Ethereum Homestead Documentation

Release 0.1

Ethereum community

August 16, 2016

ethereum

Homestead Documentation

This documentation is the result of an ongoing collaborative effort by volunteers from the Ethereum *Community*. Although it has not been authorized by the *The Ethereum Foundation*, we hope you will find it useful, and welcome new *Contributors*.

Contents

1.1 Introduction

1.1.1 What is Ethereum?

Ethereum is an open blockchain platform that lets anyone build and use decentralized applications that run on blockchain technology. Like Bitcoin, no one controls or owns Ethereum – it is an open-source project built by many people around the world. But unlike the Bitcoin protocol, Ethereum was designed to be adaptable and flexible. It is easy to create new applications on the Ethereum platform, and with the Homestead release, it is now safe for anyone to use those applications.

A next generation blockchain

Blockchain technology is the technological basis of Bitcoin, first described by its mysterious author Satoshi Nakamoto in his white paper "Bitcoin: A Peer-to-Peer Electronic Cash System", published in 2008. While the use of blockchains for more general uses was already discussed in the original paper, it was not until a few years later that blockchain technology emerged as a generic term. A blockchain is a distributed computing architecture where every network node executes and records the same transactions, which are grouped into blocks. Only one block can be added at a time, and every block contains a mathematical proof that verifies that it follows in sequence from the previous block. In this way, the blockchain's "distributed database" is kept in consensus across the whole network. Individual user interactions with the ledger (transactions) are secured by strong cryptography. Nodes that maintain and verify the network are incentivized by mathematically enforced economic incentives coded into the protocol.

In Bitcoin's case the distributed database is conceived of as a table of account balances, a ledger, and transactions are transfers of the bitcoin token to facilitate trustless finance between individuals. But as bitcoin began attracting greater attention from developers and technologists, novel projects began to use the bitcoin network for purposes other than transfers of value tokens. Many of these took the form of "alt coins" - separate blockchains with cryptocurrencies of their own which improved on the original bitcoin protocol to add new features or capabilities. In late 2013, Ethereum's inventor Vitalik Buterin proposed that a single blockchain with the capability to be reprogrammed to perform any arbitrarily complex computation could subsume these many other projects.

In 2014, Ethereum founders Vitalik Buterin, Gavin Wood and Jeffrey Wilcke began work on a next-generation blockchain that had the ambitions to implement a general, fully trustless smart contract platform.

Ethereum Virtual Machine

Ethereum is a programmable blockchain. Rather than give users a set of pre-defined operations (e.g. bitcoin transactions), Ethereum allows users to create their own operations of any complexity they wish. In this way, it serves as a platform for many different types of decentralized blockchain applications, including but not limited to cryptocurrencies.

Ethereum in the narrow sense refers to a suite of protocols that define a platform for decentralised applications. At the heart of it is the *Ethereum Virtual Machine ("EVM")*, which can execute code of arbitrary algorithmic complexity. In computer science terms, Ethereum is "Turing complete". Developers can create applications that run on the EVM using friendly programming languages modelled on existing languages like JavaScript and Python.

Like any blockchain, Ethereum also includes a peer-to-peer network protocol. The Ethereum blockchain database is maintained and updated by many nodes connected to the network. Each and every node of the network runs the EVM and executes the same instructions. For this reason, Ethereum is sometimes described evocatively as a "world computer".

This massive parallelisation of computing across the entire Ethereum network is not done to make computation more efficient. In fact, this process makes computation on Ethereum far slower and more expensive than on a traditional "computer". Rather, every Ethereum node runs the EVM in order to maintain consensus across the blockchain. Decentralized consensus gives Ethereum extreme levels of fault tolerance, ensures zero downtime, and makes data stored on the blockchain forever unchangeable and censorship-resistant.

The Ethereum platform itself is featureless or value-agnostic. Similar to programming languages, it is up to entrepreneurs and developers to decide what it should be used for. However, it is clear that certain application types benefit more than others from Ethereum's capabilities. Specifically, ethereum is **suited for applications that automate direct interaction between peers or facilitate coordinated group action across a network**. For instance, applications for coordinating peer-to-peer marketplaces, or the automation of complex financial contracts. Bitcoin allows for individuals to exchange cash without involving any middlemen like financial institutions, banks, or governments. Ethereum's impact may be more far-reaching. In theory, financial interactions or exchanges of any complexity could be carried out automatically and reliably using code running on Ethereum. Beyond financial applications, any environments where trust, security, and permanence are important – for instance, asset-registries, voting, governance, and the internet of things – could be massively impacted by the Ethereum platform.

How does Ethereum work?

Ethereum incorporates many features and technologies that will be familiar to users of Bitcoin, while also introducing many modifications and innovations of its own.

Whereas the Bitcoin blockchain was purely a list of transactions, *Ethereum's basic unit is the account*. The Ethereum blockchain tracks the state of every account, and all state transitions on the Ethereum blockchain are transfers of value and information between accounts. There are two types of accounts:

- Externally Owned Accounts (EOAs), which are controlled by private keys
- Contract Accounts, which are controlled by their contract code and can only be "activated" by an EOA

For most users, the basic difference between these is that human users control EOAs - because they can control the private keys which give control over an EOA. Contract accounts, on the other hand, are governed by their internal code. If they are "controlled" by a human user, it is because they are *programmed* to be controlled by an EOA with a certain address, which is in turn controlled by whoever holds the private keys that control that EOA. The popular term "smart contracts" refers to code in a Contract Account – programs that execute when a transaction is sent to that account. Users can create new contracts by deploying code to the blockchain.

Contract accounts only perform an operation when instructed to do so by an EOA. So it is not possible for a Contract account to be performing native operations like random number generation or API calls – it can do these things only if prompted by an EOA. This is because Ethereum requires nodes to be able to agree on the outcome of computation, which requires a guarantee of strictly deterministic execution.

Like in Bitcoin, users must pay small transaction fees to the network. This protects the Ethereum blockchain from frivolous or malicious computational tasks, like DDoS attacks or infinite loops. The sender of a transaction must pay for each step of the "program" they activated, including computation and memory storage. These fees are paid in amounts of Ethereum's native value-token, ether.

These transaction fees are collected by the nodes that validate the network. These "miners" are nodes in the Ethereum network that receive, propogate, verify, and execute transactions. The miners then group the transactions – which include many updates to the "state" of accounts in the Ethereum blockchain – into what are called "blocks", and miners then compete with one another for *their* block to be the next one to be added to the

blockchain. Miners are rewarded with ether for each successful block they mine. This provides the economic incentive for people to dedicate hardware and electricity to the Ethereum network.

Just as in the Bitcoin network, miners are tasked with solving a complex mathematical problem in order to successfully "mine" a block. This is known as a "Proof of Work". Any computational problem that requires orders of magnitude more resources to solve algorithmically than it takes to verify the solution is a good candidate for proof of work. In order to discourage centralisation due to the use of specialised hardware (e.g. ASICs), as has occurred in the Bitcoin network, Ethereum chose a memory-hard computational problem. If the problem requires memory as well as CPU, the ideal hardware is in fact the general computer. This makes Ethereum's Proof of Work ASIC-resistant, allowing a more decentralized distribution of security than blockchains whose mining is dominated by specialized hardware, like Bitcoin.

Learn about Ethereum

[to be extended]

PR videos with some pathos:

- Ethereum: the World Computer
- Ethereum – your turn

Blockchain and Ethereum 101

- Explain bitcoin like I'm five - an excellent introduction to blockchain technology and bitcoin to the mildly techsavvy layperson.
- https://medium.com/@creole/7-a-simple-view-of-ethereum-e276f76c980b
- http://blog.chain.com/post/92660909216/explaining-ethereum
- Explain Ethereum to non-technical people Q&A on stackexchange
- Reddit threads on ELI5-ing Ethereum:

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19]

Videos

- http://change.is/video/ethereum-the-world-computer-featuring-dr-gavin-wood

Infographics

- Ethereum explained...[to your mother]
- http://decentral.ca/wp-content/uploads/2016/03/infographic.jpg
- https://medium.com/@angelomilan/ethereum-explained-to-my-mom-infographic-673e32054c1c#.n9kzhme6v

Comparison to alternatives

- NXT
- MaidSafe

1.1.2 How to use this guide?

Using Ethereum: The Basics

This section captures the basic ways in which a user would want to participate in the Ethereum project. First of all becoming a node in the network you need to run an Ethereum client. Multiple implementations are listed in the section *Choosing a client* which also gives you advice what clients to choose in various setups. *Connecting to the Network* gives you basic information about networks, connectivity troubleshooting and blockchain synchronization. Advanced network topics like setting up private chains is found in *Test Networks*.

1.1.3 The Homestead Release

Homestead is the second major version of the Ethereum platform and is the first production release of Ethereum. It includes several protocol changes and a networking change that provides the ability to do further network upgrades. The first version of Ethereum, called the Frontier release, was essentially a beta release that allowed developers to learn, experiment, and begin building Ethereum decentralized apps and tools.

Milestones of the Ethereum development roadmap

The original development roadmap laid out before Ethereum went live specified the following milestones:

- Prerelease Step 0: Olympic testnet - launched May 2015
- Release Step One: Frontier - launched 30 July 2015
- Release Step Two: Homestead - launches 14 March 2016 (Pi Day)
- Release Step Three: Metropolis - TBA
- Release Step Four: Serenity - TBA

While still valid, the substance behind it has changed somewhat. The Olympic testnet phase (before the Frontier release) saw a lot of major improvements, followed by Frontier which was launched immediately after. Homestead marks the exit from a beta product to a stable release. Homestead is introduced automatically at block number 1,150,000 which should occur roughly around March 14th, 2016, Pi Day.

If you are running a node connected to the live network, it is important that you upgrade to a Homestead-compatible client. Such clients with their versions are listed under *Ethereum Clients*. Otherwise you will end up on the wrong fork and will no longer be in sync with the rest of the network.

Once the Ethereum blockchain reaches block 1,150,000, the Ethereum network will undergo a hardfork enabling a few major changes such as explained in the following section.

Homestead hard fork changes

Ethereum in the narrow formal sense is a suite of protocols. Homestead comes with a few backward-incompatible protocol changes, and therefore will require a hard fork. These changes that made their way through the process for *Ethereum Improvement Proposals* and included are:

- EIP 2:
  - cost for creating contracts via a transaction is increased from 21000 to 53000. Contract creation from a contract using the `CREATE` opcode is unaffected.
  - transaction signatures whose s-value is greater than `secp256k1n/2` are now considered invalid
  - If contract creation does not have enough gas to pay for the final gas fee for adding the contract code to the state, the contract creation fails (ie. goes out-of-gas) rather than leaving an empty contract.
  - Change the difficulty adjustment algorithm

- EIP 7: DELEGATECALL: Add a new opcode, `DELEGATECALL` at `0xf4`, which is similar in idea to `CALLCODE`, except that it propagates the sender and value from the parent scope to the child scope, ie. the call created has the same sender and value as the original call. This means contracts can store pass through information while following msg.sender and `msg.value` from its parent contract. Great for contracts which create contracts but don't repeat additional information which saves gas. See comments on EIP 7
- EIP 8: devp2p Forward Compatibility compliance with the Robustness Principle Changes to the RLPx Discovery Protocol and RLPx TCP transfer protocol to ensure that all client software in use on the Ethereum network can cope with future network protocol upgrades. For older versions of an Ethereum client, updates to the network protocol weren't being accepted by older clients and would refuse communication if the hello packets didn't meet expectations. This update means all future versions of the client will accept incoming network upgrades and handshakes.

The changes have the following benefits:

- EIP-2/1 eliminates the excess incentive to create contracts via transactions, where the cost is 21000, rather than contracts, where the cost is 32000.
- EIP-2/1 also fixes the protocol "bug" that with the help of suicide refunds, it is currently possible to make a simple ether value transfer using only 11664 gas.
- EIP-2/2 fixes a transaction malleability concern (not a security flaw, but a UI incovenience).
- EIP-2/3 creates a more intuitive "success or fail" distinction in the result of a contract creation process, rather than the current "success, fail, or empty contract" trichotomy
- EIP-2/4 eliminates the excess incentive to set the timestamp difference to exactly 1 in order to create a block that has slightly higher difficulty and that will thus be guaranteed to beat out any possible forks. This guarantees to keep block time in the 10-20 range and according to simulations restores the target 15 second blocktime (instead of the current effective 17s).
- EIP-7 makes it much easier for a contract to store another address as a mutable source of code and ''pass through" calls to it, as the child code would execute in essentially the same environment (except for reduced gas and increased callstack depth) as the parent.
- EIP-8 makes sure that all client software in use on the Ethereum network can cope with future network protocol upgrades.

Additional resources: - Reddit discussion on Homestead Release - *Ethereum Improvement Proposals (EIPs)*

1.1.4 Web 3: A platform for decentralized apps

Many have come to believe that an open, trustless blockchain platform like Ethereum is perfectly suited to serve as the shared "back end" to a decentralized, secure internet - Web 3.0. An internet where core services like DNS and digital identity are decentralized, and where individuals can engage in economic interactions with each other.

As intended by the Ethereum developers, Ethereum is a blank canvas and you have the freedom to build whatever you want with it. The Ethereum protocol is meant to be generalized so that the core features can be combined in arbitrary ways. Ideally, dapp projects on Ethereum will leverage the Ethereum blockchain to build solutions that rely on decentralized consensus to provide new products and services that were not previously possible.

Ethereum is perhaps best described as an ecosystem: the core protocol is supported by various pieces of infrastructure, code, and community that together make up the Ethereum project. Ethereum can also be understood by looking at the projects that use Ethereum. Already, there are a number of high-profile projects built on Ethereum such as Augur, Digix, Maker, and many more (see *Dapps*). In addition, there are development teams that build open source components that anyone can use. While each of these organizations are separate from the Ethereum Foundation and have their own goals, they undoubtedly benefit the overall Ethereum ecosystem.

Further Watching/Reading:

- Vitalik Buterin - TNABC 2015: https://www.youtube.com/watch?v=Fjhe0MVRHO4
- Gavin Wood - DEVCON 1 - Ethereum for Dummies: https://www.youtube.com/watch?v=U_LK0t_qaPo
- Ethereum London Meetup (best detailed here): https://www.youtube.com/watch?v=GJGIeSCgskc

Smart contracts

by Alex:

Would you enter in a contract with someone you've never met? Would you agree to lend money to some farmer in Ethiopia? Would you become an investor in a minority-run newspaper in a war zone? Would you go to the hassle of writing up a legal binding contract for a $5 dollar purchase over the internet?

The answer is no for most of these questions, the reason being that a contract requires a large infrastructure: sometimes you need a working trust relationship between the two parties, sometimes you rely on a working legal system, police force and lawyer costs.

In Ethereum you don't need any of that: if all the requisites to the contract can be put in the blockchain then they will, in a trustless environment for almost no cost.

Instead of thinking of moving your current contracts to the blockchain, think of all the thousand little contracts that you would never agree to simply because they weren't economically feasible or there was not enough legal protection..

DAO

Here is just one example: imagine you own a small business with your friends. Lawyers and accountants are expensive, and trusting a single partner to oversee the books can be a source of tension (even an opportunity for fraud). Complying strictly with a system in which more than one partner oversees the books can be trying and is subject to fraud whenever the protocol isn't followed exactly.

Using a smart contract, ownership in your company and terms for the disbursal of funds can be specified at the outset. The smart contract can be written such that it is only changeable given the approval of a majority of owners. Smart contracts like these will likely be available as open source software, so you won't even need to hire your own programmer instead of an accountant/lawyer.

A smart contract like this scales instantly. A couple of teenagers can split revenue from a lemonade stand just as transparently as a sovereign wealth fund can disburse funds to the hundred million citizens who are entitled to it. In both cases the price of this transparency is likely to be fractions of a penny per dollar.

1.1.5 History of Ethereum

For a recent historical account, see Taylor Gerring's blogpost

Inception

Ethereum was initially described by Vitalik Buterin in late 2013 as a result of his research and work in the Bitcoin community. Shortly thereafter, Vitalik published the Ethereum white paper, where he describes in detail the technical design and rationale for the Ethereum protocol and smart contracts architecture. In January 2014, Ethereum was formally announced by Vitalik at the The North American Bitcoin Conference in Miami, Florida, USA.

Around that time, Vitalik also started working with Dr. Gavin Wood and together co-founded Ethereum. By April 2014, Gavin published the Ethereum Yellow Paper that would serve as the technical specification for the Ethereum Virtual Machine (EVM). By following the detailed specification in the Yellow Paper, the Ethereum client has been implemented in seven programming languages (C++, Go, Python, Java, JavaScript, Haskell, Rust), and has resulted in better software overall.

- Ethereum launches Cryptocurrency 2.0 network - Coindesk article of 2014 Jan on the beginnings
- Ethereum announcement on bitcointalk Vitalik's original announcement to the bitcoin community. Forum thread with 5000 replies.

The Ethereum Foundation and the ether presale

In addition to developing the software for Ethereum, the ability to launch a new cryptocurrency and blockchain requires a massive bootstrapping effort in order to assemble the resources needed to get it up and running. To kickstart a large network of developers, miners, investors, and other stakeholders, Ethereum announced its plan to conduct a presale of ether tokens, the currency unit of Ethereum. The legal and financial complexities of raising funds through a presale led to the creation of several legal entities, including the *Ethereum Foundation (Stiftung Ethereum)* established June 2014 in Zug, Switzerland.

Beginning in July 2014, Ethereum distributed the initial allocation of ether via a 42-day public ether presale, netting 31,591 bitcoins, worth $18,439,086 at that time, in exchange for about 60,102,216 ether. The results of the sale were initially used to pay back mounting legal debts and also for the months of developer effort that had yet to be compensated, and to finance the ongoing development of the Ethereum.

- Launching the ether sale - original official announcement on the Ethereum blog
- Concise information-rich stats page about the presale by (since then inactive) Ether.Fund
- Overview: Ethereum's initial public sale - Blogpost by slacknation - all stats about the ether presale
- Terms and Conditions of the Presale

ETH/DEV and Ethereum development

Following the successful ether presale, Ethereum development was formalized under a non-for-profit organization called ETH DEV, which manages the development of Ethereum under contract from Ethereum Suisse – with Vitalik Buterin, Gavin Wood, and Jeffrey Wilcke as the 3 directors of the organization. Developer interest in Ethereum grew steadily throughout 2014 and the ETH DEV team delivered a series of proof-of-concept (PoC) releases for the development community to evaluate. Frequent posts by ETH DEV team on the the Ethereum blog also kept the excitement and momentum around Ethereum going. Increasing traffic and growing user-base on both the Ethereum forum and the ethereum subreddit testified that the platform is attracting a fast-growing and devoted developer community. This trend has been continuing to this very day.

DEVCON-0

In November 2014, ETH DEV organized the DEVCON-0 event, which brought together Ethereum developers from around the world to Berlin to meet and discuss a diverse range of Ethereum technology topics. Several of the presentations and sessions at DEVcon-0 would later drive important initiatives to make Ethereum more reliable, more secure, and more scalable. Overall, the event galvanized developers as they continued to work towards the launch of Ethereum.

- DEVCON-0 talks youtube playlist
- DEVCON-0 reddit post
- Gav's DEV update mentioning DEVCON-0
- DEVcon-0 recap blog post

DEVgrants program

In April 2015, the DEVgrants program was announced, which is a program that offers funding for contributions both to the Ethereum platform, and to projects based on Ethereum. Hundreds of developers were already contributing their time and thinking to Ethereum projects and in open source projects. This program served to reward and support those developers for their contributions. The DEVgrants program continues to operate today and funding of the program was recently renewed in January 2016.

- DEVgrants initial announcement
- Announcement of new funding at DEVCON-1
- DEVgrants public gitter room

- DEVgrants talk at DEVCON-1 by Wendell Davis on YouTube

Olympic testnet, bug bounty and security audit

Throughout 2014 and 2015 development went through a series of proof of concept releases leading to the 9th POC open testnet, called Olympic. The developer community was invited to test the limits of the network and a substantial prize fund was allocated to award those holding various records or having success in breaking the system in some way or other. The rewards were announced officially a month after the live release.

In early 2015, an Ethereum Bounty Program was launched, offering BTC rewards for finding vulnerabilities in any part of the Ethereum software stack. This has undoubtedly contributed to the reliability and security of Ethereum and the confidence of the Ethereum community in the technology. The bounty program is currently still active and there is no end date planned.

The Ethereum security audit began at the end of 2014 and continued through the first half of 2015. Ethereum engaged multiple third party software security firms to conduct an end-to-end audit of all protocol-critical components (Ethereum VM, networking, Proof of Work). The audits uncovered security issues that were addressed and tested again and as a result ultimately led to a more secure platform.

- Olympic testnet prerelease - Vitalik's blogpost detailing olympic rewards
- Olympic rewards announced - Vitalik's blogpost detailing the winners and prizes
- Bug bounty program launch
- Ethereum Bounty Program website
- Least Authority audit blogpost - with links to the audit report
- Deja Vu audit blogpost

The Ethereum Frontier launch

The Ethereum Frontier network launched on July 30th, 2015, and developers began writing smart contracts and decentralized apps to deploy on the live Ethereum network. In addition, miners began to join the Ethereum network to help secure the Ethereum blockchain and earn ether from mining blocks. Even though the Frontier release is the first milestone in the Ethereum project and was intended for use by developers as a beta version, it turned out to be more capable and reliable than anyone expected, and developers have rushed in to build solutions and improve the Ethereum ecosystem.

See also:

- Original announcement of the release scheme by Vinay Gupta
- Frontier is coming - Frontier launch announcement by Stephan Tual
- Frontier launch final steps - Follow-up post to announcement
- Ethereum goes live with Frontier launch
- The frontier website

DEVCON-1

The second developers' conference DEVCON-1 took place in the city of London at the beginning of November 2015. The 5-day event featured more than 100 presentations, panel discussions and lightning talks, attracted more than 400 participants, a mix of developers, entrepreneurs, thinkers, and business executives. The talks were all recorded and are freely available

The presence of large companies like UBS, IBM and Microsoft clearly indicated enterprise interest in the technology. Microsoft announced that it would offer Ethereum on its new Blockchain as a Service offering on the Microsoft Azure cloud platform. In conjunction with DEVCON-1, this announcement will be remembered as the moment when blockchain technology became mainstream, with Ethereum at the center of it.

- DEVCON-1 talks Youtube playlist
- DEVCON-1 website full listing of presentations with links to the slides if available.

History resources

- a simple graphical timeline

1.1.6 Community

Please choose your forum wisely when starting a discussion or asking a question, help keep our various forums clean and tidy.

Reddit

The Ethereum subreddit is the most inclusive Ethereum forum, where most of the community discussion is happening and where core devs are also active. This is your forum of choice for generic discussion of news, media coverage, announcements, brainstorming. In general all things Ethereum relevant to the wider community.

Strictly no price discussion.

Also, this is not the ideal place to ask for hands-on help or post questions you expect there are clear immediate answers to (use *Gitter Rooms* and *Stack Exchange* for these, respectively).

Read the Ethereum subreddit rules before posting.

Further specialised subreddits:

- /r/EthTrader - Ether trading, price and market
- /r/EtherMining - Ether mining discussion
- /r/Ethmarket - Marketplace for individuals looking to exchange goods and services for Ether
- /r/Ethinvestor - News and prospects for Ethereum investors. Following the long term trends in the Ethereum marketplace.
- /r/ethereumism/ - a bit more ism, ostic, ical, ist and tinfoil hats, pyramids and crystal ball type of views - the ethereal side of Ethereum

Stack Exchange

The Ethereum Stack Exchange is part of the StackExchange network of Q&A communities. StackExchange is a free Q&A site where all the questions and answers are preserved for posterity.

This is the best place to ask technical questions. Help your fellow etherians by answering questions and collect reputation points.

Gitter Rooms

Gitter is our forum of choice for daily chat. It is the virtual coworking space where devs hang out, so it is where you can get quick help and a bit of handholding if needed.

Gitter uses Github accounts, offers Github integration (notification of pull requests etc), private channels, provides markdown formatting, and more.

Most Gitter channels are organised around particular repositories, or generic topics like research or governance. Please choose the appropriate room and keep discussions on topic.

See the full list of gitter rooms for the Ethereum organisation. Below is the list of active public channels:

- go-ethereum - about geth (and tools related to the go implementation)

- cpp-ethereum - about eth (and tools related to the C++ implementation)
- web3.js - about web3.js, Ethereum JavaScript API library
- Solidity - The Solidity Contract-Oriented Programming Language
- serpent - The Serpent language for contract development
- mist - GUI dapp browser, official wallet app
- light-client - about light client and the LES protocol
- research - Ethereum research
- governance - about dev governance
- whisper - anonymous datagram publishing
- swarm - decentralised content storage and distribution network
- EIPs - discussion of Ethereum Improvement Proposals (EIPs)
- ethereumjs-lib - a JavaScript library of core Ethereum functions
- devp2p - ÐΞV's p2p network protocol & framework

Ethereum Improvement Proposals (EIPs)

The EIP scheme aims to be a framework and largely informal business process coordinating improvements to the protocols. People should first propose their idea as an issue or pull request to the EIPs repository. After basic filtering, the proposal will receive a number and is published in draft form. For an EIP to become Active it will require the mutual consent of the community. Those proposing changes should consider that ultimately consent may rest with the consensus of the Ethereum users. For discussion of EIPs, use the gitter channel for EIP discussions.

- EIP guidelines and sample EIP
- EIP template
- EIP repository and README
- gitter channel for EIP discussions

Meetups

- Directory hosted on Meetup
- Meetup channel on Ethereum Forum

Obsolete

Skype

Some community discussion fora still use skype rooms, but we would like to move away from that and encourage people to use gitter or slack.

Ethereum Forum

Stephan Tual's legendary Ethereum Forum is no longer maintained and likely to be decommissioned soon. We encourage people to use one of the recommended alternatives listed above.

1.1.7 The Ethereum Foundation

The Ethereum Foundation is a non-profit organization registered in Switzerland, and has the purpose of managing the funds that were raised from the Ether Sale in order to best serve the Ethereum and decentralized technology ecosystem.

Founded July 2014 in Switzerland, Stiftung Ethereum's mission is the promotion of developments of new technologies and applications, especially in the fields of new open and decentralized software architectures.

It is the aim that decentralized and open technologies will be developed, nurtured, promoted and maintained. A dominating, but not exclusive, focus is set on the promotion of the development of the Ethereum Protocol and the relevant technology to it as well as the promotion and support of applications using the Ethereum technology or protocol. Stiftung Ethereum will additionally support and advocate for a decentralized Internet in a variety of forms.

Find out about more about the Foundation Management Team on the website

Ethereum Foundation's faces to the community

- Official Homestead website - main entrypoint
- Reddit - see *Community*
- Blog
- Twitter
- Youtube
- Facebook - largely unused
- Email - use if you must

Official communication from the Ethereum foundation most often comes in the form of a comprehensive blogpost on the Ethereum blog. Some of the posts there are technical, some organisational, some personal. All blog posts are announced on Twitter and Reddit.

The foundation Youtube channel hosts our videos, including all talks of the developers conferences DEVCON0 and DEVCON1.

For community discussion forums, see *Community*.

1.1.8 Contributors

This documentation was built collectively by the Ethereum community as part of a project called the Homestead Documentation Initiative which was coordinated by:

- Viktor Trón ("zelig")
- Hudson Jameson ("Souptacular")

We would like to thank everybody who helped in this effort for their contributions:

- Ricardo de Azevedo Brandao
- Santanu Barai
- Brooks Boyd
- RJ Catalano
- Joseph Chow
- Keri Clowes
- François Deppierraz
- Bertie Dinneen
- Erik Edrosa
- Andrey Fedorov
- Rocky Fikki
- Alex Fisher
- Enrique Fynn
- Arno Gaboury
- Taylor Gerring
- Dave Hoover
- Joël Hubert
- Makoto Inoue

- Keith Irwin
- Matthias Käppler
- Bas van Kervel
- Michael Kilday
- Chandra Kumar
- Guangmian Kung
- Hugh Lang
- Yann Levreau
- Roman Mandeleil
- Kévin Maschtaler
- Andrew Mazzola
- Dominik Miszkiewicz
- John Mooney
- Chris Peel
- Craig Polley
- Colm Ragu
- Laurent Raufaste
- Christian Reitwiessner
- Josh Stark
- Scott Stevenson
- Bob Summerwill
- Alex van de Sande
- Paul Schmitzer
- Afri Schoedon
- Sudeep Singh
- Giacomo Tazzari
- Ben Tannenbaum
- Dean Alain Vernon
- Paul Worrall
- Luca Zeug
- Weiyang Zhu
- Will Zeng

And these pseudonymous contributors:

- 12v
- c0d3inj3cT
- ijcoe6ru
- LucaTony
- madhancr
- mWo

- Omkara
- tflux99
- xyzether

1.2 Ethereum Clients

1.2.1 Choosing a client

Why are there multiple Ethereum clients?

The Ethereum clients are very analogous to a Java VM or .NET runtime.

They enable you to execute "Ethereum programs" on your computer. They are implemented to a written specification (the Yellow Paper) and by design are interoperable and somewhat "commodity".

From the earlier days of the project there have been multiple interoperable client implementations across a range of different operating systems. That client diversity is a huge win for the ecosystem as a whole. It lets us verify that the protocol is unambiguous. It keeps the door open for new innovation. It keeps us all honest. However, it can be very confusing for end-users, because there is no universal "Ethereum Installer" for them to use.

As we enter the Homestead phase, the Go client is very, very dominant, but it hasn't always been that way, and won't necessarily be that way in the future. All of the clients except ethereumH have Homestead-compatible releases. The table below contains links to the latest release.

Client	Language	Developers	Latest release
go-ethereum	Go	Ethereum Foundation	go-ethereum-v1.4.10
Parity	Rust	Ethcore	Parity-v1.3.0
cpp-ethereum	C++	Ethereum Foundation	cpp-ethereum-v1.3.0
pyethapp	Python	Ethereum Foundation	pyethapp-v1.4.0
ethereumjs-lib	Javascript	Ethereum Foundation	ethereumjs-lib-v3.0.0
Ethereum(J)	Java	<ether.camp>	ethereumJ-v1.3.0-RC5-DaoHardFork
ruby-ethereum	Ruby	Jan Xie	ruby-ethereum-v0.9.3
ethereumH	Haskell	BlockApps	no Homestead release yet

1.2.2 Installing a Client

There are a number of "official" clients whose development has been funded from the resources administered by the Ethereum Foundation. There are also various other clients which have been built by the community or by other commercial entities.

Read more about the specific clients in the specific client sections in this chapter.

What should I install on my desktop/laptop?

If you have a laptop or desktop machine, you should probably just install the Ethereum Wallet and you are done.

- Download the latest Ethereum Wallet ZIP from Github.
- Unzip wherever you like
- Click on the executable (**Ethereum-Wallet, Ethereum-Wallet** or **Ethereum-Wallet.app**)
- The block-chain will be downloaded

The Ethereum Wallet is a "single dapp" deployment of the **Mist Browser** which will be the centerpiece of the Metropolis phase of development, which comes after Homestead.

Mist comes with bundled *go-ethereum* and *cpp-ethereum* binaries and if you are not running a command-line Ethereum client when Mist starts then it will start running one of the bundles clients for you.

If you want to interact with Ethereum on the command-line, and to take advantage of the Javascript console then you will want to install one of the client applications directly, as well as Mist.

go-ethereum and *cpp-ethereum* are the best place to start, because they have both been under development since the start of the project, have passed security audits, work for all platforms and have *The Ethereum Foundation* resources assigned to their ongoing maintenance and support.

- Follow the *Installing binaries* instructions for **cpp-ethereum**
- For **go-ethereum**, just unzip the released binaries

Parity is gaining in popularity fast.

Beyond that, of course, it is all a matter of personal preference. Try them all :-)

If you want to do mining then Mist will not be sufficient. Check out the *Mining* section.

What should I install on my mobile/tablet?

