

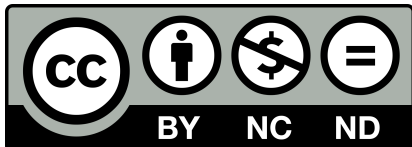
# Bitcoin & Co

Blockchain Technology from First Principles to Business  
Applications, Governance and Unsolved Issues.

Joerg Evermann

Faculty of Business Administration  
Memorial University of Newfoundland  
jevermann@mun.ca

Blockwoche Seminar HS Osnabrück May 2019



These slides are licensed under the creative commons attribution, non-commercial use and no-derivative works license (version 4). The complete text of the license can be found at <http://creativecommons.org>.

## What is a blockchain?

*A system in which a record of transactions made in bitcoin or another cryptocurrency are maintained across several computers that are linked in a peer-to-peer network.*

(Oxford English Dictionary)

## Monday

### Preliminaries

Technology

### Cryptography

The Very Basics

Hashing

Digital Signatures

## Tuesday

### Blockchain

Blocks and Transactions

### Bitcoin

Simple Overview

Bitcoin Details

Bitcoin Hands-On

## Wednesday

### Ethereum Basics

Principles

Ethereum Hands On

Ethereum Web3.js Console

Java Access Using Web3j

### Ethereum Smart Contracts

Smart Contracts Introduction

My First Smart Contract

## Thursday

### Byzantine Fault Tolerance

## Friday

### Project Presentations

# Preliminaries

- ▶ Introductions
- ▶ Schedule
- ▶ Expectations
  - ▶ "Pre-requisites"
  - ▶ Technology setup
- ▶ Presentation Groups and Topics

# Presentation Groups and Topics

- ▶ Groups of 4–5 students
  - ▶ please make sure there's an even mix
- ▶ Topics either technical or business/application oriented
- ▶ Final topics due on Tuesday morning
- ▶ Presentations on Thursday and Friday

# Presentations

- ▶ About 60 to 75 minutes
- ▶ Slides welcome but not required
- ▶ No more than 25 words on a slide: Use pictures, videos, diagrams, etc.
- ▶ You're welcome to include a demo of something practical/applied!
- ▶ Strong preference: Include discussion questions for the group!
- ▶ Use and cite appropriate sources (tiny font at slide bottom is ok)
- ▶ Include a list of approx. 5 resources for further reading
- ▶ Make it funny, make it lively, make it controversial, do something "out of the box"
- ▶ **Bonus points: Dance your blockchain!!**

# Topic Examples

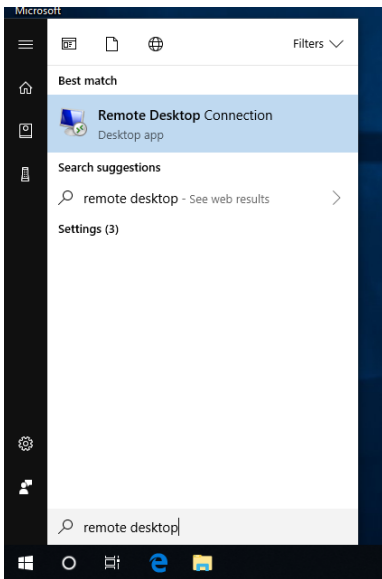
- ▶ Scalability, especially of proof-of-work consensus
- ▶ Security (attack vectors, vulnerabilities, known attacks)
- ▶ Regulation (is it money? securities? ICOs), perhaps with international perspective
- ▶ Inside a blockchain "ecosystem": Coin inflation, governance, mining power, controversies, developments
- ▶ Different public chains and their consensus mechanisms
- ▶ Highly publicized blockchain projects and their status/outcome (in particular industries/applications)
- ▶ ...

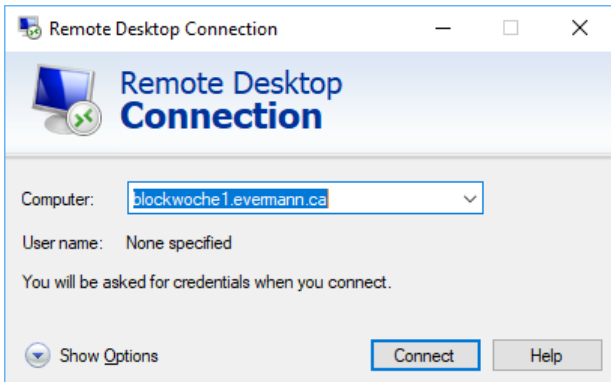


# Technology Setup

- ▶ Computers with software provided for remote access
  - ▶ `blockwoche1.evermann.ca`
  - ▶ `blockwoche2.evermann.ca`
  - ▶ ...
- ▶ Private network
- ▶ Remote access using RDP
  - ▶ Pre-installed on Windows
  - ▶ Free on the Apple AppStore
- ▶ User name: **ubuntu** Password: **password**
- ▶ Only one concurrent login to each computer :-)

# Windows 10







## Remote Desktop Connection



**The identity of the remote computer cannot be verified. Do you want to connect anyway?**

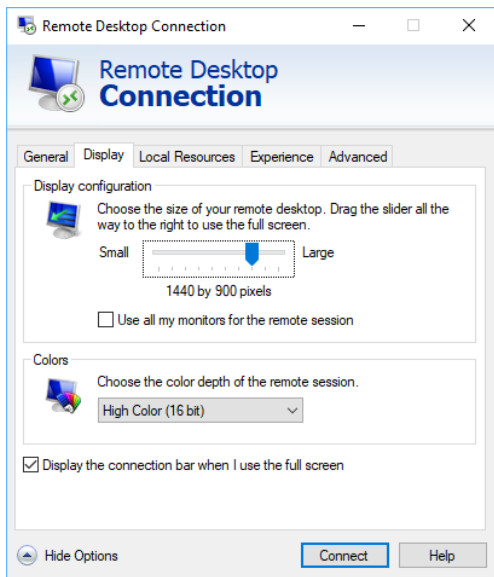
This problem can occur if the remote computer is running a version of Windows that is earlier than Windows Vista, or if the remote computer is not configured to support server authentication.

For assistance, contact your network administrator or the owner of the remote computer.

Don't ask me again for connections to this computer

Yes

No



Set resolution to fit your screen and color to 16 bits

Login to ip:10.0.0.112



Session

username

password





# MacOS

Search: microsoft remote de...

Discover

Create

Work

Play

Develop

Categories

Updates



# Microsoft Remote Desktop 10

Get work done from anywhere.

Microsoft Corporation

GET



2.7 ★★★★★  
70 Ratings

#1  
Business

4+  
Age



Use the new Microsoft Remote Desktop app to connect to a remote PC or virtual apps and desktops made available by your administrator. The app helps you be productive no matter where you are. [more](#)

Microsoft Corporation

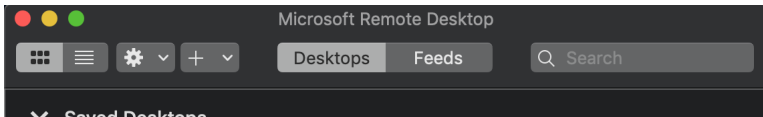
Website

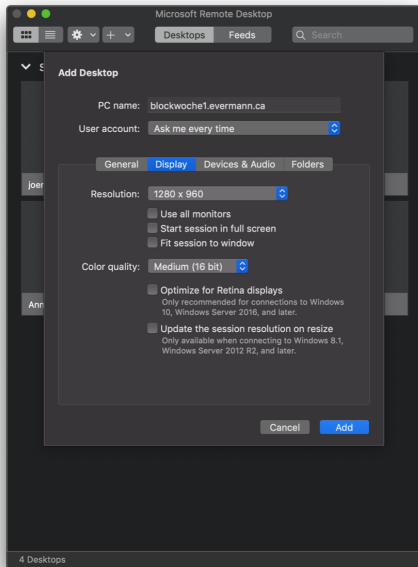
Support

Screenshot

Ratings & Reviews

See All





Set resolution to fit your screen and color to 16 bits

blockwoche1.evermann.ca

Connec

Configu



**The identity of the remote computer  
"blockwoche1.evermann.ca" can't be verified**

This problem can occur if the remote computer is running a version of Windows that is earlier than Windows Vista, or if the remote computer is not configured to support server authentication.

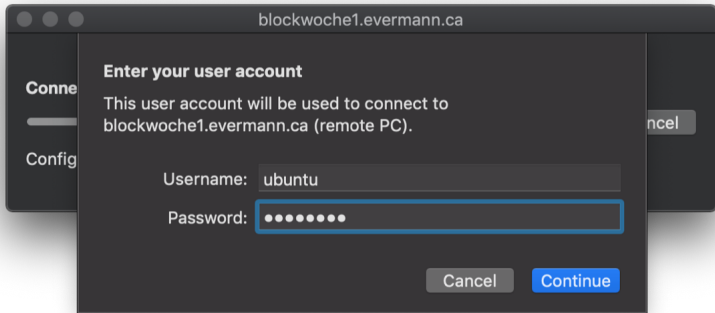
For assistance, contact your network administrator or the owner of the remote computer.

