for and managing USB hardware
wallets
  --networkid value                        Network identifier (integer, 1=Frontier, 2=Morden
(disused), 3=Ropsten, 4=Rinkeby) (default: 1)
  --testnet                                Ropsten network: pre-configured proof-of-work test
network
  --rinkeby                                Rinkeby network: pre-configured proof-of-authority
test network
  --syncmode "fast"                        Blockchain sync mode ("fast", "full", or "light")
  --ethstats value                         Reporting URL of a ethstats service
(nodename:secret@host:port)
  --identity value                         Custom node name
  --lightserv value                        Maximum percentage of time allowed for serving LES
requests (0-90) (default: 0)
  --lightpeers value                       Maximum number of LES client peers (default: 20)
  --lightkdf                               Reduce key-derivation RAM & CPU usage at some
expense of KDF strength

DEVELOPER CHAIN OPTIONS:
  --dev               Ephemeral proof-of-authority network with a pre-funded developer
account, mining enabled
  --dev.period value  Block period to use in developer mode (0 = mine only if transaction
pending) (default: 0)
```

```
ETHASH OPTIONS:
  --ethash.cachedir                          Directory to store the ethash verification
caches (default = inside the datadir)
  --ethash.cachesinmem value                 Number of recent ethash caches to keep in
memory (16MB each) (default: 2)
  --ethash.cachesondisk value                Number of recent ethash caches to keep on disk
(16MB each) (default: 3)
  --ethash.dagdir "/home/karalabe/.ethash"   Directory to store the ethash mining DAGs
(default = inside home folder)
  --ethash.dagsinmem value                   Number of recent ethash mining DAGs to keep in
memory (1+GB each) (default: 1)
  --ethash.dagsondisk value                  Number of recent ethash mining DAGs to keep on
disk (1+GB each) (default: 2)

TRANSACTION POOL OPTIONS:
  --txpool.nolocals            Disables price exemptions for locally submitted
transactions
  --txpool.journal value       Disk journal for local transaction to survive node restarts
(default: "transactions.rlp")
  --txpool.rejournal value     Time interval to regenerate the local transaction journal
(default: 1h0m0s)
  --txpool.pricelimit value    Minimum gas price limit to enforce for acceptance into the
pool (default: 1)
  --txpool.pricebump value     Price bump percentage to replace an already existing
transaction (default: 10)
  --txpool.accountslots value  Minimum number of executable transaction slots guaranteed
per account (default: 16)
  --txpool.globalslots value   Maximum number of executable transaction slots for all
accounts (default: 4096)
  --txpool.accountqueue value  Maximum number of non-executable transaction slots
permitted per account (default: 64)
  --txpool.globalqueue value   Maximum number of non-executable transaction slots for all
accounts (default: 1024)
  --txpool.lifetime value      Maximum amount of time non-executable transaction are
queued (default: 3h0m0s)
```

```
PERFORMANCE TUNING OPTIONS:
  --cache value            Megabytes of memory allocated to internal caching (min 16MB / database forced) (default: 128)
  --trie-cache-gens value  Number of trie node generations to keep in memory (default: 120)

ACCOUNT OPTIONS:
  --unlock value    Comma separated list of accounts to unlock
  --password value  Password file to use for non-interactive password input

API AND CONSOLE OPTIONS:
  --rpc                    Enable the HTTP-RPC server
  --rpcaddr value          HTTP-RPC server listening interface (default: "localhost")
  --rpcport value          HTTP-RPC server listening port (default: 8545)
  --rpcapi value           API's offered over the HTTP-RPC interface
  --ws                     Enable the WS-RPC server
  --wsaddr value           WS-RPC server listening interface (default: "localhost")
  --wsport value           WS-RPC server listening port (default: 8546)
  --wsapi value            API's offered over the WS-RPC interface
  --wsorigins value        Origins from which to accept websockets requests
  --ipcdisable             Disable the IPC-RPC server
  --ipcpath                Filename for IPC socket/pipe within the datadir (explicit paths escape it)
  --rpccorsdomain value    Comma separated list of domains from which to accept cross origin requests (browser enforced)
  --jspath loadScript      JavaScript root path for loadScript (default: ".")
  --exec value             Execute JavaScript statement
  --preload value          Comma separated list of JavaScript files to preload into the console
```