We are at the very beginning of our support for mobile devices. The Go team are publishing experimental iOS and Android libraries, which some developers are using to start bootstrapping mobile applications, but there are not yet any mobile Ethereum clients available.

The main hinderance to the use of Ethereum on mobile devices is that the Light Client support is still incomplete. The work which has been done is off in a private branch, and is only available for the Go client. doublethinkco will be starting development of Light Client for the C++ client in the coming months, following grant funding.

Check out Syng.im, who were initially using ethereumj-personal based on *Ethereum(J)*, but have recently flipped to Geth cross-builds with Light Client.

What should I install on my SBC?

You have some choice here depending on your skill level, and what you are looking to do.

- Download a fully prepared image(link to page with detailed download & install instructions)
  - If you are new to Ethereum AND SBC boards such as the Raspberry Pi then this is for you! Simply download the image specific to the dev board you are working with, burn it to an SD card, boot your device, and run Ethereum!
- Download a pre-compiled application(link to page with detailed download & install instructions)
  - If you already have an SBC running and have a specific, preferred OS or setup that you want to keep, then this is your best option! Depending on the platform, you can simply download the apropriate executable, and with minimal linking of libraries and setting of PATH you can have Ethereum running in your existing environment!
- Build from source using customizable scripts(link to page with more detail and individual SBC links to https://github.com/ethembedded)
  - Looking to perform a custom install? We have scripts available to compile from source "on device". Our scripts contain auto-install of dependencies as well as the client itself. This will allow you to install a specific version of the Ethereum client(i.e.-"develop", "master", etc.), compile your own forked version of a client, and generally play around with the intracacies of the build process.

1.2.3 cpp-ethereum

Quick Start

- Welcome to the Ethereum C++ project :-)
- The Github repository for this project is ethereum/cpp-ethereum
- Automation runs on Appveyor and TravisCI.
- We have instructions for *Installing binaries* and *Building from source*.
- Most project communication happens in our User and Developer Gitter channels.
- Issues are tracked in our Github issue tracker.
- cpp-ethereum is extremely portable and is used on a *very broad range of platforms*.

Details

Current status

We blog about the codebase periodically on the Official Ethereum blog and elsewhere. Here are some recent articles from the development team:

- Ethereum DEV Update: C++ Roadmap (February 2016)
- C++ DEV Update: Announcing Remix (May 2016)
- C++ DEV Update – July edition (July 2016)
- Ethereum Everywhere (July 2016)
- C++ re-licensing plan (July 2016)

We simplified the project naming at Homestead (March 2016), although some naming shadows of the past still linger. With the homecoming we have another name to retire - **webthree-umbrella**.

At the time of writing (August 2016), we are just completing our "Homecoming", where the code has been reconsolidated into the ethereum/cpp-ethereum repository. From October 2015 until August 2016 it was split across multiple repositories under ethereum/webthree-umbrella

The re-licensing plan is the culmination of a very long-term plan to liberalize the core. An effort was begun in 2015 to re-license the cpp-ethereum core as MIT, but it was never completed.

This is a revival of that effort, especially with a view towards the potential for collaboration with the Linux Foundation's Hyperledger project, and with other corporations outside of Hyperledger who wish to build Ethereum private/consortium chain solutions similar to HydraChain. The Rubix by Deloitte project is an example of that approach.

Building from source

Overview The **cpp-ethereum** codebase lives on Github.com in the cpp-ethereum repository.

Between October 2015 and August 2016 it was split into various repositories which were grouped as sub-modules under the webthree-umbrella repository, and you will likely see many references to **webthree-umbrella** online. Those all refer to the **cpp-ethereum** codebase during that period of its development.

We use a common CMake build system to generate platform-specific build files, meaning that the workflow is very similar whatever operating system you use:

- Install build tools and external packages (these are platform dependent)
- Clone the source code from the **webthree-umbrella** git repository
- Run CMake to generate a build file (makefile, Visual Studio solution, etc)
- Build it

Platform-specific instructions

Building for Linux NOTE - It may be possible to get the client working for Linux 32-bit, by disabling EVMJIT and maybe other features too. We might accept pull-requests to add such support, but we will not put any of our own development time into supporting Linux 32-bit builds.

Linux has a horror-show of distro-specific packaging system steps which are the first thing which we need to do before we can start on *Building from source*. The sections below attempt to capture those steps. If you are using as different distro and hit issues, please let us know.

Clone the repository To clone the source code, execute the following command:

```
git clone --recursive https://github.com/ethereum/cpp-ethereum.git
cd cpp-ethereum
```

Installing dependencies (the easy way) For the "Homecoming" release (v1.3.0) in July 2016 we added a new "one-button" script for installing external dependencies, which identifies your distro and installs the packages which you need. This script is new and incomplete, but is a way easier experience than the manual steps described in the next section of this document. Give it a go!

It works for Debian, Ubuntu and macOS and a few other distros already

If you try it, and it doesn't work for you, please let us know and we will prioritize fixing your distro!:

```
./scripts/install_deps.sh
```

Installing dependencies manually (distro-specific)

Installing dependencies for Fedora

Fedora 20 Steps:

```
yum install git automake autoconf libtool cmake gcc gcc-c++ xkeyboard-config \
        leveldb-devel boost-devel gmp-devel cryptopp-devel miniupnpc-devel \
        qt5-qtbase-devel qt5-qtdeclarative-devel qt5-qtquick1-devel qt5-qtwebkit-devel \
        mesa-dri-drivers snappy-devel ncurses-devel readline-devel curl-devel \
        python-devel
```

Fedora 21 Steps:

```
yum install git automake autoconf libtool cmake gcc gcc-c++ xkeyboard-config \
        leveldb-devel boost-devel gmp-devel cryptopp-devel miniupnpc-devel \
        qt5-qtbase-devel qt5-qtdeclarative-devel qt5-qtquick1-devel qt5-qtwebkit-devel \
        mesa-dri-drivers snappy-devel ncurses-devel readline-devel curl-devel \
        python-devel jsoncpp-devel argtable-devel
```

Build json-rpc from github as per https://github.com/ethereum/cpp-ethereum/issues/617:

```
git clone https://github.com/cinemast/libjson-rpc-cpp
cd libjson-rpc-cpp
git checkout tags/v0.3.2
mkdir -p build
cd build
cmake .. && make
sudo make install
sudo ldconfig
```

Fedora 22 Fedora 22 dependencies there may be more depends what you have already installed:

```
dnf install git automake autoconf libtool cmake gcc gcc-c++ xkeyboard-config \
leveldb-devel boost-devel gmp-devel cryptopp-devel miniupnpc-devel \
mesa-dri-drivers snappy-devel ncurses-devel readline-devel curl-devel \
python-devel jsoncpp-devel argtable-devel
```

Install gcc version 4.9! Fedora 22 comes with a different compiler (CC v5.3). This one wont compile webthree-umbrella 4 me so i installed gcc version 4.9 from SRC!

Check that you have a working gcc4.9 install in /usr/local i installed it in /home/app/gcc49 its your choice read manual how to compile gcc in google! After that you have to compile everything you need 4 webthree-umbrella with gcc4.9 so before every cmake:

```
export CXX=/home/app/gcc49/bin/g++
    export CC=/home/app/gcc49/bin/gcc
```

With this you use gcc4.9 to compile instead of the one that comes with the distro F22. Its not recommended to uninstall the compiler that comes with your distro! You can also work with symlinking.

Install from Fedora COPR REPO LLVM3.7 with:

```
dnf copr enable alonid/llvm-3.7
dnf install llvm-3.7 llvm-3.7-devel llvm-3.7-static llvm-3.7-libs
```

I had to do this because Fedora 22 comes with llvm-3.5 from stock repos! There may be other solutions but this one worked 4 me

Install CryptoPP from SRC https://github.com/weidai11/cryptopp CRYPTOPP_5_6_2:

```
git clone https://github.com/weidai11/cryptopp
cd cryptopp
git checkout release/CRYPTOPP_5_6_2
mkdir build
cd build
export CXX=/home/app/gcc49/bin/g++ <- be sure to compile with gcc4.9
export CC=/home/app/gcc49/bin/gcc <- be sure to compile with gcc4.9
cmake ..
make
make install
```

Install QT5 from COPR "dnf copr enable @kdesig/Qt5" newer QT5 version:

```
dnf install qt5-*
```

this should install QT5 version 5.6.0 in COPR repo are other QT5. Packages from other users i didnt test them

Install qtwebengine from https://github.com/qtproject/qtwebengine i installed version 5.6.0 others may also work find it out :D

```
git clone https://github.com/qtproject/qtwebengine
cd qtwebengine
git checkout release/v5.6.0
qmake-qt5 <- in other distros its just called qmake in fedora 22 qmake-qt5
make
make install
```

Install json-rpc from github https://github.com/ethereum/cpp-ethereum/issues/617:

```
git clone https://github.com/cinemast/libjson-rpc-cpp
cd libjson-rpc-cpp
git checkout tags/v0.4.2
mkdir -p build
cd build
export CXX=/home/app/gcc49/bin/g++ <- be sure to compile with gcc4.9
export CC=/home/app/gcc49/bin/gcc <- be sure to compile with gcc4.9
cmake .. && make
sudo make install
sudo ldconfig
```

Be sure to check if jsonrpcstub works in console enter "jsonrpcstub" and look if its responding. If it answers No Argument or s-l-t it works but if you get no such file to blabla.so you have to symlinking the missing ones to your libs dir /usr/local/lib64 or usr/local/lib depends where the file blabla.so is try to find it with "updatedb" and than "locate blabla.so"

Try to compile now it should work if not there a missing symlinks cause of no such file easyfix or there are some missing Packages try to find them with dnf like this "dnf search packname*" or "dnf list packname*" all i can say its not a 5 min compile of webthree-umbrella enjoy Tflux99.

Installing dependencies for openSUSE Here is how to get the dependencies needed to build the latest webthree-umbrella on OpenSUSE. This was done on Leap 42.1, but there should be equivalent packages available for Tumbleweed and 13.x.

First install dependencies provided by the main repos:

```
zypper in git automake autoconf libtool cmake gcc gcc-c++ \
    xkeyboard-config leveldb-devel boost-devel gmp-devel \
    cryptopp-devel libminiupnpc-devel libqt5-qtbase-common-devel \
    libqt5-qtdeclarative-devel libQTWebKit-devel libqt5-qtwebengine-devel \
    libQt5Concurrent-devel Mesa ncurses-devel readline-devel libcurl-devel \
    llvm llvm-clang llvm-clang-devel llvm-devel libLLVM binutils \
    libmicrohttp-devel jsoncpp-devel opencl-headers-1.2 zlib-devel
```

It may be possible to use the generic *libOpenCL1*, but I have only tested with the AMD proprietary package from the AMD drivers repo *fglrx64_opencl_SUSE421*

These packages are not in the standard repos but can be found using the OpenSUSE build service package search and YaST 1-Click Install:

- libargtable2-devel
- libv8-3
- v8-devel

If you also have v8 from the chromium repo installed the devel package will default to the 4.x branch which will not work. Use YaST or zypper to downgrade this package to 3.x

Installing dependencies for Arch Linux Compiling webthree-umbrella on Arch Linux requires dependencies from both the official repositories and the Arch User Repository (AUR). To install packages from the official repositories pacman is used. For installation of packages from the AUR, a number of AUR helpers is available. For this guide, yaourt AUR helper is used.

Installing dependencies

from official repositories sudo pacman -Sy git base-devel cmake boost crypto++ leveldb llvm miniupnpc libcl opencl-headers libmicrohttpd qt5-base qt5-webengine

from AUR yaourt -Sy libjson-rpc-cpp

Compiling the source code During this step, an installation folder for the Ethereum can be specified. Specification of the folder is optional though. If not given, the binary files will be located in the build folder. However, for this guide, it is assumed that the Ethereum files will be installed under */opt/eth*. The reason for using */opt* is that it makes much easier to delete the Ethereum files later on, as compared to having them installed under, e.g., */usr*. Also */opt* is commonly used to install software that is not managed by packaging systems, such as manually compiled programs.

```
# enter webthree-umbrella folder after cloning its github repository
cd webthree-umbrella

# make a build folder and enter into it
mkdir -p build && cd build

# create build files and specify Ethereum installation folder
cmake .. -DCMAKE_INSTALL_PREFIX=/opt/eth

# compile the source code
make

# alternatively it is possible to specify number of compilation threads
# for example to use 4 threads execute make as follows:
# make -j 4

# install the resulting binaries, shared libraries and header files into /opt
sudo make install
```

After successful compilation and installation, Ethereum binaries can be found in */opt/eth/bin*, shared libraries in */opt/eth/lib*, and header files in */opt/eth/include*.

Specifying Ethereum libraries path Since Ethereum was installed in */opt/eth*, executing its binaries can result in linker error due to not being able to find the Ethereum shared libraries. To rectify this issue, it is needed to add the folder containing Ethereum shared libraries into *LD_LIBRARY_PATH* environmental variable:

```
# update ~/.bashrc
echo "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/eth/lib" >> ~/.bashrc

# reload ~/.bashrc
source ~/.bashrc
```

Build on the command-line When you have installed your dependencies you can build.

```
mkdir build                                                    Make a directory for the build output
cd build                                                       Switch into that directory

cmake ..                                                       To generate a makefile.
make                                                           To build that makefile on the command-li
make -j<number>                                                (or) Execute makefile with multiple cores
```

Building for Windows We support **only 64-bit** builds and only for the following versions of Windows:

- Windows 7
- Windows 8/8.1
- Windows 10
- Windows Server 2012 R2

It may be possible to get the client working for Windows 32-bit, by disabling EVMJIT and maybe other features too. We might accept pull-requests to add such support, but we will not put any of our own development time into supporting Windows 32-bit builds.

Pre-requisites You will need to install the following dependencies

Software	Notes
Git for Windows	Command-line tool for retrieving source from Github.
CMake	Cross-platform build file generator.
Visual Studio 2015	C++ compiler and dev environment.

Get the source Clone the git repository containing all the source code by executing the following command:

```
git clone --recursive https://github.com/ethereum/cpp-ethereum.git
cd cpp-ethereum
```

Get the external dependencies Execute the CMake script that downloads and unpacks pre-built external libraries needed to build the project:

```
scripts\install_deps.bat
```

Generate Visual Studio project files Then execute the following commands, which will generate a Visual Studio solution file using CMake:

```
mkdir build
cd build
cmake -G "Visual Studio 14 2015 Win64" ..
```

Which should result in the creation of **cpp-ethereum.sln** in that build directory.

NOTE: We only support Visual Studio 2015 as of cpp-ethereum-v.1.3.0.

Double-clicking on that file should result in Visual Studio firing up. We suggest building **RelWithDebugInfo** configuration, but all others work.

Build on the command-line Alternatively, you can build the project on the command-line, like so:

```
cmake --build . --config RelWithDebInfo
```

Building for OS X

Overview - Here be dragons! It is impossible for us to avoid OS X build breaks because Homebrew is a "rolling release" package manager which means that the ground will forever be moving underneath us unless we add all external dependencies to our Homebrew tap, or add them as git sub-modules. End-user results vary depending on when they are build the project. Building yesterday may have worked for you, but that doesn't guarantee that your friend will have the same result today on their machine. Needless to say, this isn't a happy situation.

If you hit build breaks for OS X please look through the Github issues to see whether the issue you are experiencing has already been reported. If so, please comment on that existing issue. If you don't see anything which looks similar, please create a new issue, detailing your OS X version, cpp-ethereum version, hardware and any other details you think might be relevant. Please add verbose log files via gist.github.com or a similar service.

The cpp-ethereum-development gitter channel is where we hang out, and try to work together to get known issues resolved.

We only support the following OS X versions:

- OS X Mavericks (10.9)
- OS X Yosemite (10.10)
- OS X El Capitan (10.11)

The cpp-ethereum code base does not build on older OS X versions and this is not something which we will ever support. If you are using an older OS X version, we recommend that you update to the latest release, not just so that you can build cpp-ethereum, but for your own security.

Clone the repository To clone the source code, execute the following command:

```
git clone --recursive https://github.com/ethereum/cpp-ethereum.git
cd cpp-ethereum
```

Pre-requisites and external dependencies Ensure that you have the latest version of xcode installed. This contains the Clang C++ compiler, the xcode IDE and other Apple development tools which are required for building C++ applications on OS X. If you are installing xcode for the first time, or have just installed a new version then you will need to agree to the license before you can do command-line builds:

```
sudo xcodebuild -license accept
```

Our OS X builds require you to install the Homebrew package manager for installing external dependencies. Here's how to uninstall Homebrew, if you ever want to start again from scratch.

We now have a "one button" script which installs all required external dependencies on macOS and on numerous Linux distros. This used to a multi-step manual process:

```
./scripts/install_deps.sh
```

Command-line build From the project root:

```
mkdir build
cd build
cmake ..
make -j4              (or different value, depending on your number of CPU cores)
```

Install your own build You can also use the same Makefile to install your own build globally on your machine:

```
make install
```

This will install binaries into **/usr/local/** and **/usr/bin/**.

Generate xcode project From the project root:

```
mkdir build_xc
cd build_xc
cmake -G Xcode ..
```

This will generate an Xcode project file called **cpp-ethereum.xcodeproj**, which you can then open with xcode and build/debug/run.

Building for FreeBSD NOTE - Once the packages are in the FreeBSD main ports this guide should be changed to something much more simple

Install the ports manually For some of this steps you must require a root access to modify the ports directory.

The webthree-umbrella depends on [libjson-rpc-cpp.shar](https://raw.githubusercontent.com/enriquefynn/webthree-umbrella-port/master/libjson-rpc-cpp.shar) that is also not in the ports system.

First you need to download the shar file and place it on your ports directory under the "devel" session, usually /usr/ports/devel

```
curl https://raw.githubusercontent.com/enriquefynn/webthree-umbrella-port/master/libjson-rpc-cpp.s
```

Now we execute the script with:

```
cd /usr/ports/devel
sh libjson-rpc-cpp.shar
```

This will create the libjson-rpc-cpp port. Now you should do the same for the webthree-umbrella port, we should get the [webthree-umbrella](https://raw.githubusercontent.com/enriquefynn/webthree-umbrella-port/master/webthree-umbrella.shar) file and create the port under "net-p2p" directory.

```
curl https://raw.githubusercontent.com/enriquefynn/webthree-umbrella-port/master/webthree-umbrella
cd /usr/ports/net-p2p
sh webthree-umbrella.shar
```

Build and Install Now you can navigate to the webthree-umbrella directory and install the port:

```
cd /usr/ports/net-p2p/webthree-umbrella
make install clean
```

Building for Android We don't currently have a working Android build, though that is on the roadmap for doublethinkco. Android uses the Linux kernel, but has a different API than the ARM Linux cross-builds, meaning that specific binaries will be required.

ARM Linux distros use the GLIBC runtime library, where Android uses bionic.

Building for iOS We don't currently have a working iOS build, though that is on the roadmap for doublethinkco. iOS is a UNIX-like operating system based on Darwin (BSD) using ARM chips. This is a different API than the ARM Linux cross-builds, meaning that specific binaries will be required.

Building for Raspberry Pi Model A, B+, Zero, 2 and 3 EthEmbedded maintain build scripts for all Raspberry Mi models. They are on Github in the Raspi-Eth-Install repository. It is also possible to cross-build for these platforms.

Building for Odroid XU3/XU4 EthEmbedded maintain build scripts for both of these Odroid models. Support for a broader range of Odroid devices is likely in the future. They are on Github in the OdroidXU3-Eth-Install repository. It is also possible to cross-build for these platforms.

Building for BeagleBone Black EthEmbedded maintain build scripts for BBB on Github in the BBB-Eth-Install repository. It is also possible to cross-build for this platform.

Building for WandBoard EthEmbedded maintain build scripts for the WandBoard on Github in the WandBoard-Eth-Install repository. It is also possible to cross-build for this platform.

Building for Linux for ARM (cross builds) doublethinkco maintain a Docker-based cross-build infrastructure which is hosted on Github in the cpp-ethereum-cross repository.

At the time of writing, these cross-built binaries have been successfully used on the following devices:

- Jolla Phone (Sailfish OS)
- Nexus 5 (Sailfish OS)
- Meizu MX4 Ubuntu Edition (Ubuntu Phone)
- Raspberry Pi Model B+, Rpi2 (Raspbian)
- Odroid XU3 (Ubuntu MATE)
- BeagleBone Black (Debian)
- Wandboard Quad (Debian)
- C.H.I.P. (Debian)

Still TODO:

- Tizen
- Android
- iOS

Installing binaries

The cpp-ethereum development team and the broader Ethereum community publish binary releases in many different forms for a variety of platforms. This aims to be a complete list of those releases.

If you are aware of other third-party packaging efforts, please let us know on the cpp-ethereum gitter channel, and we will add them to this list.

Ubuntu PPA (Personal Package Archive) We have set up PPA instances for the following Ubuntu versions:

- Ubuntu Trusty Tahr (14.04)
- Ubuntu Utopic Unicorn (14.10)
- Ubuntu Vivid Vervet (15.04)
- Ubuntu Wily Werewolf (15.10)
- Ubuntu Xenial Xerus (16.04)

We only support 64-bit builds. It may be possible to get the client working for Ubuntu 32-bit, by building from source and disabling EVMJIT and maybe other features too. We might accept pull-requests to add such support, but we will not put any of our development time into supporting Ubuntu 32-bit builds.

For the latest stable version:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install cpp-ethereum
```

If you want to use the cutting edge developer version:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install cpp-ethereum
```

Windows Chocolatey NuGet packages We aren't generating Chocolatey packages at the time of writing, though we have done so in the past.

For anybody who isn't already familiar with the technology, this is essentially *apt-get for Windows* - a global silent installer for tools.

We would like to support Chocolatey again in the near future for all the same reasons we support Homebrew on OS X and have PPAs for Ubuntu. For technically competent users, doing command-line operations like so would be very convenient:

```
choco install cpp-ethereum
choco update cpp-ethereum
```

OS X Homebrew packages We generate Homebrew packages within our automated build system for the following OS X / Mac versions:

- **'OS X Mavericks (10.9) <https://en.wikipedia.org/wiki/OS_X_Mavericks'_**
- OS X Yosemite (10.10)
- OS X El Capitan (10.11)

We only support 64-bit builds.

If your system does not support these OS X versions then you are out of luck. Sorry!

All OS X builds require you to install the Homebrew package manager before doing anything else. Here's how to uninstall Homebrew, if you ever want to start again from scratch.

To install the Ethereum C++ components from Homebrew, execute these commands:

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install cpp-ethereum
brew linkapps cpp-ethereum
```

Here is the Homebrew Formula which details all the supported command-line options.

Raspberry Pi, Odroid, BeagleBone Black, Wandboard John Gerryts of EthEmbedded builds binary images for a variety of SBCs at major milestones, in addition to testing and maintaining build scripts for these devices. EthEmbedded was a devgrant recipient in May 2015. He builds binaries for both eth and geth.

Here are the Homestead binaries from EthEmbedded

Linux ARM cross-builds for mobile, wearables, SBCs Bob Summerwill, of doublethinkco cross-builds ARM binaries which work on a very broad variety of hardware, from mobile and wearables Linux distros (Sailfish OS, Tizen OS, Ubuntu Touch) to the same SBCs which EthEmbedded target - and more. doublethinkco was a BlockGrantX recipient in Feb 2016.

See the cpp-ethereum-cross README for a full matrix of platforms and known status.

Here are the cross-build binaries from doublethinkco: RELEASED – Cross-build eth binaries for Homestead.

ArchLinux User Repository (AUR) Arch Linux packages are community maintained by Afri Schoedon.

Check out the following packages on aur.archlinux.org.

- ethereum (stable, latest release)
- ethereum-git (unstable, latest develop)

To build and install the package, follow the AUR installing package instructions:

- Acquire the tarball which contains the PKGBUILD
- Extract the tarball
- Run `makepkg -sri` as simple user in the directory where the files are saved
- Install the resulting package with `pacman -U` as superuser

You can also use AUR helpers like `yaourt` or `pacaur` to install the packages directly on your system.

Architecture

- bench: trie benchmarking
- cmake: cmake files for build system, contains specification of inter-dependencies
- eth A command-line Ethereum full-node that can be controlled via RPC.
- ethkey: stand-alone key management
- ethminer: stand-alone ethash miner
- ethvm: stand-alone EVM execution utility
- evmjit: library for the EVM just-in-time compiler
- libdevcore: data structures, utilities, rlp, trie, memory db
- libdevcrypto: crypto primitives. Depends on libsecp256k1 and libcrypto++.
- libp2p: core peer to peer networking implementation (excluding specific sub-protocols)
- libethash: ethash mining POW algorithm implementation
- libethash-cl: ethash mining code for GPU mining (OpenCL)
- libethashseal: generic wrapper around the POW block seal engine. Also contains the genesis states for all ethash-based chains.
- libethcore: collection of core data structures and concepts
- libethereum: main consensus engine (minus EVM). Includes the State and BlockChain classes.
- libevm: Ethereum Virtual Machine implementation (interpreter).

- libevmasm: EVM assembly tools, also contains the optimizer.
- libevmcore: elementary data structures of the EVM, opcodes, gas costs, ...
- libweb3jsonrpc: json-rpc server-side endpoint, provides http and IPC (unix socket, windows pipe) connectors
- libwebthree: service connectors for ethereum, swarm/ipfs and whisper.
- libwhisper: whisper implementation
- rlp: stand-alone rlp en-/decoder
- testeth: tests for the modules formerly within the **libethereum** repo
- testweb3core: tests for the modules formerly within the **libweb3core** repo
- testweb3: tests for the modules formerly within the **webthree** repo
- utils/json_spirit: JSON parser written for Boost's Spirit library.
- utils/libscrypt: scrypt implementation
- utils/secp256k1: implementation of the SECP 256k1 ECDSA signing algorithm.

Portability

The Ethereum C++ client code is exceedingly portable, and is being successfully used on a huge range of different operating systems and devices.

We continue to expand our range and are very open to pull-requests which add support for additional operating systems, compilers or devices.

Operating systems verified as working

- **Linux**
  - Alpine Linux
  - Arch Linux
  - Debian 8 (Jessie)
  - Fedora 20
  - Fedora 21
  - Fedora 22
  - openSUSE Leap 42.1
  - Raspbian
  - Sailfish OS 2.0
  - Ubuntu 14.04 (Trusty)
  - Ubuntu 14.10 (Utopic)
  - Ubuntu 15.04 (Vivid)
  - Ubuntu 15.10 (Wily)
  - Ubuntu 16.04 (Xenial)
  - Ubuntu Touch
  - Ubuntu 15.04 MATE
- **BSD**
  - FreeBSD

- **OS X**
  - OS X Yosemite (10.10)
  - OS X El Capitan (10.11)
  - OS X 10.10 (Yosemite Server 4.0)
  - OS X 10.11 (Yosemite Server 5.0)
  - OS X 10.11 (Yosemite Server 5.1)
- **Windows**
  - Windows 7
  - Windows 8
  - Windows 8.1
  - Windows 10
  - Windows Server 2012 R2

Operating systems - work in progress

- **Linux**
  - Maemo
  - MeeGo
  - Tizen
- **BSD**
  - iOS
  - tvOS
  - WatchOS
- Android

Devices verified as working

- **All varieties of desktop and laptop devices (Windows, OS X, Desktop Linux)**
  - 64-bit (with rebuilt binaries)
  - 32-bit (not official supported, but they work)
- **Smartphones**
  - **Linux**
    - Jolla Phone
    - Meizu MX4 Ubuntu Edition
    - Nexus 5 (SailfishOS 2.0)
- **SBCs**
  - **Linux**
    - BeagleBone Black
    - Odroid XU3
    - Project C.H.I.P.
    - Raspberry Pi Model A

        * Raspberry Pi Model B+
        * Raspberry Pi Zero
        * Raspberry Pi 2
        * Raspberry Pi 3
        * Wandboard Quad

Devices - work in progress

- **Smartwatches**
    - **Linux**
        * Samsung Gear S2
    - **BSD**
        * Apple Watch
- **Smartphones**
    - **Linux**
        * Nokia N9 (MeeGo)
        * Nokia N900 (Meemo)
        * Samsung Z1
        * Samsung Z3
    - **Android**
        * Samsung Galaxy S3
        * Samsung Galaxy S4
    - **BSD**
        * iPhone 3GS
        * iPhone 5
- **Developer phones**
    - **Linux**
        * Samsung RD-210
        * Samsung RD-PQ
        * Samsung TM1
- **Tablets**
    - **Android**
        * Samsung Galaxy Tab S 10.5
        * Nexus 7
    - **BSD**
        * iPad Air 2
- **SBCs**
    - **Linux**
        * DragonBoard 410c
        * Intel Curie

* Intel Edison
* Intel NUC
* Minnowboard Max
* Odroid XU4

Running

Running eth without any argument will synchronise your node to the public blockchain. It is also possible to create or synchronise to another blockchain (see *custom blockchain using eth*).

Interacting with your node can be done using either geth or the ethereum console:

Using geth

Using the ethereum console

The ethereum console is a node.js application which connect to a running eth/geth node and provide access to the web3 object.

Note: https://github.com/ethereum/ethereum-console

It can be installed using npm:

Note:

> npm install -g ethereum-console
> ethconsole

Note:

Usage:
ethconsole [javascript file] [ipc socket]
Connects to an ethereum node via ipc in order to control it remotely
through global variable web3 (web3.admin is also present).
If no arguments are given, connects to the default ipc socket
and drops into interactive mode.
Arguments:
<ipc socket path> connect to the given ipc socket (use ipc://<path> if it does not end with .ipc)
<javascript file> execute the given javascript file that has to end in .js non-interactively.
The script has to call process.exit() in order to terminate the console.

1.2.4 go-ethereum

The go-ethereum client is commonly referred to as **geth**, which is the the command line interface for running a full ethereum node implemented in Go. By installing and running geth, you can take part in the ethereum frontier live network and:

- mine real ether
- transfer funds between addresses
- create contracts and send transactions
- explore block history

- and much much more

Links:

- Website: http://ethereum.github.io/go-ethereum/
- Github: https://github.com/ethereum/go-ethereum
- Wiki: https://github.com/ethereum/go-ethereum/wiki/geth
- Gitter: https://gitter.im/ethereum/go-ethereum

1.2.5 pyethapp

pyethapp is the python-based client implementing the Ethereum cryptoeconomic state machine. The python implementation aims to provide an easily hackable and extendable codebase.

pyethapp leverages two ethereum core components to implement the client:

- pyethereum - the core library, featuring the blockchain, the ethereum virtual machine, mining
- pydevp2p - the p2p networking library, featuring node discovery for and transport of multiple services over multiplexed and encrypted connections

Links:

- Github: https://github.com/ethereum/pyethapp
- Wiki: https://github.com/ethereum/pyethapp/wiki/Getting-Started
- Gitter chat: https://gitter.im/ethereum/pyethapp

1.2.6 ethereumjs-lib

ethereumjs-lib is the javascript library of core Ethereum functions as described in the Yellow Paper. This is a simple meta-module that provides the following modules. Most JS modules are tracked in ethereumjs

- VM - The Ethereum virtual machine and state processing functions
- Blockchain - Blockchain managment
- Block - Block Schema definition and validation
- Transaction - Transaction Schema definition and validation
- Account - Account Schema definition and validation
- rlp - Recursive Length Prefix serialization
- Trie - Modified Merkle Patricia Tree
- Ethash - Ethereum's Proof of Work algorithm
- utils - Miscellaneous helper functions
- devp2p - The networking protocol
- devp2p-dpt - The disputed peer table

Links:

- Github: https://github.com/ethereumjs/ethereumjs-lib
- Join the Gitter chat: https://gitter.im/ethereum/ethereumjs-lib

1.2.7 Ethereum(J)

Ethereum(J) is a pure-Java implementation of the Ethereum protocol. It is provided as a library that can be embedded in any Java/Scala project and to provide full support for Ethereum protocol and sub-services. Ethereum(J) was first developed by Roman Mandeleil and is now sponsored by <ether.camp>.

Ethereum(J) supports CPU mining. It is currently implemented in pure Java and can be used in private and test networks. You may even mine on the live Ethereum network, even though it is not economically feasible.

Links:

- Blog: http://ethereumj.io/
- Github: https://github.com/ethereum/ethereumj
- Gitter chat: https://gitter.im/ethereum/ethereumj

1.2.8 ethereumH

This package provides a tool written in Haskell to allow you to connect to the Ethereum blockchain

Links:

- Github: https://github.com/blockapps/ethereumH
- BlockApps: http://www.blockapps.net/

1.2.9 Parity

Parity claims to be the world's fastest and lightest Ethereum client. It is written in the Rust language, which offers improved reliability, performance, and code clarity. Parity is being developed by Ethcore, which was founded by several members of the Ethereum Foundation.

- Website: https://ethcore.io/parity.html
- Github: https://github.com/ethcore/parity
- Gitter chat: https://gitter.im/ethcore/parity

Arch Linux packages are community maintained by Afri Schoedon and quininer.

- https://aur.archlinux.org/packages/parity/ (stable, latest release)
- https://aur.archlinux.org/packages/parity-git/ (unstable, latest develop)

Some people have reported success with Parity on Raspberry Pi 2.

1.2.10 ruby-ethereum

ruby-ethereum is an implementation of the *Ethereum Virtual Machine* written in Ruby.

Links:

- Github: https://github.com/janx/ruby-ethereum
- Gem: https://rubygems.org/gems/ruby-ethereum

Related:

- ruby-serpent: Ruby binding to the Ethereum Serpent compiler.
- ethereum-ruby: a pure-Ruby JSON-RPC wrapper for communicating with an Ethereum node. To use this library you will need to have a running Ethereum node with IPC support enabled (default). Currently, the *go-ethereum* client is supported.

1.3 Account Management

1.3.1 Accounts

Accounts play a central role in Ethereum. There are two types of accounts: *externally owned accounts* (EOAs) and *contract accounts*. Here we focus on externally owned accounts, which will be referred to simply as *accounts*. Contract accounts will be referred to as *contracts* and are *discussed in detail in Contracts*. This generic notion of account subsuming both externally owned accounts and contracts is justified in that these entities are so called *state objects*. These entities have a state: accounts have balance and contracts have both balance and contract storage. The state of all accounts is the state of the Ethereum network which is updated with every block and which the network really needs to reach a consensus about. Account are essential for users to interact with the Ethereum blockchain via transactions.

If we restrict Ethereum to only externally owned accounts and allow only transactions between them, we arrive at an "altcoin" system that is less powerful than bitcoin itself and can only be used to transfer ether.

Accounts represent identities of external agents (e.g., human personas, mining nodes or automated agents). Accounts use public key cryptography to sign transaction so that the EVM can securely validate the identity of a transaction sender.

1.3.2 Keyfiles

Every account is defined by a pair of keys, a private key and public key. Accounts are indexed by their *address* which is derived from the public key by taking the last 20 bytes. Every private key/address pair is encoded in a *keyfile*. Keyfiles are JSON text files which you can open and view in any text editor. The critical component of the keyfile, your account's private key, is always encrypted, and it is encrypted with the password you enter when you create the account. Keyfiles are found in the `keystore` subdirectory of your Ethereum node's data directory. Make sure you backup your keyfiles regularly! See the section *Backup and restore accounts* for more information.

Creating a key is tantamount to creating an account.

- You don't need to tell anybody else you're doing it
- You don't need to synchronize with the blockchain
- You don't need to run a client
- You don't even need to be connected to the internet

Of course your new account will not contain any Ether. But it'll be yours and you can be certain that without your key and your password, nobody else can ever access it.

It is safe to transfer the entire directory or any individual keyfile between Ethereum nodes.

> **Warning:** Note that in case you are adding keyfiles 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.

1.3.3 Creating an account

> **Warning: Remember your passwords and 'backup your keyfiles <backup-and-restore-accounts>'_.** In order to send transactions from an account, including sending ether, you must have BOTH the keyfile and the password. Be absolutely sure to have a copy of your keyfile AND remember the password for that keyfile, and store them both as securely as possible. There are no escape routes here; lose the keyfile or forget your password and all your ether is gone. It is NOT possible to access your account without a password and there is no *forgot my password* option here. Do not forget it.

Using `geth account new`

Once you have the geth client installed, creating an account is merely a case of executing the `geth account new` command in a terminal.

Note that you do not have to run the geth client or sync up with the blockchain to use the `geth account` command.