Don't ask me for connections to this computer

Cancel

Continue

ancel





# Basics

## Bits and Bytes

- ▶ Computers represent information in the binary system (base 2)
- ▶ Each digit ("bit") represents a power of 2
  - ▶ From the right:  $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 16$ , etc.
  - ▶ Example:  $00100110_2 = 38_{10}$
- ▶ Base 16 is popular too, called "hexadecimal" or just "hex"
- ▶ Uses digits 0 – 9, A – E, each represents a power of 16
  - ▶ From right:  $16^0 = 1, 16^1 = 16, 16^2 = 256, 16^3 = 4096$ , etc.
  - ▶ Ex:  $2B4E_{16} = 0x2B4E = 11086_{10} = 00101011\ 01001110_2$
- ▶ A byte is made of 8 bits
- ▶ Two hex digit represent a byte
- ▶ **Use a calculator to convert!**



# Basics

## Integer Division and Modulo Arithmetic

- ▶ `div` or `/`: Integer quotient
  - ▶  $7 \text{ div } 2 = 3$
- ▶ `mod` or `%`: Integer modulus (remainder)
  - ▶  $7 \text{ mod } 2 = 1$

# Basics

## Bit operations

- ▶ XOR: "Either or" (but not both)
  - ▶  $0 \text{ XOR } 0 = 0$ ,  $1 \text{ XOR } 0 = 1$ ,  $0 \text{ XOR } 1 = 1$ ,  $1 \text{ XOR } 1 = 0$
- ▶  $\ll$ : "left shift" is multiplying by powers of two
  - ▶  $00100110_2 \ll 2 = 10011000_2 = 152_{10} = 4_{10} \times 38_{10}$
- ▶  $\gg$ : "right shift" is integer division by powers of two
  - ▶  $00100110_2 \gg 2 = 00001001_2 = 9_{10} = 38_{10} \text{ div } 4_{10}$
- ▶  $\lll$ : "left shift" with rotate
- ▶  $\ggg$ : "right shift" with rotate

# Hashing

# Hash Function

- ▶ Maps arbitrary sized data onto a fixed size.
- ▶ Returns *hash values*, *hash codes*, *digests*
- ▶ Used in
  - ▶ Information lookup (hash tables)
  - ▶ Duplicate detection (dedup)
  - ▶ **Protecting data integrity**

# Very Simple Hash Function Example

## Mapping Numbers to {"A", "B"}

1. `Random("A", "B")`
2. If  $(n < 1000) \rightarrow$  "A" else "B"

## Mapping Numbers to Numbers

1.  $n \rightarrow n \times 2$

What is wrong with these hash functions?

# Hash Function Properties

## Good Hash Functions

- ▶ Deterministic
  - ▶ Same input leads to same output
- ▶ Uniform output distribution
- ▶ Low probability of *collisions*
  - ▶ Different input leads to different output
- ▶ **Non-invertible**
  - ▶ Cannot (easily) get input from output ("one-way")

# Let's Go Play

## Exploring Hash Functions in Java

- ▶ Connect to your remote desktop
- ▶ Launch the Eclipse Java Integrated Development Environment (IDE)
- ▶ Select the **HashTest** project and open the **Hashtest.java** file
- ▶ The "F3" key leads you the definition of methods/functions
- ▶ The "F11" key lets you run/debug the application
- ▶ Define breakpoints and step into Java functions ("F5", "F6", "F7" keys)

## Questions

- ▶ What is the hash function for different objects in Java?
- ▶ What is the hash function for multiple objects?
- ▶ What properties does the hash function have (distribution, collision, one-way)?
- ▶ Can you engineer a hash collision?



# Cryptographic Hash Functions

## Properties

- ▶ Map byte arrays to byte arrays
- ▶ Fixed output size
- ▶ Deterministic
- ▶ Quick to compute
- ▶ Small changes in input should lead to large changes in output (**Why is this important?**)
- ▶ Very low collision probability

## Applications

- ▶ File verification (for downloads, etc.)
- ▶ Password storage
- ▶ Unique data identifiers
- ▶ **Blockchains**

# Pearson Hash

- ▶ Simple 8-bit hash function
- ▶ **Not cryptographically secure**
- ▶ Message  $M$  is an array of bytes  $b$
- ▶ Table  $T$  is arbitrary

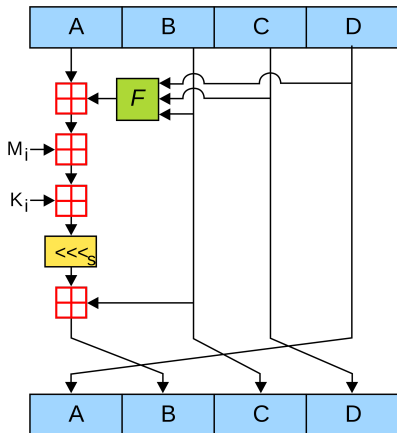
```
1 int h = 0;
  for (byte b : M) {
3   h = T[h xor c];
  }
5 return h;
```

# Popular Hash Functions

- ▶ MD5 (1992, R. Rivest) (No longer secure)
- ▶ SHA-1 (1995, NSA) (No longer secure)
- ▶ SHA-256, SHA-384, SHA-512 (2002, NSA)
- ▶ SHA-3 (Keccak) (2008, Bertoni et al.)

# MD5

- ▶ Break message into chunks of 512 bits, pad as required:
  - ▶ Add a single 1-bit
  - ▶ Add 0-bits until length is 64 bits less than multiple of 512
  - ▶ Fill remaining 64 bits with message length mod  $2^{64}$
- ▶ Initialize MD5 state (A, B, C, D, each 32 bits)
- ▶ For each chunk
  - ▶ Divide into 16 sets of 32 bits
  - ▶ Run 64 MD5 operations (16 in each of 4 rounds) (next slide)
- ▶ Concatenate final A, B, C, D for MD5 hash

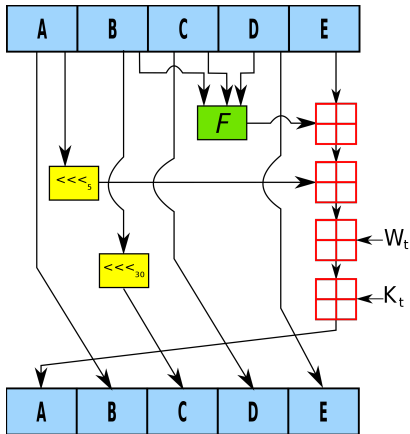


One MD5 operation. MD5 consists of 64 of these operations, grouped in four rounds of 16 operations.  $F$  is a nonlinear function; a different function is used in each round.  $M_i$  denotes a 32-bit block of the message input, and  $K_i$  denotes a 32-bit constant, different for each operation.  $\lll_s$  denotes a left bit rotation by  $s$  places;  $s$  varies for each operation. Addition denotes addition modulo  $2^{32}$ .

# MD5 Functions

## F(B,C,D)

- ▶  $(B \wedge C) \vee (\neg B \wedge D)$  (for rounds 0–15)
- ▶  $(B \wedge D) \vee (C \wedge \neg D)$  (for rounds 16–31)
- ▶  $B \oplus C \oplus D$  (for rounds 32–47)
- ▶  $C \oplus (B \vee \neg D)$  (for rounds 48–63)



One iteration within the SHA-1 compression function: A, B, C, D and E are 32-bit words of the state; F is a nonlinear function that varies;

$\lll_n$  denotes a left bit rotation by n places; n varies for each operation;  $W_t$  is the expanded message word of round t;  $K_t$  is the round constant of round t; Addition denotes addition modulo  $2^{32}$ .

# SHA-1 Functions

## $F(B,C,D)$

- ▶  $(B \wedge C) \vee (\neg B \wedge D)$  (for rounds 0–19)
- ▶  $B \oplus C \oplus D$  (for rounds 20–39)
- ▶  $(B \wedge C) \vee (B \wedge \neg D) \vee (C \wedge D)$  (for rounds 40–59)
- ▶  $C \oplus C \oplus D$  (for rounds 60–79)

## $K_t$

- ▶ 0x5A827999
- ▶ 0x6ED9EBA1
- ▶ 0x8F1BBCDC
- ▶ 0xCA62C1D6



# Let's Go Play

## Explore Cryptographic Hashes (Message Digests) in Java

- ▶ Select the **CryptoHashes** project and open the **CryptoHashTest.java** file
- ▶ The "F3" key leads you the definition of methods/functions
- ▶ The "F11" key lets you run/debug the application

## Questions

- ▶ Debug-step through the "computeMD5" method of the MD5 class
- ▶ How long does it take to calculate an MD5 hash?
- ▶ Can you engineer a hash collision for MD5?

## BONUS

- ▶ Extend the example to calculate the MD5 hash of a file.