```
NETWORKING OPTIONS:
  --bootnodes value     Comma separated enode URLs for P2P discovery bootstrap (set v4+v5
instead for light servers)
  --bootnodesv4 value   Comma separated enode URLs for P2P v4 discovery bootstrap (light
server, full nodes)
  --bootnodesv5 value   Comma separated enode URLs for P2P v5 discovery bootstrap (light
server, light nodes)
  --port value          Network listening port (default: 30303)
  --maxpeers value      Maximum number of network peers (network disabled if set to 0)
(default: 25)
  --maxpendpeers value  Maximum number of pending connection attempts (defaults used if
set to 0) (default: 0)
  --nat value           NAT port mapping mechanism (any|none|upnp|pmp|extip:<IP>)
(default: "any")
  --nodiscover          Disables the peer discovery mechanism (manual peer addition)
  --v5disc              Enables the experimental RLPx V5 (Topic Discovery) mechanism
  --netrestrict value   Restricts network communication to the given IP networks (CIDR
masks)
  --nodekey value       P2P node key file
  --nodekeyhex value    P2P node key as hex (for testing)

MINER OPTIONS:
  --mine                    Enable mining
  --minerthreads value      Number of CPU threads to use for mining (default: 8)
  --etherbase value         Public address for block mining rewards (default = first
account created) (default: "0")
  --targetgaslimit value    Target gas limit sets the artificial target gas floor for the
blocks to mine (default: 4712388)
  --gasprice "18000000000"  Minimal gas price to accept for mining a transactions
  --extradata value         Block extra data set by the miner (default = client version)

GAS PRICE ORACLE OPTIONS:
  --gpoblocks value      Number of recent blocks to check for gas prices (default: 10)
  --gpopercentile value  Suggested gas price is the given percentile of a set of recent
transaction gas prices (default: 50)

VIRTUAL MACHINE OPTIONS:
  --vmdebug  Record information useful for VM and contract debugging
```

```
LOGGING AND DEBUGGING OPTIONS:
  --metrics                  Enable metrics collection and reporting
  --fakepow                  Disables proof-of-work verification
  --nocompaction             Disables db compaction after import
  --verbosity value          Logging verbosity: 0=silent, 1=error, 2=warn, 3=info, 4=debug,
5=detail (default: 3)
  --vmodule value            Per-module verbosity: comma-separated list of
<pattern>=<level> (e.g. eth/*=5,p2p=4)
  --backtrace value          Request a stack trace at a specific logging statement (e.g.
"block.go:271")
  --debug                    Prepends log messages with call-site location (file and line
number)
  --pprof                    Enable the pprof HTTP server
  --pprofaddr value          pprof HTTP server listening interface (default: "127.0.0.1")
  --pprofport value          pprof HTTP server listening port (default: 6060)
  --memprofilerate value     Turn on memory profiling with the given rate (default: 524288)
  --blockprofilerate value   Turn on block profiling with the given rate (default: 0)
  --cpuprofile value         Write CPU profile to the given file
  --trace value              Write execution trace to the given file

WHISPER (EXPERIMENTAL) OPTIONS:
  --shh                        Enable Whisper
  --shh.maxmessagesize value   Max message size accepted (default: 1048576)
  --shh.pow value              Minimum POW accepted (default: 0.2)

DEPRECATED OPTIONS:
  --fast   Enable fast syncing through state downloads
  --light  Enable light client mode

MISC OPTIONS:
  --help, -h  show help

COPYRIGHT:
   Copyright 2013-2017 The go-ethereum Authors
```

JavaScript Console

Felix Lange edited this page on 21 Dec 2017 · 88 revisions

Ethereum implements a javascript runtime environment (JSRE) that can be used in either interactive (console) or non-interactive (script) mode.

Ethereum's Javascript console exposes the full web3 JavaScript Dapp API and the admin API.

Interactive use: the JSRE REPL Console

The `ethereum CLI` executable `geth` has a JavaScript console (a Read, Evaluate & Print Loop= REPL exposing the JSRE), which can be started with the `console` or `attach` subcommand. The `console` subcommands starts the geth node and then opens the console. The `attach`subcommand will not start the geth node but instead tries to open the console on a running geth instance.