```
$ 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}
```

For non-interactive use you supply a plaintext password file as argument to the `--password` flag. The data in the file consists of the raw bytes of the password optionally followed by a single newline.

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

Warning: Using the `--password` flag is meant to be used only for testing or automation in trusted environments. It is a bad idea to save your password to file or expose it in any other way. If you do use the `--password` flag with a password file, make sure the file is not readable or even listable for anyone but you. You can achieve this in Mac/Linux systems with:

```
touch /path/to/password
chmod 600 /path/to/password
cat > /path/to/password
>I type my pass
```

To list all the accounts with keyfiles currently in you're `keystore` folder use the `list` subcommand of the `geth account` command:

```
$ geth account list

account #0: {a94f5374fce5edbc8e2a8697c15331677e6ebf0b}
account #1: {c385233b188811c9f355d4caec14df86d6248235}
account #2: {7f444580bfef4b9bc7e14eb7fb2a029336b07c9d}
```

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

Using geth console

In order to create a new account using geth, we must first start geth in console mode (or you can use `geth attach` to attach a console to an already running instance):

```
> geth console 2>> file_to_log_output
instance: Geth/v1.4.0-unstable/linux/go1.5.1
coinbase: coinbase: [object Object]
at block: 865174 (Mon, 18 Jan 2016 02:58:53 GMT)
datadir: /home/USERNAME/.ethereum
```

The console allows you to interact with your local node by issuing commands. For example, try the command to list your accounts:

```
> eth.accounts

{
code: -32000,
```

```
message: "no keys in store"
}
```

This shows that you have no accounts. You can also create an account from the console:

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xb2f69ddf70297958e582a0cc98bce43294f1007d"
```

Note: Remember to use a strong and randomly generated password.

We just created our first account. If we try to list our accounts again we can see our new account:

```
> eth.accounts
["0xb2f69ddf70297958e582a0cc98bce43294f1007d"]
```

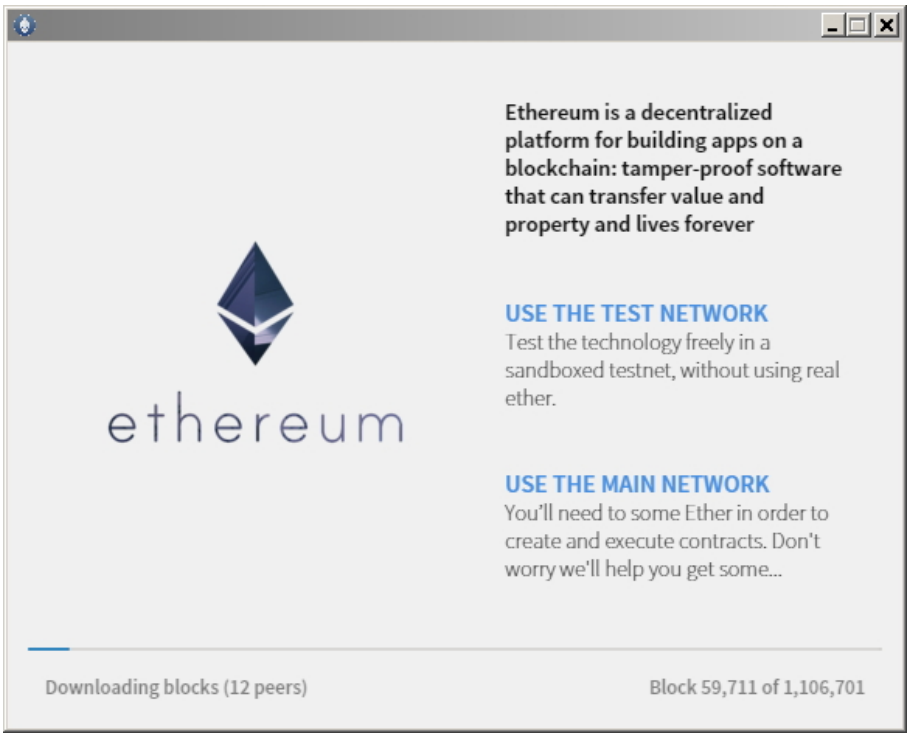
Using Mist Ethereum wallet

For the command line averse, there is now a GUI-based option for creating accounts: The "official" Mist Ethereum wallet. The Mist Ethereum wallet, and its parent Mist project, are being developed under the auspices of the Ethereum Foundation, hence the "official" status. Versions of the wallet app are available for Linux, Mac OS X, and Windows.

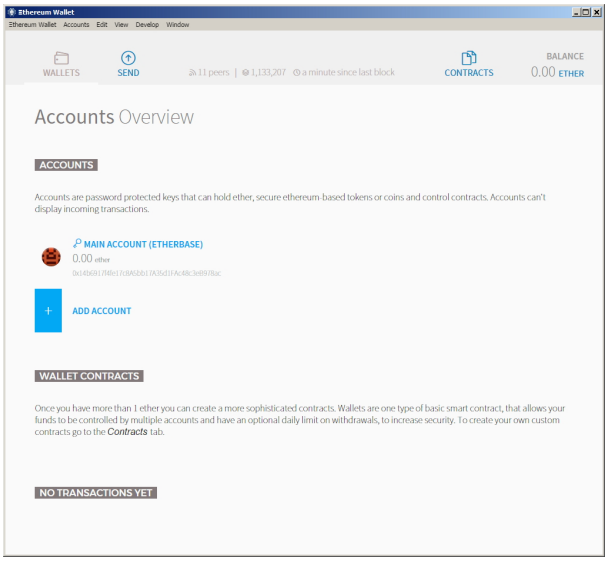
Warning: The Mist wallet is beta software. Please beware and use it at your own risk.

Creating an account using the GUI Mist Ethereum wallet couldn't be easier. In fact, your first account is created during the installation of the app.

1. Download the latest version of the wallet app for your operating system. Opening the Wallet App will kick off syncing a full copy of the Ethereum blockchain on your computer, since you will in effect be running a full geth node.
2. Unzip the downloaded folder and run the Ethereum-Wallet executable file.



3. Wait for the blockchain to fully sync, then follow the instructions on the screen and your first account will be created.
4. When you launch the Mist Ethereum wallet for the first time, you will see the account you created during the installation process. By default it will be named MAIN ACCOUNT (ETHERBASE).



5. Creating additional accounts is easy; just click on ADD ACCOUNT in the app's main screen and enter the required password.

Note: The Mist wallet is still in active development, so details of the steps outlined above may change with upgrades.

Creating a Multi-Signature Wallet in Mist

The Mist Ethereum wallet has an option to secure your wallet balance with a multisig wallet. The advantage of using a multisig wallet is that it requires authorization from more than one account to withdrawal larger amounts from your balance. Before you can create a multisig wallet, you'll need to create more than one account.

It's very easy to create account files in Mist. In the 'Accounts' section click 'Add Account'. Pick a strong yet easy-to-remember password (remember there is no password recovery option), confirm it, and your account is created. Create at least 2 accounts. Secondary accounts can be created on separate computers running Mist if you prefer (and theoretically make your multisig more secure doing it this way). You only need the public keys (your deposit addresses) of your secondary accounts when creating the multisig wallet (copy/paste them, do not ever type them by hand). Your primary account will be needed to create the multisig wallet contract, so it must be on the computer you are creating the multisig wallet on.

Now that you have your accounts setup, be safe and back them up (if your computer crashes, you will lose your balance if you do not have a backup). Click 'Backup' in the top menu. Choose the 'keystore' folder, opposite-click on it / choose 'copy' (do NOT choose 'cut', that would be very bad). Navigate to your desktop, opposite-click in a blank area and choose 'paste'. You may want to rename this new copy of the 'keystore' folder to something like 'Ethereum-keystore-backup-year-month-day' so you have quick recognition of it later. At this point you can then add the folder contents to a zip / rar file (and even password-protect the archive with another strong yet easy-to-remember password if backing up online), copy it to a USB Drive, burn it to a CD / DVD, or upload it to online storage (Dropbox / Google Drive / etc).

You now should add approximately no less than 0.02 ETH to your primary account (the account you will initiate creation of a multisig wallet with). This is required for the transaction fee when you create the multisig wallet contract. An additional 1 ETH (or more) is also needed, because Mist currently requires this to assure wallet contract transactions have enough 'gas' to execute properly...so no less than about 1.02 ETH total for starters.

You will be entering the full addresses of all the accounts you are attaching to this multisig wallet, when you create it. I recommend copying / pasting each address into a plain text editor (notepad / kedit / etc), after going to each account's details page in Mist, and choosing 'copy address' from the right-side column of buttons. Never type an address by hand, or you run a very high risk of typos and could lose your balance sending transactions to the wrong address.

We are now ready to create the multisig wallet. Under 'Wallet Contracts', select 'Add Wallet Contract'. Give it a name, select the primary account owner, and choose 'Multisignature Wallet Contract'. You will see something like this appear:

"This is a joint account controlled by X owners. You can send up to X ether per day. Any transaction over that daily limit requires the confirmation of X owners."

Set whatever amount of owners (accounts) you are attaching to this multisig wallet, whatever you want for a daily withdrawal limit (that only requires one account to withdrawal that amount), and how many owners (accounts) are required to approve any withdrawal amount over the daily limit.

Now add the addresses of the accounts that you copied / pasted into your text editor earlier, confirm all your settings are correct, and click 'Create' at the bottom. You will then need to enter your password to send the transaction. In the 'Wallet Contracts' section it should show your new wallet, and say 'creating'.

When wallet creation is complete, you should see your contract address on the screen. Select the entire address, copy / paste it into a new text file in your text editor, and save the text file to your desktop as 'Ethereum-Wallet-Address.txt', or whatever you want to name it.

Now all you need to do is backup the 'Ethereum-Wallet-Address.txt' file the same way you backed up your account files, and then you are ready to load your new multisig wallet with ETH using this address.

If you are restoring from backup, simply copy the files inside the 'Ethereum-keystore-backup' folder over into the 'keystore' folder mentioned in the first section of this walkthrough. FYI, you may need to create the 'keystore' folder if it's a brand new install of Mist on a machine it was never installed on before (the first time you create an account is when this folder is created). As for restoring a multisig wallet, instead of choosing 'Multisignature Wallet Contract' like we did before when creating it, we merely choose 'Import Wallet' instead.

Troubleshooting:

- Mist won't sync. One solution that works well is syncing your PC hardware clock with an NTP server so the time is exactly correct...then reboot.
- Mist starts after syncing, but is a blank white screen. Chances are you are running the "xorg" video drivers on a Linux-based OS (Ubuntu, Linux Mint, etc). Try installing the manufacturer's video driver instead.
- "Wrong password" notice. This seems to be a false notice on occasion on current Mist versions. Restart Mist and the problem should go away (if you indeed entered the correct password).

Using Eth

Every options related to key management available using geth can be used the same way in eth.

Below are "account" related options:

```
> eth account list  // List all keys available in wallet.
> eth account new   // Create a new key and add it to the wallet.
> eth account update [<uuid>|<address> , ... ]  // Decrypt and re-encrypt given keys.
> eth account import [<uuid>|<file>|<secret-hex>] // Import keys from given source and place in wa
```

Below are "wallet" related option:

```
> eth wallet import <file> //Import a presale wallet.
```

Note: the 'account import' option can only be used to import generic key file. the 'wallet import' option can only be used to import a presale wallet.

It is also possible to access keys management from the integrated console (using the built-in console or geth attach):

```
> web3.personal
{
      listAccounts: [],
      getListAccounts: function(callback),
      lockAccount: function(),
      newAccount: function(),
      unlockAccount: function()
}
```

Using EthKey (deprecated)

Ethkey is a CLI tool of the C++ implementation that allows you to interact with the Ethereum wallet. With it you can list, inspect, create, delete and modify keys and inspect, create and sign transactions.

We will assume you have not yet run a client such as eth or anything in the Aleth series of clients. If you have, you can skip this section. To create a wallet, run `ethkey` with the `createwallet` command:

```
> ethkey createwallet
```

Please enter a MASTER passphrase to protect your key store (make it strong!): You'll be asked for a "master" passphrase. This protects your privacy and acts as a default password for any keys. You'll need to confirm it by entering the same text again.

Note: Use a strong randomly generated password.

We can list the keys within the wallet simply by using the list command:

```
> ethkey list

No keys found.
```

We haven't yet created any keys, and it's telling us so! Let's create one.

To create a key, we use the `new` command. To use it we must pass a name - this is the name we'll give to this account in the wallet. Let's call it "test":

```
> ethkey new test
```

Enter a passphrase with which to secure this account (or nothing to use the master passphrase). It will prompt you to enter a passphrase to protect this key. If you just press enter, it'll use the default "master" passphrase. Typically this means you won't need to enter the passphrase for the key when you want to use the account (since it remembers the master passphrase). In general, you should try to use a different passphrase for each key since it prevents one compromised passphrase from giving access to other accounts. However, out of convenience you might decide that for low-security accounts to use the same passphrase.

Here, let's give it the incredibly imaginative passphrase of 123. (Never ever use simple passwords like this for anything else than ephemeral test accounts). Once you enter a passphrase, it'll ask you to confirm it by entering again. Enter 123 a second time. Because you gave it its own passphrase, it'll also ask you to provide a hint for this password which will be displayed to you whenever it asks you to enter it. The hint is stored in the wallet and is itself protected by the master passphrase. Enter the truly awful hint of 321 backwards.

```
> ethkey new test

Enter a passphrase with which to secure this account (or nothing to use the master passphrase):
Please confirm the passphrase by entering it again:
Enter a hint to help you remember this passphrase: 321 backwards
Created key 055dde03-47ff-dded-8950-0fe39b1fa101
  Name: test
```

```
  Password hint: 321 backwards
  ICAP: XE472EVKU3CGMJF2YQ0J9RO1Y90BC0LDFZ
  Raw hex: 0092e965928626f8880629cec353d3fd7ca5974f
```

All normal (aka direct) ICAP addresses begin with XE so you should be able to recognize them easily. Notice also that the key has another identifier after Created key. This is known as the UUID. This is a unique identifier for the key that has absolutely nothing to do with the account itself. Knowing it does nothing to help an attacker discover who you are on the network. It also happens to be the filename for the key, which you can find in either ~/.web3/keys (Mac or Linux) or $HOME/AppData/Web3/keys (Windows). Now let's make sure it worked properly by listing the keys in the wallet:

```
> ethkey list
055dde03-47ff-dded-8950-0fe39b1fa101 0092e965... XE472EVKU3CGMJF2YQ0J9RO1Y90BC0LDFZ test
```

It reports one key on each line (for a total of one key here). In this case our key is stored in a file 055dde... and has an ICAP address beginning XE472EVK.... Not especially easy things to remember so rather helpful that it has its proper name, test, too.

1.3.4 Importing your presale wallet

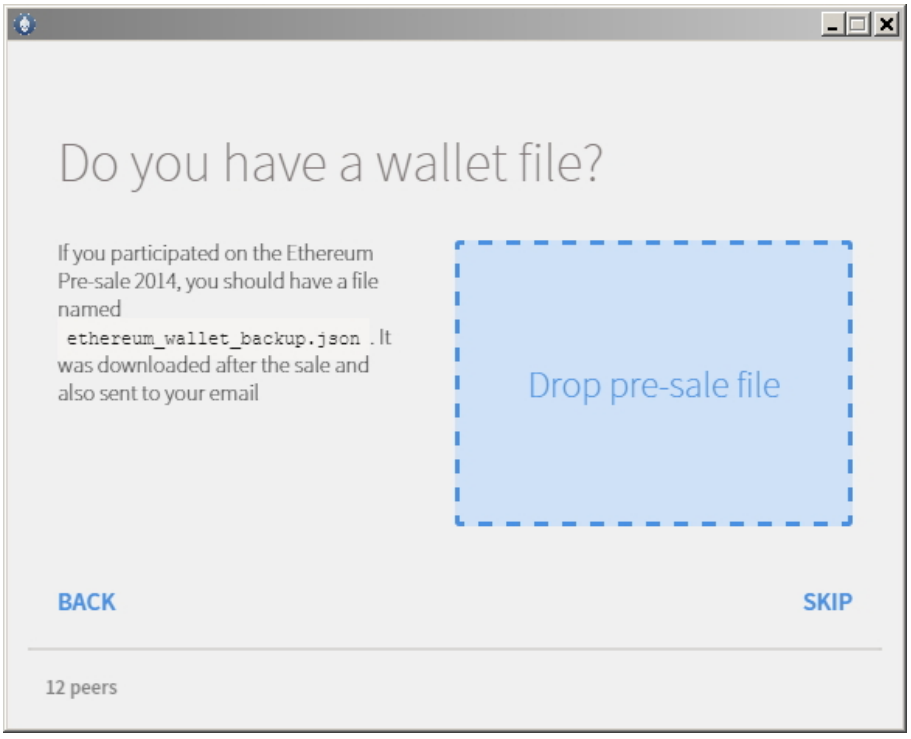
Using Mist Ethereum wallet

Importing your presale wallet using the GUI Mist Ethereum wallet is very easy. In fact, you will be asked if you want to import your presale wallet during the installation of the app.

> **Warning:** Mist wallet is beta software. Beware and use it at your own risk.

Instructions for installing the Mist Ethereum wallet are given in the section *Creating an account: Using Mist Ethereum wallet*.

Simply drag-and-drop your `.json` presale wallet file into the designated area and enter your password to import your presale account.



If you choose not to import your presale wallet during installation of the app, you can import it at any time by selecting the `Accounts` menu in the app's menu bar and then selecting `Import Pre-sale Accounts`.

Note: The Mist wallet is still in active development, so details of the steps outlined above may change with upgrades.

Using geth

If you have a standalone installation of geth, importing your presale wallet is accomplished by executing the following command in a terminal:

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

You will be prompted to enter your password.

1.3.5 Updating an account

You are able to upgrade your keyfile to the latest keyfile format and/or upgrade your keyfile password.

Using geth

You can update an existing account on the command line with the `update` subcommand with the account address or index as parameter. Remember that the account index reflects the order of creation (lexicographic order of keyfile names containing the creation time).

```
geth account update b0047c606f3af7392e073ed13253f8f4710b08b6
```

or

```
geth account update 2
```

For example:

```
$ 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
```

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 be used to migrate an account of a deprecated format to the newest format or change the password for an account.

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

```
geth --password <passwordfile> account update a94f5374fce5edbc8e2a8697c15331677e6ebf0bs
```

Since only one password can be given, only format update can be performed, changing your password is only possible interactively.

Note: account update has the side effect that the order of your accounts may change. After a successful update, all previous formats/versions of that same key will be removed!

1.3.6 Backup and restore accounts

Manual backup/restore

You must have an account's keyfile to be able to send any transaction from that account. Keyfiles are found in the keystore subdirectory of your Ethereum node's data directory. The default data directory locations are platform specific:

- Windows: `C:\Users\username\%appdata%\Roaming\Ethereum\keystore`
- Linux: `~/.ethereum/keystore`
- Mac: `~/Library/Ethereum/keystore`

To backup your keyfiles (accounts), copy either the individual keyfiles within the `keystore` subdirectory or copy the entire `keystore` folder.

To restore your keyfiles (accounts), copy the keyfiles back into the `keystore` subdirectory, where they were originally.

Importing an unencrypted private key

Importing an unencrypted private key is supported by `geth`

```
geth account import /path/to/<keyfile>
```

This command imports an unencrypted private key from the plain text file `<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.

An example where the data directory is specified. If the `--datadir` flag is not used, the new account will be created in the default data directory, i.e., the keyfile will be places in the `keyfiles` subdirectory of the data directory.

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

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

```
geth --password <passwordfile> account import <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 double-check and update the indexes used in your scripts.

1.4 Ether

1.4.1 What is ether?

Ether is the name of the currency used within Ethereum. It is used to pay for computation within the EVM. This is done indirectly by purchasing gas for ether as explained in gas.

Denominations

Ethereum has a metric system of denominations used as units of Ether. Each denomination has its own unique name (some bear the family name of seminal figures playing a role in evolution of computer science and cryptoeconomics). The smallest denomination aka *base unit* of Ether is called Wei. Below is a list of the named denominations and their value in Wei. Following a common (although somewhat ambiguous) pattern, Ether also designates a unit (of 1e18 or one quintillion Wei) of the currency. Note that the currency is not called Ethereum as many mistakenly think, nor is Ethereum a unit.

Unit	Wei Value	Wei
wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

1.4.2 Ether supply

- https://blog.ethereum.org/2014/04/10/the-issuance-model-in-ethereum/
- https://www.reddit.com/r/ethereum/comments/44zy88/clarification_on_ether_supply_and_cost_of_gas/
- https://www.reddit.com/r/ethereum/comments/45vj4g/question_about_scarcity_of_ethereum_and_its/
- https://www.reddit.com/r/ethtrader/comments/48yqg6/is_there_a_cap_like_with_btc_with_how_many_ether/

1.4.3 Getting ether

In order to obtain Ether, you need to either

- become an Ethereum miner (see Mining) or
- trade other currencies for Ether using centralised or trustless services
- use the user friendly Mist Ethereum GUI Wallet that as of Beta 6 introduced the ability to purchase ether using the http://shapeshift.io/ API.

Trustless services

Note that the Ethereum platform is special in that the smart contracts enable trustless services that obviate the need for trusted third parties in a currency exchange transaction, ie. disintermediate currency exchange businesses.

Such projects (alpha/prelaunch status at the time of writing) are:

- **BTCrelay**
  - More information (about ETH/BTC 2-way peg without modifying bitcoin code).
  - BTCrelay audit
- EtherEx decentralised exchange.

List of centralised exchange marketplaces

Exchange	Currencies
Poloniex	BTC
Kraken	BTC, USD, EUR, CAD, GBP
Gatecoin	BTC, EUR
Bitfinex	BTC, USD
Bittrex	BTC
Bluetrade	BTC, LTC, DOGE
HitBTC	BTC
Livecoin	BTC
Coinsquare	BTC
Bittylicious	GBP
BTER	CNY
Yunbi	CNY
Metaexchange	BTC

Centralised fixed rate exchanges

Exchange	Currencies
Shapeshift	BTC, LTC, DOGE, Other
Bity	BTC, USD, EUR, CHF

Trading and price analytics

- ETH markets exhaustive listing by volume on coinmarketcap
- Aggregating realtime stats of major ETH markets:
  - Tradeblock
  - EthereumWisdom
  - Cryptocompare
  - Coinmarketcap

1.4.4 Online wallets, paper wallets, and cold storage

Todo

This is here just a dumping ground of links and notes Please move this over in a listing form to ecosystem

Keep examples here, maybe explain paranoid practices, list dangers

- **Mist Ethereum Wallet**
  - Releases to download
  - Mist Ethereum Wallet developer preview - foundation blog post
  - How to easily set up the Ethereum Mist wallet! - Tutorial by Tommy Economics
- **Kryptokit Jaxx**
  - Jaxx main site
  - Mobile release
- **Etherwall**

  - Etherwall website
  - Etherwall source
- **MyEtherWallet**
  - MyEtherWallet website
  - MyEtherWallet source
  - Chrome extension
- **Cold storage**
  - Icebox by ConsenSys - Cold storage based on lightwallet with HD wallet library integrated.
  - Reddit discussion 1
  - How to setup a cold storage wallet
- **Hardware wallet**
  - reddit discussion 2
  - reddit discussion 3
- **Brain wallet**
  - brain wallets are not safe, do not use them. https://www.reddit.com/r/ethereum/comments/45y8m7/brain_wallets_ar
  - Extreme caution with brain wallets. Read the recent controversy: https://reddit.com/r/ethereum/comments/43fhb5/brainwallets vs http://blog.ether.camp/post/138376049438/why-brain-wallet-is-the-best
- **Misc**
  - Kraken Wallet Sweeper Tool - Pre-sale wallet import
  - Recommended ways to safely store ether
  - How to buy and store ether
  - A laymen's intro into brute forcing and why not to use brain wallets
  - Pyethsaletool
  - Account vs wallet

1.4.5 Sending ether

The Ethereum Wallet supports sending ether via a graphical interface.

Ether can also be transferred using the **geth console**.

```
> var sender = eth.accounts[0];
> var receiver = eth.accounts[1];
> var amount = web3.toWei(0.01, "ether")
> eth.sendTransaction({from:sender, to:receiver, value: amount})
```

For more information of Ether transfer transactions, see *Account Types, Gas, and Transactions*.

Ethereum is unique in the realm of cryptocurrencies in that ether has utility value as a cryptofuel, commonly referred to as "gas". Beyond transaction fees, gas is a central part of every network request and requires the sender to pay for the computing resources consumed. The gas cost is dynamically calculated, based on the volume and complexity of the request and multiplied by the current gas price. Its value as a cryptofuel has the effect of increasing the stability and long-term demand for ether and Ethereum as a whole. For more information, see *Account Types, Gas, and Transactions*.

1.4.6 Gas and ether

- https://www.reddit.com/r/ethereum/comments/271qdz/can_someone_explain_the_concept_of_gas_in_ethereum/
- https://www.reddit.com/r/ethereum/comments/3fnpr1/can_someone_possibly_explain_the_concept_of/
- https://www.reddit.com/r/ethereum/comments/49gol3/can_ether_be_used_as_a_currency_eli5_ether_gas/

Gas is supposed to be the constant cost of network resources/utilisation. You want the real cost of sending a transaction to always be the same, so you can't really expect Gas to be issued, currencies in general are volatile.

So instead, we issue Ether whose value is supposed to vary, but also implement a Gas Price in terms of Ether. If the price of Ether goes up, the Gas Price in terms of Ether should go down to keep the real cost of Gas the same.

Gas has multiple associated terms with it: Gas Prices, Gas Cost, Gas Limit, and Gas Fees. The principle behind Gas is to have a stable value for how much a transaction or computation costs on the Ethereum network.

- Gas Cost is a static value for how much a computation costs in terms of Gas, and the intent is that the real value of the Gas never changes, so this cost should always stay stable over time.
- Gas Price is how much Gas costs in terms of another currency or token like Ether. To stabilise the value of gas, the Gas Price is a floating value such that if the cost of tokens or currency fluctuates, the Gas Price changes to keep the same real value. The Gas Price is set by the equilibrium price of how much users are willing to spend, and how much processing nodes are willing to accept.
- Gas Limit is the maximum amount of Gas that can be used per block, it is considered the maximum computational load, transaction volume, or block size of a block, and miners can slowly change this value over time.
- Gas Fee is effectively the amount of Gas needed to be paid to run a particular transaction or program (called a contract). The Gas Fees of a block can be used to imply the computational load, transaction volume, or size of a block. The gas fees are paid to the miners (or bonded contractors in PoS).

1.5 The Ethereum network

Network info.

1.5.1 Connecting to the Network

This section

The Ethereum network

The basis for decentralised consensus is the peer-to-peer network of participating nodes which maintain and secure the blockchain. See *Mining*.

Ethereum network stats

EthStats.net is a dashboard of live statistics of the Ethereum network. This dashboard displays important information such as the current block, hash difficulty, gas price, and gas spending. The nodes shown on the page are only a selection of actual nodes on the network. Anyone is allowed to add their node to the EthStats dashboard. The Eth-Netstats README on Github describes how to connect.

EtherNodes.com displays current and historical data on node count and other information on both the Ethereum mainnet and Morden testnet.

Distribution of client implementations on the current live network - Realtime stats on EtherChain.

Public, private, and consortium blockchains

Most Ethereum projects today rely on Ethereum as a public blockchain, which grants access to a larger audience of users, network nodes, currency, and markets. However, there are often reasons to prefer a private blockchain or consortium blockchain (among a group of trusted participants). For example, a number of companies in verticals, like banking, are looking to Ethereum as a platform for their own private blockchains.

Below is an excerpt from the blog post On Public and Private Blockchains that explains the difference between the three types of blockchains based on permissioning:

- **Public blockchains**: a public blockchain is a blockchain that anyone in the world can read, anyone in the world can send transactions to and expect to see them included if they are valid, and anyone in the world can participate in the consensus process – the process for determining what blocks get added to the chain and what the current state is. As a substitute for centralized or quasi-centralized trust, public blockchains are secured by cryptoeconomics – the combination of economic incentives and cryptographic verification using mechanisms such as proof of work or proof of stake, following a general principle that the degree to which someone can have an influence in the consensus process is proportional to the quantity of economic resources that they can bring to bear. These blockchains are generally considered to be "fully decentralized".
- **Consortium blockchains**: a consortium blockchain is a blockchain where the consensus process is controlled by a pre-selected set of nodes; for example, one might imagine a consortium of 15 financial institutions, each of which operates a node and of which 10 must sign every block in order for the block to be valid. The right to read the blockchain may be public, or restricted to the participants, and there are also hybrid routes such as the root hashes of the blocks being public together with an API that allows members of the public to make a limited number of queries and get back cryptographic proofs of some parts of the blockchain state. These blockchains may be considered "partially decentralized".
- **Private blockchains**: a fully private blockchain is a blockchain where write permissions are kept centralized to one organization. Read permissions may be public or restricted to an arbitrary extent. Likely applications include database management, auditing, etc internal to a single company, and so public readability may not be necessary in many cases at all, though in other cases public auditability is desired.

While these private/consortium blockchains may not have any connection to the public blockchain, they still contribute to the overall Ethereum ecosystem by investing in Ethereum software development. Over time, this translates into software improvements, shared knowledge, and job opportunities.

How to connect

Geth continuously attempts to connect to other nodes on the network until it has peers. If you have UPnP enabled on your router or run Ethereum on an Internet-facing server, it will also accept connections from other nodes.

Geth finds peers through something called the *discovery protocol*. In the discovery protocol, nodes are gossipping with each other to find out about other nodes on the network. In order to get going initially, geth uses a set of bootstrap nodes whose endpoints are recorded in the source code.

Checking connectivity and ENODE IDs

To check how many peers the client is connected to in the interactive console, the `net` module has two attributes that give you info about the number of peers and whether you are a listening node.