# Digital Signatures

# Digital Signatures

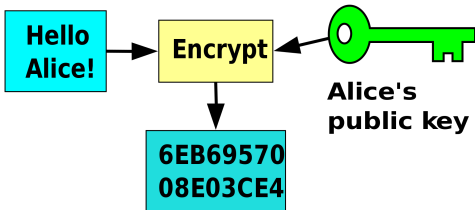
## Purpose

- ▶ Authentication
- ▶ Integrity
- ▶ Non-repudiation

## Public-Key Cryptography

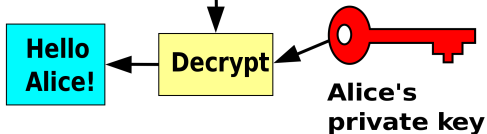
- ▶ Private Key
- ▶ Public Key

**Bob**

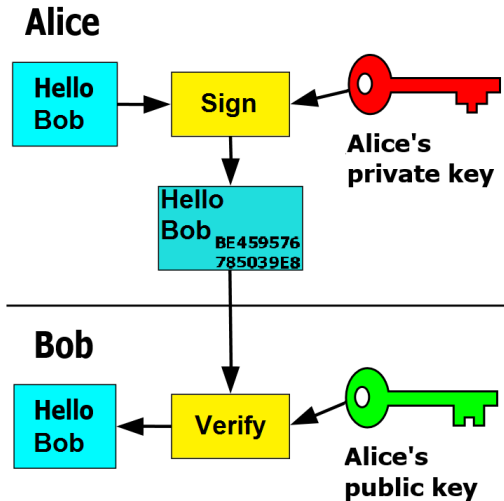


---

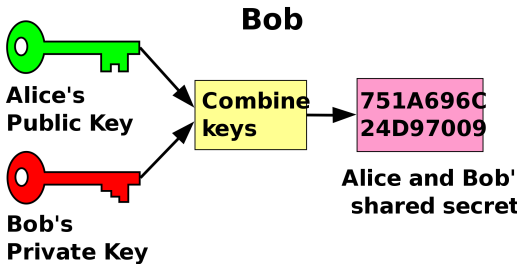
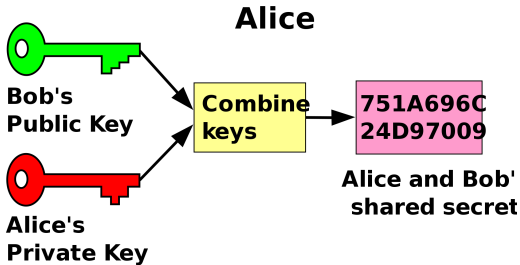
**Alice**



Source: Wikipedia, Public Domain



(c): FlippyFlink:Private key signing / Wikimedia Commons / CC-BY-SA-4.0



Source: Wikipedia, Public Domain

# Example: DSA

## Parameters

- ▶ Hash function  $H$  (e.g. SHA-2)
- ▶ Key lengths  $L$  and  $N$  (e.g. 512 bits)
- ▶ Choose an  $N$ -bit prime  $q$
- ▶ Choose an  $L$ -bit prime  $p$  such that  $p - 1$  is a multiple of  $q$
- ▶ Choose a  $g$  such that  $q$  is the smallest positive integer for which  $g^q = 1 \pmod{p}$



# Exercise

Select DSA parameters for a key of length 7 ( $2^7 = 128$ )

## Exercise – Example Parameters

$$q = 3 \rightarrow p = 7, p = 13, p = 19, \dots$$

$$q = 5 \rightarrow p = 11, p = 31, p = 61, \dots$$

$$q = 7 \rightarrow p = 29, p = 43, p = 71, \dots$$

$$q = 11 \rightarrow p = 23, p = 67, p = 89, \dots$$

For  $q=11, p=67$  ( $N=4, L=7$ )

$$g = 9 \rightarrow 9^{11} = 31381059609 \bmod 67 = 1$$

# Public and Private Keys

- ▶ Choose a secret private key  $x$  such that  $0 < x < q$
- ▶ Calculate the public key  $y = g^x \pmod{p}$   
(this is hard to invert)

## Exercise

- ▶ Choose a public and private key for  $q = 11, p = 67, g = 9$

## Exercise – Public and Private Key

- ▶  $x = 7$  (private)
- ▶  $y = 9^7 \bmod 67 = 4782969 \bmod 67 = 40$  (public)

# Signing a Message

- ▶ Choose a random per-message value  $k$  such that  $1 < k < q$
- ▶ Calculate  $r = (g^k \bmod p) \bmod q$
- ▶ If  $r = 0$ , use a different  $k$
- ▶ Calculate  $s = k^{-1} \times (H(m) + x \times r) \bmod q$ 
  - ▶ ("modular multiplicative inverse")
  - ▶  $k^{-1} \bmod q = x \rightarrow x \times k = 1 \pmod{q}$
- ▶ If  $s = 0$ , use a different  $k$
- ▶ The signature is  $(r, s)$

## Exercise – Signing a Message

- ▶ For  $x = 7$ , sign the message  $m = \text{"blockweek"}$
- ▶ Hash function: Pearson hashing

## Exercise – Signed Message

- ▶ Choose  $k = 8$

$$\begin{aligned}r &= (9^8 \bmod 67) \bmod 11 \\ &= (43046721 \bmod 67) \bmod 11 \\ &= 25 \bmod 11 = 3\end{aligned}$$

- ▶  $H(\text{"blockweek"}) = 117$
- ▶  $8^{-1} \bmod 11 = 7 \rightarrow 7 \times 8 = 56 = 1 \pmod{11}$

$$\begin{aligned}s &= 7 \times (117 + 7 \times 3) \bmod 11 \\ &= 7 \times 138 \bmod 11 \\ &= 966 \bmod 11 = 9\end{aligned}$$

- ▶ Signature is  $(3, 9)$

# Verify Signature

- ▶  $w = s^{-1} \bmod q$  ("modular multiplicative inverse")
- ▶  $u_1 = H(m) \times w \bmod q$
- ▶  $u_2 = r \times w \bmod q$
- ▶  $v = (g^{u_1} y^{u_2} \bmod p) \bmod q$
- ▶ Signature is valid iff  $v = r$



## Exercise – Verify Signature

$$w = 9^{-1} \bmod 11 = 5 \quad (5 \times 9 = 45 \bmod 11 = 1)$$

$$u_1 = H(\text{"blockweek"}) \times 5 \bmod 11$$

$$= 117 \times 5 \bmod 11 = 585 \bmod 11 = 2$$

$$u_2 = 3 \times 5 \bmod 11 = 4$$

$$v = (9^2 40^4 \bmod 67) \bmod 11$$

$$= (207360000 \bmod 67) \bmod 13$$

$$= 25 \bmod 11 = 3$$

- ▶ Signature is valid!

# Elliptic Curves

# Elliptic Curve Cryptograph

## Traditionally

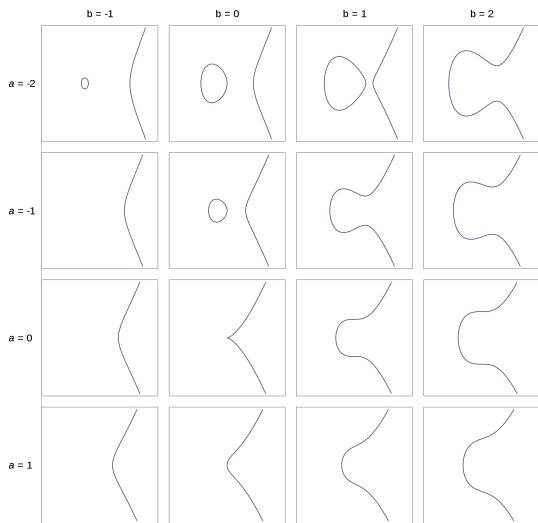
- ▶ Multiply two big prime numbers
- ▶ Factorization to primes was hard
- ▶ New techniques make this easier
- ▶ Requires longer key lengths

## Elliptic Curves

- ▶ Based on discrete logarithm
- ▶ Considered harder than prime factorization
- ▶ Shorter keys possible

# Elliptic Curves

$$y^2 = (x^3 + ax + b)$$

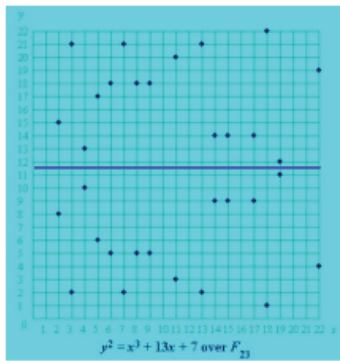
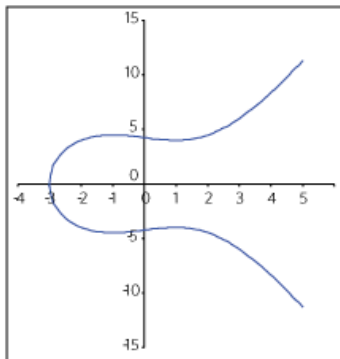


# Elliptic Curves on Finite (Prime) Fields

$$y^2 = (x^3 + ax + b) \pmod{p} \quad \text{where } p \text{ is prime}$$

$$y^2 = (x^3 + 13x + 7) \pmod{23}$$

mod23



# Intuition

- ▶ Each  $x$  value has 2  $y$  values (because of the square, note the symmetry)
- ▶ Because of the  $\text{mod } p$ ,  $y \in 0 \dots p - 1$
- ▶ Few of the  $y$  ( $N$ ) will be a squares of integers
- ▶ Since each  $x$  yields two points, there are  $N/2$  possible  $x$  values