```
$ geth console
$ geth attach
```

The attach node accepts an endpoint in case the geth node is running with a non default ipc endpoint or you would like to connect over the rpc interface.

```
$ geth attach ipc:/some/custom/path
$ geth attach http://191.168.1.1:8545
$ geth attach ws://191.168.1.1:8546
```

Note that by default the geth node doesn't start the http and weboscket service and not all functionality is provided over these interfaces due to security reasons. These defaults can be overridden when the `--rpcapi` and `--wsapi` arguments when the geth node is started, or with admin.startRPC and admin.startWS.

If you need log information, start with:

```
$ geth --verbosity 5 console 2>> /tmp/eth.log
```

Otherwise mute your logs, so that it does not pollute your console:

```
$ geth console 2>> /dev/null
```

or

```
$ geth --verbosity 0 console
```

Geth has support to load custom JavaScript files into the console through the `--preload`argument. This can be used to load often used functions, setup web3 contract objects, or ...

```
geth --preload "/my/scripts/folder/utils.js,/my/scripts/folder/contracts.js"
console
```

Non-interactive use: JSRE script mode

It's also possible to execute files to the JavaScript interpreter. The `console` and `attach`subcommand accept the `--exec` argument which is a javascript statement.

```
$ geth --exec "eth.blockNumber" attach
```

This prints the current block number of a running geth instance.

Or execute a local script with more complex statements on a remote node over http:

```
$ geth --exec 'loadScript("/tmp/checkbalances.js")' attach
http://123.123.123.123:8545
$ geth --jspath "/tmp" --exec 'loadScript("checkbalances.js")' attach
http://123.123.123.123:8545
```

Use the `--jspath <path/to/my/js/root>` to set a libdir for your js scripts. Parameters to `loadScript()` with no absolute path will be understood relative to this directory.

You can exit the console cleanly by typing `exit` or simply with `CTRL-C`.

Caveat

The go-ethereum JSRE uses the Otto JS VM which has some limitations:

- "use strict" will parse, but does nothing.
- The regular expression engine (re2/regexp) is not fully compatible with the ECMA5 specification.

Note that the other known limitation of Otto (namely the lack of timers) is taken care of. Ethereum JSRE implements both `setTimeout` and `setInterval`. In addition to this, the console provides `admin.sleep(seconds)` as well as a "blocktime sleep" method `admin.sleepBlocks(number)`.

Since `web3.js` uses the `bignumber.js` library (MIT Expat Licence), it is also autoloded.

Timers

In addition to the full functionality of JS (as per ECMA5), the ethereum JSRE is augmented with various timers. It implements `setInterval`, `clearInterval`, `setTimeout`, `clearTimeout` you may be used to using in browser windows. It also provides implementation for `admin.sleep(seconds)` and a block based timer, `admin.sleepBlocks(n)` which sleeps till the number of new blocks added is equal to or greater than `n`, think "wait for n confirmations".

Management APIs

Beside the official DApp API interface the go ethereum node has support for additional management API's. These API's are offered using JSON-RPC and follow the same conventions as used in the DApp API. The go ethereum package comes with a console client which has support for all additional API's.

The management API has its own wiki page.

Private network

Felix Lange edited this page on 27 Mar 2017 · 3 revisions

An Ethereum network is a private network if the nodes are not connected to the main network nodes. In this context private only means reserved or isolated, rather than protected or secure.

Choosing A Network ID

Since connections between nodes are valid only if peers have identical protocol version and network ID, you can effectively isolate your network by setting either of these to a non default value. We recommend using the `--networkid` command line option for this. Its argument is an integer, the main network has id 1 (the default). So if you supply your own custom network ID which is different than the main network your nodes will not connect to other nodes and form a private network.

Creating The Genesis Block

Every blockchain starts with the genesis block. When you run geth with default settings for the first time, the main net genesis block is committed to the database. For a private network, you usually want a different genesis block.

Here's an example of a custom genesis.json file. The `config` section ensures that certain protocol upgrades are immediately available. The `alloc` section pre-funds accounts.