```
> net.listening
true

> net.peerCount
4
```

To get more information about the connected peers, such as IP address and port number, supported protocols, use the `peers()` function of the `admin` object. `admin.peers()` returns the list of currently connected peers.

```
> admin.peers
[{
      ID: 'a4de274d3a159e10c2c9a68c326511236381b84c9ec52e72ad732eb0b2b1a2277938f78593cdbe734e6002
      Name: 'Geth/v0.9.14/linux/go1.4.2',
      Caps: 'eth/60',
      RemoteAddress: '5.9.150.40:30301',
      LocalAddress: '192.168.0.28:39219'
 }, {
      ID: 'a979fb575495b8d6db44f750317d0f4622bf4c2aa3365d6af7c284339968eef29b69ad0dce72a4d8db5ebb
      Name: 'Geth/v0.9.15/linux/go1.4.2',
      Caps: 'eth/60',
      RemoteAddress: '52.16.188.185:30303',
      LocalAddress: '192.168.0.28:50995'
 }, {
      ID: 'f6ba1f1d9241d48138136ccf5baa6c2c8b008435a1c2bd009ca52fb8edbbc991eba36376beaee9d45f16d5
      Name: 'pyethapp_dd52/v0.9.13/linux2/py2.7.9',
      Caps: 'eth/60, p2p/3',
      RemoteAddress: '144.76.62.101:30303',
      LocalAddress: '192.168.0.28:40454'
 }, {
  ID: 'f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae2e826f293c481b5325
  Name: '++eth/Zeppelin/Rascal/v0.9.14/Release/Darwin/clang/int',
  Caps: 'eth/60, shh/2',
  RemoteAddress: '129.16.191.64:30303',
  LocalAddress: '192.168.0.28:39705'
 } ]
```

To check the ports used by geth and also find your enode URI run:

```
> admin.nodeInfo
{
  Name: 'Geth/v0.9.14/darwin/go1.4.2',
  NodeUrl: 'enode://3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60f14998a3a98c0
  NodeID: '3414c01c19aa75a34f2dbd2f8d0898dc79d6b219ad77f8155abf1a287ce2ba60f14998a3a98c0cf14915ea
  IP: '::',
  DiscPort: 30303,
  TCPPort: 30303,
  Td: '2044952618444',
  ListenAddr: '[::]:30303'
}
```

Download the blockchain faster

When you start an Ethereum client, the Ethereum blockchain is automatically downloaded. The time it takes to download the Ethereum blockchain can vary based on client, client settings, connection speed, and number of peers available. Below are some options for more quickly obtaining the Ethereum blockchain.

Using geth

If you are using the geth client, there are some things you can do to speed up the time it takes to download the Ethereum blockchain. If you choose to use the `--fast` flag to perform an Ethereum fast sync, you will not retain past transaction data.

Note: You cannot use this flag after performing all or part of a normal sync operation, meaning you should not have any portion of the Ethereum blockchain downloaded before using this command. See this Ethereum Stack.Exchange answer for more information.

Below are some flags to use when you want to sync your client more quickly.

`--fast`

This flag enables fast syncing through state downloads rather than downloading the full block data. This will also reduce the size of your blockchain dramatically. NOTE: `--fast` can only be run if you are syncing your blockchain from scratch and only the first time you download the blockchain for security reasons. See this Reddit post for more information.

`--cache=1024`

Megabytes of memory allocated to internal caching (min 16MB / database forced). Default is 16MB, so increasing this to 256, 512, 1024 (1GB), or 2048 (2GB) depending on how much RAM your computer has should make a difference.

`--jitvm`

This flag enables the JIT VM.

Full example command with console:

```
geth --fast --cache=1024 --jitvm console
```

For more discussion on fast syncing and blockchain download times, see this Reddit post.

Exporting/Importing the blockchain

If you already have a full Ethereum node synced, you can export the blockchain data from the fully synced node and import it into your new node. You can accomplish this in geth by exporting your full node with the command `geth export filename` and importing the blockchain into your node using `geth import filename`. see this link

Static Nodes, Trusted Nodes, and Boot Nodes

Geth supports a feature called static nodes if you have certain peers you always want to connect to. Static nodes are re-connected on disconnects. You can configure permanent static nodes by putting something like the following into `<datadir>/static-nodes.json` (this should be the same folder that your `chaindata` and `keystore` folders are in)

```
[
      "enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae2e826f293c
      "enode://pubkey@ip:port"
]
```

You can also add static nodes at runtime via the Javascript console using `admin.addPeer()`

```
> admin.addPeer("enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54e8b995de102e0ceae
```

Common problems with connectivity

Sometimes you just can't get connected. The most common reasons are:

- Your local time might be incorrect. An accurate clock is required to participate in the Ethereum network. Check your OS for how to resync your clock (example `sudo ntpdate -s time.nist.gov`) because even 12 seconds too fast can lead to 0 peers.
- Some firewall configurations can prevent UDP traffic from flowing. You can use the static nodes feature or `admin.addPeer()` on the console to configure connections by hand.

To start geth without the discovery protocol, you can use the `--nodiscover` parameter. You only want this if you are running a test node or an experimental test network with fixed nodes.

1.5.2 Test Networks

Morden testnet

Morden is a public Ethereum alternative testnet. It is expected to continue throughout the Frontier and Homestead milestones of the software.

Usage

eth (C++ client) This is supported natively on 0.9.93 and above. Pass the `--morden` argument in when starting any of the clients. e.g.:

PyEthApp (Python client) PyEthApp supports the morden network from v1.0.5 onwards:

geth (Go client)

Details

All parameters are the same as the main Ethereum network except:

- Network Name: **Morden**
- Network Identity: 2
- genesis.json (given below);
- Initial Account Nonce (`IAN`) is 2^20 (instead of 0 in all previous networks).
  - All accounts in the state trie have nonce >= `IAN`.
  - Whenever an account is inserted into the state trie it is initialised with nonce = `IAN`.
- Genesis generic block hash: `0cd786a2425d16f152c658316c423e6ce1181e15c3295826d7c9904cba9ce303`
- Genesis generic state root: `f3f4696bbf3b3b07775128eb7a3763279a394e382130f27c21e70233e04946a9`

Morden's genesis.json

Getting Morden testnet ether

Two ways to obtain Morden testnet ether:

- Mine using your CPU/GPU, (see *Mining*).
- Use the Ethereum wei faucet.

1.5.3 Setting up a local private testnet

eth (C++ client)

It is possible to connect to or create a new network by using the –genesis and –config.

It is possible to use both –config and –genesis.

In that case, the genesis block description provided by –config will be overwritten by the –genesis option.

Note: <filename> contains a JSON description of the network:

- sealEngine (engine use to mine block)

  "Ethash" is the Ethereum proof of work engine (used by the live network).

  "NoProof" no proof of work is needed to mine a block.
- params (general network information like minGasLimit, minimumDifficulty, blockReward, networkID)
- genesis (genesis block description)
- accounts (setup an original state that contains accounts/contracts)

Here is a Config sample (used by the Olympic network):

Note: <filename> contains a JSON description of the genesis block:

The content is the same as the genesis field provided by the 'config' parameter:

geth (Go client)

You either pre-generate or mine your own Ether on a private testnet. It is a much more cost effective way of trying out Ethereum and you can avoid having to mine or find Morden test ether.

The things that are required to specify in a private chain are:

- Custom Genesis File
- Custom Data Directory
- Custom NetworkID
- (Recommended) Disable Node Discovery

The genesis file

The genesis block is the start of the blockchain - the first block, block 0, and the only block that does not point to a predecessor block. The protocol ensures that no other node will agree with your version of the blockchain unless they have the same genesis block, so you can make as many private testnet blockchains as you'd like!

`CustomGenesis.json`

```
{
    "nonce": "0x0000000000000042",     "timestamp": "0x0",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": "0x0",     "gasLimit": "0x8000000",     "difficulty": "0x400",
    "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "coinbase": "0x3333333333333333333333333333333333333333",     "alloc": {     }
}
```

Save a file called `CustomGenesis.json`. You will reference this when starting your geth node using the following flag:

```
--genesis /path/to/CustomGenesis.json
```

Command line parameters for private network

There are some command line options (also called "flags") that are necessary in order to make sure that your network is private. We already covered the genesis flag, but we need a few more. Note that all of the commands below are to be used in the geth Ethereum client.

```
--nodiscover
```

Use this to make sure that your node is not discoverable by people who do not manually add you. Otherwise, there is a chance that your node may be inadvertently added to a stranger's blockchain if they have the same genesis file and network id.

`--maxpeers 0`

Use maxpeers 0 if you do not want anyone else connecting to your test chain. Alternatively, you can adjust this number if you know exactly how many peers you want connecting to your node.

`--rpc`

This will enable RPC interface on your node. This is generally enabled by default in Geth.

`--rpcapi "db,eth,net,web3"`

This dictates what APIs that are allowed to be accessed over RPC. By default, Geth enables the web3 interface over RPC.

IMPORTANT: Please note that offering an API over the RPC/IPC interface will give everyone access to the API who can access this interface (e.g. dapp's). Be careful which API's you enable. By default geth enables all API's over the IPC interface and only the db,eth,net and web3 API's over the RPC interface.

`--rpcport "8080"`

Change 8000 to any port that is open on your network. The default for geth is 8080.

`--rpccorsdomain "http://chriseth.github.io/browser-solidity/"`

This dictates what URLs can connect to your node in order to perform RPC client tasks. Be very careful with this and type a specific URL rather than the wildcard (*) which would allow any URL to connect to your RPC instance.

`--datadir "/home/TestChain1"`

This is the data directory that your private chain data will be stored in (under the `nubits` . Choose a location that is separate from your public Ethereum chain folder.

`--port "30303"`

This is the "network listening port", which you will use to connect with other peers manually.

`--identity "TestnetMainNode"`

This will set up an identity for your node so it can be identified more easily in a list of peers. Here is an example of how these identities show up on the network.

Launching `geth`

After you have created your custom genesis block JSON file and created a directory for your blockchain data, type the following command into your console that has access to geth:

```
geth --identity "MyNodeName" --genesis /path/to/CustomGenesis.json --rpc --rpcport "8080" --rpcco
```

Note: Please change the flags to match your custom settings.

You will need to start your geth instance with your custom chain command every time you want to access your custom chain. If you just type "geth" in your console, it will not remember all of the flags you have set.

Pre-allocating ether to your account

A difficulty of "0x400" allows you to mine Ether very quickly on your private testnet chain. If you create your chain and start mining, you should have hundreds of Ether in a matter of minutes which is way more than enough to test transactions on your network. If you would still like to pre-allocate Ether to your account, you will need to:

1. Create a new Ethereum account after you create your private chain
2. Copy your new account address
3. Add the following command to your Custom_Genesis.json file:

```
"alloc":
{
        "<your account address e.g. 0x1fb891f92eb557f4d688463d0d7c560552263b5a>":
        { "balance": "200000000000000000000" }
}
```

Note: Replace `0x1fb891f92eb557f4d688463d0d7c560552263b5a` with your account address.

Save your genesis file and rerun your private chain command. Once geth is fully loaded, close it by .

We want to assign an address to the variable `primary` and check its balance.

Run the command `geth account list` in your terminal to see what account # your new address was assigned.

```
> geth account list
Account #0: {d1ade25ccd3d550a7eb532ac759cac7be09c2719}
Account #1: {da65665fc30803cb1fb7e6d86691e20b1826dee0}
Account #2: {e470b1a7d2c9c5c6f03bbaa8fa20db6d404a0c32}
Account #3: {f4dd5c3794f1fd0cdc0327a83aa472609c806e99}
```

Take note of which account # is the one that you pre-allocated Ether to. Alternatively, you can launch the console with `geth console` (keep the same parameters as when you launched `geth` first). Once the prompt appears, type

```
> eth.accounts
```

This will return the array of account addresses you possess.

```
> primary = eth.accounts[0]
```

Note: Replace `0` with your account's index. This console command should return your primary Ethereum address.

Type the following command:

```
> balance = web3.fromWei(eth.getBalance(primary), "ether");
```

This should return `7.5` indicating you have that much Ether in your account. The reason we had to put such a large number in the alloc section of your genesis file is because the "balance" field takes a number in wei which is the smallest denomination of the Ethereum currency Ether (see Ether).

- https://www.reddit.com/r/ethereum/comments/3kdnus/question_about_private_chain_mining_dont_upvote/
- http://adeduke.com/2015/08/how-to-create-a-private-ethereum-chain/

1.6 Mining

1.6.1 Introduction

The word mining originates in the context of the gold analogy for crypto currencies. Gold or precious metals are scarce, so are digital tokens, and the only way to increase the total volume is through mining. This is appropriate to the extent that in Ethereum too, the only mode of issuance post launch is via mining. Unlike these examples

however, mining is also the way to secure the network by creating, verifying, publishing and propagating blocks in the blockchain.

- Mining Ether = Securing the Network = Verifying Computation

What is mining?

Ethereum, like all blockchain technologies, uses an incentive-driven model of security. Consensus is based on choosing the block with the highest total difficulty. Miners produce blocks which the others check for validity. Among other well-formedness criteria, a block is only valid if it contains *proof of work* (PoW) of a given *difficulty*. Note that in the Ethereum Serenity milestone, this is likely going to be replaced by a (see *proof of stake model*).

The Ethereum blockchain is in many ways similar to the Bitcoin blockchain, although it does have some differences. The main difference between Ethereum and Bitcoin with regard to the blockchain architecture is that, unlike Bitcoin, Ethereum blocks contain a copy of both the transaction list and the most recent state (the root hash of the merkle patricia trie encoding the state to be more precise). Aside from that, two other values, the block number and the difficulty, are also stored in the block.

The proof of work algorithm used is called Ethash (a modified version of the Dagger-Hashimoto algorithm) and involves finding a *nonce* input to the algorithm so that the result is below a certain difficulty threshold. The point in PoW algorithms is that there is no better strategy to find such a nonce than enumerating the possibilities, while verification of a solution is trivial and cheap. Since outputs have a uniform distribution (as they are the result of the application of a hash function), we can guarantee that, on average, the time needed to find such a nonce depends on the difficulty threshold. This makes it possible to control the time of finding a new block just by manipulating the difficulty.

As dictated by the protocol, the difficulty dynamically adjusts in such a way that on average one block is produced by the entire network every 15 seconds. We say that the network produces a blockchain with a *15 second block time*. This "heartbeat" basically punctuates the synchronisation of system state and guarantees that maintaining a fork (to allow double spend) or rewriting history by malicious actors are impossible unless the attacker possesses more than half of the network mining power (this is the so called *51% attack*).

Any node participating in the network can be a miner and their expected revenue from mining will be directly proportional to their (relative) mining power or *hashrate*, i.e., the number of nonces tried per second normalised by the total hashrate of the network.

Ethash PoW is *memory hard*, making it *ASIC resistant*. Memory hardness is achieved with a proof of work algorithm that requires choosing subsets of a fixed resource dependent on the nonce and block header. This resource (a few gigabyte size data) is called a **DAG**. The DAG is totally different every 30000 blocks, a 125-hour window called *epoch* (roughly 5.2 days) and takes a while to generate. Since the DAG only depends on block height, it can be pregenerated but if its not, the client needs to wait until the end of this process to produce a block. If clients do not pregenerate and cache DAGs ahead of time the network may experience massive block delay on each epoch transition. Note that the DAG does not need to be generated for verifying the PoW essentially allowing for verification with both low CPU and small memory.

As a special case, when you start up your node from scratch, mining will only start once the DAG is built for the current epoch.

Mining rewards

The successful PoW miner of the winning block receives:

- a *static block reward* for the 'winning' block, consisting of exactly 5.0 Ether
- cost of the gas expended within the block – an amount of ether that depends on the current gas price
- an extra reward for including uncles as part of the block, in the form of an extra 1/32 per uncle included

All the gas consumed by the execution of all the transactions in the block submitted by the winning miner is paid by the senders of each transaction. The gas cost incurred is credited to the miner's account as part of the consensus protocol. Over time, it is expected these will dwarf the static block reward.

Uncles are stale blocks i.e. with parents that are ancestors (max 6 blocks back) of the including block. Valid uncles are rewarded in order to neutralise the effect of network lag on the dispersion of mining rewards, thereby increasing security (this is called the GHOST protocol). Uncles included in a block formed by the successful PoW miner receive 7/8 of the static block reward (=4.375 ether). A maximum of 2 uncles are allowed per block.

- Uncles ELI5 on reddit
- Forum thread explaining uncles

Mining success depends on the set block difficulty. Block difficulty dynamically adjusts each block in order to regulate the network hashing power to produce a 12 second blocktime. Your chances of finding a block therefore follows from your hashrate relative to difficulty.

Ethash DAG

Ethash uses a *DAG* (directed acyclic graph) for the proof of work algorithm, this is generated for each *epoch*, i.e., every 30000 blocks (125 hours, ca. 5.2 days). The DAG takes a long time to generate. If clients only generate it on demand, you may see a long wait at each epoch transition before the first block of the new epoch is found. However, the DAG only depends on the block number, so it can and should be calculated in advance to avoid long wait times at each epoch transition. Both `geth` and `ethminer` implement automatic DAG generation and maintains two DAGs at a time for smooth epoch transitions. Automatic DAG generation is turned on and off when mining is controlled from the console. It is also turned on by default if `geth` is launched with the `--mine` option. Note that clients share a DAG resource, so if you are running multiple instances of any client, make sure automatic dag generation is switched off in all but one instance.

To generate the DAG for an arbitrary epoch:

```
geth makedag <block number> <outputdir>
```

For instance `geth makedag 360000 ~/.ethash`. Note that ethash uses `~/.ethash` (Mac/Linux) or `~/AppData/Ethash` (Windows) for the DAG so that it can shared between different client implementations as well as multiple running instances.

1.6.2 The algorithm

Our algorithm, Ethash (previously known as Dagger-Hashimoto), is based around the provision of a large, transient, randomly generated dataset which forms a DAG (the Dagger-part), and attempting to solve a particular constraint on it, partly determined through a block's header-hash.

It is designed to hash a fast verifiability time within a slow CPU-only environment, yet provide vast speed-ups for mining when provided with a large amount of memory with high-bandwidth. The large memory requirements mean that large-scale miners get comparatively little super-linear benefit. The high bandwidth requirement means that a speed-up from piling on many super-fast processing units sharing the same memory gives little benefit over a single unit. This is important in that pool mining have no benefit for nodes doing verification, thus discourageing centralisation.

Communication between the external mining application and the Ethereum daemon for work provision and submission happens through the JSON-RPC API. Two RPC functions are provided; `eth_getWork` and `eth_submitWork`.

These are formally documented on the JSON-RPC API wiki article under miner.

In order to mine you need a fully synced Ethereum client that is enabled for mining and at least one ethereum account. This account is used to send the mining rewards to and is often referred to as *coinbase* or *etherbase*. Visit the "*Creating an account*" section of this guide to learn how to create an account.

> **Warning:** Ensure your blockchain is fully synchronised with the main chain before starting to mine, otherwise you will not be mining on the main chain.

1.6.3 CPU mining

You can use your computer's central processing unit (CPU) to mine ether. This is no longer profitable, since GPU miners are roughly two orders of magnitude more efficient. However, you can use CPU mining to mine on the Morden testnet or a private chain for the purposes of creating the ether you need to test contracts and transactions without spending your real ether on the live network.

Note: The testnet ether has no value other than using it for testing purposes (see *Test Networks*).

Using geth

When you start up your ethereum node with `geth` it is not mining by default. To start it in CPU mining mode, you use the `--mine` command line option. The `-minerthreads` parameter can be used to set the number parallel mining threads (defaulting to the total number of processor cores).

```
geth --mine --minerthreads=4
```

You can also start and stop CPU mining at runtime using the console. `miner.start` takes an optional parameter for the number of miner threads.

```
> miner.start(8)
true
> miner.stop()
true
```

Note that mining for real ether only makes sense if you are in sync with the network (since you mine on top of the consensus block). Therefore the eth blockchain downloader/synchroniser will delay mining until syncing is complete, and after that mining automatically starts unless you cancel your intention with `miner.stop()`.

In order to earn ether you must have your **etherbase** (or **coinbase**) address set. This etherbase defaults to your primary account. If you don't have an etherbase address, then `geth --mine` will not start up.

You can set your etherbase on the command line:

```
geth --etherbase 1 --mine  2>> geth.log // 1 is index: second account by creation order OR
geth --etherbase '0xa4d8e9cae4d04b093aac82e6cd355b6b963fb7ff' --mine 2>> geth.log
```

You can reset your etherbase on the console too:

```
miner.setEtherbase(eth.accounts[2])
```

Note that your etherbase does not need to be an address of a local account, just an existing one.

There is an option to add extra Data (32 bytes only) to your mined blocks. By convention this is interpreted as a unicode string, so you can set your short vanity tag.

```
miner.setExtra("ΞTHΞSPHΞΞ")
...
debug.printBlock(131805)
BLOCK(be465b020fdbedc4063756f0912b5a89bbb4735bd1d1df84363e05ade0195cb1): Size: 531.00 B TD: 643485
NoNonce: ee48752c3a0bfe3d85339451a5f3f411c21c8170353e450985e1faab0a9ac4cc
Header:
[
...
        Coinbase:           a4d8e9cae4d04b093aac82e6cd355b6b963fb7ff
        Number:             131805
        Extra:              ΞTHΞSPHΞΞ
...
}
```

You can check your hashrate with miner.hashrate, the result is in H/s (Hash operations per second).

```
> miner.hashrate
712000
```

After you successfully mined some blocks, you can check the ether balance of your etherbase account. Now assuming your etherbase is a local account:

```
> eth.getBalance(eth.coinbase).toNumber();
'34698870000000'
```

In order to spend your earnings on gas to transact, you will need to have this account unlocked.

```
> personal.unlockAccount(eth.coinbase)
Password
true
```

You can check which blocks are mined by a particular miner (address) with the following code snippet on the console:

```
function minedBlocks(lastn, addr) {
  addrs = [];
  if (!addr) {
    addr = eth.coinbase
  }
  limit = eth.blockNumber - lastn
  for (i = eth.blockNumber; i >= limit; i--) {
    if (eth.getBlock(i).miner == addr) {
      addrs.push(i)
    }
  }
  return addrs
}
// scans the last 1000 blocks and returns the blocknumbers of blocks mined by your coinbase
// (more precisely blocks the mining reward for which is sent to your coinbase).
minedBlocks(1000, eth.coinbase);
//[352708, 352655, 352559]
```

Note that it will happen often that you find a block yet it never makes it to the canonical chain. This means when you locally include your mined block, the current state will show the mining reward credited to your account, however, after a while, the better chain is discovered and we switch to a chain in which your block is not included and therefore no mining reward is credited. Therefore it is quite possible that as a miner monitoring their coinbase balance will find that it may fluctuate quite a bit.

1.6.4 GPU mining

Hardware

The algorithm is memory hard and in order to fit the DAG into memory, it needs 1-2GB of RAM on each GPU. If you get `Error GPU mining. GPU memory fragmentation?` you do not have enough memory. The GPU miner is implemented in OpenCL, so AMD GPUs will be 'faster' than same-category NVIDIA GPUs. ASICs and FPGAs are relatively inefficient and therefore discouraged. To get openCL for your chipset and platform, try:

- AMD SDK openCL
- NVIDIA CUDA openCL

Ubuntu Linux set-up

For this quick guide, you'll need Ubuntu 14.04 or 15.04 and the fglrx graphics drivers. You can use NVidia drivers and other platforms, too, but you'll have to find your own way to getting a working OpenCL install with them, such as Genoil's ethminer fork.

If you're on 15.04, Go to "Software and Updates > Additional Drivers" and set it to "Using video drivers for the AMD graphics accelerator from fglrx".

If you're on 14.04, go to "Software and Updates > Additional Drivers" and set it to "Using video drivers for the AMD graphics accelerator from fglrx". Unfortunately, for some of you this will not work due to a known bug in Ubuntu 14.04.02 preventing you from switching to the proprietary graphics drivers required to GPU mine.

So, if you encounter this bug, and before you do anything else, go to "Software and updates > Updates" and select "Pre-released updates trusty proposed". Then, go back to "Software and Updates > Additional Drivers" and set it to "Using video drivers for the AMD graphics accelerator from fglrx"). After rebooting, it's well worth having a check that the drivers have now indeed been installed correctly (For example by going to "Additional Drivers" again).

Whatever you do, if you are on 14.04.02 do not alter the drivers or the drivers configuration once set. For example, the usage of aticonfig –initial (especially with the -f, –force option) can 'break' your setup. If you accidentally alter their configuration, you'll need to de-install the drivers, reboot, reinstall the drivers and reboot.

Mac set-up

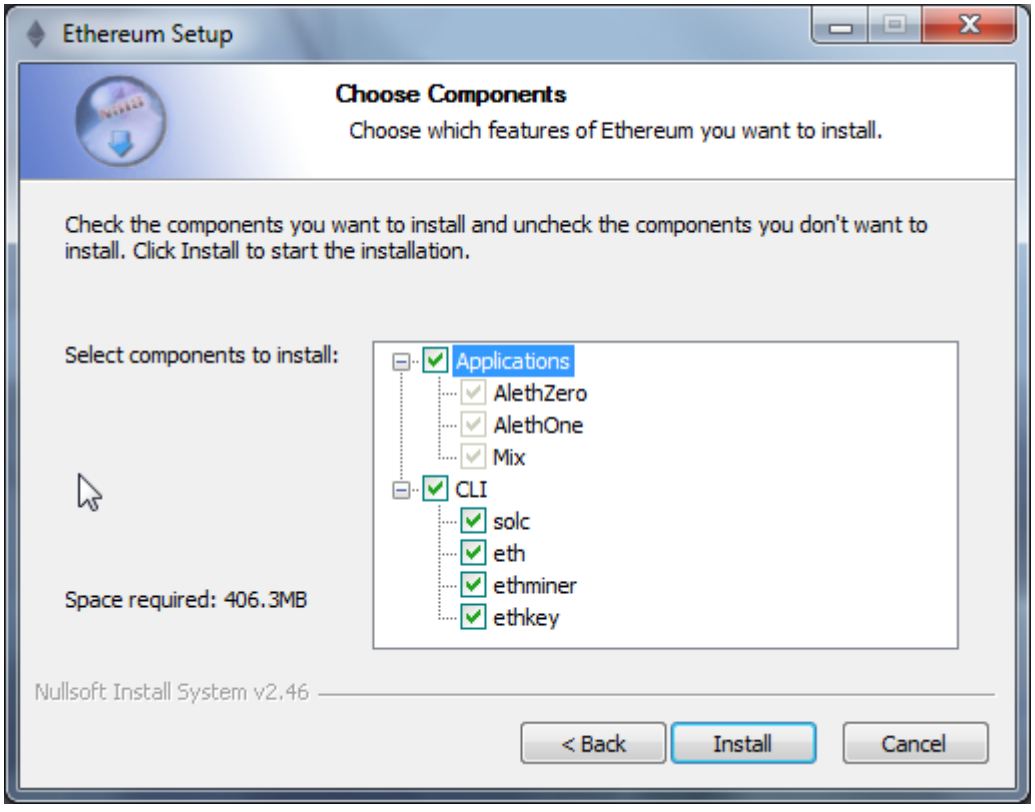
```
wget http://developer.download.nvidia.com/compute/cuda/7_0/Prod/local_installers/cuda_7.0.29_mac.p
sudo installer -pkg ~/Desktop/cuda_7.0.29_mac.pkg -target /
brew update
brew tap ethereum/ethereum
brew reinstall cpp-ethereum --with-gpu-mining --devel --headless --build-from-source
```

You check your cooling status:

```
aticonfig --adapter=0 --od-gettemperature
```

Windows set-up

Download the latest Eth++ installation and choose ethminer at the "Choose Components" screen of the installation screen.



Using ethminer with geth

```
geth account new // Set-up ethereum account if you do not have one
geth --rpc --rpccorsdomain localhost 2>> geth.log &
ethminer -G  // -G for GPU, -M for benchmark
tail -f geth.log
```

`ethminer` communicates with geth on port 8545 (the default RPC port in geth). You can change this by giving the `--rpcport` option to `geth`. Ethminer will find geth on any port. Note that you need to set the CORS header with `--rpccorsdomain localhost`. You can also set port on `ethminer` with `-F http://127.0.0.1:3301`. Setting the ports is necessary if you want several instances mining on the same computer, although this is somewhat pointless. If you are testing on a private chain, we recommend you use CPU mining instead.

Note: You do **not** need to give `geth` the `--mine` option or start the miner in the console unless you want to do CPU mining on TOP of GPU mining.

If the default for `ethminer` does not work try to specify the OpenCL device with: `--opencl-device X` where X is {0, 1, 2,...}. When running `ethminer` with `-M` (benchmark), you should see something like:

```
Benchmarking on platform: { "platform": "NVIDIA CUDA", "device": "GeForce GTX 750 Ti", "version":


Benchmarking on platform: { "platform": "Apple", "device": "Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.7
```

To debug `geth`:

```
geth  --rpccorsdomain "localhost" --verbosity 6 2>> geth.log
```

To debug the miner:

```
make -DCMAKE_BUILD_TYPE=Debug -DETHASHCL=1 -DGUI=0
gdb --args ethminer -G -M
```

Note: hashrate info is not available in `geth` when GPU mining.

Check your hashrate with `ethminer`, `miner.hashrate` will always report 0.

Using ethminer with eth

Mining on a single GPU

In order to mine on a single GPU all that needs to be done is to run eth with the following arguments:

```
eth -v 1 -a 0xcadb3223d4eebcaa7b40ec5722967ced01cfc8f2 --client-name "OPTIONALNAMEHERE" -x 50 -m
```

- `-v 1` Set verbosity to 1. Let's not get spammed by messages.
- `-a YOURWALLETADDRESS` Set the coinbase, where the mining rewards will go to. The above address is just an example. This argument is really important, make sure to not make a mistake in your wallet address or you will receive no ether payout.
- `--client-name "OPTIONAL"` Set an optional client name to identify you on the network
- `-x 50` Request a high amount of peers. Helps with finding peers in the beginning.
- `-m on` Actually launch with mining on.
- `-G` set GPU mining on.

While the client is running you can interact with it using either geth attach‘ or [ethconsole](https://github.com/ethereum/ethereum-console).

Mining on a multiple GPUs

Mining with multiple GPUs and eth is very similar to mining with geth and multiple GPUs. Ensure that an eth node is running with your coinbase address properly set:

```
eth -v 1 -a 0xcadb3223d4eebcaa7b40ec5722967ced01cfc8f2 --client-name "OPTIONALNAMEHERE" -x 50 -j
```

Notice that we also added the -j argument so that the client can have the JSON-RPC server enabled to communicate with the ethminer instances. Additionally we removed the mining related arguments since ethminer will now do the mining for us. For each of your GPUs execute a different ethminer instance:

```
ethminer --no-precompute -G --opencl-device X
```

Where X is the index number corresponding to the openCL device you want the ethminer to use {0, 1, 2,...}. In order to easily get a list of OpenCL devices you can execute `ethminer --list-devices` which will provide a list of all devices OpenCL can detect, with also some additional information per device.

Below is a sample output:

```
[0] GeForce GTX 770
    CL_DEVICE_TYPE: GPU
    CL_DEVICE_GLOBAL_MEM_SIZE: 4286345216
    CL_DEVICE_MAX_MEM_ALLOC_SIZE: 1071586304
    CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024
```

Finally the `--no-precompute` argument requests that the ethminers don't create the DAG of the next epoch ahead of time. Although this is not recommended since you'll have a mining interruption every time when there's an epoch transition.

Benchmarking

Mining power tends to scale with memory bandwidth. Our implementation is written in OpenCL, which is typically supported better by AMD GPUs over NVidia. Empirical evidence confirms that AMD GPUs offer a better mining performance in terms of price than their NVidia counterparts.

To benchmark a single-device setup you can use ethminer in benchmarking mode through the -M option:

```
ethminer -G -M
```

If you have many devices and you'll like to benchmark each individually, you can use the –opencl-device option similarly to the previous section:

```
ethminer -G -M --opencl-device X
```

Use ethminer `--list-devices` to list possible numbers to substitute for the X {0, 1, 2,...}.

To start mining on Windows, first download the geth windows binary.

- Unzip Geth (right-click and select unpack) and launch Command Prompt. Use *cd* to navigate to the location of the Geth data folder. (e.g. `cd /` to go to the `C:` drive)
- Start geth by typing `geth --rpc`.

As soon as you enter this, the Ethereum blockchain will start downloading. Sometimes your firewall may block the synchronisation process (it will prompt you when doing so). If this is the case, click "Allow access".

- First download and install ethminer, the C++ mining software (your firewall or Windows itself may act up, allow access)
- Open up another Command Prompt (leave the first one running!), change directory by typing `cd /Program\ Files/Ethereum(++)/release`

- Now make sure *geth* has finished syncing the blockchain. If it is not syncing any longer, you can start the mining process by typing `ethminer -G` at the command prompt

At this point some problems may appear. If you get an error, you can abort the miner by pressing `Ctrl+C`. If the error says "Insufficient Memory", your GPU does not have enough memory to mine ether.

1.6.5 Pool mining

Mining pools are cooperatives that aim to smooth out expected revenue by pooling the mining power of participating miners. In return, they usually charge you 0-5% of your mining rewards. The mining pool submits blocks with proof of work from a central account and redistributes the reward to participants in proportion to their contributed mining power.

Warning: Most mining pools involve third party, central components which means they are not trustless. In other words, pool operators can run away with your earnings. Act with caution. There are a number of trustless, decentralised pools with open source codebase.

Warning: Mining pools only outsource proof of work calculation, they do not validate blocks or run the VM to check state transitions brought about by executing the transactions. This effectively make pools behave like single nodes in terms of security, so their growth poses a centralisation risk of a 51% attack. Make sure you follow the network capacity distribution and do not allow pools to grow too large.

Mining pools

- coinotron
- nanopool
- ethpool - Predictable solo mining, unconventional payout scheme, affiliated with etherchain.org.
- supernova
- coinmine.pl
- eth.pp.ua
- talkether - Unconventional payout scheme, partially decentralized
- weipool
- ethereumpool
- pooleum
- alphapool
- cryptopool
- unitedminers
- dwarfpool - Try to avoid this (currently over 50% of the network)
- laintimes - Discontinued

1.6.6 Mining resources

- Top miners of last 24h on etherchain
- pool hashrate distribution for august 2015
- Unmaintained list of pools on Forum
- Mining profitability calculator on cryptocompare

- Mining profitability calculator on cryptowizzard
- Mining profitability calculator on etherscan
- Mining profitability calculator on In The Ether
- Mining difficulty chart on etherscan

POS vs POW

- https://www.reddit.com/r/ethereum/comments/38db1z/eli5_the_difference_between_pos_and_pow/
- https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity/
- https://www.reddit.com/r/ethereum/comments/42o8oy/can_someone_explain_the_switch_to_pos_how_and_when/

1.7 Contracts and Transactions

1.7.1 Account Types, Gas, and Transactions

EOA vs contract accounts

There are two types of accounts in Ethereum

- Externally Owned Accounts
- Contracts Accounts

This distinction might be eliminated in Serenity.

Externally owned accounts (EOAs)

An externally controlled account

- has an Ether balance,
- can send transactions (ether transfer or trigger contract code),
- is controlled by private keys,
- has no associated code.

Contract accounts

A contract

- has an Ether balance,
- has associated code,
- code execution is triggered by transactions or messages (calls) received from other contracts.
- when executed - perform operations of arbitrary complexity (Turing completeness) - manipulate its own persistent storage, i.e., can have its own permanent state - can call other contracts

All action on the Ethereum block chain is set in motion by transactions fired from externally owned accounts. Every time a contract account receives a transaction, its code is executed as instructed by the input parameters sent as part of the transaction. The contract code is executed by the Ethereum Virtual Machine on each node participating in the network as part of their verification of new blocks.

This execution needs to be completely deterministic, its only context is the position of the block on the blockchain and all data available. The blocks on the blockchain represent units of time, the blockchain itself is a temporal

dimension and represents the entire history of states at the discrete time points designated by the blocks on the chain.

All Ether balances and values are denominated in units of wei: 1 Ether is 1e18 wei.

Note: "Contracts" in Ethereum should not be seen as something that should be "fulfilled" or "complied with"; rather, they are more like "autonomous agents" that live inside of the Ethereum execution environment, always executing a specific piece of code when "poked" by a message or transaction, and having direct control over their own ether balance and their own key/value store to store their permanent state.

What is a transaction?

The term "transaction" is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account to another account on the blockchain.

Transactions contain:

- the recipient of the message,
- a signature identifying the sender and proving their intention to send the message via the blockchain to the recipient,
- `VALUE` field - The amount of wei to transfer from the sender to the recipient,
- an optional data field, which can contain the message sent to a contract,
- a `STARTGAS` value, representing the maximum number of computational steps the transaction execution is allowed to take,
- a `GASPRICE` value, representing the fee the sender is willing to pay for gas. One unit of gas corresponds to the execution of one atomic instruction, i.e., a computational step.

What is a message?

Contracts have the ability to send "messages" to other contracts. Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment. They can be conceived of as function calls.

A message contains:

- the sender of the message (implicit).
- the recipient of the message
- `VALUE` field - The amount of wei to transfer alongside the message to the contract address,
- an optional data field, that is the actual input data to the contract
- a `STARTGAS` value, which limits the maximum amount of gas the code execution triggered by the message can incur.

Essentially, a message is like a transaction, except it is produced by a contract and not an external actor. A message is produced when a contract currently executing code executes the `CALL` or `DELEGATECALL` opcodes, which produces and executes a message. Like a transaction, a message leads to the recipient account running its code. Thus, contracts can have relationships with other contracts in exactly the same way that external actors can.

What is gas?

Ethereum implements an execution environment on the blockchain called the Ethereum Virtual Machine (EVM). Every node participating in the network runs the EVM as part of the block verification protocol. They go through the transactions listed in the block they are verifying and run the code as triggered by the transaction within the EVM. Each and every full node in the network does the same calculations and stores the same values. Clearly Ethereum is not about optimising efficiency of computation. Its parallel processing is redundantly parallel. This

is to offer an efficient way to reach consensus on the system state without needing trusted third parties, oracles or violence monopolies. But importantly they are not there for optimal computation. The fact that contract executions are redundantly replicated across nodes, naturally makes them expensive, which generally creates an incentive not to use the blockchain for computation that can be done offchain.

When you are running a decentralized application (dapp), it interacts with the blockchain to read and modify its state, but dapps will typically only put the business logic and state that are crucial for consensus on the blockchain.

When a contract is executed as a result of being triggered by a message or transaction, every instruction is executed on every node of the network. This has a cost: for every executed operation there is a specified cost, expressed in a number of gas units.

Gas is the name for the execution fee that senders of transactions need to pay for every operation made on an Ethereum blockchain. The name gas is inspired by the view that this fee acts as cryptofuel, driving the motion of smart contracts. Gas is purchased for ether from the miners that execute the code. Gas and ether are decoupled deliberately since units of gas align with computation units having a natural cost, while the price of ether generally fluctuates as a result of market forces. The two are mediated by a free market: the price of gas is actually decided by the miners, who can refuse to process a transaction with a lower gas price than their minimum limit. To get gas you simply need to add ether to your account. The Ethereum clients automatically purchase gas for your Ether in the amount you specify as your maximum expenditure for the transaction.

The Ethereum protocol charges a fee per computational step that is executed in a contract or transaction to prevent deliberate attacks and abuse on the Ethereum network. Every transaction is required to include a gas limit and a fee that it is willing to pay per gas. Miners have the choice of including the transaction and collecting the fee or not. If the total amount of gas used by the computational steps spawned by the transaction, including the original message and any sub-messages that may be triggered, is less than or equal to the gas limit, then the transaction is processed. If the total gas exceeds the gas limit, then all changes are reverted, except that the transaction is still valid and the fee can still be collected by the miner. All excess gas not used by the transaction execution is reimbursed to the sender as Ether. You do not need to worry about overspending, since you are only charged for the gas you consume. This means that it is useful as well as safe to send transactions with a gas limit well above the estimates.

Estimating transaction costs

The total ether cost of a transaction is based on 2 factors:

`gasUsed` is the total gas that is consumed by the transaction

`gasPrice` price (in ether) of one unit of gas specified in the transaction

Total cost = gasUsed * gasPrice

gasUsed

Each operation in the EVM was assigned a number of how much gas it consumes. `gasUsed` is the sum of all the gas for all the operations executed. There is a spreadsheet which offers a glimpse to some of the analysis behind this.

For estimating `gasUsed`, there is an estimateGas API that can be used but has some caveats.

gasPrice

A user constructs and signs a transaction, and each user may specify whatever `gasPrice` they desire, which can be zero. However, the Ethereum clients launched at Frontier had a default gasPrice of 0.05e12 wei. As miners optimize for their revenue, if most transactions are being submitted with a gasPrice of 0.05e12 wei, it would be difficult to convince a miner to accept a transaction that specified a lower, or zero, gasPrice.

Example transaction cost

Let's take a contract that just adds 2 numbers. The EVM OPCODE `ADD` consumes 3 gas.

The approximate cost, using the default gas price (as of January 2016), would be:

3 * 0.05e12 = 1.5e11 wei

Since 1 Ether is 1e18 wei, the total cost would be 0.00000015 Ether.

This is a simplification since it ignores some costs, such as the cost of passing the 2 numbers to contract, before they can even be added.

- question
- gas fees
- gas cost calculator
- Ethereum Gas Prices

Operation Name	Gas Cost	Remark
step	1	default amount per execution cycle
stop	0	free
suicide	0	free
sha3	20	
sload	20	get from permanent storage
sstore	100	put into permanent storage
balance	20	
create	100	contract creation
call	20	initiating a read-only call
memory	1	every additional word when expanding memory
txdata	5	every byte of data or code for a transaction
transaction	500	base fee transaction
contract creation	53000	changed in homestead from 21000

Account interactions example - betting contract

As previously mentioned, there are two types of accounts:

- **Externally owned account (EOAs)**: an account controlled by a private key, and if you own the private key associated with the EOA you have the ability to send ether and messages from it.
- **Contract**: an account that has its own code, and is controlled by code.

By default, the Ethereum execution environment is lifeless; nothing happens and the state of every account remains the same. However, any user can trigger an action by sending a transaction from an externally owned account, setting Ethereum's wheels in motion. If the destination of the transaction is another EOA, then the transaction may transfer some ether but otherwise does nothing. However, if the destination is a contract, then the contract in turn activates, and automatically runs its code.

The code has the ability to read/write to its own internal storage (a database mapping 32-byte keys to 32-byte values), read the storage of the received message, and send messages to other contracts, triggering their execution in turn. Once execution stops, and all sub-executions triggered by a message sent by a contract stop (this all happens in a deterministic and synchronous order, ie. a sub-call completes fully before the parent call goes any further), the execution environment halts once again, until woken by the next transaction.

Contracts generally serve four purposes:

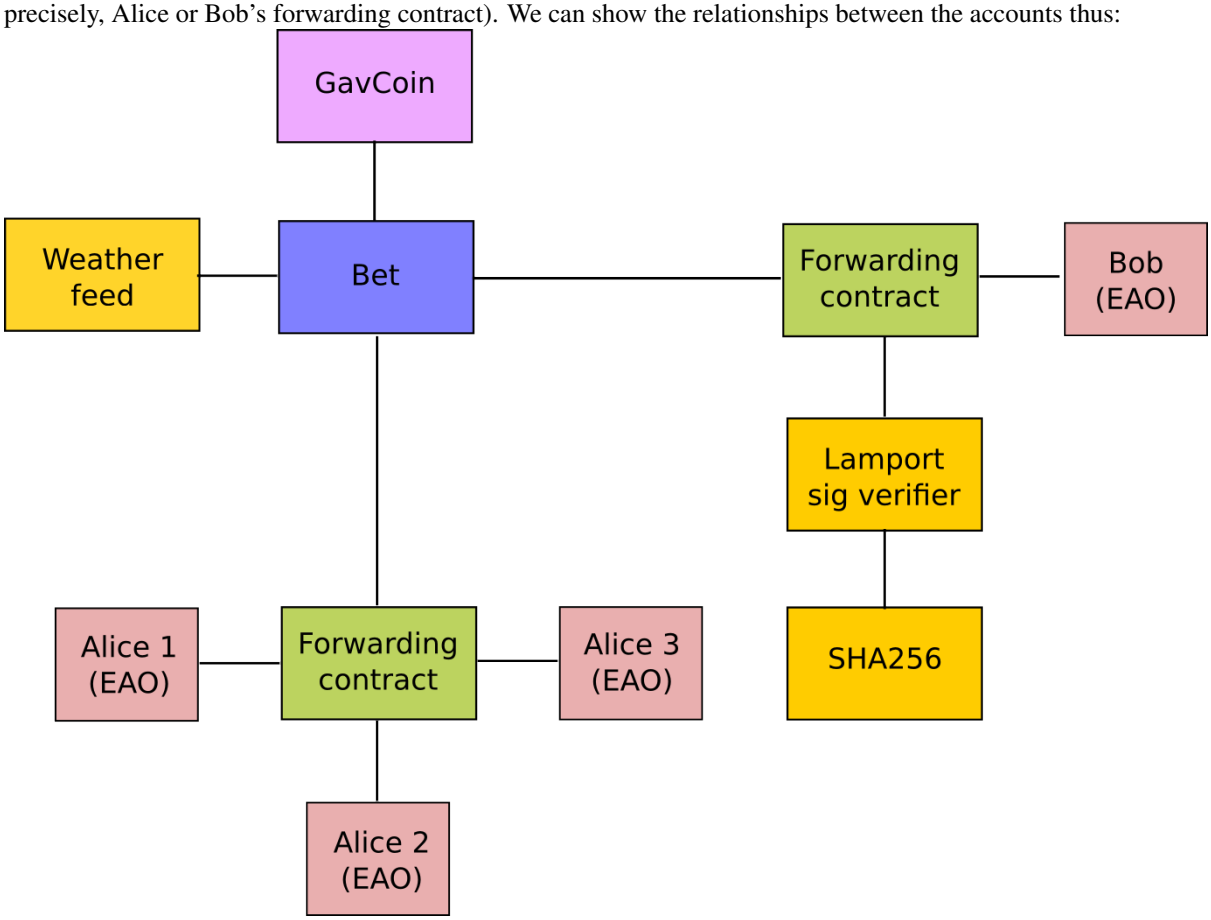
- Maintain a data store representing something which is useful to either other contracts or to the outside world; one example of this is a contract that simulates a currency, and another is a contract that records membership in a particular organization.

- Serve as a sort of externally-owned account with a more complicated access policy; this is called a "forwarding contract" and typically involves simply resending incoming messages to some desired destination only if certain conditions are met; for example, one can have a forwarding contract that waits until two out of a given three private keys have confirmed a particular message before resending it (ie. multisig). More complex forwarding contracts have different conditions based on the nature of the message sent. The simplest use case for this functionality is a withdrawal limit that is overrideable via some more complicated access procedure. A wallet contract is a good example of this.
- Manage an ongoing contract or relationship between multiple users. Examples of this include a financial contract, an escrow with some particular set of mediators, or some kind of insurance. One can also have an open contract that one party leaves open for any other party to engage with at any time; one example of this is a contract that automatically pays a bounty to whoever submits a valid solution to some mathematical problem, or proves that it is providing some computational resource.
- Provide functions to other contracts, essentially serving as a software library.

Contracts interact with each other through an activity that is alternately called either "calling" or "sending messages". A "message" is an object containing some quantity of ether, a byte-array of data of any size, the addresses of a sender and a recipient. When a contract receives a message, it has the option of returning some data, which the original sender of the message can then immediately use. In this way, sending a message is exactly like calling a function.

Because contracts can play such different roles, we expect that contracts will be interacting with each other. As an example, consider a situation where Alice and Bob are betting 100 GavCoin that the temperature in San Francisco will not exceed 35ºC at any point in the next year. However, Alice is very security-conscious, and as her primary account uses a forwarding contract which only sends messages with the approval of two out of three private keys. Bob is paranoid about quantum cryptography, so he uses a forwarding contract which passes along only messages that have been signed with Lamport signatures alongside traditional ECDSA (but because he's old fashioned, he prefers to use a version of Lamport sigs based on SHA256, which is not supported in Ethereum directly).

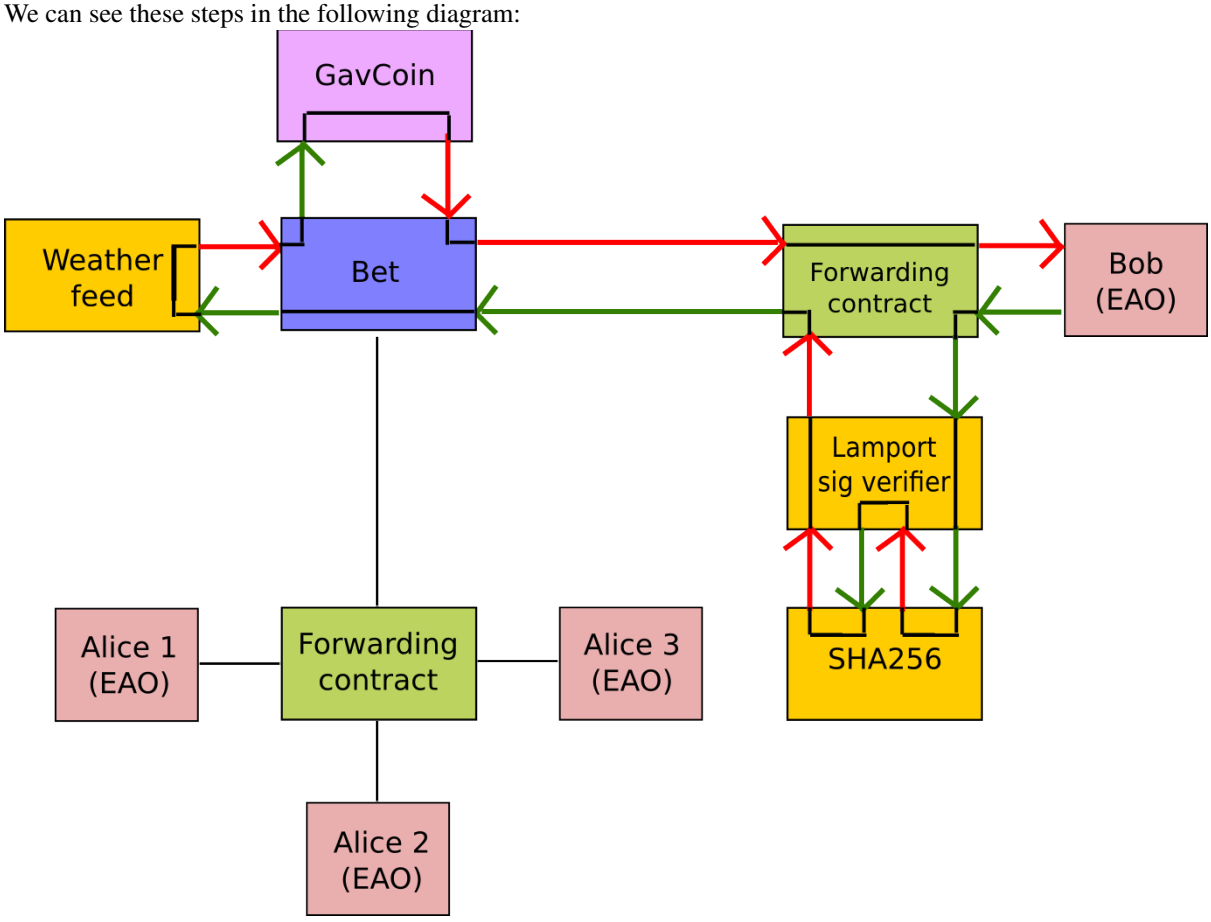
The betting contract itself needs to fetch data about the San Francisco weather from some contract, and it also needs to talk to the GavCoin contract when it wants to actually send the GavCoin to either Alice or Bob (or, more precisely, Alice or Bob's forwarding contract). We can show the relationships between the accounts thus:



When Bob wants to finalize the bet, the following steps happen:

1. A transaction is sent, triggering a message from Bob's EOA to his forwarding contract.
2. Bob's forwarding contract sends the hash of the message and the Lamport signature to a contract which functions as a Lamport signature verification library.
3. The Lamport signature verification library sees that Bob wants a SHA256-based Lamport sig, so it calls the SHA256 library many times as needed to verify the signature.
4. Once the Lamport signature verification library returns 1, signifying that the signature has been verified, it sends a message to the contract representing the bet.
5. The bet contract checks the contract providing the San Francisco temperature to see what the temperature is.
6. The bet contract sees that the response to the messages shows that the temperature is above 35ºC, so it sends a message to the GavCoin contract to move the GavCoin from its account to Bob's forwarding contract.

Note that the GavCoin is all "stored" as entries in the GavCoin contract's database; the word "account" in the context of step 6 simply means that there is a data entry in the GavCoin contract storage with a key for the bet contract's address and a value for its balance. After receiving this message, the GavCoin contract decreases this value by some amount and increases the value in the entry corresponding to Bob's forwarding contract's address. We can see these steps in the following diagram:



Signing transactions offline

[Maybe add this to the FAQ and point to the ethkey section of turboethereum guide?]

- Resilience Raw Transaction Broadcaster

1.7.2 Contracts

What is a contract?

A contract is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. Contract accounts are able to pass messages between themselves as well as doing practically Turing complete computation. Contracts live on the blockchain in a Ethereum-specific binary format called Ethereum Virtual Machine (EVM) bytecode.

Contracts are typically written in some high level language such as Solidity and then compiled into bytecode to be uploaded on the blockchain.

See also:

Other languages also exist, notably Serpent and LLL, which are described further in the *Ethereum high level languages* section of this documentation.

Dapp development resources lists the integrated development environments, developer tools that help you develop in these languages, offering testing, and deployment support among other features.

Ethereum high level languages

Contracts live on the blockchain in an Ethereum-specific binary format (EVM bytecode) that is executed by the Ethereum Virtual Machine (EVM). However, contracts are typically written in a higher level language and then compiled using the EVM compiler into byte code to be deployed to the blockchain.

Below are the different high level languages developers can use to write smart contracts for Ethereum.

Solidity

Solidity is a language similar to JavaScript which allows you to develop contracts and compile to EVM bytecode. It is currently the flagship language of Ethereum and the most popular.

- Solidity Documentation - Solidity is the flagship Ethereum high level language that is used to write contracts.
- Solidity online realtime compiler
- Standardized Contract APIs
- Useful Đapp Patterns - Code snippets which are useful for Đapp development.

Serpent

Serpent is a language similar to Python which can be used to develop contracts and compile to EVM bytecode. It is intended to be maximally clean and simple, combining many of the efficiency benefits of a low-level language with ease-of-use in programming style, and at the same time adding special domain-specific features for contract programming. Serpent is compiled using LLL.

- Serpent on the ethereum wiki
- Serpent EVM compiler

LLL

Lisp Like Language (LLL) is a low level language similar to Assembly. It is meant to be very simple and minimalistic; essentially just a tiny wrapper over coding in EVM directly.

- LIBLLL in GitHub
- Examples of LLL

Mutan (deprecated)

Mutan is a statically typed, C-like language designed and developed by Jeffrey Wilcke. It is no longer maintained.

Writing a contract

No language would be complete without a Hello World program. Operating within the Ethereum environment, Solidity has no obvious way of "outputting" a string. The closest we can do is to use a *log event* to place a string into the blockchain:

```
contract HelloWorld {
        event Print(string out);
        function() { Print("Hello, World!"); }
}
```

This contract will create a log entry on the blockchain of type Print with a parameter "Hello, World!" each time it is executed.

See also:

Solidity docs has more examples and guidelines to writing Solidity code.

Compiling a contract

Compilation of solidity contracts can be accomplished via a number of mechanisms.

- Using the `solc` compiler via the command line.
- Using `web3.eth.compile.solidity` in the javascript console provided by `geth` or `eth` (This still requires the `solc` compiler to be installed).
- The online Solidity realtime compiler.
- The Meteor dapp Cosmo for building solidity contracts.
- The Mix IDE.
- The Ethereum Wallet.

Note: More information on solc and compiling Solidity contract code can be found here.

Setting up the solidity compiler in geth

If you start up your `geth` node, you can check which compilers are available.

```
> web3.eth.getCompilers();
["lll", "solidity", "serpent"]
```

This command returns an array of strings indicating which compilers are currently available.

Note: The `solc` compiler is installed with `cpp-ethereum`. Alternatively, you can build it yourself.

If your `solc` executable is in a non-standard location you can specify a custom path to the `solc` executable using th `--solc` flag.

```
$ geth --solc /usr/local/bin/solc
```

Alternatively, you can set this option at runtime via the console:

```
> admin.setSolc("/usr/local/bin/solc")
solc, the solidity compiler commandline interface
Version: 0.2.2-02bb315d/.-Darwin/appleclang/JIT linked to libethereum-1.2.0-8007cef0/.-Darwin/appl
path: /usr/local/bin/solc
```

Compiling a simple contract

Let's compile a simple contract source:

```
> source = "contract test { function multiply(uint a) returns(uint d) { return a * 7; } }"
```

This contract offers a single method **multiply** which is called with a positive integer `a` and returns `a * 7`.

You are ready to compile solidity code in the `geth` JS console using eth.compile.solidity():

```
> contract = eth.compile.solidity(source).test
{
  code: '605280600c6000396000f3006000357c010000000000000000000000000000000000000000000000000000000
  info: {
    language: 'Solidity',
    languageVersion: '0',
    compilerVersion: '0.9.13',
    abiDefinition: [{
      constant: false,
      inputs: [{
        name: 'a',
        type: 'uint256'
      } ],
      name: 'multiply',
      outputs: [{
        name: 'd',
        type: 'uint256'
      } ],
      type: 'function'
    } ],
    userDoc: {
      methods: {
      }
    },
    developerDoc: {
      methods: {
      }
    },
    source: 'contract test { function multiply(uint a) returns(uint d) { return a * 7; } }'
  }
}
```

Note: The compiler is also available via RPC and therefore via web3.js to any in-browser Đapp connecting to `geth` via RPC/IPC.

The following example shows how you interface `geth` via JSON-RPC to use the compiler.

```
$ geth --datadir ~/eth/ --loglevel 6 --logtostderr=true --rpc --rpcport 8100 --rpccorsdomain '*' -
$ curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["contract test {
```

The compiler output for one source will give you contract objects each representing a single contract. The actual return value of `eth.compile.solidity` is a map of contract name to contract object pairs. Since our contract's name is `test`, `eth.compile.solidity(source).test` will give you the contract object for the test contract containing the following fields:

code The compiled EVM bytecode

info Additional metadata output from the compiler

source The source code

language The contract language (Solidity, Serpent, LLL)

languageVersion The contract language version

compilerVersion The solidity compiler version that was used to compile this contract.

abiDefinition The Application Binary Interface Definition

userDoc The NatSpec Doc for users.

developerDoc The NatSpec Doc for developers.

The immediate structuring of the compiler output (into `code` and `info`) reflects the two very different **paths of deployment**. The compiled EVM code is sent off to the blockchain with a contract creation transaction while the rest (info) will ideally live on the decentralised cloud as publicly verifiable metadata complementing the code on the blockchain.

If your source contains multiple contracts, the output will contain an entry for each contact, the corresponding contract info object can be retrieved with the name of the contract as attribute name. You can try this by inspecting the most current GlobalRegistrar code:

```
contracts = eth.compile.solidity(globalRegistrarSrc)
```

Create and deploy a contract

Before you begin this section, make sure you have both an unlocked account as well as some funds.

You will now create a contract on the blockchain by sending a transaction to the empty address with the EVM code from the previous section as data.

Note: This can be accomplished much easier using the online Solidity realtime compiler or the Mix IDE program.

```
var primaryAddress = eth.accounts[0]
var abi = [{ constant: false, inputs: [{ name: 'a', type: 'uint256' } ]
var MyContract = eth.contract(abi)
var contract = MyContract.new(arg1, arg2, ..., {from: primaryAddress, data: evmByteCodeFromPrevio
```

All binary data is serialised in hexadecimal form. Hex strings always have a hex prefix `0x`.

Note: Note that `arg1, arg2, ...` are the arguments for the contract constructor, in case it accepts any. If the contract does not require any constructor arguments then these arguments can be omitted.

It is worth pointing out that this step requires you to pay for execution. Your balance on the account (that you put as sender in the `from` field) will be reduced according to the gas rules of the EVM once your transaction makes it into a block. After some time, your transaction should appear included in a block confirming that the state it brought about is a consensus. Your contract now lives on the blockchain.

The asynchronous way of doing the same looks like this:

```
MyContract.new([arg1, arg2, ...,]{from: primaryAccount, data: evmCode}, function(err, contract) {
  if (!err && contract.address)
    console.log(contract.address);
});
```

Interacting with a contract

Interaction with a contract is typically done using an abstraction layer such as the eth.contract() function which returns a javascript object with all of the contract functions available as callable functions in javascript.

The standard way to describe the available functions of a contract is the ABI definition. This object is an array which describles the call signature and return values for each available contract function.