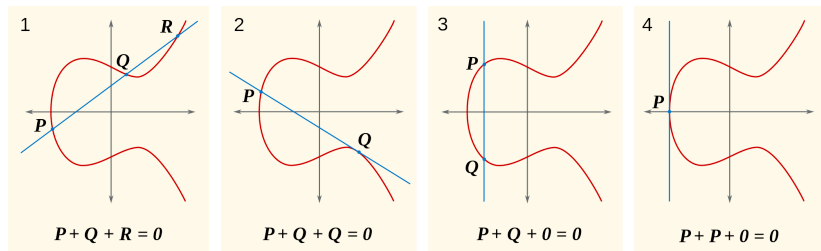
# Groups on Elliptic Curves

- ▶ A group  $G$  is a set for which a binary operation "addition" is defined
- ▶ Elements of  $G$  are the points on an elliptic curve
- ▶ Addition has the properties:
  - ▶ Closure
  - ▶ Associativity
  - ▶ Identity element
  - ▶ Inverse
  - ▶ Commutativity

# Addition on Elliptic Curves

## Geometric Interpretation

If  $P$  and  $Q$  are points on the curve, then  $P + Q$  is defined in the following way: Draw the line that intersects  $P$  and  $Q$ . This intersects the curve at a third point  $R$ . Then  $-R = P + Q$



(c) Emmanuel.boutet:ECCLines / Wikimedia Commons / CC-BY-SA-3.0



# Addition on Elliptic Curves

## Algebraic Definition

$$R = P + Q = \begin{pmatrix} p_x \\ p_y \end{pmatrix} + \begin{pmatrix} q_x \\ q_y \end{pmatrix} = \begin{pmatrix} r_x \\ r_y \end{pmatrix}$$

$$r_x = s^2 - p_x - q_x$$

$$r_y = s(p_x - r_x) - p_y$$

$$s = \frac{p_y - q_y}{p_x - q_x}$$

- ▶ Works the same in prime fields, everything is just mod  $p$

# Scalar Multiplication on Elliptic Curves

## Scalar Multiplication

$$Q = n \times P = P + P + \dots + P$$

## Discrete Logarithm

$$n = \frac{P}{Q}$$

Believed to be a Hard problem

# Elliptic Curve Digital Signature Algorithm (ECDSA)

## Choose a Curve (Parameters)

- ▶  $a, b$
- ▶  $p$  (prime modulus)
- ▶  $G$  "generating point", "point of origin" (a point on the curve)
- ▶  $n$  (prime order, number of points on curve)

## Example curve (secp256k1) (used by bitcoin)

- ▶  $a = 0, b = 7$
- ▶  $p = 0xffffffff ffffffff ffffffff ffffffff ffffffff ffffffff fffffffe fffffc2f$
- ▶  $n = 0xffffffff ffffffff fffffffe baaedce6 af48a03b bfd25e8c d0364141$
- ▶  $g_x = 0x79be667e f9dcbbac 55a06295 ce870b07 029bfcdb 2dce28d9 59f2815b 16f81798$
- ▶  $g_y = 0x483ada77 26a3c465 5da4fbfc 0e1108a8 fd17b448 a6855419 9c47d08f fb10d4b8$

# ECDSA

## Keys

- ▶ Choose an integer  $d_A \in [1, n - 1]$  (private key)
- ▶ Calculate public key  $Q_A = d_A \times G$  (this is hard to invert)

## Signing (same as DSA)

- ▶ Choose a random per-message value  $k \in [1, n - 1]$
- ▶ Calculate  $r = (g^k \bmod p) \bmod q$
- ▶ Calculate the curve point  $(x_1, y_1) = k \times G$  (this is hard to invert)
- ▶  $r = x_1 \bmod n$
- ▶ If  $r = 0$ , use a different  $k$
- ▶ Calculate  $s = k^{-1}(H(m) + r d_A) \bmod n$   
("modular multiplicative inverse")
- ▶ If  $s = 0$ , use a different  $k$
- ▶ The signature is  $(r, s)$

# ECDSA

## Preliminaries

- ▶ Check that  $Q_A \neq O$
- ▶ Check that  $Q_A$  is on the curve
- ▶ Check that  $n \times Q_A = O$

## Signature Verification

- ▶  $w = s^{-1} \bmod n$  ("modular multiplicative inverse")
- ▶  $u_1 = H(m) \times w \bmod n$
- ▶  $u_2 = r \times w \bmod n$
- ▶ Calculate  $X = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = u_1 \times G + u_2 \times Q_A$
- ▶ If  $X = O$  then signature is invalid
- ▶ If  $r = x_1 \bmod n$  then signature is valid

## Some Curves

- ▶ NIST (<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>)
- ▶ RFC4492 (<https://tools.ietf.org/html/rfc4492>)
- ▶ RFC7748 (<https://tools.ietf.org/html/rfc7748>)
- ▶ RFC8422 (<https://tools.ietf.org/html/rfc8422>)
- ▶ IANA (<https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml>)

## Recommendations

- ▶ When it says "random", really use random numbers
- ▶ Don't use your own curves
- ▶ Don't build your own implementations

## Security Issues

- ▶ Standard curves defined by NIST may be "backdoored"
- ▶ Discrete logarithm not proven hard, only believed to be
- ▶ Side-channel attacks

# Let's Go Play

Digital Signatures in Java

## Explore Digital Signatures in Java

- ▶ Select the **DigitalSignatures** project and open the **DigitalSignaturesDemo.java** file

## Questions

- ▶ Debug-step through the "ellipticCurveDemo" method and "primeFactorizationDemo" methods
- ▶ Make sure you understand how the theory is implemented
- ▶ How long does it take to sign the first message?
- ▶ How long does it take to sign subsequent messages?
- ▶ How long does it take to verify the first signature?
- ▶ How long does it take to verify subsequent messages?

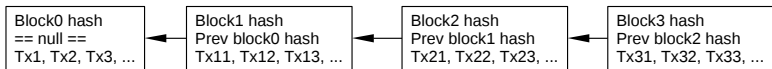


# Blocks and Transactions

# Block

## A very simple blockchain

- ▶ A block is a set of transactions (messages)
- ▶ A transaction is arbitrary information
- ▶ Transactions contain the public key of their creator
- ▶ Transactions are digitally signed by their creator
- ▶ Each block contains a hash of its content
- ▶ Each block's content includes the hash of the previous block



## Questions

- ▶ What must you do to verify the integrity of a block? Of the entire blockchain?
- ▶ Why is the blockchain record considered "immutable"?
- ▶ Under what conditions is the blockchain "immutable"?
- ▶ What would you have to do to change Tx12?

# Let's Go Play

Basic Blockchain in Java

# Exercise

## Blocks and Transactions

- ▶ Create a new project
- ▶ Create a "Transaction" class with methods to create, sign, and verify a transaction
  - ▶ Use an array of bytes (`byte[]`) as transaction data
- ▶ Create a "Block" class with methods to create and verify a block
  - ▶ Creation uses a collection of transactions and the hash of the previous block

## Exercise – Tips

- ▶ Copy code from earlier projects for hashing and signatures
- ▶ Bookmark this URL:  
<https://docs.oracle.com/javase/10/docs/api/index.html?overview-summary.html>
- ▶ You can catch and quietly ignore exceptions, or just throw them up the stack
- ▶ Use the following code to put things into an array of bytes (for signing, hashing, etc.)

```
1 byte[] a = ...;
  byte[] b = ...;
3
  ByteArrayOutputStream stream = new ByteArrayOutputStream
    ();
5   stream.write(a);
   stream.write(b);
7 return stream.toByteArray();
```

# Exercise

## Blockchain

- ▶ Create a "Blockchain" class with methods to add a block and verify the chain
  - ▶ Tip: Use appropriate Java Collections classes (in the `java.util` package)
- ▶ Implement a sensible `toString()` method for your classes
  - ▶ Use `Hex.encodeHex()` to convert byte arrays to readable character Strings
- ▶ Implement a `main()` method to test your blockchain

**Congratulations!**  
You've built your first blockchain!



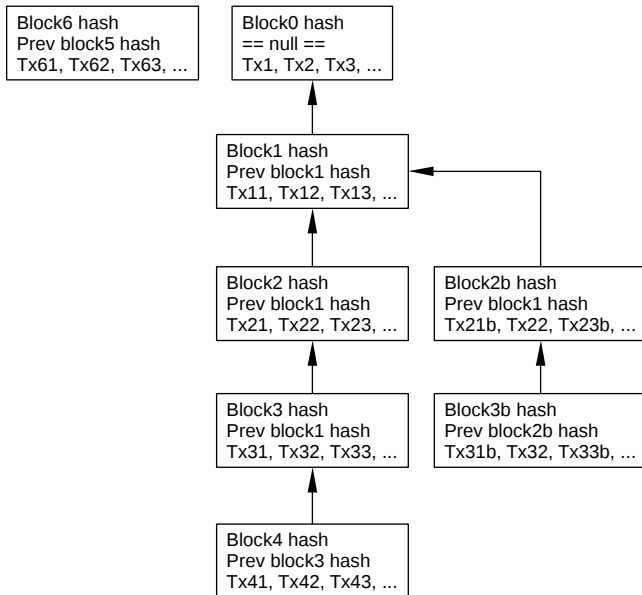
# Exercise

## Test Cases

- ▶ What happens when you create and add the first block?
- ▶ What happens when you add a block and you're missing the predecessor?
- ▶ What happens when you add another block with the same predecessor?