```
{
    "config": {
        "chainId": 15,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
    "difficulty": "200000000",
    "gasLimit": "2100000",
    "alloc": {
        "7df9a875a174b3bc565e6424a0050ebc1b2d1d82": { "balance": "300000" },
        "f41c74c9ae680c1aa78f42e5647a62f353b7bdde": { "balance": "400000" }
```

```
    }
}
```

To create a database that uses this genesis block, run the following command. This will import and set the canonical genesis block for your chain.

```
geth --datadir path/to/custom/data/folder init genesis.json
```

Future runs of geth on this data directory will use the genesis block you have defined.

```
geth --datadir path/to/custom/data/folder --networkid 15
```

Network Connectivity

With all nodes that you want to run initialized to the desired genesis state, you'll need to start a bootstrap node that others can use to find each other in your network and/or over the internet. The clean way is to configure and run a dedicated bootnode:

```
bootnode --genkey=boot.key
bootnode --nodekey=boot.key
```

With the bootnode online, it will display an enode URL that other nodes can use to connect to it and exchange peer information. Make sure to replace the displayed IP address information (most probably [::]) with your externally accessible IP to get the actual enode URL.

Note: You can also use a full fledged Geth node as a bootstrap node.

Starting Up Your Member Nodes

With the bootnode operational and externally reachable (you can try `telnet <ip> <port>` to ensure it's indeed reachable), start every subsequent Geth node pointed to the bootnode for peer discovery via the --bootnodes flag. It will probably also be desirable to keep the data directory of your private network separated, so do also specify a custom `--datadir` flag.

```
geth --datadir path/to/custom/data/folder --networkid 15 --bootnodes
<bootnode-enode-url-from-above>
```

Since your network will be completely cut off from the main and test networks, you'll also need to configure a miner to process transactions and create new blocks for you.

Running A Private Miner

Mining on the public Ethereum network is a complex task as it's only feasible using GPUs, requiring an OpenCL or CUDA enabled ethminer instance. For information on such a setup, please consult the EtherMining subreddit and the Genoil miner repository.

In a private network setting however, a single CPU miner instance is more than enough for practical purposes as it can produce a stable stream of blocks at the correct intervals without needing heavy resources (consider running on a single thread, no need for multiple ones either). To start a Geth instance for mining, run it with all your usual flags, extended by:

```
$ geth <usual-flags> --mine --minerthreads=1
--etherbase=0x0000000000000000000000000000000000000000
```

Which will start mining bocks and transactions on a single CPU thread, crediting all proceedings to the account specified by --etherbase. You can further tune the mining by changing the default gas limit blocks converge to (`--targetgaslimit`) and the price transactions are accepted at (`--gasprice`).

Developers' Guide

Felix Lange edited this page on 21 Dec 2017 · 36 revisions

NOTE: These instructions are for people who want to contribute Go source code changes. If you just want to run ethereum, use the normal Installation Instructions

This document is the entry point for developers of the Go implementation of Ethereum. Developers here refer to the hands-on: who are interested in build, develop, debug, submit a bug report or pull request or contribute code to go-ethereum.

Building and Testing

Go Environment

We assume that you have `go` v1.8 installed, and `GOPATH` is set.

Note:You must have your working copy under `$GOPATH/src/github.com/ethereum/go-ethereum`.

Since `go` does not use relative path for import, working in any other directory will have no effect, since the import paths will be appended to `$GOPATH/src`, and if the lib does not exist, the version at master HEAD will be downloaded.

Most likely you will be working from your fork of `go-ethereum`, let's say from `github.com/nirname/go-ethereum`. Clone or move your fork into the right place:

```
git clone git@github.com:nirname/go-ethereum.git
$GOPATH/src/github.com/ethereum/go-ethereum
```

Managing Vendored Dependencies

All other dependencies are tracked in the vendor/ directory. We use govendor to manage them.

If you want to add a new dependency, run `govendor fetch <import-path>`, then commit the result.

If you want to update all dependencies to their latest upstream version, run `govendor fetch +v`.

You can also use govendor to run certain commands on all go-ethereum packages, excluding vendored code. Example: to recreate all generated code, run `govendor generate +l`.

Building Executables

Switch to the go-ethereum repository root directory.

You can build all code using the go tool, placing the resulting binary in `$GOPATH/bin`.