```
var Multiply7 = eth.contract(contract.info.abiDefinition);
var myMultiply7 = Multiply7.at(address);
```

Now all the function calls specified in the ABI are made available on the contract instance. You can just call those methods on the contract instance in one of two ways.

```
> myMultiply7.multiply.sendTransaction(3, {from: address})
"0x12345"
> myMultiply7.multiply.call(3)
21
```

When called using `sendTransaction` the function call is executed via sending a transaction. This will cost ether to send and the call will be recorded forever on the blockchain. The return value of calls made in this manner is the hash of the stransaction.

When called using `call` the function is executed locally in the EVM and the return value of the function is returned with the function. Calls made in this manner are not recorded on the blockchain and thus, cannot modify the internal state of the contract. This manner of call is referred to as a **constant** function call. Calls made in this manner do not cost any ether.

You should use `call` if you are interested only in the return value and use `sendTransaction` if you only care about *side effects* on the state of the contract.

In the example above, there are no side effects, therefore `sendTransaction` only burns gas and increases the entropy of the universe.

Contract metadata

In the previous sections we explained how you create a contract on the blockchain. Now we will deal with the rest of the compiler output, the **contract metadata** or contract info.

When interacting with a contract you did not create you might want documentation or to look at the source code. Contract authors are encouraged to make such information available by registering it on the blockchain or through a third party service, such as EtherChain. The `admin` API provides convenience methods to fetch this bundle for any contract that chose to register.

```
// get the contract info for contract address to do manual verification
var info = admin.getContractInfo(address) // lookup, fetch, decode
var source = info.source;
var abiDef = info.abiDefinition
```

The underlying mechanism that makes this work is is that:

- contract info is uploaded somewhere identifiable by a *URI* which is publicly accessible
- anyone can find out what the *URI* is only knowing the contracts address

These requirements are achieved using a 2 step blockchain registry. The first step registers the contract code (hash) with a content hash in a contract called `HashReg`. The second step registers a url with the content hash in the `UrlHint` contract. These registry contracts were part of the Frontier release and have carried on into Homestead.

By using this scheme, it is sufficient to know a contract's address to look up the url and fetch the actual contract metadata info bundle.

So if you are a conscientious contract creator, the steps are the following:

1. Deploy the contract itself to the blockchain

2. Get the contract info json file.
3. Deploy contract info json file to any url of your choice
4. Register codehash ->content hash -> url

The JS API makes this process very easy by providing helpers. Call `admin.register` to extract info from the contract, write out its json serialisation in the given file, calculates the content hash of the file and finally registers this content hash to the contract's code hash. Once you deployed that file to any url, you can use `admin.registerUrl` to register the url with your content hash on the blockchain as well. (Note that in case a fixed content addressed model is used as document store, the url-hint is no longer necessary.)

```
source = "contract test { function multiply(uint a) returns(uint d) { return a * 7; } }"
// compile with solc
contract = eth.compile.solidity(source).test
// create contract object
var MyContract = eth.contract(contract.info.abiDefinition)
// extracts info from contract, save the json serialisation in the given file,
contenthash = admin.saveInfo(contract.info, "~/dapps/shared/contracts/test/info.json")
// send off the contract to the blockchain
MyContract.new({from: primaryAccount, data: contract.code}, function(error, contract){
  if(!error && contract.address) {
    // calculates the content hash and registers it with the code hash in `HashReg`
    // it uses address to send the transaction.
    // returns the content hash that we use to register a url
    admin.register(primaryAccount, contract.address, contenthash)
    // here you deploy ~/dapps/shared/contracts/test/info.json to a url
    admin.registerUrl(primaryAccount, hash, url)
  }
});
```

Testing contracts and transactions

Often you need to resort to a low level strategy of testing and debugging contracts and transactions. This section introduces some debug tools and practices you can use. In order to test contracts and transactions without real-word consequences, you best test it on a private blockchain. This can be achieved with configuring an alternative network id (select a unique integer) and/or disable peers. It is recommended practice that for testing you use an alternative data directory and ports so that you never even accidentally clash with your live running node (assuming that runs using the defaults. Starting your `geth` with in VM debug mode with profiling and highest logging verbosity level is recommended:

```
geth --datadir ~/dapps/testing/00/ --port 30310 --rpcport 8110 --networkid 4567890 --nodiscover --
```

Before you can submit any transactions, you need set up your private test chain. See *Test Networks*.

```
// create account. will prompt for password
personal.newAccount();
// name your primary account, will often use it
primary = eth.accounts[0];
// check your balance (denominated in ether)
balance = web3.fromWei(eth.getBalance(primary), "ether");
```

```
// assume an existing unlocked primary account
primary = eth.accounts[0];

// mine 10 blocks to generate ether

// starting miner
miner.start(4);
// sleep for 10 blocks (this can take quite some time).
admin.sleepBlocks(10);
// then stop mining (just not to burn heat in vain)
```

```
miner.stop();
balance = web3.fromWei(eth.getBalance(primary), "ether");
```

After you create transactions, you can force process them with the following lines:

```
miner.start(1);
admin.sleepBlocks(1);
miner.stop();
```

You can check your pending transactions with:

```
// shows transaction pool
txpool.status
// number of pending txs
eth.getBlockTransactionCount("pending");
// print all pending txs
eth.getBlock("pending", true).transactions
```

If you submitted contract creation transaction, you can check if the desired code actually got inserted in the current blockchain:

```
txhash = eth.sendTansaction({from:primary, data: code})
//... mining
contractaddress = eth.getTransactionReceipt(txhash);
eth.getCode(contractaddress)
```

1.7.3 Accessing Contracts and Transactions

RPC

In previous sections we have seen how contracts can be written, deployed and interacted with. Now it's time to dive in the details of communicating with the Ethereum network and smart contracts.

An Ethereum node offers a RPC interface. This interface gives Đapp's access to the Ethereum blockchain and functionality that the node provides, such as compiling smart contract code. It uses a subset of the JSON-RPC 2.0 specification (no support for notifications or named parameters) as serialisation protocol and is available over HTTP and IPC (unix domain sockets on linux/OSX and named pipe's on Windows).

If you are not interested in the details but are looking for an easy to use javascript library you can skip the following sections and continue with *Using Web3*.

Conventions

The RPC interface uses a couple of conventions that are not part of the JSON-RPC 2.0 specification:

- Numbers are hex encoded. This decision was made because some languages have no or limited support for working with extremly large numbers. To prevent these type of errors numbers are hex encoded and it is up to the deverloper to parse these numbers and handle them appropriately. See the hex encoding section on the wiki for examples.
- Default block number, several RPC methods accept a block number. In some cases it's not possible to give a block number or not very convenient. For these cases the default block number can be one of these strings [”earliest”, “latest”, “pending”]. See the wiki page for a list of RPC methods that use the default block parameters.

Deploy contract

We will go through the different steps to deploy the following contract using only the RPC interface.

```
contract Multiply7 {
   event Print(uint);
   function multiply(uint input) returns (uint) {
      Print(input * 7);
      return input * 7;
   }
}
```

The first thing to do is make sure the HTTP RPC interface is enabled. This means for geth we supply the `--rpc` flag on startup and for eth the `-j` flag. In this example we use the geth node on a private development chain. Using this approach we don't need ether on the real network.

```
> geth --rpc --dev --mine --minerthreads 1 --unlock 0 console 2>>geth.log
```

This will start the HTTP RPC interface on `http://localhost:8545`.

Note: geth supports CORS, see the `--rpccorsdomain` flag for more information.

We can verify that the interface is running by retrieving the coinbase address and balance using curl. Please note that data in these examples will differ on your local node. If you want to try these command replace the request params accordingly.

```
> curl --data '{"jsonrpc":"2.0","method":"eth_coinbase", "id":1}' localhost:8545
{"id":1,"jsonrpc":"2.0","result":["0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a"]}

> curl --data '{"jsonrpc":"2.0","method":"eth_getBalance", "params": ["0xeb85a5557e5bdc18ee1934a8
{"id":2,"jsonrpc":"2.0","result":"0x1639e49bba16280000"}
```

Remember when we said that numbers are hex encoded? In this case the balance is returned in Wei as a hex string. If we want to have the balance in Ether as a number we can use web3 from the geth console.

```
> web3.fromWei("0x1639e49bba16280000", "ether")
"410"
```

Now that we have some ether on our private development chain we can deploy the contract. The first step is to verify that the solidity compiler is available. We can retrieve available compilers using the `eth_getCompilers` RPC method.

```
> curl --data '{"jsonrpc":"2.0","method": "eth_getCompilers", "id": 3}' localhost:8545
{"id":3,"jsonrpc":"2.0","result":["Solidity"]}
```

We can see that the solidity compiler is available. If it's not available follow these instructions.

The next step is to compile the Multiply7 contract to byte code that can be send to the EVM.

```
> curl --data '{"jsonrpc":"2.0","method": "eth_compileSolidity", "params": ["contract Multiply7 {
{"id":4,"jsonrpc":"2.0","result":{"Multiply7":{"code":"0x6060604052605f806010600039600f360606040
```

Now that we have the compiled code we need to determine how much gas it costs to deploy it. The RPC interface has an `eth_estimateGas` method that will give us an estimate.

```
> curl --data '{"jsonrpc":"2.0","method": "eth_estimateGas", "params": [{"from": "0xeb85a5557e5bd
{"id":5,"jsonrpc":"2.0","result":"0xb8a9"}
```

And finally deploy the contract.

```
> curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction", "params": [{"from": "0xeb85a5557
{"id":6,"jsonrpc":"2.0","result":"0x3a90b5face52c4c5f30d507ccf51b0209ca628c6824d0532bcd6283df7c08
```

The transaction is accepted by the node and a transaction hash is returned. We can use this hash to track the transaction.

The next step is to determine the address where our contract is deployed. Each executed transaction will create a receipt. This receipt contains various information about the transaction such as in which block the transaction

was included and how much gas was used by the EVM. If a transaction creates a contract it will also contain the contract address. We can retrieve the receipt with the `eth_getTransactionReceipt` RPC method.

```
> curl --data '{"jsonrpc":"2.0","method": "eth_getTransactionReceipt", "params": ["0x3a90b5face52c
{"id":7,"jsonrpc":"2.0","result":{"transactionHash":"0x3a90b5face52c4c5f30d507ccf51b0209ca628c682
```

We can see that our contract was created on `0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d`. If you got null instead of a receipt the transaction has not been included in a block yet. Wait for a moment and check if your miner is running and retry it.

Interacting with smart contracts

Now that our contract is deployed we can interact with it. There are 2 methods for this, sending a transaction or *using call as previously explained*. In this example we will be sending a transaction to the multiply method of the contract.

If we look at the documentation for the eth_sendTransaction we can see that we need to supply several arguments. In our case we need to specify the `from`, `to` and `data` arguments. `From` is the public address of our account and `to` the contract address. The `data` argument is a bit harder. It contains a payload that defines which method must be called and with which arguments. This is were the ABI comes into play. The ABI defines how to define and encode data for the EVM. You can read all the details about the ABI here.

The bytes of the payload is the function selector and defines which method is called. This is done by taking the first 4 bytes from the Keccak hash over the function name and its argument types and hex encode it. The *multiply* function accepts an *uint* which is an alias for *uint256*. This leaves us with:

```
> web3.sha3("multiply(uint256)").substring(0, 8)
"c6888fa1"
```

See for details this page.

The next step is to encode the arguments. We only have one uint256, lets assume we supply the value 6. The ABI has a section which specifies how to encode uint256 types.

> *int<M>: enc(X) is the big-endian two's complement encoding of X, padded on the higher-oder (left) side with 0xff for negative X and with zero bytes for positive X such that the length is a multiple of 32 bytes.*

This encodes to `0000000000000000000000000000000000000000000000000000000000000006`.

Combining the function selector and the encoded argument our `data` will be `0xc6888fa10000000000000000000000000000000000000000000000000000000000000006`.

Lets try it:

```
> curl --data '{"jsonrpc":"2.0","method": "eth_sendTransaction", "params": [{"from": "0xeb85a5557e
{"id":8,"jsonrpc":"2.0","result":"0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869
```

Since we sent a transaction we got the transaction hash returned. If we retrieve the receipt we can see something new:

```
{
   blockHash: "0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55",
   blockNumber: 268,
   contractAddress: null,
   cumulativeGasUsed: 22631,
   gasUsed: 22631,
   logs: [{
      address: "0x6ff93b4b46b41c0c3c9baee01c255d3b4675963d",
      blockHash: "0xbf0a347307b8c63dd8c1d3d7cbdc0b463e6e7c9bf0a35be40393588242f01d55",
      blockNumber: 268,
      data: "0x000000000000000000000000000000000000000000000000000000000000002a",
      logIndex: 0,
      topics: ["0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"],
```

```
      transactionHash: "0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74",
      transactionIndex: 0
  }],
  transactionHash: "0x759cf065cbc22e9d779748dc53763854e5376eea07409e590c990eafc0869d74",
  transactionIndex: 0
}
```

The receipt contains a log. This log was generated by the EVM on transaction execution and included in the receipt. If we look at the multipy function we can see that the Print event was raised with the input times 7. Since the argument for the Print event was a uint256 we can decode it according to the ABI rules which will leave us with the expected decimal 42. Apart from the data it is worth noting that topics can be used to determine which event created the log:

```
> web3.sha3("Print(uint256)")
"24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
```

You can read more about events, topics and indexing in the Solidity tutorial.

This was just a brief introduction into some of the most common tasks. See for a full list of available RPC methods the RPC wiki page.

Web3.js

As we have seen in the previous example using the JSON-RPC interface can be quite tedious and error-prone, especially when we have to deal with the ABI. Web3.js is a javascript library that works on top of the Ethereum RPC interface. Its goal is to provide a more user friendly interface and reducing the chance for errors.

Deploying the Multiply7 contract using web3 would look like:

```
var source = 'contract Multiply7 { event Print(uint); function multiply(uint input) returns (uint
var compiled = web3.eth.compile.solidity(source);
var code = compiled.Multiply7.code;
var abi = compiled.Multiply7.info.abiDefinition;

web3.eth.contract(abi).new({from: "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a", data: code}, funct
   if (!err && contract.address)
      console.log("deployed on:", contract.address);
   }
);

deployed on: 0x0ab60714033847ad7f0677cc7514db48313976e2
```

Load a deployed contract and send a transaction:

```
var source = 'contract Multiply7 { event Print(uint); function multiply(uint input) returns (uint
var compiled = web3.eth.compile.solidity(source);
var Multiply7 = web3.eth.contract(compiled.Multiply7.info.abiDefinition);
var multi = Multiply7.at("0x0ab60714033847ad7f0677cc7514db48313976e2")
multi.multiply.sendTransaction(6, {from: "0xeb85a5557e5bdc18ee1934a89d8bb402398ee26a"})
```

Register a callback which is called when the `Print` event created a log.

```
multi.Print(function(err, data) { console.log(JSON.stringify(data)) })
{"address":"0x0ab60714033847ad7f0677cc7514db48313976e2","args": {"":"21"},"blockHash":"0x259c7dc0
```

See for more information the web3.js wiki page.

Console

The geth console offers a command line interface with a javascript runtime. It can connect to a local or remote geth or eth node. It will load the web3.js library that users can use. This allows users to deploy and interact with

smart contract from the console using web3.js. In fact the examples in the *Web3.js* section can by copied into the console.

Viewing Contracts and Transactions

There are several online blockchain explorers available that will allow you to inspect the Ethereum blockchain. See for a list: *Blockchain explorers*.

Hosted blockchain explorers

- EtherChain
- EtherCamp
- EtherScan (and for Testnet)

Other Resources

- EtherNodes - Geographic distribution of nodes and split by client
- EtherListen - Realtime Ethereum transaction visualizer and audializer

1.7.4 Mix

The IDE Mix is intended to help you as a developer to create, debug and deploy contracts and dapps (both contracts backend and frontend).

WARNING - There are numerous reports of crash-at-boot issues for Mix on OS X. The issue is a Heisenbug which we have been chasing for a month or two. The best workaround we have for right now is to use the Debug configuration, like so:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

WARNING - A replacement for Mix called Remix is being worked on, so if you are experiencing issues with Mix, you might be better to look for alternatives until Remix is more mature.

Start by creating a new project that consists of

- contracts
- html files
- JavaScript files
- style files
- image files

Project Editor

You can use projects to manage the creation and testing of a dapp. The project will contain data related to both backend and frontend as well as the data related to your scenarios (blockchain interaction) for debugging and testing. The related files will be created and saved automatically in the project directory.

Creating a new project

The development of a dapp start with the creation of a new project. Create a new project in the “edit” menu. Enter the project name, e.g. “Ratings” and select a path for the project file.

Editing backend contract file

By default, a new project contains a contract "Contract" for backend development on the blockchain using the Solidity language and the "index.html" for the frontend. Check the Solidity tutorial for references.

Edit the empty default contract "Contract", e.g.

```
contract Rating {
        function setRating(bytes32 _key, uint256 _value) {
                ratings[_key] = _value;
        }
        mapping (bytes32 => uint256) public ratings;
}
```

Check the Solidity tutorial for help getting started with the solidity programming language.

Save changes

Editing frontend html files Select default index.html file and enter the following code

```
 .... <script>

function getRating() {
  var param = document.getElementById("query").value;
  var res = contracts["Rating"].contract.ratings(param);
  document.getElementById("queryres").innerText = res;
}

function setRating() {
        var key = document.getElementById("key").value;
        var value = parseInt(document.getElementById("value").value);
        var res = contracts["Rating"].contract.setRating(key, value);
}

</script>
    </head>
    <body bgcolor="#E6E6FA">
        <h1>Ratings</h1>
        <div>
            Store:
            <input type="string" id="key">
            <input type="number" id="value">
            <button onclick="setRating()">Save</button>
        </div>
        <div>
            Query:
            <input type="string" id="query" onkeyup='getRating()'>
            <div id="queryres"></div>
        </div>
    </body>
    </html>
```

Then it is possible to add many contract files as well as many HTML, JavaScript, css files

Scenarios Editor

Scenarios can be used to test and debug contracts.

A scenario is effectively a local blockchain where blocks can be mined without PoW – otherwise testing would be quite slow ;).

A scenario consists of a sequence of transactions. Usually, a scenario would start with the contract creation scenarios of the dapp. In addition, further transactions can be added to test and debug the dapp. Scenarios can be

modified, i.e. transactions can be removed. Note that a scenario needs to be rebuilt for modifications to become effective. Further testing can be done using local JS calls via the JS API.

In case it's not open, access the scenario and debugger pane by pressing F7 or Windows > Show right or the debug button in the upper right corner of the main window.

Creating and setting up a new scenario

When you launch Mix for the first time, an empty scenario, i.e. not containing any transactions, will be created. Add an account named "MyAccount" and set it's initial balance to 1 ether. Click OK. Rename the scenario to "Deploy".

Modifying initial ether balance of an account

Actually, we want to do a lot of tests Edit the Genesis block parameters and set your initial account balance to 1000 ether. Rebuild the scenario for the change to become effective.

Rebuilding a scenario

Each time a transaction is modified or an account added, the scenario has to be rebuilt for modifications to become effective. Note that if a scenario is rebuilt the web frontend (local storage) may also need to be reset (this is not done automatically be Mix).

Creating a transaction

Let's get some ether sent to Bob. Create another account named "Bob" with zero ether balance. Create a new transaction in the scenario pane. Click "Add Tx. . . " and send 300 ether to Bob. Add a block.

Altering and reusing scenarios

Create a new scenario or start from a scenario with several transactions that you duplicate first

Rename the scenario

Modify scenario by specifying transactions that shall be removed

Rebuild the scenario

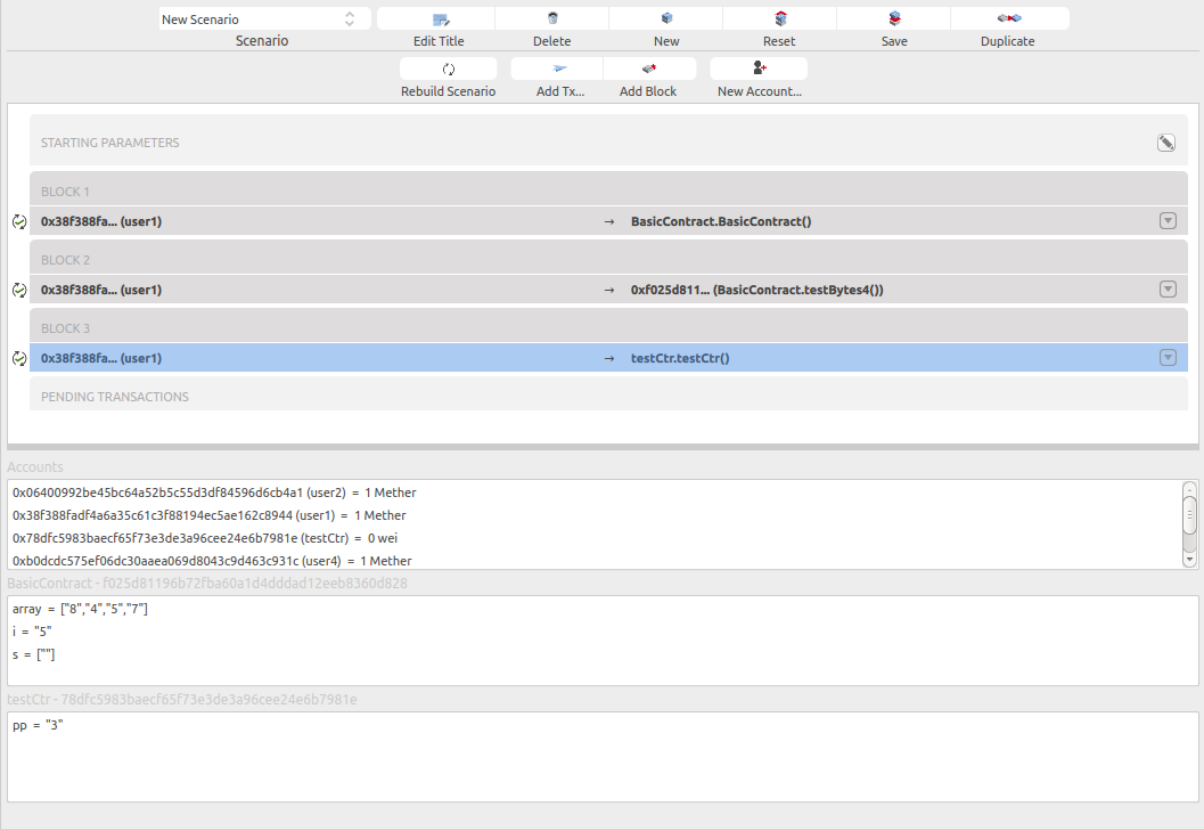
Display calls

A contract call is a function invokation. This is not a transaction as a contract call cannot change the state. A contract call is not part of the blockchain but for practical and ux design reason, it is convenient to display calls at the same functional level as a transaction. The JS icon warn you that this is not a transaction but a call. To show/hide call, click on the menu Scenario -> Display calls.

State Viewer

This panel is located below the block chain panel, in the scenario view. Once the blockchain has been run, this panel shows the state of the blockchain.

By state we mean all accounts balance (including contract and normal account), and the storage (global variable of all deployed contract). The content of this panel is not static, it depends on the selected transaction on the blockchain panel. The state shown here is the state resulting of the execution of the selected transaction.



In that case, 2 contracts are deployed, the selected transaction (deployment of testCtr) is the last one. so the state view shows the storage of both TestCtr and BasicContract.

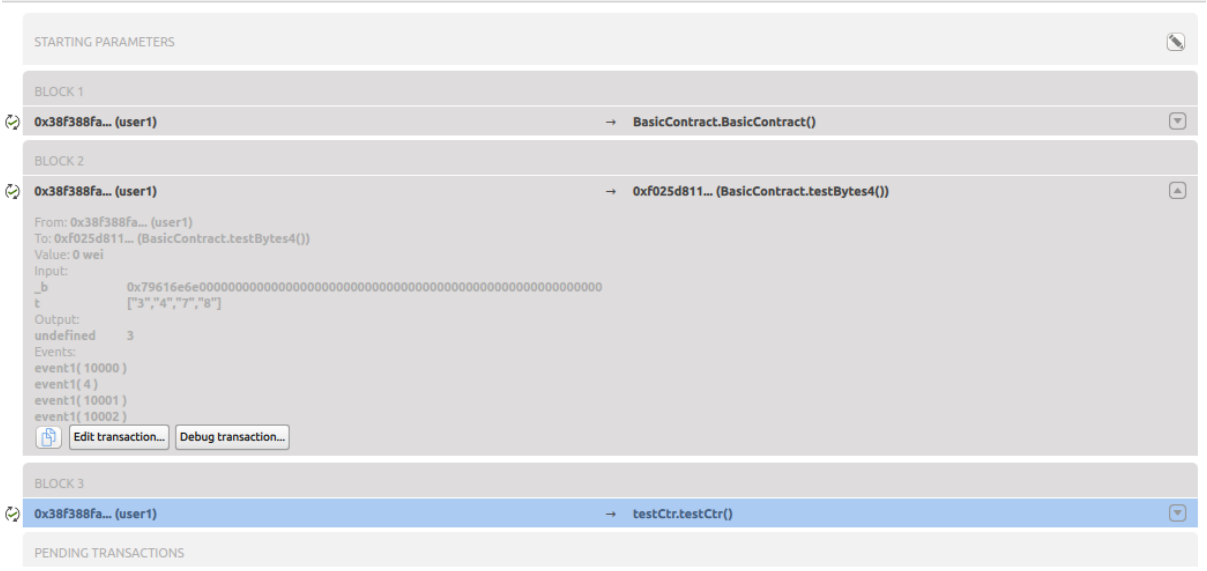
Transaction Explorer

Using the transaction pane

The transaction pane enables you to explore transactions receipts, including

- Input parameters
- Return parameters
- Event logs

To display the transaction explorer, click on the down triangle icon which is on the right of each transaction, this will expand transaction details:



Then you can either copy the content of this transaction in the clipboard, Edit the current transaction (you will have to rerun the blockchain then), or debug the transaction.

JavaScript console

Mix exposes the following objects into the global window context

web3 - Ethereum JavaScript API

contracts: A collection of contract objects. keys represents contracts name. values are is an objects containing the following properties:

- contract: contract object instance (created as in web3.eth.contract)
- address: contract address from the last deployed state (see below)
- interface: contract ABI

Check the JavaScript API Reference for further information.

Using the JS console to add transactions and local calls

In case the name of the contract is “Sample” with a function named “set”, it is possible to make a transaction to call “set” by writing:

```
contracts["Sample"].contract.set(14)
```

If a call can be made this will be done by writing:

```
contracts["Sample"].contract.get.call()
```

It is also possible to use all properties and functions of the web3 object:
https://github.com/ethereum/wiki/wiki/JavaScript-API

Transaction debugger

Mix supports both Solidity and assembly level contract code debugging. You can toggle between the two modes to retrieve the relevant information you need.

At any execution point the following information is available:

VM stack – See Yellow Paper for VM instruction description

Call stack – Grows when contract is calling into another contract. Double click a stack frame to view the machine state in that frame

Storage – Storage data associated with the contract

Memory – Machine memory allocated up to this execution point

Call data – Transaction or call parameters

Accessing the debug mode

When transaction details are expanded, you can switch to the debugger view by clicking on the "Debug Transaction" button

Toggling between debug modes and stepping through transactions

This opens the Solidity debugging mode. Switch between Solidity and EVM debugging mode using the Menu button (Debug -> Show VM code)

- Step through a transaction in solidity debugging mode
- Step through a transaction in EVM debugging mode

Dapps deployment

This feature allows users to deploy the current project as a Dapp in the main blockchain.
This will deploy contracts and register frontend resources.

The deployment process includes three steps:

- **Deploy contract**:
  This step will deploy contracts in the main blockchain.
- **Package dapp**:
  This step is used to package and upload frontend resources.
- **Register**:
  To render the Dapp, the Ethereum browser (Mist or AlethZero) needs to access this package. This step will register the URL where the resources are stored.

To Deploy your Dapp, Please follow these instructions:

Click on `Deploy`, `Deploy to Network`.
This modal dialog displays three parts (see above):

- **Deploy contract**
- *Select Scenario*

"Ethereum node URL" is the location where a node is running, there must be a node running in order to initiate deployment.

"Pick Scenario to deploy" is a mandatory step. Mix will execute transactions that are in the selected scenario (all transactions except transactions that are not related to contract creation or contract call). Mix will display all the transactions in the panel below with all associated input parameters.

"Gas Used": depending on the selected scenario, Mix will display the total gas used.

- *Deploy Scenario*

"Deployment account" allow selecting the account that Mix will use to execute transactions.

"Gas Price" shows the default gas price of the network. You can also specify a different value.

"Deployment cost": depending on the value of the gas price that you want to use and the selected scenario. this will display the amount ether that the deployment need.

"Deployed Contract": before any deployment this part is empty. This will be filled once the deployment is finished by all contract addresses that have been created.

"Verifications". This will shows the number of verifications (number of blocks generated on top of the last block which contains the last deployed transactions). Mix keep track of all the transactions. If one is missing (unvalidated) it will be displayed in this panel.

- *Package dapp*

The action "Generate Package" will create a new folder named 'www', this folder will contain all the resources and scripts will be mapped to the current deployed contract. In order to publish your dapp, you need to host the www folder in a webserver (to be replace soon by IPFS and SWARM). by default the library web3.js is not included. If you want to be able to use the dapp in a standard web browser, you wiil need to include this library.

Code Editor

This editor provides basic functionalities of a code editor.

- In Solidity or JavaScript mode, an autocompletion plugin is available (Ctrl + Space).
- Increasing/decreasing the font size (Ctrl +, Ctrl -)
- In Solidity mode, you can display the gas estimation (Tools -> Display Gas Estimation). This will highlight all statements which requires a minimum amount of gas. Color turns to red if the gas required becomes important. It will also display the max execution cost of a transaction (for each function).

1.7.5 Dapps

A dapp is service that enables direct interaction between end users and providers (e.g. connecting buyers and sellers in some marketplace, owners and storers in file storage). Ethereum dapps typically interface users via an HTML/Javascript web application using a Javascript API to communicate with the blockchain. Dapps would typically have their own suite of associated contracts on the blockchain which they use to encode business logic and allow persistent storage of their consensus-critical state. Remember that because of the redundant nature of computation on the Ethereum network, the gas costs of execution will always be higher than private execution offchain. This incentivizes dapp developers to restrict the amount of code they execute and amount of data they store on the blockchain.

Dapp directories

Dapps that use Ethereum are compiled to the following lists. They are listed in various stages of development (concept, working prototype, live/deployed). If you are developing a dapp, consider adding an entry to these listings:

- Ethercasts State of the Đapps

- Dappslist
- Dappcentral - Sortable pages for Dapps with instructions, code validation, and network stats.
- Dapps Mailing List - Mailing list for developers on Ethereum (discontinued).

The offered decentralised services listed cover a wide range of areas including finance, insurance, prediction markets, social networks, distributed computation and storage, gambling, marketplace, internet of things, governance, collaboration, development and games.

- What apps can we eventually expect? https://www.reddit.com/r/ethereum/comments/2mnl7f/the_top_10_ether_dapps_of_201

In the future, dapps are likely to be listed and distributed in dappstores integrated in dapp browsers.

Dapp browsers

- Mist - official GUI dapp browser developed by the foundation, alpha stage. Mist as Wallet dapp is in beta.
- Syng - Mobile Ethereum browser (alpha) by Jarrad Hope - supported by DEVgrants
- MetaMask - Aaron Kumavis Davis's in-browser GUI. Epicenter Bitcoin interview on github - supported by DEVgrants
- AlethZero - C++ eth client GUI, (discontinued).
- Supernova - (discontinued).

1.7.6 Developer Tools

Dapp development requires an understanding of the Web3 Javascript API, the JSON RPC API, and the Solidity programming language.

Note: There are developer tools that help you develop, test, and deploy dapps in a way that automatically utilizes the resources listed below.

- Web3 JavaScript API - This is the main JavaScript SDK to use when you want to interact with an Ethereum node.
- JSON RPC API - This is the low level JSON RPC 2.0 interface to interface with a node. This API is used by the Web3 JavaScript API.
- Solidity Docs - Solidity is the Ethereum developed Smart Contract language, which compiles to EVM (Ethereum Virtual Machine) opcodes.
- *Test Networks* - Test networks help developers develop and test Ethereum code and network interactions without spending their own Ether on the main network. Test network options are listed below.
- *Dapp development resources*. This assists you in developing, debugging, and deploying Ethereum applications.

Dapp development resources

- Smart contracts ELI5
- https://blog.slock.it/a-primer-to-the-decentralized-autonomous-organization-dao-69fb125bd3cd
- A 101 noob's intro to programming smart contracts
- Standardised contract APIs listing

Examples

- example use of pricefeed - web3 script printing all account balances
- Example Ethereum contracts

https://dappsforbeginners.wordpress.com/tutorials/your-first-dapp/

https://github.com/ethereum/wiki/wiki/Dapp-Developer-Resources

Tutorials

- Dapp tutorials on ethereum.org
- Dapps for beginners tutorial series
- Eris' Solidity Tutorial Series
- Tutorials on advanced Solidity
- http://ethereumj.io/blog/2015/09/09/friendly-ether-bot/
- https://github.com/ConsenSys/ether-pudding

Mix-IDE

Mix is the official Ethereum IDE that allows developers to build and deploy contracts and decentralized applications on top of the Ethereum blockchain. It includes a Solidity source code debugger. *Mix*

IDEs/Frameworks

Below are developer frameworks and IDEs used for writing Ethereum dapps.

- Truffle - Truffle is a development environment, testing framework and asset pipeline for Ethereum.
- Dapple - Dapple is a tool for Solidity developers to help build and manage complex contract systems on Ethereum-like blockchains.
- Populus - Populus is a Smart Contract development framework written in python.
- Eris-PM - The Eris Package Manager deploys and tests smart contract systems on private and public chains.
- Embark - Embark is a Đapp development framework written in JavaScript.
- EtherScripter (obsolete, discontinued)
- Resilience Raw Transaction Broadcaster

Ethereum-console

Commandline console for Ethereum nodes.

Ethconsole connects to an Ethereum node running in the background (tested with eth and geth) via IPC and provides an interactive javascript console containing the web3 object with admin additions.

Here you could find a list of available commands ethereum node control commands

To use this console you would need to start a local ethereum node with ipc communication socket enabled (file `geth.ipc` in data directory). By default ipc socket should be located at you local home directory in .ethereum after you started a node. You could also set `--test` option to use specific node test commands.

In the console you could then type

Here the defenition of `--test` mode node commands:

More information about node configuration file.

Base layer services

Whisper

- What is Whisper and what is it used for - stackexchange Q&A
- Gavin Wood: Shh! Whisper - DEVCON-1 talk youtube video
- Whisper overview and dream API usage -
- ELI5

Swarm

Swarm is a distributed storage platform and content distribution service, a native base layer service of the Ethereum web 3 stack. The primary objective of Swarm is to provide a sufficiently decentralized and redundant store of Ethereum's public record, in particular to store and distribute dapp code and data as well as block chain data. From an economic point of view, it allows participants to efficiently pool their storage and bandwidth resources in order to provide the aforementioned services to all participants.

From the end user's perspective, Swarm is not that different from WWW, except that uploads are not to a specific server. The objective is to peer-to-peer storage and serving solution that is DDOS-resistant, zero-downtime, fault-tolerant and censorship-resistant as well as self-sustaining due to a built-in incentive system which uses peer to peer accounting and allows trading resources for payment. Swarm is designed to deeply integrate with the devp2p multiprotocol network layer of Ethereum as well as with the Ethereum blockchain for domain name resolution, service payments and content availability insurance.

ÐΞVcon talks on swarm

- Viktor Trón, Daniel A. Nagy: Swarm - Ethereum ÐΞVcon-1 talk on youtube
- Daniel A. Nagy: Keeping the Public Record Safe and Accessible - Ethereum ÐΞVcon-0 talk on youtube

Code and status

- [source](https://github.com/ethereum/go-ethereum/tree/swarm)
- [issues on github](https://github.com/ethereum/go-ethereum/labels/swarm)
- [development roadmap]()
- ethersphere on twitter
- swarm gitter room
- swarm subreddit

Storage on and offchain

- https://www.reddit.com/r/ethereum/comments/3hkv2f/eli5_storage_in_the_ethereum_blockchain/
- https://www.reddit.com/r/ethereum/comments/3npsoz/ethereum_ipfs_and_filecoin/
- What is swarm and what is it used for? - stackexchange Q&A

Ethereum Alarm Clock

- **Author:** Piper Merriam
- **Website:** alarm_main_website.
- **Documentation:** alarm_documentation.

A marketplace that facilitates scheduling transactions to occur at a later time. Serves a similar role to things like *crontab* in unix, or *setTimeout* in javascript.

- Decentralized cron service in Ethereum proposal - by Peter Szilagyi

Ethereum Computation Market

- **Author:** Piper Merriam
- **Website:** computation_market_main_website.
- **Documentation:** computation_market_documentation.

A marketplace that facilitates verifiable execution of computations off-chain. Allows for very expernsive computations to be used within the EVM without having to actually pay the high gas costs of executing them on-chain.

BTCRelay

BTCrelay

- More information (about ETH/BTC 2-way peg without modifying bitcoin code).
- BTCrelay audit

RANDAO

Random number * https://www.reddit.com/r/ethereum/comments/49yld7/eli5_how_does_a_service_like_szabodice_grab_a/

The EVM

The Ethereum Virtual Machine (EVM) is the runtime environment for smart contracts in Ethereum. It is not only sandboxed, but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem, or other processes. Smart contracts even have limited access to other smart contracts.

Contracts live on the blockchain in an Ethereum-specific binary format (EVM bytecode). However, contracts are typically written in an Ethereum high level language, compiled into byte code using an EVM compiler, and finally uploaded on the blockchain using an Ethereum client.

1.7.7 Ethereum Tests

Common tests for all clients to test against. The git repo updated regulary with new tests.
This section describes basic test concepts and templates which are created by cpp-client.

Using Testeth

Ethereum cpp-client testeth tool for creation and execution of ethereum tests.

To run tests you should open folder (see also Installing and building)

```
/build/libethereum/test
```

and execute a command `./testeth` This will run all test cases automatically. To run a specific test case you could use parameter `-t` in the command line option:

```
./testeth -t <TEST_SUITE>/<TEST_CASE>
```

Or just the test suite:

```
./testeth -t <TEST_SUITE>
```

You could also use `--filltests` option to rerun test creation from .json files which are located at `../cpp-ethereum/test/<TEST_FILLER>.json`

```
./testeth -t <TEST_SUITE>/<TEST_CASE> --filltests
```

By default using `--filltests` option `testeth` recreate tests to the `ETHEREUM_TEST_PATH` folder. You might want to set this variable globally on your system like:

```
nano /etc/environment
ETHEREUM_TEST_PATH="/home/user/ethereum/tests"
```

Filler files are test templates which are used to fill initial parameters defined at test specification Ethereum Tests and then create a complete test `.json` file. You might find filler files very useful when creating your own tests.

The `--checkstate` option adds a BOOST error if the post state of filled test differs from it's `expected` section.

To specify a concrete test in a TEST_CASE file for filling/running procedure use `--singletest` option:

```
./testeth -t <TEST_SUITE>/<TEST_CASE> --singletest <TEST_NAME>
```

If you want to debug (note: testeth should be build with VMTRACE=1) a single test within a result test `.json` file, you might use the following command:

```
./testeth --log_level=test_suite --run_test=<TEST_SUITE>/<TEST_CASE>
--singletest <TEST_FILE>.json
    <TEST_NAME> --vmtrace --verbosity 12
```

or

```
./testeth -t <TEST_SUITE>/<TEST_CASE> --singletest <TEST_NAME> --vmtrace
--verbosity 12
```

Some tests may use excessive resources when running, so by default they are disabled. Such tests require specific flag to be set in order to be executed. Like `--performance`, `--inputLimits`, `--memory`, `--quadratic`. You may also enable all of the tests by setting `--all` flag. Be careful. Enabled memory tests may stress your system to use 4GB of RAM and more.

That's it for test execution. To read more about command line options you may run `testeth` with `--help` option.

Now let's see what test cases are available.

Test Cases

Almost each test case has it's filler file available at `/webthree-umbrella/libethereum/test`

TEST_SUITE = BlockTests TEST_CASES = blValidBlockTest blInvalidTransactionRLP blTransactionTest blInvalidHeaderTest userDefinedFile

TEST_SUITE = TransactionTests TEST_CASES = ttTransactionTest ttWrongRLPTransaction tt10mbDataField userDefinedFile

TEST_SUITE = StateTests TEST_CASES = stExample stSystemOperationsTest stPreCompiledContracts stLogTests stRecursiveCreate stTransactionTest stInitCodeTest stSpecialTest stRefundTest stBlockHashTest stQuadraticComplexityTest stSolidityTest stMemoryTest stCreateTest userDefinedFileState

TEST_SUITE = VMTests TEST_CASES = vm_tests vmArithmeticTest vmBitwiseLogicOperationTest vmSha3Test vmEnvironmentalInfoTest vmBlockInfoTest vmIOandFlowOperationsTest vmPushDupSwapTest vmLogTest vmSystemOperationsTest vmPerformanceTest vmInputLimitsTest1 vmInputLimitsTest2 vmRandom userDefinedFile

Blockchain Tests

Found in `/BlockTests`, the blockchain tests aim is to test the basic verification of a blockchain.

`/BlockTests` - general blockchain tests. All blocks are built on network: **Frontier** `/BlockTests/Homestead` - homestead blockchain tests. All blocks are built on network: **Homestead** `/BlockTests/TestNetwork` - transition blockchain tests. All blocks before 5th are built on network: **Frontier**, then each block should correspond to Homestead rules.

It is based around the notion of executing a list of single blocks, described by the `blocks` portion of the test. The first block is the modified genesis block as described by the `genesisBlockHeader` portion of the test. A set of pre-existing accounts are detailed in the `pre` portion and form the world state of the genesis block.

It is generally expected that the test implementer will read `genesisBlockHeader` and `pre` and build the corresponding blockchain in the client. Then the new blocks, described by its RLP found in the `rlp` object of the `blocks` (RLP of a complete block, not the block header only), is read. If the client concludes that the block is valid, it should execute the block and verify the parameters given in `blockHeader` (block header of the new block), `transactions` (transaction list) and `uncleHeaders` (list of uncle headers). If the client concludes that the block is invalid, it should verify that no `blockHeader`, `transactions` or `uncleHeaders` object is present in the test. The client is expected to iterate through the list of blocks and ignore invalid blocks.

Basic structure

```
{
   "ValidBlocks": {
           "genesisBlockHeader": { ... },
           "pre": { ... },
           "blocks" : [
                   {
                           "chainname" : "A",
                           "blocknumber" : "1",
                           "rlp": { ... },
                           "blockHeader": { ... },
                           "transactions": { ... },
                           "uncleHeaders": { ... }
                   },
                   {
                           "chainname" : "A",
                           "blocknumber" : "2",
                           "rlp": { ... },
                           "blockHeader": { ... },
                           "transactions": { ... },
                           "uncleHeaders": { ... }
                   }
           ]
   },
```

```
   "SomeInvalidBlocks": {
           "genesisBlockHeader": { ... },
           "pre": { ... },
           "blocks" : [
                   {
                           "chainname" : "B",
                           "blocknumber" : "3",
                           "chainnetwork" : "Frontier",
                           "rlp": { ... },
                   },
                   {
                           "blocknumber" : "1",
                           "rlp": { ... },
                           "blockHeader": { ... },
                           "transactions": { ... },
                           "uncleHeaders": { ... }
                   },
                   {
                           "blocknumber" : "1",
                           "chainnetwork" : "Homestead",
                           "rlp": { ... },
                   },
                   {
                           "blocknumber" : "2",
                           "rlp": { ... },
                           "blockHeader": { ... },
                           "transactions": { ... },
                           "uncleHeaders": { ... }
                   }
           ]
   },
   ...
}
```

Sections

- The `genesisBlockHeader` section

`coinbase`: The 160-bit address to which all fees collected from the successful mining of this block be transferred, as returned by the **COINBASE** instruction.

`difficulty`: A scalar value corresponding to the difficulty level of this block. This can be alculated from the previous block's difficulty level and the timestamp, as returned by the **DIFFICULTY** instruction.

`gasLimit`: A scalar value equal to the current limit of gas expenditure per block, as returned by the **GASLIMIT** instruction.

`number`: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero.

`timestamp`: A scalar value equal to the reasonable output of Unix's time() at this block's inception, as returned by the **TIMESTAMP** instruction.

`parentHash`: The Keccak 256-bit hash of the parent block's header, in its entirety

`bloom`: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list.

`extraData`: An arbitrary byte array containing data relevant to this block. This must be 1024 bytes or fewer.

`gasUsed`: A scalar value equal to the total gas used in transactions in this block.

`nonce`: A 256-bit hash which proves that a sufficient amount of computation has been carried out on this block.

`receiptTrie`: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block.

stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied.

transactionsTrie: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block.

uncleHash: The Keccak 256-bit hash of the uncles list portion of this block

- `pre` section: as described in State Tests.
- `postState` section: as described in State Tests (section - post).
- `blocks` section is a list of block objects, which have the following format:
- `rlp` section contains the complete rlp of the new block as described in the yellow paper in section 4.3.3.
- `blockHeader` section describes the block header of the new block in the same format as described in *genesisBlockHeader*.
- `transactions` section is a list of transactions which have the same format as in Transaction Tests.
- `uncleHeaders` section is a list of block headers which have the same format as descibed in *genesisBlockHeader*.

Optional BlockHeader Sections (Information fields)

`"blocknumber" = "int"` is section which defines what is the order of this block. It is used to define a situation when you have 3 blocks already imported but then it comes new version of the block 2 and 3 and thus you might have new best blockchain with blocks 1 2' 3' instead previous. If *blocknumber* is undefined then it is assumed that blocks are imported one by one. When running test, this field could be used for information purpose only.

`"chainname" = "string"` This is used for defining forks in the same test. You could mine blocks to chain "A": 1, 2, 3 then to chain "B": 1, 2, 3, 4 (chainB becomes primary). Then again to chain "A": 4, 5, 6 (chainA becomes primary) and so on. chainname could also be defined in uncle header section. If defined in uncle header it tells on which chain's block uncle header would be populated from. When running test, this field could be used for information purpose only.

`"chainnetwork" = "string"` Defines on which network rules this block was mined. (see the difference https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki). When running test, this field could be used for information purpose only.

State Tests

Found in `/StateTest`, the state tests aim is to test the basic workings of the state in isolation.

It is based around the notion of executing a single transaction, described by the `transaction` portion of the test. The overarching environment in which it is executed is described by the `env` portion of the test and includes attributes of the current and previous blocks. A set of pre-existing accounts are detailed in the `pre` portion and form the world state prior to execution. Similarly, a set of accounts are detailed in the `post` portion to specify the end world state. Since the data of the blockchain is not given, the opcode `BLOCKHASH` could not return the hashes of the corresponding blocks. Therefore we define the hash of block number `n` to be `SHA256("n")`.

The log entries (`logs`) as well as any output returned from the code (`output`) is also detailed.

It is generally expected that the test implementer will read `env`, `transaction` and `pre` then check their results against `logs`, `out`, and `post`.

Basic structure

```
{
   "test name 1": {
           "env": { ... },
           "logs": { ... },
           "out": { ... },
           "post": { ... },
           "pre": { ... },
           "transaction": { ... },
   },
   "test name 2": {
           "env": { ... },
           "logs": { ... },
           "out": { ... },
           "post": { ... },
           "pre": { ... },
           "transaction": { ... },
   },
   ...
}
```

Sections

- **The `env` section:**

`currentCoinbase`
: The current block's coinbase address, to be returned by the *COINBASE* instruction.

`currentDifficulty`
: The current block's difficulty, to be returned by the *DIFFICULTY* instruction.

`currentGasLimit`
: The current block's gas limit.

`currentNumber`
: The current block's number. Also indicates network rules for the transaction. Since blocknumber = **1000000** Homestead rules are applied to transaction. (see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki)

`currentTimestamp`
: The current block's timestamp.

`previousHash`
: The previous block's hash.

- **The `transaction` section:**

`data`
: The input data passed to the execution, as used by the *CALLDATA*... instructions. Given as an array of byte values. See $DATA_ARRAY.

`gasLimit`
: The total amount of gas available for the execution, as would be returned by the *GAS* instruction were it be executed first.

`gasPrice`
: The price of gas for the transaction, as used by the *GASPRICE* instruction.

`nonce`
: Scalar value equal to the number of transactions sent by the sender.

`address`

The address of the account under which the code is executing, to be returned by the *ADDRESS* instruction.

`secretKey`
: The secret key as can be derived by the v,r,s values if the transaction.

`to`
: The address of the transaction's recipient, to be returned by the *ORIGIN* instruction.

`value`
: The value of the transaction (or the endowment of the create), to be returned by the *CALLVALUE*' instruction (if executed first, before any *CALL*).

- **The `pre` and `post` sections each have the same format of a mapping between addresses and accounts. Each account has the format:**

`balance`
: The balance of the account.

`nonce`
: The nonce of the account.

`code`
: The body code of the account, given as an array of byte values. See $DATA_ARRAY.

`storage`
: The account's storage, given as a mapping of keys to values. For key used notion of string as digital or hex number e.g: `"1200"` or `"0x04B0"` For values used $DATA_ARRAY.

The `logs` sections is a mapping between the blooms and their corresponding logentries.
Each logentry has the format:
`address` The address of the logentry.
`data` The data of the logentry.
`topics` The topics of the logentry, given as an array of values.

Finally, there is one simple key `output`

`output`
The data, given as an array of bytes, returned from the execution (using the `RETURN` instruction). See $DATA_ARRAY. In order to avoid big data files, there is one exception. If the output data is prefixed with `#`, the following number represents the size of the output, and not the output directly.

$DATA_ARRAY - type that intended to contain raw byte data and for convenient of the users is populated with three types of numbers, all of them should be converted and concatenated to a byte array for VM execution.

The types are:

1. number - (unsigned 64bit)
2. "longnumber" - (any long number)
3. "0xhex_num" - (hex format number)

e.g: ````[1, 2, 10000, "0xabc345dFF", "1999999999999999999999999999999999999999"]````

RLP Tests

Describes an **RLP** (https://github.com/ethereum/wiki/wiki/RLP) encoding using the .json file. The client should read the rlp byte stream, **decode** and check whether the contents match it's json representation. Then it should try do it reverse - **encode** json rlp representation into rlp byte stream and check whether it matches the given rlp byte stream.

If it is an invalid RLP byte stream in the test, then 'in' field would contain string 'INVALID'

Some RLP byte streams are expected to be generated by fuzz test suite. For those examples 'in' field would contain string 'VALID' as it means that rlp should be easily decoded.

RLP tests are located in in `/RLPTests`

Note that RLP tests are testing a single RLP object encoding. Not a stream of RLP objects in one array.

Basic structure

```
{
   "rlpTest": {
        "in": "dog",
        "out": "83646f67"
   },

   "multilist": {
        "in": [ "zw", [ 4 ], 1 ],
        "out": "c6827a77c10401"
   },

   "validRLP": {
        "in": "VALID",
        "out": "c7c0c1c0c3c0c1c0"
   },

   "invalidRLP": {
        "in": "INVALID",
        "out": "bf0f000000000000021111"
   },
   ...
}
```

Sections

- `in` - json object (array, int, string) representation of the rlp byte stream (*except values 'VALID' and 'INVALID')
- `out` - string of rlp bytes stream

Difficulty Tests

Found in `\Basic Tests\difficulty*.json` files. This tests are designed to just check the difficulty formula of a block.

difficulty = DIFFICULTY(currentBlockNumber, currentTimestamp, parentTimestamp, parentDifficulty)

described at [EIP2](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki) point 4 with homestead changes.

So basically this .json tests are just to check how this function is calculated on different function parameters (parentDifficulty, currentNumber) in it's extremum points.

There are several test files:

difficulty.json Normal Frontier/Homestead chain difficulty tests defined manually

difficultyFrontier.json Same as above, but auto-generated tests

difficultyMorden.json Tests for testnetwork difficulty. (it has different homestead transition block)

difficultyOlimpic.json Olympic network. (no homestead)

difficultyHomestead.json Tests for homestead difficulty (regardless of the block number)

difficultyCustomHomestead.json Tests for homestead difficulty (regardless of the block number)

Basic structure

```
{
        "difficultyTest" : {
                "parentTimestamp" : "42",
                "parentDifficulty" : "1000000",
                "currentTimestamp" : "43",
                "currentBlockNumber" : "42",
                "currentDifficulty" : "1000488"
        }
}
```

Sections

- `parentTimestamp` - indicates the timestamp of a previous block
- `parentDifficulty` - indicates the difficulty of a previous block
- `currentTimestamp` - indicates the timestamp of a current block
- `currentBlockNumber` - indicates the number of a current block (previous block number = currentBlockNumber - 1)
- `currentDifficulty` - indicates the difficulty of a current block

Transaction Tests

Describes a complete transaction and its RLP representation using the .json file. The client should read the rlp and check whether the transaction is valid, has the correct sender and corresponds to the transaction parameters. If it is an invalid transaction, the transaction and the sender object will be missing.

Basic structure

```
{
   "transactionTest1": {
          "rlp" : "bytearray",
          "sender" : "address",
          "blocknumber" : "1000000"
          "transaction" : {
                        "nonce" : "int",
                        "gasPrice" : "int",
                        "gasLimit" : "int",
                        "to" : "address",
                        "value" : "int",
```

```
                        "v" : "byte",
                        "r" : "256 bit unsigned int",
                        "s" : "256 bit unsigned int",
                        "data" : "byte array"
        }
    },

    "invalidTransactionTest": {
        "rlp" : "bytearray",
    },
    ...
}
```

Sections

- `rlp` - RLP encoded data of this transaction
- `transaction` - transaction described by fields
- `nonce` - A scalar value equal to the number of transactions sent by the sender.
- `gasPrice` - A scalar value equal to the number of Wei to be paid per unit of gas.
- `gasLimit` - A scalar value equal to the maximum amount of gas that should be used in executing this transaction.
- `to` - The 160-bit address of the message call's recipient or empty for a contract creation transaction.
- `value` - A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account.
- `v, r, s` - Values corresponding to the signature of the transaction and used to determine the sender of the transaction.
- `sender` - the address of the sender, derived from the v,r,s values.
- `blocknumber` - indicates network rules for the transaction. Since blocknumber = **1000000** Homestead rules are applied to transaction. (see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki)

VM Tests

Found in `/VMTest`, the VM tests aim is to test the basic workings of the VM in isolation. This is specifically not meant to cover transaction, creation or call processing, or management of the state trie. Indeed at least one implementation tests the VM without calling into any Trie code at all.

It is based around the notion of executing a single piece of code as part of a transaction, described by the `exec` portion of the test. The overarching environment in which it is executed is described by the `env` portion of the test and includes attributes of the current and previous blocks. A set of pre-existing accounts are detailed in the `pre` portion and form the world state prior to execution. Similarly, a set of accounts are detailed in the `post` portion to specify the end world state.

The gas remaining (`gas`), the log entries (`logs`) as well as any output returned from the code (`output`) is also detailed.

Because the data of the blockchain is not given, the opcode BLOCKHASH could not return the hashes of the corresponding blocks. Therefore we define the hash of block number n to be SHA3-256("n").

Since these tests are meant only as a basic test of VM operation, the `CALL` and `CREATE` instructions are not actually executed. To provide the possibility of testing to guarantee they were actually run at all, a separate portion `callcreates` details each `CALL` or `CREATE` operation in the order they would have been executed. Furthermore, gas required is simply that of the VM execution: the gas cost for transaction processing is excluded.

It is generally expected that the test implementer will read `env`, `exec` and `pre` then check their results against `gas`, `logs`, `out`, `post` and `callcreates`. If an exception is expected, then latter sections are absent in the test. Since the reverting of the state is not part of the VM tests.

Basic structure