How should we handle this?

# Block Forest



# Block Forest

- ▶ One genesis block (i.e. a block without a previous hash)
- ▶ Multiple "orphans"
- ▶ Side-branches
- ▶ One current main branch head

# Distributed Blockchain

## Context

- ▶ Peer-to-peer network
- ▶ Not necessarily fully connected
- ▶ Peers leave and join network at any time
- ▶ No reliable timestamps, no synchronized clocks
- ▶ Possible malicious actors

## Consensus on "Current State"

- ▶ Which transactions and blocks are valid? ("Validation")
- ▶ Order of transactions and blocks? ("Ordering")

# Consensus in Distributed Blockchains

## Main Assumptions

- ▶ Every peer validates and participates in ordering
- ▶ No privileged/trusted peers
- ▶ There must be an incentive to keep peers honest
  - ▶ Bitcoin
  - ▶ Eth

## Common Consensus Methods

- ▶ Proof-of-work (Bitcoin, Ethereum)
- ▶ Proof-of-stake ("voting")
- ▶ Byzantine fault tolerance (BFT) (Hyperledger Fabric)

# Bitcoin

A Simplified Intro

# Bitcoin

- ▶ The first widely-used blockchain
- ▶ A specialized blockchain (financial transactions)
- ▶ Described and first implemented by Satoshi Nakamoto (?)
  - ▶ <https://bitcoin.org/bitcoin.pdf>
- ▶ Proof-of-work consensus
- ▶ Based on the cryptocurrency bitcoin

# Bitcoin

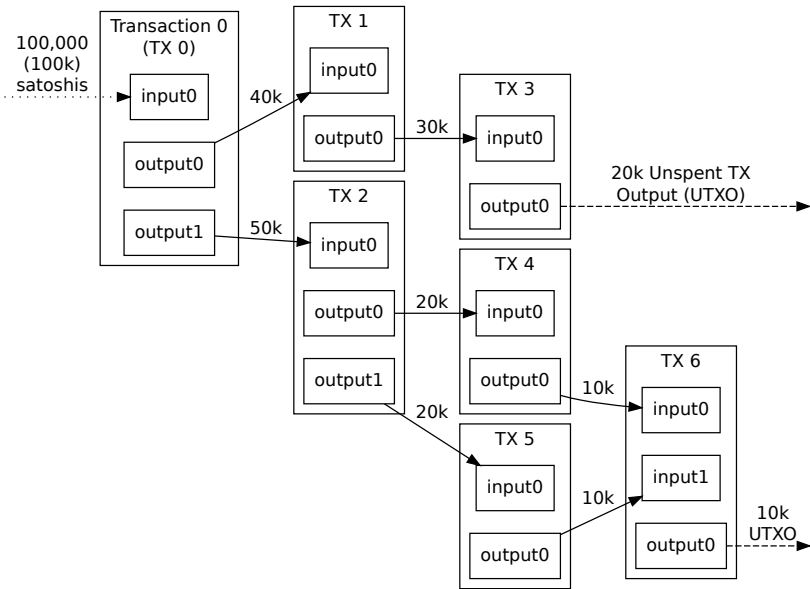
## Bitcoins

- ▶ Peers/accounts are identified by their public key
  - ▶ Actually, a hash of their public key
- ▶ Peers can "hold" a number of "bitcoins" (1 "bitcoin" = 100,000,000 "satoshis")
- ▶ Transactions transfer bitcoins from one peer/account to another
- ▶ Bitcoins are generated by "mining"



# Transactions

- ▶ Transactions are identified by their hash
- ▶ Transactions have numbered inputs and numbered outputs
- ▶ Inputs reference outputs from an earlier transaction sent to this account
- ▶ Outputs from transactions not already spent ("UTXO") is "bitcoin credit"
- ▶ You can only fully use up a UTXO but you can give yourself "change"

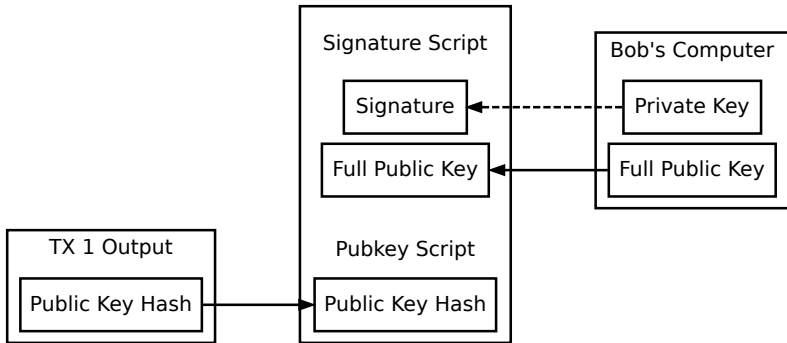


Triple-Entry Bookkeeping (Transaction-To-Transaction Payments) As Used By Bitcoin

Source: <https://www.bitcoin.org>

# Transactions

- ▶ Each output specified a recipient, identified by hash of public key
- ▶ To spend a UTXO, sender must verify it controls the private key to the recipient public key
- ▶ Sender includes public key
  - ▶ To verify it hashes to the specified recipient key and allow verification of signature
- ▶ Sender signs transaction (with private key)
  - ▶ If the included public key verifies the signature,
  - ▶ and it equals the recipient's public key,
  - ▶ then sender rightfully owns the UTXO



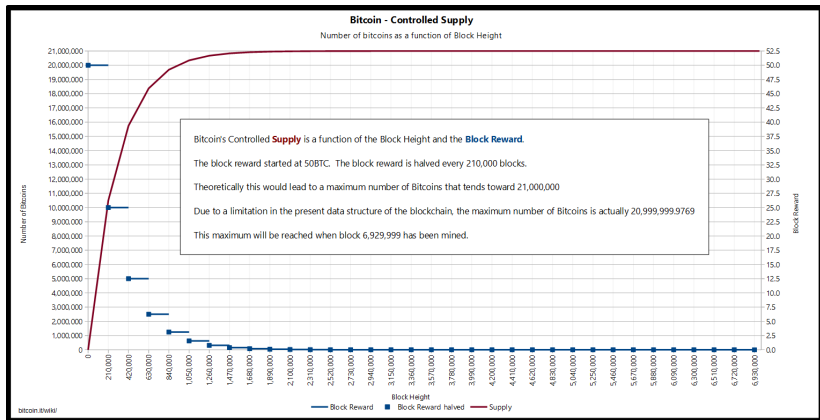
## Spending A P2PKH Output

Source: <https://www.bitcoin.org>

# Coinbase Transaction

- ▶ Special transaction with no inputs and one output
- ▶ Added by miner of a block as first transaction
- ▶ "Creates" new bitcoins
- ▶ Also includes "transaction fees"
  - ▶ Difference between sum of outputs and sum of inputs

Date reached	Block	BTC/block	BTC Added	BTC Increase	% of Limit
2009-01-03	0	50.00	2625000	$\infty$	12.500%
2010-04-22	52500	50.00	2625000	100.00%	25.000%
2011-01-28	105000	50.00	2625000	50.00%	37.500%
2011-12-14	157500	50.00	2625000	33.33%	50.000%
2012-11-28	210000	25.00	1312500	12.50%	56.250%
2013-10-09	262500	25.00	1312500	11.11%	62.500%
2014-08-11	315000	25.00	1312500	10.00%	68.750%
2015-07-29	367500	25.00	1312500	9.09%	75.000%
2016-07-09	420000	12.50	656250	4.17%	78.125%
2017-06-23	472500	12.50	656250	4.00%	81.250%
2018-05-29	525000	12.50	656250	3.85%	84.375%
	577500	12.50	656250	3.70%	87.500%
	630000	6.25	328125	1.79%	89.063%
	682500	6.25	328125	1.75%	90.625%
	735000	6.25	328125	1.72%	92.188%
	787500	6.25	328125	1.69%	93.750%

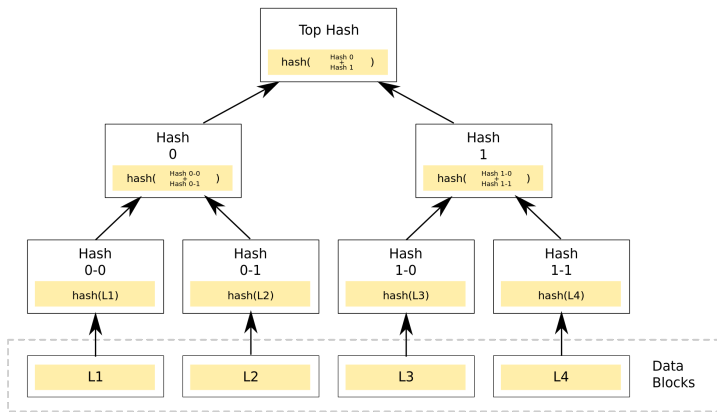


Source: <https://en.bitcoin.it>

# Block

- ▶ Identified by its hash
- ▶ Contains exactly one coinbase transaction and multiple other transactions
- ▶ Contains hash of previous block
- ▶ Contains hash of its transactions ("**Root of Merkle Tree**")
- ▶ Block size is limited
- ▶ Block includes a "nonce"