```
go install -v ./...
```

go-ethereum exectuables can be built individually. To build just geth, use:

```
go install -v ./cmd/geth
```

Read about cross compilation of go-ethereum here.

Git flow

To make life easier try git flow it sets this all up and streamlines your work flow.

Testing

Testing one library:

```
go test -v -cpu 4 ./eth
```

Using options `-cpu` (number of cores allowed) and `-v` (logging even if no error) is recommended.

Testing only some methods:

```
go test -v -cpu 4 ./eth -run TestMethod
```

Note: here all tests with prefix *TestMethod* will be run, so if you got TestMethod, TestMethod1, then both!

Running benchmarks, eg.:

```
go test -v -cpu 4 -bench . -run BenchmarkJoin
```

for more see go test flags

Metrics and monitoring

`geth` can do node behaviour monitoring, aggregation and show performance metric charts. Read about metrics and monitoring

Getting Stack Traces

If `geth` is started with the `--pprof` option, a debugging HTTP server is made available on port 6060. You can bring up http://localhost:6060/debug/pprof to see the heap, running routines etc. By clicking full goroutine stack dump (clicking http://localhost:6060/debug/pprof/goroutine?debug=2) you can generate trace that is useful for debugging.

Note that if you run multiple instances of `geth`, this port will only work for the first instance that was launched. If you want to generate stacktraces for these other instances, you need to start them up choosing an alternative pprof port. Make sure you are redirecting stderr to a logfile.

```
geth -port=30300 -verbosity 5 --pprof --pprofport 6060 2>> /tmp/00.glog
geth -port=30301 -verbosity 5 --pprof --pprofport 6061 2>> /tmp/01.glog
geth -port=30302 -verbosity 5 --pprof --pprofport 6062 2>> /tmp/02.glog
```

Alternatively if you want to kill the clients (in case they hang or stalled syncing, etc) but have the stacktrace too, you can use the `-QUIT` signal with `kill`:

```
killall -QUIT geth
```

This will dump stack traces for each instance to their respective log file.

Contributing

Thank you for considering to help out with the source code! We welcome contributions from anyone on the internet, and are grateful for even the smallest of fixes!

GitHub is used to track issues and contribute code, suggestions, feature requests or documentation.

If you'd like to contribute to go-ethereum, please fork, fix, commit and send a pull request (PR) for the maintainers to review and merge into the main code base. If you wish to submit more complex changes though, please check up with the core devs

first on our gitter channel to ensure those changes are in line with the general philosophy of the project and/or get some early feedback which can make both your efforts much lighter as well as our review and merge procedures quick and simple.

PRs need to be based on and opened against the `master` branch (unless by explicit agreement, you contribute to a complex feature branch).

Your PR will be reviewed according to the Code Review Guidelines.

We encourage a PR early approach, meaning you create the PR the earliest even without the fix/feature. This will let core devs and other volunteers know you picked up an issue. These early PRs should indicate 'in progress' status.

Dev Tutorials (mostly outdated)

- Private networks, local clusters and monitoring
- P2P 101: a tutorial about setting up and creating a p2p server and p2p sub protocol.
- How to Whisper: an introduction to whisper.

Whisper Overview

gluk256 edited this page on 10 May 2017 · 4 revisions

Whisper Overview

Whisper is a pure identity-based messaging system. Whisper provides a simple low-level API without being based upon or influenced by the low-level hardware attributes and characteristics. Peer-to-peer communication between the nodes of Whisper network uses the underlying ĐΞVp2p Wire Protocol. Whisper was not designed to provide a connection-oriented system, nor for simply delivering data between a pair of particular network endpoints. However, this might be necessary in some very specific cases (e.g. delivering the expired messages in case they were missed), and Whisper protocol will accommodate for that. Whisper is designed for easy and efficient broadcasting, and also for low-level asynchronous communications. It is designed to be a building block in next generation of unstoppable ĐApps. It was designed to provide resilience and privacy at considerable expense. At its most secure mode of operation, Whisper can theoretically deliver 100% darkness. Whisper should also allow the users to configure the level of privacy (how much information it leaks concerning the ĐApp content and ultimately, user activities) as a trade-off for performance.

Basically, all Whisper messages are supposed to be sent to every Whisper node. In order to prevent a DDoS attack, proof-of-work (PoW) algorithm is used. Messages will be processed (and forwarded further) only if their PoW exceeds a certain threshold, otherwise they will be dropped.

Encryption in version 5

All Whisper messages are encrypted and then sent via underlying ĐΞVp2p Protocol, which in turn uses its own encryption, on top of Whisper encryption. Every Whisper message must be encrypted either symmetrically or asymmetrically. Messages could be decrypted by anyone who possesses the corresponding key.

In previous versions unencrypted messages were allowed, but since it was necessary to exchange the topic (more on that below), the nodes might as well use the same communication channel to exchange encryption key.

Every node may possess multiple symmetric and asymmetric keys. Upon Envelope receipt, the node should try to decrypt it with each of the keys, depending on Envelope's encryption mode -- symmetric or asymmetric. In case of success, decrypted message is passed to the corresponding Đapp. In any case, every Envelope should be forwarded to each of the node's peers.

Asymmetric encryption uses the standard Elliptic Curve Integrated Encryption Scheme with SECP-256k1 public key. Symmetric encryption uses AES GCM algorithm with random 96-bit nonce. If the same nonce will be used twice, then all the previous messages encrypted with the same key will be compromised. Therefore no more than 2^48 messages should be encrypted with the same symmetric key (for detailed explanation please see the Birthday Paradox). However, since Whisper uses proof-of-work, this number could possibly be reached only under very special circumstances (e.g. private network with extremely high performance and reduced PoW). Still, usage of one-time session keys is strongly encouraged for all Đapps.

Although we assume these standard encryption algorithms to be reasonably secure, users are encouraged to use their own custom encryption on top of the default Whisper encryption.

Envelopes

Envelopes are the packets sent and received by Whisper nodes. Envelopes contain the encrypted payload and some metadata in plain format, because these data is essential for decryption. Envelopes are transmitted as RLP-encoded structures of the following format:

```
[ Version, Expiry, TTL, Topic, AESNonce, Data, EnvNonce ]
```

Version: up to 4 bytes (currently one byte containing zero). Version indicates encryption method. If Version is higher than current, envelope could not be decrypted, and therefore only forwarded to the peers.

Expiry time: 4 bytes (UNIX time in seconds).

TTL: 4 bytes (time-to-live in seconds).

Topic: 4 bytes of arbitrary data.

AESNonce: 12 bytes of random data (only present in case of symmetric encryption).

Data: byte array of arbitrary size (contains encrypted message).

EnvNonce: 8 bytes of arbitrary data (used for PoW calculation).

Whisper nodes know nothing about content of envelopes which they can not decrypt. The nodes pass envelopes around regardless of their ability to decrypt the message, or their interest in it at all. This is an important component in Whisper's dark communications strategy.

Messages

Message is the content of Envelope's payload in plain format (unencrypted).

Plaintext (unencrypted) message is formed as a concatenation of a single byte for flags, additional metadata (as stipulated by the flags) and the actual payload. The message has the following structure:

```
flags: 1 byte
optional padding: byte array of arbitrary size
payload: byte array of arbitrary size
optional signature: 65 bytes
```

Those unable to decrypt the message data are also unable to access the signature. The signature, if provided, is the ECDSA signature of the Keccak-256 hash of the unencrypted data using the secret key of the originator identity. The signature is serialised as the concatenation of the `R`, `S` and `V` parameters of the SECP-256k1 ECDSA signature, in that order. `R` and `S` are both big-endian encoded, fixed-width 256-bit unsigned. `V` is an 8-bit big-endian encoded, non-normalised and should be either 27 or 28.

The padding is introduced in order to align the message size, since message size alone might reveal important metainformation. The padding is supposed to contain random data (at least this is the default behaviour in version 5). However, it is possible to set arbitrary padding data, which might even be used for steganographic purposes. The API allows easy access to the padding data.

Default version 5 implementation ensures the size of the message to be multiple of 256. However, it is possible to set arbitrary padding through Whisper API. It might be useful for private Whisper networks, e.g. in order to ensure that all Whisper messages have the same (arbitrary) size. Incoming Whisper messages might have arbitrary padding size, and still be compatible with version 5.

The first several bytes of padding (up to four bytes) indicate the total size of padding. E.g. if padding is less than 256 bytes, then one byte is enough; if padding is less than 65536 bytes, then 2 bytes; and so on.

Flags byte uses only three bits in v.5. First two bits indicate, how many bytes indicate the padding size. The third byte indicates if signature is present. Other bits must be set to zero for backwards compatibility of future versions.

Topics

It might not be feasible to try to decrypt ALL incoming envelopes, because decryption is quite expensive. In order to facilitate the filtering, Topics were introduced to the Whisper protocol. Topic gives a probabilistic hint about encryption key. Single Topic corresponds to a single key (symmetric or asymmetric).

Upon receipt of a message, if the node detects a known Topic, it tries to decrypt the message with the corresponding key. In case of failure, the node assumes that Topic collision occurs, e.g. the message was encrypted with another key, and should be just forwarded further. Collisions are not only expected, they are necessary for plausible deniability.

Any Envelope could be encrypted only with one key, and therefore it contains only one Topic.

Topic field contains 4 bytes of arbitrary data. It might be generated from the key (e.g. first 4 bytes of the key hash), but we strongly discourage it. In order to avoid any compromise on security, Topics should be completely unrelated to the keys.

In order to use symmetric encryption, the nodes must exchange symmetric keys via some secure channel anyway. They might use the same channel in order to exchange the corresponding Topics as well.

In case of asymmetric encryption, it might be more complicated since public keys are meant to be exchanged via the open channels. So, the Đapp has a choice of either publishing its Topic along with the public key (thus compromising on privacy), or trying to decrypt all asymmetrically encrypted Envelopes (at considerable expense). Alternatively, PoW requirement for asymmetric Envelopes might be set much higher than for symmetric ones, in order to limit the number of futile attempts.

It is also possible to publish a partial Topic (first bytes), and then filter the incoming messages correspondingly. In this case the sender should set the first bytes as required, and rest should be randomly generated.

Examples:

```
Partial Topic: 0x12 (first byte must be 0x12, the last three bytes - random)
Partial Topic: 0x1234 (first two bytes must be {0x12, 0x34}, the last two bytes - random)
Partial Topic: 0x123456 (first three bytes must be {0x12, 0x34, 0x56}, the last byte - random)
```

Filters

Any Đapp can install multiple Filters utilising the Whisper API. Filters contain the secret key (symmetric or asymmetric), and some conditions, according to which the Filter should try to decrypt the incoming Envelopes. If Envelope does not satisfy those conditions, it should be ignored. Those are:

- array of possible Topics (or partial Topics)
- Sender address
- Recipient address
- PoW requirement

- AcceptP2P: boolean value, indicating whether the node accepts direct messages from trusted peers (reserved for some specific purposes, like Client/MailServer implementation)

All incoming messages, that have satisfied the Filter conditions AND have been successfully decrypted, will be saved by the corresponding Filter until the Đapp requests them. Đapps are expected to poll for incoming messages at regular time intervals. All installed Filters are independent of each other, and their conditions might overlap. If a message satisfies the conditions of multiple Filters, it will be stored in each of the Filters.

In future versions subscription will be used instead of polling.

In case of partial Topic, the message will match the Filter if first X bytes of the message Topic are equal to the corresponding bytes of partial Topic in Filter (where X can be 1, 2 or 3). The last bytes of the message Topic are ignored in this case.

Proof of Work

The purpose of PoW is spam prevention, and also reducing the burden on the network. The cost of computing PoW can be regarded as the price you pay for allocated resources if you want the network to store your message for a specific time (TTL). In terms of resources, it does not matter if the network stores X equal messages for Y seconds, or Y messages for X seconds. Or N messages of Z bytes each versus Z messages of N bytes. So, required PoW should be proportional to both message size and TTL.

After creating the Envelope, its Nonce should be repeatedly incremented, and then its hash should be calculated. This procedure could be run for a specific predefined time, looking for the lowest hash. Alternatively, the node might run the loop until certain predefined PoW is achieved.

In version 5, PoW is defined as average number of iterations, required to find the current BestBit (the number of leading zero bits in the hash), divided by message size and TTL:

```
PoW = (2^BestBit) / (size * TTL)
```

Thus, we can use PoW as a single aggregated parameter for the message rating. In the future versions every node will be able to set its own PoW requirement dynamically and

communicate this change to the other nodes via the Whisper protocol. Now it is only possible to set PoW requirement at the Đapp startup.

Packet Codes (ĐΞVp2p level)

As a sub-protocol of ĐΞVp2p, Whisper sends and receives its messages within ĐΞVp2p packets. Whisper v5 supports the following packet codes:

```
Status (0x0)

Messages (0x1)

P2PMessage (0x2)

P2PRequest (0x3)
```

Also, the following codes might be supported in the future:

```
PoWRequirement (0x4)

BloomFilterExchange (0x5)
```

Basic Operation

Nodes are expected to receive and send envelopes continuously. They should maintain a map of envelopes, indexed by expiry time, and prune accordingly. They should also efficiently deliver messages to the front-end API through maintaining mappings between Đapps, their filters and envelopes.

When a node's envelope memory becomes exhausted, a node may drop envelopes it considers unimportant or unlikely to please its peers. Nodes should rate peers higher if they pass them envelopes with higher PoW. Nodes should blacklist peers if they pass invalid envelopes, i.e., expired envelopes or envelopes with an implied insertion time in the future.

Nodes should always treat messages that its ĐApps have created no different than incoming messages.

Creating and Sending Messages

To send a message, the node should place the envelope its envelope pool. Then this envelope will be forwarded to the peers in due course along with the other envelopes. Composing an envelope from a basic payload, is done in a few steps:

- Compose the Envelope data by concatenating the relevant flag byte, padding, payload (randomly generated or provided by user), and an optional signature.
- Encrypt the data symmetrically or asymmetrically.
- Add a Topic.
- Set the TTL attribute.
- Set the expiry as the present Unix time plus TTL.
- Set the nonce which provides the best PoW.

Mail Server

Suppose, a Đapp waits for messages with certain Topic and suffers an unexpected network failure for certain period of time. As a result, a number of important messages will be lost. Since those messages are expired, there is no way to resend them via the normal Whisper channels, because they will be rejected and the peer punished.

One possible way to solve this problem is to run a Mail Server, which would store all the messages, and resend them at the request of the known nodes. Even though the server might repack the old messages and provide sufficient PoW, it's not feasible to resend all of them at whim, because it would tantamount to DDoS attack on the entire network. Instead, the Mail Server should engage in peer-to-peer communication with the node, and resend the expired messages directly. The recipient will consume the messages and will not forward them any further.

In order to facilitate this task, protocol-level support is provided in version 5. New message types are introduced to Whisper v.5: mailRequestCode and p2pCode.

- mailRequestCode is used by the node to request historic (expired) messages from the Mail Server. The payload of this message should be understood by the Server. The Whisper protocol is entirely agnostic about it. It might contain a time frame, the node's authorization details, filtering information, payment details, etc.

- p2pCode is a peer-to-peer message, that is not supposed to be forwarded to other peers. It will also bypass the protocol-level checks for expiry and PoW threshold.

There is MailServer interface defined in the codebase for easy Mail Server implementation. One only needs to implement two functions:

type MailServer interface { Archive(env *Envelope) DeliverMail(whisperPeer *Peer, data []byte) }

Archive should just save all incoming messages. DeliverMail should be able to process the request and send the historic messages to the corresponding peer (with p2pCode), according to additional information in the data parameter. This function will be invoked upon receipt of protocol-level mailRequestCode.