```
{
   "test name 1": {
           "env": { ... },
           "pre": { ... },
           "exec": { ... },
           "gas": { ... },
           "logs": { ... },
           "out": { ... },
           "post": { ... },
           "callcreates": { ... }
   },
   "test name 2": {
           "env": { ... },
           "pre": { ... },
           "exec": { ... },
           "gas": { ... },
           "logs": { ... },
           "out": { ... },
           "post": { ... },
           "callcreates": { ... }
   },
   ...
}
```

Sections

The `env` section:

- `currentCoinbase`: The current block's coinbase address, to be returned by the `COINBASE` instruction.
- `currentDifficulty`: The current block's difficulty, to be returned by the `DIFFICULTY` instruction.
- `currentGasLimit`: The current block's gas limit.
- `currentNumber`: The current block's number.
- `currentTimestamp`: The current block's timestamp.
- `previousHash`: The previous block's hash.

The `exec` section:

- `address`: The address of the account under which the code is executing, to be returned by the `ADDRESS` instruction.
- `origin`: The address of the execution's origin, to be returned by the `ORIGIN` instruction.
- `caller`: The address of the execution's caller, to be returned by the `CALLER` instruction.
- `value`: The value of the call (or the endowment of the create), to be returned by the `CALLVALUE` instruction.
- `data`: The input data passed to the execution, as used by the `CALLDATA`... instructions. Given as an array of byte values. See $DATA_ARRAY.
- `code`: The actual code that should be executed on the VM (not the one stored in the state(address)) . See $DATA_ARRAY.
- `gasPrice`: The price of gas for the transaction, as used by the `GASPRICE` instruction.

- `gas`: The total amount of gas available for the execution, as would be returned by the `GAS` instruction were it be executed first.

The `pre` and `post` sections each have the same format of a mapping between addresses and accounts. Each account has the format:

- `balance`: The balance of the account.
- `nonce`: The nonce of the account.
- `code`: The body code of the account, given as an array of byte values. See $DATA_ARRAY.
- `storage`: The account's storage, given as a mapping of keys to values. For key used notion of string as digital or hex number e.g: `"1200"` or `"0x04B0"` For values used $DATA_ARRAY.

The `callcreates` section details each `CALL` or `CREATE` instruction that has been executed. It is an array of maps with keys:

- `data`: An array of bytes specifying the data with which the `CALL` or `CREATE` operation was made. In the case of `CREATE`, this would be the (initialisation) code. See $DATA_ARRAY.
- `destination`: The receipt address to which the `CALL` was made, or the null address (`"0000..."`) if the corresponding operation was `CREATE`.
- `gasLimit`: The amount of gas with which the operation was made.
- `value`: The value or endowment with which the operation was made.

The `logs` sections is a mapping between the blooms and their corresponding logentries. Each logentry has the format:

- `address`: The address of the logentry.
- `data`: The data of the logentry.
- `topics`: The topics of the logentry, given as an array of values.

Finally, there are two simple keys, `gas` and `output`:

- `gas`: The amount of gas remaining after execution.
- `output`: The data, given as an array of bytes, returned from the execution (using the `RETURN` instruction). See $DATA_ARRAY.

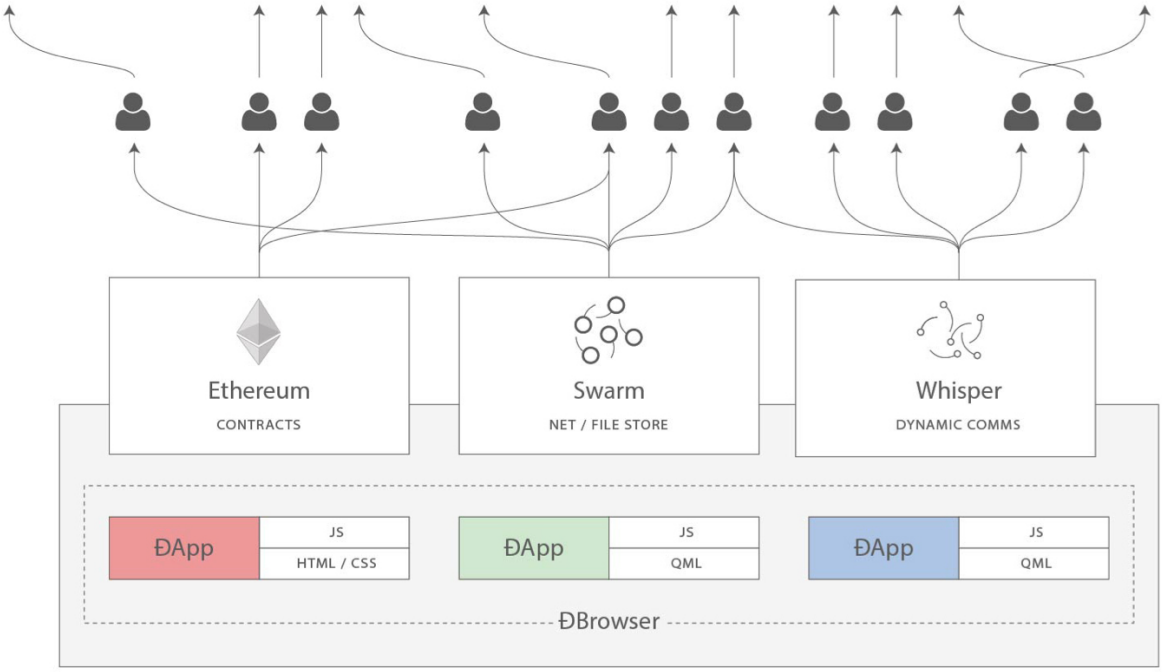
  $DATA_ARRAY - type that intended to contain raw byte data and for convenient of the users is populated with three types of numbers, all of them should be converted and concatenated to a byte array for VM execution.

- The types are: 1. number - (unsigned 64bit) 2. "longnumber" - (any long number) 3. "0xhex_num" - (hex format number)

  e.g: ````[1, 2, 10000, "0xabc345dFF", "19999999999999999999999999999999999999"]``

1.7.8 Web3 Base Layer Services

In addition to the Ethereum blockchain, more components are being developed that decentralise other important aspects of web applications.



Swarm - Decentralised data storage and distribution

Swarm is a peer to peer data sharing network in which files are addressed by the hash of their content. Similar to Bittorrent, it is possible to fetch the data from many nodes at once and as long as a single node hosts a piece of data, it will remain accessible everywhere. This approach makes it possible to distribute data without having to host any kind of server - data accessibility is location independent.

Other nodes in the network can be incentivised to replicate and store the data themselves, obviating the need for hosting services when the original nodes are not connected to the network.

Whisper - Decentralised messaging

A protocol for private, secure communication directly between nodes.

Furthermore, standard contracts are being created to make the development and usage of distributed applications easier:

Name registry

Because dapps can be stored anywhere, including the Swarm network, the name registry maps names to their content or location. This is a decentralised alternative to the Domain Name System (DNS).

See https://github.com/ethereum/EIPs/issues/26

Contract registry

To publish the source code of a specific contract, its address has to be mapped to it. The contract registry stores this mapping. Users can then look up this mapping and verify the contract byte code.

See * global registrar code * namereg API

1.8 Frequently Asked Questions

1.8.1 Questions

What is Ethereum?

Ethereum is a decentralized smart contracts platform that is powered by a cryptocurrency called Ether. A good starting point to learn more about it's workings would be the "*What is Ethereum?*" page.

I have heard of Ethereum, but what are Geth, Mist, Ethminer, Mix?

- **Geth**: This is the Go implementation of an Ethereum node, and is the basis for any interactions with the Ethereum blockchain. Running this locally will allow you to easily interact with the Ethereum blockchain. Read the go-ethereum installation instructions.
- **Mist**: This is the equivalent of a web browser, but for the Ethereum platform. It acts as a GUI to display the accounts and contracts that you interact with. It also allows you to create and interact with contracts in a graphical user interface without ever touching the command line. If you are not a developer and just want to store Ether and interact with Ethereum contracts, then Mist is the program to use. Downloads can be found on the Mist releases page.
- **Ethminer**: A standalone miner. This can be used to mine or benchmark a mining set-up. It is compatible with eth, geth, and pyethereum. Check out the :ref: *mining* page for more information.
- **Mix**: The integrated development environment for DApp authoring. Quickly prototype and debug decentralised applications on the Ethereum platform. More information can be found at the Mix GitHub Page.

How can I store big files on the blockchain?

In general you do not want to store large files or pieces of data in the Ethereum blockchain because of the high cost of storage. You will need to use a third party storage solution, such as Swarm or IPFS. Swarm is an Ethereum-

specific project for distributed file storage. IPFS is an non-Ethereum project which has close ties to Ethereum; it will be used independently and may be used as an added layer underlying Swarm in the future. See this Ethereum StackExchange post on the topic for more information.

Is Ethereum based on Bitcoin?

Only in the sense that it uses a blockchain, which Bitcoin pioneered. Ethereum has a separate blockchain that has several significant technical differences from Bitcoin's blockchain. See this Ethereum StackExchange answer for a detailed explanation.

What's the future of Ethereum?

Ethereum developers are planning a switch from a Proof-of-Work consensus model to a Proof-of-Stake consensus model in the future. They are also investigating scalability solutions and how to store secrets on the blockchain.

What's the difference between account and "wallet contract"?

An account is your public / private key pair file that serves as your identity on the blockchain. See "account" in the glossary. A "wallet contract" is an Ethereum contract that secures your ether and identity with features such as multisignature signing and programmed deposit/withdrawal limits. A wallet contract can be easily created in the Mist Ethereum Wallet GUI client.

Are keyfiles only accessible from the computer you downloaded the client on?

No, you are welcome to export or move the keyfile, but always remember to backup your keyfiles and be aware of which computers you store your keyfile on.

How long should it take to download the blockchain?

The Ethereum blockchain is constantly growing, and is nearing 10GB as of March 2016. The amount of time it will take to download depends on the amount of peers you are able to connect to, your internet connection speed, and other factors. See the :ref: *download-the-blockchain-faster* section for tips on syncing the blockchain more quickly.

How do I get a list of transactions into/out of an address?

You would have to pull the transactions manually out of the blockchain to achieve this. Alternatively, you can rely on third party explorers' API's like Etherchain. For contract execution transactions however, you can filter the contract logs to achieve this.

Can a contract pay for its execution?

No this is not possible. The gas for the execution must be provided by the address submitting the execution request.

Can a contract call another contract?

Yes, this is possible, read about interactions between contracts.

Can a transaction be signed offline and then submitted on another online device?

Yes, you can refer to the solution from Icebox.

How to get testnet Ether?

See :ref: *test-networks*.

Can a transaction be sent by a third party? i.e can transaction broadcasting be outsourced

Technically yes, but there is an important restriction as opposed to bitcoin signed transactions: in ethereum the transaction has a nonce (more precisely, each account increases a counter when sending a transaction based on how many transactions total have been sent. If 3 transactions have ever been sent from the account, the account nonce would be 3).

Can Ethereum contracts pull data using third-party APIs?

No, Ethereum contracts cannot pull data from external information sources in this way. It is however possible to push data from external sites (e.g. weather sites, stock prices) to Ethereum contracts through transactions. There are "oracle" services that are compatible with the Ethereum network that will pull/push data to the Ethereum network for a fee.

Is the content of the data and contracts sent over the Ethereum network encrypted?

Data and contracts on the Ethereum network are encoded, but not encrypted. Everyone can audit the behavior of the contracts and the data sent to them. However, you are always free to encrypt data locally before broadcasting it to the network.

Can I store secrets or passwords on the Ethereum network?

All data on Ethereum is public. It is not possible to store secrets or passwords in Ethereum contracts without it being seen by all. There is work being done to make this a possibility through code obfuscation and other techniques. A good read would be this article by Vitalik Buterin.

How will Ethereum combat centralisation of mining pools?

There are two primary ways that the Ethereum PoW based consensus algorithm combats mining centralisation (Source).

- The first is by reducing losses due to orphaned blocks, which independent miners are more likely to experience.
  - This portion of the Ethereum mining algorithm, a technique referred to as GHOST, includes the headers only of recently orphaned blocks in return for a reduced reward to both the block producer and the includer of the (otherwise orphaned) block. These included orphans from "grandparent" or earlier blocks are frequently referred to as "uncle" blocks because the gender neutral term "ommer" isn't widely known or understood.
- The second way that the Ethereum PoW consensus algorithm combats mining centralisation is by its use of a Proof of Work function that is ASIC resistant.
  - By preventing mining from becoming dominated by specially designed and produced hardware, independent miners are kept competitive or even given an advantage in terms of their profits and/or levels of hardware investment, because they can make use of readily available commodity hardware (i.e. consumer graphics cards).

How will Ethereum deal with ever increasing blockchain size?

There are many discussions around blockchain scalability. This questioned has been partially answered on this Ethereum StackExchange post and this blog post from Vitalik Buterin.

How will Ethereum ensure the network is capable of making 10,000+ transactions-per-second?

Ethereum is planning on implementing a proof-of-stake consensus protocol change during the Serenity phase of their development roadmap. More information on the likely Ethereum PoS candidate and how it may increase transactions-per-second can be found here.

Where do the contracts reside?

TODO

Your question is still not answered?

Ask the community on Ethereum StackExchange.

1.9 Glossary

Ð Ð, D with stroke, is used in Old English, Middle English, Icelandic, and Faroese to stand for an uppercase letter "Eth". It is used in words like ÐEV or Ðapp (decentralized application), where the Ð is the Norse letter "eth". The uppercase eth (Ð) is also used to symbolize the cryptocurrency Dogecoin.

decentralized application (= *dapp*) Service that operates without a central trusted party. An application that enables direct interaction/agreements/communication between end users and/or resources without a middleman. See *Dapps*.

DAO decentralized autonomous organization DAO is type of contract on the blockchain (or a suite of contracts) that is supposed to codify, enforce or automate the workings of an organization including governance, fund-raising, operations, spending and expansion.

identity A set of cryptographically verifiable interactions that have the property that they were all created by the same person.

digital identity The set of cryptographically verifiable transactions signed by the same public key define the digital identity's behavior. In many real world scenarios (voting) it is desireable that digital identities coincide with real world identities. Ensuring this without violence is an unsolved problem.

unique identity A set of cryptographically verifiable interactions that have the property that they were all created by the same person, with the added constraint that one person cannot have multiple unique identities.

reputation The property of an identity that other entities believe that identity to be either (1) competent at some specific task, or (2) trustworthy in some context, i.e., not likely to betray others even if short-term profitable.

escrow If two mutually-untrusting entities are engaged in commerce, they may wish to pass funds through a mutually trusted third party and instruct that party to send the funds to the payee only when evidence of product delivery has been shown. This reduces the risk of the payer or payee committing fraud. Both this construction and the third party is called escrow.

deposit Digital property placed into a contract involving another party such that if certain conditions are not satisfied that property is automatically forfeited and either credited to a counterparty as insurance against the conditions, or destroyed (= burnt = equally distributed) or donated to some charitable funds.

web of trust The idea that if A highly rates B, and B highly rates C, then A is likely to trust C. Complicated and powerful mechanisms for determining the reliability of specific individuals in specific concepts can theoretically be gathered from this principle.

incentive compatibility A protocol is incentive-compatible if everyone is better off "following the rules" than attempting to cheat, at least unless a very large number of people agree to cheat together at the same time (collusion).

collusion In an incentivized protocol scenario, when a number of participants *play together* (conspire) to game the rules to their own benefit.

token system A fungible virtual good that can be traded. More formally, a token system is a database mapping addresses to numbers with the property that the primary allowed operation is a transfer of N tokens from A to B, with the conditions that N is non-negative, N is not greater than A's current balance, and a document authorizing the transfer is digitally signed by A. Secondary "issuance" and "consumption" operations may also exist, transaction fees may also be collected, and simultaneous multi-transfers with many parties may be possible. Typical use cases include currencies, cryptographic tokens inside of networks, company shares and digital gift cards.

block A block is a package of data that contains zero or more transactions, the hash of the previous block ("parent"), and optionally other data. The total set of blocks, with every block except for the initial "genesis block" containing the hash of its parent, is called the blockchain and contains the entire transaction history of a network. Note that some blockchain-based cryptocurrencies use the word "ledger" instead of blockchain; the two are roughly equivalent, although in systems that use the term "ledger" each block generally contains a full copy of the current state (e.g. currency balances, partially fulfilled contracts, registrations) of every account allowing users to discard outdated historical data.

dapp Đapp Stands for "decentralized application". Some say it is pronounced Ethapp due to the use of the uppercase eth letter Đ.

address An Ethereum address represents an account. For *EOA*, the address is derived as the last 20 bytes of the public key controlling the account, e.g., `cd2a3d9f938e13cd947ec05abc7fe734df8dd826`. This is a *hexadecimal* format (base 16 notation), which is often indicated explicitly by appending `0x` to the address. Web3.js and console functions accept addresses with or without this prefix but for transparency we encourage their use. Since each byte of the address is represented by 2 hex characters, a prefixed address is 42 characters long. Several apps and APIs are also meant to implement the new checksum-enabled address scheme introduced in the Mist Ethereum wallet as of version 0.5.0.

hexadecimal Common representation format for byte sequencing. Its advantage is that values are represented in a compact format using two characters per byte (the characters `[0-9][a-f]`).

ether Ether is the name of the currency used within Ethereum. It is used to pay for computations within the EVM. Ambiguously, ether is also the name of a unit in the system;

EOA Externally Owned Account. An account controlled by a private key. If you own the private key associated with the EOA you have the ability to send ether and messages from it. Contract accounts also have an address, see *Accounts*. EOAs and contract accounts may be combined into a single account type during Serenity.

gas Name for the *cryptofuel* that is consumed when code is executed by the EVM. The gas is paid for execution fee for every operation made on an Ethereum blockchain.

gas limit Gas limit can apply to both individual transactions, see *transaction gas limit* and to blocks, *block-gas-limit*. For individual transactions, the gas limit represents the maximum amount of gas you indicate you are willing to pay for a contract execution transaction. It is meant to protect users from getting their ether depleted when trying to execute buggy or malicious contracts. The block gas limit represents the maximum cumulative gas used for all the transactions in a block. With the launch of Homestead, the block gas limit floor will increase from 3,141,592 gas to 4,712,388 gas (~50% increase).

gas price Price in ether of one unit of gas specified in a transaction. With the launch of Homestead, the default gas price reduces from 50 shannon to 20 shannon (~60% reduction).

transaction The signed data package that stores a message to be sent from an externally owned account. Simply put, a transaction describes a transfer of information from an EOA to another

EOA or a contract account.

message A data transfer mechanism contracts use to communicate with other contracts. Messages can also be described as virtual objects that are never serialized and exist only in the Ethereum execution environment.

Web3 The exact definition of the Web3 paradigm is still taking form, but it generally refers to the phenomenon of increased connectedness between all kinds of devices, decentralization of services and applications, semantic storage of information online and application of artificial intelligence to the web.

DAO See Decentralized Autonomous Organization.

epoch Epoch is the interval between each regeneration of the DAG used as seed by the PoW algorithm Ethash. The epoch in specified as 30000 blocks.

elliptic curve (cryptography) Refers to an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. See elliptic curve cryptography.

wallet A wallet, in the most generic sense, refers to anything that can store ether or any other crypto token. In the crypto space in general, the term wallet is used to mean anything from a single private/public key pair (like a single paper wallet) all the way to applications that manage multiple key pairs, like the Mist Ethereum wallet.

contract A persistent piece of code on the Ethereum blockchain that encompasses a set of data and executable functions. These functions execute when Ethereum transactions are made to them with certain input parameters. Based on the input parameters, the functions will execute and interact with data within and outside of the contract.

suicide See self-destruct. `selfdestruct` acts as an alias to the deprecated `suicide` terminology in accordance with EIP 6 - Renaming SUICIDE OPCODE.

selfdestruct A global variable in the Solidity language that allows you to "destroy the current contract, sending its funds to the given address". `selfdestruct` acts as an alias to the deprecated `suicide` terminology in accordance with EIP 6 - Renaming SUICIDE OPCODE. It frees up space on the blockchain and prevents future execution of the contract. The contract's address will still persist, but ether sent to it will be lost forever. The possibility to kill a contract has to be implemented by the contract creator him/herself using the Solidity `selfdestruct` function.

transaction fee Also known as gas cost, it is the amount of ether that the miners will charge for the execution of your transaction.

mining The process of verifying transactions and contract execution on the Ethereum blockchain in exchange for a reward in ether with the mining of every block.

mining pool The pooling of resources by miners, who share their processing power over a network, to split the reward equally, according to the amount of work they contributed to solving a block.

mining reward The amount of cryptographic tokens (in this case ether) that is given to the miner who mined a new block.

state Refers to a snapshot of all balances and data at a particular point in time on the blockchain, normally referring to the condition at a particular block.

blockchain An ever-extending series of data blocks that grows as new transactions are confirmed as part of a new block. Each new block is chained to the existing blockchain by a cryptographic proof-of-work.

peer Other computers on the network also running an Ethereum node (Geth) with an exact copy of the blockchain that you have.

signing Producing a piece of data from the data to be signed using your private key, to prove that the data originates from you.

discovery (peer) The process of 'gossiping' with other nodes in the network to find out the state of other nodes on the network.

gas price oracle A helper function of the Geth client that tries to find an appropriate default gas price when sending transactions.

light client A client program that allows users in low-capacity environments to still be able to execute and check the execution of transactions without needing to run a full Ethereum node (Geth).

etherbase It is the default name of the account on your node that acts as your primary account. If you do mining, mining rewards will be credited to this account.

coinbase Coinbase is analogous to etherbase, but is a more generic term for all cryptocurrency platforms.

balance The amount of cryptocurrency (in this case) belonging to an account.

solidity Solidity is a high-level language whose syntax is similar to that of JavaScript and it is designed to compile to code for the Ethereum Virtual Machine.

serpent Serpent is a high-level language whose syntax is similar to that of Python and it is designed to compile to code for the Ethereum Virtual Machine.

EVM Ethereum Virtual Machine, the decentralized computing platform which forms the core of the Ethereum platform.

virtual machine In computing, it refers to an emulation of a particular computer system.

peer to peer network A network of computers that are collectively able to perform functionalities normally only possible with centralized, server-based services.

decentralization The concept of moving the control and execution of computational processes away from a central entity.

distributed hash table A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key.

NAT Network address translation (NAT) is a methodology of remapping one IP address space into another by modifying network address information in Internet Protocol (IP) datagram packet headers while they are in transit across a traffic routing device.

nonce Number Used Once or Number Once. A nonce, in information technology, is a number generated for a specific use, such as session authentication. Typically, a nonce is some value that varies with time, although a very large random number is sometimes used. In general usage, nonce means "for the immediate occasion" or "for now." In the case of Blockchain Proof of Work scenarios, the hash value, found by a Miner, matching the network's Difficulty thus proving the Block Validity is called Nonce as well.

proof-of-work Often seen in its abbreviated form "PoW", it refers to a mathematical value that can act as the proof of having solved a resource and time consuming computational problem.

proof-of-stake An alternative method of mining blocks that require miners to demonstrate their possession of a certain amount of the currency of the network in question. This works on the principle that miners will be disincentivized to try to undermine a network in which they have a stake. PoS is less wasteful than PoW, but is still often used together with it to provide added security to the network.

CASPER Casper is a security-deposit based economic consensus protocol. This means that nodes, so called "bonded validators", have to place a security deposit (an action we call "bonding") in order to serve the consensus by producing blocks. If a validator produces anything that Casper considers "invalid", the deposit is forfeited along with the privilege of participating in the consensus process.

consensus The agreement among all nodes in the network about the state of the Ethereum network.

homestead Homestead is the second major version release of the Ethereum platform. Homestead includes several protocol changes and a networking change that makes possible further network upgrades: EIP-2 Main homestead hardfork changes; EIP-7 Hardfork EVM update (DEL-

EGATECALL); EIP-8 devp2p forward compatibility. Homestead will launch when block 1,150,000 is reached on the Mainnet. On the Testnet, Homestead will launch at block 494,000.

metropolis The third stage of Ethereum's release. This is the stage when the user interfaces come out (e.g. Mist), including a dapp store, and non-technical users should feel comfortable joining at this point.

serenity The fourth stage of Ethereum's release. This is when things are going to get fancy: the network is going to change its mining process from Proof-of-Work to Proof-of-Stake.

frontier Ethereum was planned to be released in four major steps with Frontier being the name for the first phase. The Frontier release went live on July 30th, 2015. The command line Frontier phase was mainly meant to get mining operations going with the full reward of 5 ether per block and also to promote the emergence of ether exchanges. Frontier surpassed earlier modest expectations and has nurtured tremendous growth of the ecosystem.

olympic The Frontier pre-release, which launched on May 9th 2015. It was meant for developers to help test the limits of the Ethereum blockchain.

morden Morden is the first Ethereum alternative testnet. It is expected to continue throughout the Frontier and Homestead era.

testnet A mirror network of the production Ethereum network that is meant for testing. See Morden.

private chain A fully private blockchain is a blockchain where write permissions are kept centralized to one organization.

consortium chain A blockchain where the consensus process is controlled by a pre-selected set of nodes.

micropayment A micropayment is a financial transaction involving a very small sum of money (<1 USD) and usually one that occurs online.

sharding The splitting of the space of possible accounts (contracts are accounts too) into subspaces, for example, based on first digits of their numerical addresses. This allows for contract executions to be executed within 'shards' instead of network wide, allowing for faster transactions and greater scalability.

hash A cryptographic function which takes an input (or 'message') and returns a fixed-size alphanumeric string, which is called the hash value (sometimes called a message digest, a digital fingerprint, a digest or a checksum). A hash function (or hash algorithm) is a process by which a document (i.e. a piece of data or file) is processed into a small piece of data (usually 32 bytes) which looks completely random, and from which no meaningful data can be recovered about the document, but which has the important property that the result of hashing one particular document is always the same. Additionally, it is crucially important that it is computationally infeasible to find two documents that have the same hash. Generally, changing even one letter in a document will completely randomize the hash; for example, the SHA3 hash of "Saturday" is `c38bbc8e93c09f6ed3fe39b5135da91ad1a99d397ef16948606cdcbd14929f9d`, whereas the SHA3 hash of "Caturday" is `b4013c0eed56d5a0b448b02ec1d10dd18c1b3832068fbbdc65b98` Hashes are usually used as a way of creating a globally agreed-upon identifier for a particular document that cannot be forged.

crypto-fuel Similar to 'gas', referring to the amount of cryptocurrency required to power a transaction.

cryptoeconomics The economics of cryptocurrencies.

protocol A standard used to define a method of exchanging data over a computer network.

block validation The checking of the coherence of the cryptographic signature of the block with the history stored in the entire blockchain.

blocktime The average time interval between the mining of two blocks.

network hashrate The number of hash calculations the network can make per second collectively.

hashrate The number of hash calculations made per second.

serialization The process of converting a data structure into a sequence of bytes. Ethereum internally uses an encoding format called recursive-length prefix encoding (RLP), described in the RLP section of the wiki.

double spend A deliberate blockchain fork, where a user with a large amount of mining power sends a transaction to purchase some produce, then after receiving the product creates another transaction sending the same coins to themselves. The attacker then creates a block, at the same level as the block containing the original transaction but containing the second transaction instead, and starts mining on the fork. If the attacker has more than 50% of all mining power, the double spend is guaranteed to succeed eventually at any block depth. Below 50%, there is some probability of success, but it is usually only substantial at a depth up to about 2-5; for this reason, most cryptocurrency exchanges, gambling sites and financial services wait until six blocks have been produced ("six confirmations") before accepting a payment.

SPV client A client that downloads only a small part of the blockchain, allowing users of low-power or low-storage hardware like smartphones and laptops to maintain almost the same guarantee of security by sometimes selectively downloading small parts of the state without needing to spend megabytes of bandwidth and gigabytes of storage on full blockchain validation and maintenance. See light client.

uncle Uncles are blockchain blocks found by a miner, when a different miner has already found another block for the corresponding place in the blockchain. They are called "stale blocks". The parent of an Uncle is an ancestor of the inserting block, located at the tip of the blockchain. In contrast to the Bitcoin network, Ethereum rewards stale blocks as well in order to avoid to penalize miners with a bad connection to the network. This is less critical in the Bitcoin network, because the Block Time there is much higher (~10 minutes) than on the Ethereum network (aimed to ~15 seconds).

GHOST Greedy Heaviest-Observed Sub-Tree is an alternative chain-selection method that is designed to incentivize stale blocks (uncles) as well, thus reducing the incentive for pool mining. In GHOST, even the confirmation given by stale blocks to previous blocks are considered valid, and the miners of the stale blocks are also rewarded with a mining reward.

merkle patricia tree Merkle Patricia trees provide a cryptographically authenticated data structure that can be used to store all (key, value) bindings. They are fully deterministic, meaning that a Patricia tree with the same (key,value) bindings is guaranteed to be exactly the same down to the last byte and therefore have the same root hash, provide O(log(n)) efficiency for inserts, lookups and deletes, and are much easier to understand and code than more complex comparison-based alternatives like red-black trees.

DAG DAG stands for Directed Acyclic Graph. It is a graph, a set of nodes and links between nodes, that has very special properties. Ethereum uses a DAG in Ethash, the Ethereum Proof of Work (POW) algorithm.The Ethash DAG takes a long time to be generated, which is done by a Miner node into a cache file for each Epoch. The file data is then used when a value from this graph is required by the algorithm.

uncle rate The number of uncles produced per block.

issuance The minting and granting of new cryptocurrency to a miner who has found a new block.

presale Sale of cryptocurrency before the actual launch of the network.

static node A feature supported by Geth, the Golang Ethereum client, which makes it possible to always connect to specific peers. Static nodes are re-connected on disconnects. For details, see the *section on static nodes*.

bootnode The nodes which can be used to initiate the discovery process when running a node. The endpoints of these nodes are recorded in the Ethereum source code.

exchange An online marketplace which facilitates the exchange of crypto or fiat currencies based on the market exchange rate.

compiler A program that translates pieces of code written in high level languages into low level executable code.

genesis block The first block in a blockchain.

network id A number which identifies a particular version of the Ethereum network.

block header The data in a block which is unique to its content and the circumstances in which it was created. It includes the hash of the previous block's header, the version of the software the block is mined with, the timestamp and the merkle root hash of the contents of the block.

pending transaction A transaction that is not yet confirmed by the Ethereum network.

block propagation The process of transmitting a confirmed block to all other nodes in the network.

sidechain A blockchain that branches off a main blockchain and checks in periodically with the main blockchain. Besides that it runs independently from the main chain, and any security compromises in the sidechain will not affect the main chain.

pegging Locking down the exchange rate of the coins/tokens in two chains (usually a main and a side chain) in a certain direction.

2-way pegging Locking down the exchange rate of the coins/tokens in two chains (usually a main and a side chain) in both directions.

trustless Refers to the ability of a network to trustworthily mediate transactions without any of the involved parties needing to trust anyone else.

faucet A website that dispenses (normally testnet) cryptocurrencies for free.

checksum A count of the number of bits in a transmission that is included with the unit so that the receiving end can verify that the entirety of the message has been transmitted.

ICAP Interexchange Client Address Protocol, an IBAN-compatible system for referencing and transacting to client accounts aimed to streamline the process of transferring funds, worry-free between exchanges and, ultimately, making KYC and AML concerns a thing of the past.

private key A private key is a string of characters known only to the owner, that is paired with a public key to set off algorithms for text encryption and decryption.

public key A string of characters derived from a private key that can be made public. The public key can be used to verify the authenticity of any signature created using the private key.

encryption Encryption is the conversion of electronic data into a form unreadable by anyone except the owner of the correct decryption key. It can further be described as a process by which a document (plaintext) is combined with a shorter string of data, called a key (e.g. `c85ef7d79691fe79573b1a7064c19c1a9819ebdbd1faaab1a8ec92344438aaf4`), to produce an output (ciphertext) which can be "decrypted" back into the original plaintext by someone else who has the key, but which is incomprehensible and computationally infeasible to decrypt for anyone who does not have the key.

digital signature A mathematical scheme for demonstrating the authenticity of a digital message or documents.

port A network port is a communication endpoint used by a one of the existing standards of establishing a network conversation (e.g. TCP, UDP).

RPC Remote Procedure Call, a protocol that a program uses to request a service from a program located in another computer in a network without having to understand the network details.

IPC Interprocess communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system.

attach The command used to initiate the Ethereum Javascript console.

daemon A computer program that runs as a background process instead of in direct control by an interactive user.

system service See base layer service

base layer service Services such as SWARM and Whisper which are built into the Ethereum platform.

js Javascript.

syncing The process of downloading the entire blockchain.

fast sync Instead of processing the entire block-chain one link at a time, and replay all transactions that ever happened in history, fast syncing downloads the transaction receipts along the blocks, and pulls an entire recent state database.

ASIC Application-specific integrated circuit, in this case referring to an integrated circuit custom built for cryptocurrency mining.

memory-hard Memory hard functions are processes that experience a drastic decrease in speed or feasibility when the amount of available memory even slightly decreases.

keyfile Every account's private key/address pair exists as a single keyfile. These are JSON text files which contains the encrypted private key of the account, which can only be decrypted with the password entered during account creation.

ICAP format The format of the IBANs defined using the Inter-exchange Client Address Protocol.

block(chain) explorer A website that allows easy searching and extraction of data from the blockchain.

geth Ethereum client implemented in the Golang programming language, based on the protocol as defined in the Ethereum Yellow Paper.

eth Ethereum client implemented in the C++ programming language, based on the protocol as defined in the Ethereum Yellow Paper.

ethereumjs Ethereum client implemented in the Javascript/Node programming language, based on the protocol as defined in the Ethereum Yellow Paper.

pyethereum Ethereum client implemented in the Python programming language, based on the protocol as defined in the Ethereum Yellow Paper.

ethereumj Ethereum client implemented in the Java programming language, based on the protocol as defined in the Ethereum Yellow Paper.

ethereumh Ethereum client implemented in the Haskell programming language, based on the protocol as defined in the Ethereum Yellow Paper.

parity Ethereum client implemented in the Rust programming language, based on the protocol as defined in the Ethereum Yellow Paper.

difficulty In very general terms, the amount of effort required to mine a new block. With the launch of Homestead, the difficulty adjustment algorithm will change.

account Accounts are a central part of the Ethereum network and are an essential part of any transaction or contract. In Ethereum, there are two types of accounts: Externally Owned accounts (EOA) and Contract accounts.

HLL (obsolete) Acronym for Higher Level Language, which is what Serpent and Solidity are. HLL is what early Đapp developers called Ethereum programming languages that did not touch the low level elements. This phrase has been phased out.

CLL (obsolete) Acronym for C Like Language, which Mutan was. This acronym has been phased out.

ES1, ES2, and ES3 (obsolete) "Ethereum Script" versions 1,2 and 3. There were early versions of what would become the Ethereum Virtual Machine (EVM).

log event Contracts are triggered by transactions executed as part of the block verification. If conceived of as a function call, contract execution is asynchronous, and therefore they have no return value. Instead contracts communicate to the outside world with log events. The log events are part of the transaction receipt which is produced when the transaction is executed. The receipts are stored in the receipt trie, the integrity of which is guaranteed by the fact that the current root

of the receipt trie is part of the block header alongside the roots of state and state-trie. In a broad sense from the external perspective receipts are part of the Ethereum system state except that they are not readable contracts internally.

1.10 The Homestead Documentation Initiative

1.10.1 Purpose and Audience

This guide should serve to be an entry level for all Ethereum users and developers. The goal is to create documentation with information, short tutorials, and examples that will cover all of the basic and intermediate functionality of using Ethereum to interact with dapps or develop a dapp.

Any information that is overly specific, technical, or not necessary to accomplish the documentation's goal will remain on the Ethereum Github Wiki. It may be referenced in this guide if necessary.

Although much of the information will be similar between the Frontier Guide and the Homestead Guide, efforts need to be made to make sure the information ported over is still accurate. This document is client agnostic and examples and tutorials may be based on any client that the author decides to write on, as long as a distinction is made as to what client is being used in the examples/tutorials.

Although overly specific and technical documentation will not be included in the first iterations of this guide, community use and popularity of this guide will dictate future decisions to move Github wiki documentation to this format.

Examples of overly specific and technical documentation include:

- ETHash, CASPER, ABI, RLP, or other technical specs.
- Full API specs for protocols. Caveat: If an example, information, or tutorial needs to reference API calls for a client or interface in order to fulfill its example it is acceptable to reference the specific call. Be sure to make a reference where the user can find remaining pieces of the specific documentation that may be on the GitHub Wiki.

1.10.2 Resources for Exemplary Documentation

Here are some examples of previous Ethereum documentation + good examples of documentation.

- Solidity Docs - https://ethereum.github.io/solidity/docs/home/
- Frontier Guide - https://ethereum.gitbooks.io/frontier-guide/content/
- Gav's TurboEthereum Guide - https://gavofyork.gitbooks.io/turboethereum/content/
- Ancient EthereumBuilder's Guide - https://ethereumbuilders.gitbooks.io/guide/content/en/index.html
- Other Ethereum Links: https://souptacular.gitbooks.io/ethereum-tutorials-and-tips-by-hudson/content/giant_ethereum_resource_list.html
- Django Docs - https://docs.djangoproject.com/en/1.9/

1.10.3 Restructured Text Markup, Sphinx

- Best Cheat Sheet - https://github.com/ralsina/rst-cheatsheet/blob/master/rst-cheatsheet.rst
- Quick Reference - http://docutils.sourceforge.net/docs/user/rst/quickref.html
- Official Cheat Sheet - http://docutils.sourceforge.net/docs/user/rst/cheatsheet.txt -> http://docutils.sourceforge.net/docs/user/rst/cheatsheet.html

- RST Primer http://sphinx-doc.org/rest.html
- http://sphinx-doc.org/markup/inline.html

1.10.4 Compilation and Deployment

We use *make* with the autogenerated read-the-docs *Makefile* to build the doc.

```
git clone https://github.com/ethereum/homestead-guide
cd homestead-guide
make html
```

1.10.5 Processing Tips

Fix section delimiter lines (always use 80-long ones to have correct length, unless the title is greater than 80 chars in length)

```
for f in `ls source/*/*.rst`; do cat $f|perl -pe 's/\=+$/================================================
for f in `ls source/*/*.rst`; do cat $f|perl -pe 's/\*+$/************************************************
for f in `ls source/*/*.rst`; do cat $f|perl -pe 's/\-+$/------------------------------------------------
for f in `ls source/*/*.rst`; do cat $f|perl -pe 's/\^+$/^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
for f in `ls source/*/*.rst`; do cat $f|perl -pe 's/\#+$/################################################
```

1.10.6 Referencing Old Documentation

old-docs-for-reference folder has all of the Frontier Gitbook and Ethereum Wiki doc. Feel free to copy/paste information from those documents that is still relevant.

1.10.7 Migrate and Convert Old Wiki Content Using Pandoc

If you still want to clone the absolute latest Ethereum Wiki and Frontier Guide docs:

```
git clone git@github.com:ethereum/go-ethereum.wiki.git
git clone git@github.com:ethereum/wiki.wiki.git

mkdir main-wiki.rst
mkdir go-ethereum-wiki.rst

for f in `ls wiki.wiki/*.md`; do pandoc $f -o main-wiki.rst/`basename $f .md`.rst; done
for f in `ls go-ethereum.wiki/*.md`; do pandoc $f -o go-ethereum-wiki.rst/`basename $f .md`.rst; d
```

CHAPTER 2

Improve the Documentation

See this page to help us improve the documentation.