# Merkle Trees Allow Verification of Partial Data



(c) Azaghal:Hash Tree / Wikimedia Commons / CC-1.0

- ▶ R. Merkle, 1987
- ▶ Example: Can verify data block L2 if we have top hash ("root"), hash 0-0 and hash 1
  - ▶ Calculate hash 0-1, hash 0, top hash and check against given root
- ▶ Bitcoin separates block headers from block body/data



# Mining a Block

- ▶ Transactions are passed around on the peer-to-peer network
- ▶ Each peer maintains a transaction pool of new transactions ("mempool")
- ▶ Miners decides which transactions to include in a block (will focus on those with large transaction fees)
- ▶ Miner must find a hash for the block below a certain value (target difficulty)
  - ▶ Example: Hash must begin with 8 zeros
- ▶ Repeatedly change nonce value until block hash satisfies difficulty
- ▶ New blocks are passed around on the peer-to-peer network
- ▶ Difficulty is adjusted for a constant block rate (about 5 per hour for bitcoin)

# Let's Go Play

Extending the Basic Blockchain in Java

## Exercise

- ▶ Create a new class "Miner" with methods to create new block from a set of transactions and the hash of the previous block
- ▶ Assume a certain difficulty (not too hard, or it'll take forever)
- ▶ Include the root of the Merkle Tree of the transactions in the block

## Questions

- ▶ How many hashes do you have to try for different difficulty levels, for example, 1, 2, or 3?
- ▶ How many hashes can you compute per second?

# Bitcoin

Examining the Details of the Bitcoin Protocol

# Bitcoin Protocol: Adding a Transaction (1)

1. Check syntactic correctness
2. Make sure neither input or output lists are empty
3. Size in bytes  $\leq$  MAX\_BLOCK\_SIZE
4. Each output value, as well as the total, must be in legal money range
5. Make sure none of the inputs have hash=0, n=-1 (coinbase transactions)
6. Reject if we already have matching tx in the pool, or in a block in the main branch
7. For each input, if the referenced output exists in any other tx in the pool, reject this transaction.
8. For each input, look in the main branch and the transaction pool to find the referenced output transaction. If the output transaction is missing for any input, this will be an *orphan transaction*. Add to the orphan transactions, if a matching transaction is not in there already.

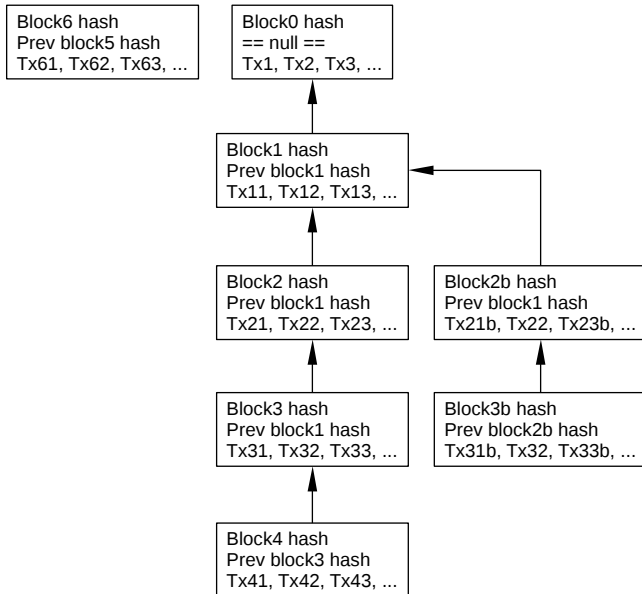
## Bitcoin Protocol: Adding a Transaction (2)

10. For each input, if the referenced output transaction is coinbase, it must have at least COINBASE\_MATURITY (100) confirmations; else reject this transaction
11. For each input, if the referenced output does not exist (e.g. never existed or has already been spent), reject this transaction
12. Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in legal money range
13. Reject if the sum of input values  $<$  sum of output values
14. Reject if transaction fee would be too low to get into an empty block
15. Verify the public keys for each input; reject if any are bad
16. Add to transaction pool
17. Relay transaction to peers
18. For each orphan transaction that uses this one as one of its inputs, run all these steps (including this one) recursively on that orphan

# Bitcoin Protocol: Adding a Block

1. Check syntactic correctness
2. Reject if duplicate exists in main branch, side branch or orphan
3. Transaction list must not be empty
4. Block hash must satisfy claimed difficulty proof of work
5. Block timestamp must not be more than two hours in the future
6. First transaction must be coinbase, the rest must not be
7. Verify Merkle hash
8. Check if prev block is in main branch or side branches. If not, add this to orphan blocks, then query peer we got this from for 1st missing orphan block in prev chain; done with block
9. Check that difficulty value matches the difficulty rules
10. Reject if timestamp is earlier than median time of last 11 blocks
11. Add block into the tree. There are three cases
  - a. block further extends the main branch;
  - b. block extends a side branch but does not add enough difficulty to make it become the new main branch;
  - c. block extends a side branch and makes it the new main branch.

# Block Forest





# Validating Block Transactions

1. For each input, look in the main branch to find the referenced output transaction. Reject if the output transaction is missing for any input.
2. For each input, if we are using the  $n$ th output of the earlier transaction, but it has fewer than  $n+1$  outputs, reject.
3. For each input, if the referenced output transaction is coinbase, it must have at least `COINBASE_MATURITY` (100) confirmations; else reject.
4. Verify crypto signatures for each input; reject if any are bad
5. For each input, if the referenced output has already been spent by a transaction in the main branch, reject
6. Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in legal money range
7. Reject if the sum of input values  $<$  sum of output values

# Adding to the Main Branch

1. For all but the coinbase transaction, validate block transactions  
(*previous slide*)
2. Reject if coinbase value  $>$  sum of block creation fee and transaction fees
3. For each transaction in the block, delete any matching transaction from the transaction pool
4. Relay block to our peers

# Adding to a Side Branch

1. Nothing to do.

# Adding to a Side Branch Becoming Main Branch

1. Find the fork block on the main branch which this side branch forks off of
2. Redefine the main branch to only go up to this fork block
3. For each block on the side branch, from the child of the fork block to the leaf, add it to the main branch:
  - 3.1 Do transaction checks 3–11
  - 3.2 Add it to the main branch (*previous slide*)
4. For each block in the old main branch, from the leaf down to the child of the fork block:
  - 4.1 For each non-coinbase transaction in the block:
    - 4.1.1 Apply transaction checks 2–9, except in step 8, only look in the transaction pool for duplicates, not the main branch
    - 4.1.2 Add to transaction pool if accepted, else go on to next transaction
5. For each block in the new main branch, from the child of the fork node to the leaf:
  - 5.1 For each transaction in the block, delete any matching transaction from the transaction pool
6. Relay block to peers

Note: If we reject at any point, leave the main branch as what it was, done

# Adding a Block

## Final step

13. For each orphan block for which this block is its prev, run all these steps (including this one) recursively on that orphan

# Transaction Stages

- ▶ Pending
- ▶ Mined
- ▶ Confirmed (at a certain height/depth)

## Discussion Questions

- ▶ What problems can a transaction encounter at each stage?
- ▶ How can/should you deal with these problems?

# Wallet

- ▶ A simplified bitcoin peer
- ▶ Does not participate in mining
- ▶ May not maintain a copy of the blockchain
- ▶ Capabilities:
  - ▶ Maintain a set of private and public keys ("accounts")
  - ▶ Listens to bitcoin peer-to-peer network to identify relevant transactions in blocks (i.e. UTXO designated for it)
  - ▶ Keeps track of UTXO (transaction hashes and output number)
  - ▶ Signs transactions to spend UTXO
  - ▶ Broadcasts new transactions to the network
- ▶ These capabilities may be in separate software applications

**Balances**


Available: **149.99996260 BTC**


Pending: **0.00000000 BTC**


Immature: **5 000.00003740 BTC**


Total: **5 150.00000000 BTC**


**Recent transactions**

 14 Apr 2019 17:34 [+50.00000000 BTC]  
(mnPm8sstca5KV66fGg27XwjLZ4K5zMBkgv)

 14 Apr 2019 16:49 [+50.00003740 BTC]  
(mhdVmzDmAYvprZWQXntSXzr9s4GTMEf1y3)

 14 Apr 2019 16:46 **-0.00003740 BTC**  
(n/a)

 14 Apr 2019 16:43 [+50.00000000 BTC]  
(mnRVMtCVZGVMs9PuyLUvm7qXRtSZ27HBHo)

 14 Apr 2019 16:43 [+50.00000000 BTC]  
(mnRVMtCVZGVMs9PuyLUvm7qXRtSZ27HBHo)



# P2P Network

## Message Types (Simplified)

- ▶ Peers can ask other peers for peer addresses ("getaddr" message)
- ▶ Peers send known peer addresses to other peers ("addr" message)
- ▶ Peers can ask other peers for transaction pool ("mempool" message)
- ▶ Peers send transactions in pool to other peers ("inv" message)
- ▶ Peers can ask other peers for known block headers ("getheaders" message)
- ▶ Peers send headers to other peers ("headers" message)
- ▶ Peers can ask other peers for blocks ("getblocks" message)
- ▶ Peers can send blocks to other peers ("block" message)

# P2P Network

## Startup (Simplified)

- ▶ Connect to previously known peers (fall-back discovery mechanisms exit)
- ▶ Ask for known block headers by sending currently known headers to peer
- ▶ Receive a list of available block headers (batch mode)
- ▶ Ask for blocks for headers
- ▶ Receive blocks for headers (batch mode)

# Discussion

## Questions

- ▶ How do these rules ensure consensus?
- ▶ What makes bitcoin trustworthy?
- ▶ How are bitcoin generated?
- ▶ How does this make bitcoin a currency?
- ▶ What are possible attacks on the system?
- ▶ All transactions are public! Can you see any problems?

# Bitcoin Statistics

## Some randomly selected sites

- ▶ <https://www.blockchain.com/explorer>
- ▶ <https://bitinfocharts.com/bitcoin/>
- ▶ <https://live.blockcypher.com/btc/>

## Discussion

- ▶ Anything in these sites you do not understand?
- ▶ Anything you find interesting?

# Let's Go Play

Bitcoin - Hands On

# Bitcoin Software Architecture

## Bitcoin

- ▶ Peer-to-Peer software that interacts with other peers
- ▶ Provides JSON RPC interface to clients

## bitcoin-cli

- ▶ Command line client to interact with bitcoind
- ▶ Provides wallet services
- ▶ Uses JSON RPC interface to bitcoind

## bitcoin-qt

- ▶ Graphical user interface client to interact with bitcoind
- ▶ Provides wallet services
- ▶ Uses JSON RPC interface to bitcoind

# Bitcoin Configuration

Configuration in `/etc/bitcoin/bitcoin.conf`

Data directory in `~/.bitcoin`

# Bitcoin Modes

## Mainnet

- ▶ Connects to the main bitcoin network and processes actual transactions

## Testnet

- ▶ Connects to the bitcoin test network
- ▶ Behaves identical to the main network
- ▶ No actual bitcoins are used

## Regtest

- ▶ Creates a private blockchain just for yourself
- ▶ Allows precise control over block creation
- ▶ No actual bitcoins are used



# Exercise

```
bitcoind -regtest -daemon
```

Starts the bitcoin application with a private chain and initializes a wallet (public/private key) for you

```
bitcoin-cli -regtest <command>
```

Uses the command line interface to connect to the local bitcoind to issue <command>.

A summary of commands is here:

<https://bitcoin.org/en/developer-reference#rpc-quick-reference>

# Exercise

- ▶ `GetNewAddress <account>`, creates a new bitcoin address
- ▶ `GenerateToAddress 101 <address>`, "mines" 101 blocks (Why?)
- ▶ `GetWalletInfo`, shows the new balance in our wallet
- ▶ `GetNewAddress <account>`, creates another bitcoin address
- ▶ `ListUnspent`, lists the UTXO
- ▶ `Sendtoaddress <address> <amount>`, sends bitcoin to address
- ▶ `GenerateToAddress 1 <address>`, "mines" another block
- ▶ `ListUnspent`, lists the UTXS (Why are there three now?)

# Useful Commands (Excerpt)

## Blocks and Chain

- ▶ GetBlockChainInfo
- ▶ GetBlock <hash>
- ▶ GetMemPoolInfo
- ▶ VerifyChain <level>

## Utility and Regtest

- ▶ Stop
- ▶ Generate <n>
- ▶ GenerateToAddress <n> <address>
- ▶ GetMiningInfo

# More Useful Commands (Excerpt)

## Networking

- ▶ AddNode <ip>:<port> [add|remove|onetry]
- ▶ GetConnectionCount
- ▶ GetNetworkInfo
- ▶ GetPeerInfo

## Transactions and Wallets

- ▶ AbandonTransaction <txid>
- ▶ DumpPrivKey <address>
- ▶ GetAccount <address>
- ▶ GetBalance <account> <confirmations>
- ▶ GetNewAddress <account>
- ▶ GetTransaction <txid>

# More Useful Commands (Excerpt)

## Transactions and Wallets

- ▶ `GetUnconfirmedBalance`
- ▶ `GetWalletInfo`
- ▶ `ListAddressGroupings`
- ▶ `ListTransactions <account> <n>`
- ▶ `ListUnspent <min_confirmations> <max_confirmations>`
- ▶ `SendToAddress <addr> <amount> <comment> <comment>`  
`<subtractfee>`
- ▶ `WalletPassPhraseChange <oldpassword> <newpassword>`
- ▶ `WalletLock`
- ▶ `WalletPassphrase <password> <seconds>`

# Exercise – Peer-to-Peer Network

## Connect

- ▶ (Start your "bitcoind" software)
- ▶ `bitcoin-cli -regtest AddNode <ipaddress> onetry`
- ▶ `bitcoin-cli -regtest GetConnectionCount`
- ▶ `bitcoin-cli -regtest GetPeerInfo`

## Transactions

- ▶ Create yourselves some addresses
- ▶ Mine some blocks with "GenerateToAddress" to your address
- ▶ Once you have some coins, send them around
- ▶ Generate new blocks as needed
- ▶ Use "getblockchaininfo", "getblock", "gettransaction" to examine the blockchain
- ▶ Use "getbalance", "listunspent", "listtransactions" to work with your money

# Bitcoin GUI






- ▶ Stop the bitcoind using the "stop" command from bitcoin-cli
- ▶ Start the graphical user interface `bitcoin-qt -regtest`
- ▶ Bitcoin-Qt can be accessed from bitcoin-cli as well!



The screenshot shows the Bitcoin Core - Wallet [regtest] interface. The window title is "Bitcoin Core - Wallet [regtest]". The menu bar includes "File", "Settings", and "Help". The main navigation bar has "Overview" (selected), "Send", "Receive", and "Transactions".

**Balances**

Available:	<b>149.99996260 BTC</b>
Pending:	<b>0.00000000 BTC</b>
Immature:	<b>5 000.00003740 BTC</b>
<hr/>	
Total:	<b>5 150.00000000 BTC</b>

**Recent transactions**

	14 Apr 2019 17:34	[+50.00000000 BTC]
	(mnPm8sstca5KV66fGg27XwjLZ4K5zMBkgv)	
	14 Apr 2019 16:49	[+50.00003740 BTC]
	(mhdVmzDmAYvprZWQXntSXzr9s4GTMEf1y3)	
	14 Apr 2019 16:46	<b>-0.00003740 BTC</b>
	(n/a)	
	14 Apr 2019 16:43	[+50.00000000 BTC]
	(mnRVmtCVZGVMs9PuyLUvm7qXRtSZ27HBHo)	
	14 Apr 2019 16:43	[+50.00000000 BTC]
	(mnRVmtCVZGVMs9PuyLUvm7qXRtSZ27HBHo)	

BTC HD  

# Ethereum



# The Basics

# Ethereum

## Characteristics

- ▶ A generic "universal" blockchain (not only for financial transactions)
- ▶ Developed by Vitalik Buterin (2013)
  - ▶ "White paper"
  - ▶ <https://github.com/ethereum/wiki/wiki/White-Paper>
- ▶ Formalized by Gavin Wood
  - ▶ "Yellow paper"
  - ▶ <https://ethereum.github.io/yellowpaper/paper.pdf>
- ▶ Cryptocurrency driven
- ▶ Proof-of-work consensus ("mining")
  - ▶ Proof-of-stake is discussed/envisioned/planned ("Casper")
- ▶ Describes "world state" distributed across blocks in chain
- ▶ A transaction/message is a "state transition operation"
- ▶ Smart contracts in Solidity

# Ethereum

## Differences to Bitcoin

- ▶ Messages in Transactions
- ▶ Ether and Wei, not Bitcoin and Satoshi
- ▶ Addresses, not UTXO
- ▶ Arbitrary smart contracts
- ▶ Different coin issuance model (pre-sale, steady supply)
- ▶ 12 seconds block time, not 10 minutes
- ▶ Different proof-of-work function ("Ethhash")
- ▶ Built on 'Keccak' hashes (not quite the SHA-3 standard)

# Ethereum Accounts

## Externally Owned Accounts

- ▶ Controlled by a private key
- ▶ Ether balance
- ▶ Send transactions to other accounts (messages in signed transaction)
- ▶ Nonce (number of transactions sent)

## Contract Account

- ▶ Controlled by code
- ▶ Ether balance
- ▶ Internal storage
- ▶ Responds to transactions sent to it
- ▶ Can send message to other contracts as response
- ▶ Nonce (number of messages sent)

# Ethereum Transactions

- ▶ Send Ether between accounts
- ▶ Create ("install", "deploy") a new contract
- ▶ Call methods on a contract

# Ethereum Computations

- ▶ Ethereum virtual machine (EVM) with VM language
- ▶ Turing complete ("can compute anything that is computable")
- ▶ Computations and storage cost "gas"
- ▶ Transaction originator
  - ▶ Specifies quantity of gas willing to spend ("startgas")
  - ▶ Specifies fee per unit of gas ("gasprice")
  - ▶ Pays  $\text{startgas} \times \text{gasprice}$  as transaction fee
- ▶ Contracts written in Solidity (other language possible)

# Mining — The GHOST protocol

- ▶ A block must specify a parent, and it must specify 0 or more uncles
- ▶ An uncle included in block B must have the following properties:
  - ▶ It must be a direct child of the  $k$ -th generation ancestor of B, where  $2 \leq k \leq 7$ .
  - ▶ It cannot be an ancestor of B
  - ▶ An uncle must be a valid block header, but does not need to be a previously verified or even valid block
  - ▶ An uncle must be different from all uncles included in previous blocks and all other uncles included in the same block (non-double-inclusion)

# Block Validation

- ▶ Check if the previous block referenced exists and is valid.
- ▶ Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
- ▶ Check that the block number, difficulty, transaction root, uncle root and gas limit are valid.
- ▶ Check that the proof of work on the block is valid.
- ▶ Let  $S[0]$  be the state at the end of the previous block.
- ▶ Let  $TX$  be the block's transaction list, with  $n$  transactions. For all  $i \in 0 \dots n - 1$ , set  $S[i + 1] = \text{APPLY}(S[i], TX[i])$ . If any application returns an error, or if the total gas consumed in the block up until this point exceeds the  $\text{GASLIMIT}$ , return an error.
- ▶ Let  $S_{\text{FINAL}}$  be  $S[n]$ , but adding the block reward paid to the miner.
- ▶ Check if the Merkle tree root of the state  $S_{\text{FINAL}}$  is equal to the final state root provided in the block header. If it is, the block is valid; otherwise, it is not valid.



# Ethereum — Coin Issuance

## Initial

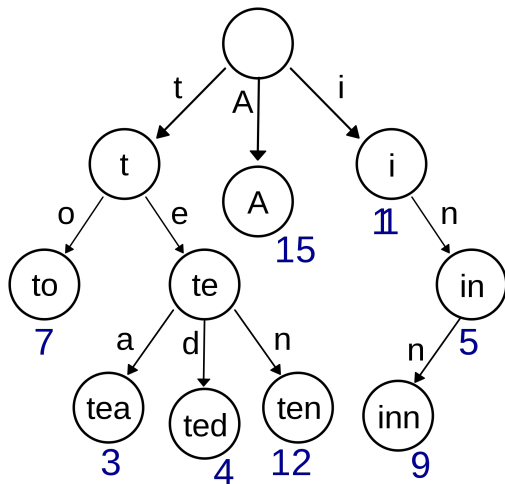
- ▶ 60102216 Eth pre-sale for Bitcoin
- ▶ 5950120 Eth development fund
- ▶ 5950120 Eth expense fund

## Ongoing

- ▶ Aim is to create 15626576 Eth every year, 26% of pre-sale
- ▶ Miner of block receives block reward  $R$  (currently 3 Eth)
- ▶ Miner of block receives  $1/32R = 0.09375Eth$  for each included uncle (maximum two uncles)
- ▶ Miner of included uncle receives  $R - 1/8(B_i - U_i)$  (maximum two uncles).

# Tries

## ► Retrieval trees



Source: Wikipedia, Public Domain

# Radix Tree

- ▶ Space-saving over tries: Only children merged with parents
- ▶ Number of children limited to radix (base) of alphabet
- ▶ Lookup and deletion are straightforward
- ▶ Operations in  $O(k)$  time where  $k$  is key length

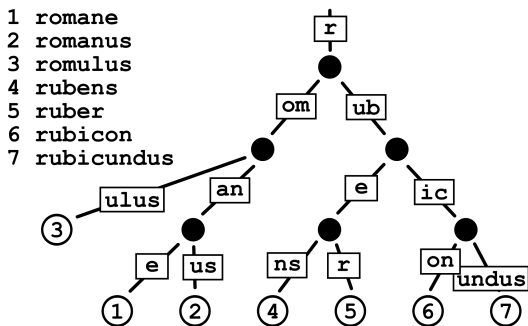
# Radix Tree Exercise

## Exercise

- ▶ Beginning with an empty tree, insert the following in order (comparison to be done by letter)
  - ▶ "test"
  - ▶ "slow"
  - ▶ "water"
  - ▶ "slower"
  - ▶ "tester"
  - ▶ "team"
  - ▶ "toast"

# Radix ("Patricia") Tree

- ▶ "Practical Algorithm To Retrieve Information Coded In Alphanumeric" (Morrison, JACM, 1968)



(c) Rocchini:Patricia trie (radix tree) / Wikimedia Commons / CC-BY-SA-3.0

# Patricia Tries and Radix Trees

- ▶ Instead of letters, we use bits
- ▶ How many bits are compared atomically?
  - ▶ 1 bit: at most 2 descendants, "binary radix tree", "Patricia trie"
  - ▶ 2 bits: at most 4 descendants, radix (base) is 4, chunks of 2 bits are compared
  - ▶ 3 bits: at most 8 descendants, radix (base) is 8, chunks of 3 bits are compared
  - ▶ etc.

# Merkle–Patricia Tries in Ethereum

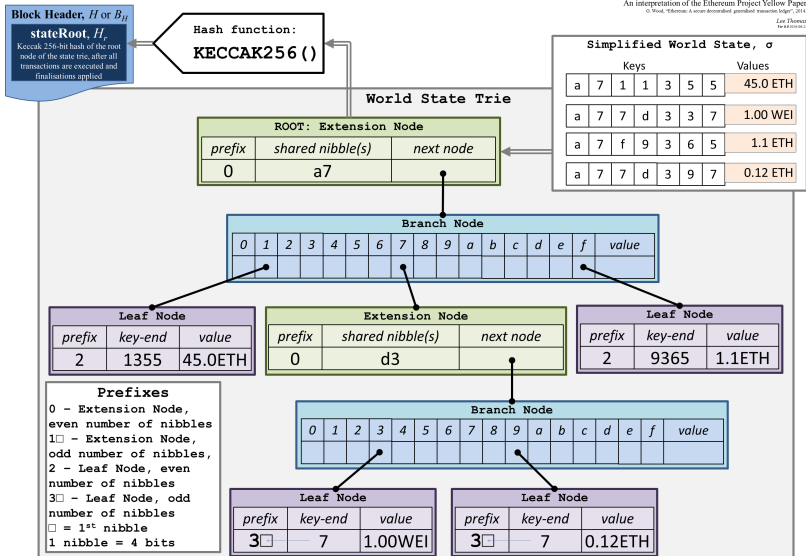
## Ethereum Adaptation

- ▶ Keys are hashes
- ▶ Hexadecimal alphabet (i.e., 4-bit chunks, "nibbles")
- ▶ "Extension nodes" describe long shared paths (parts of keys)
- ▶ Each block header has roots of state, transaction, and receipts trie

## Tries in Ethereum

- ▶ State Trie:  $\text{sha3}(\text{address}) \rightarrow \text{rlp}(\text{nonce}, \text{balance}, \text{storageRoot}, \text{codeHash})$
- ▶ Storage Trie:  $\text{sha3}(\text{variable}) \rightarrow \text{value}$
- ▶ Transaction Trie:  $\text{rlp}(\text{transactionIndex}) \rightarrow \text{transaction}$
- ▶ Receipts Trie:  $\text{rlp}(\text{transactionIndex}) \rightarrow \text{transaction}$

Note: RLP is "recursive length prefix", a method to serialize objects to arrays of bytes.

(c) Lee Thomas: Merkle-Patricia Trees as used in Ethereum, <https://i.stack.imgur.com/YZGxe.png>

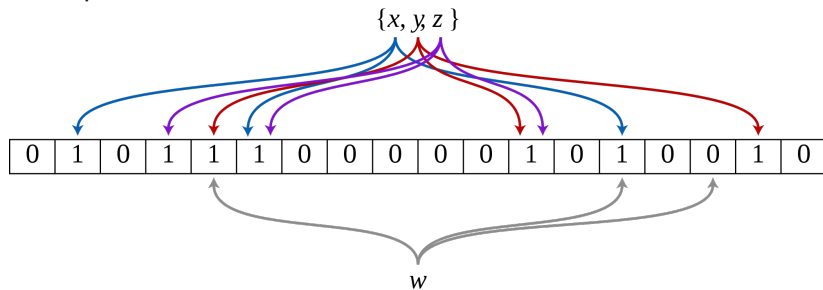


# Bloom Filters

- ▶ Decide if an element is in a set
- ▶ Bit array of size  $m$
- ▶ Set of  $k$  hash functions, which map input into one of the  $m$  array positions
- ▶ Ideally, hash functions yield a uniform distribution for all inputs
- ▶ When inserting element  $a$  into set, apply all  $k$  hash functions and set corresponding bits to 1
- ▶ When querying for a set  $a$ 
  - ▶ Use all  $k$  hash functions to identify relevant bit positions
  - ▶ If any of the  $k$  positions are 0, then element is not in set
  - ▶ If all of the  $k$  positions are 1, then element *may be* in set
- ▶ The error rate diminishes with more hash functions and a larger array

# Bloom Filters

Example:



Source: Wikipedia, public domain

# Bloom Filters

Probability that a bit is not set by a hash function is

$$1 - \frac{1}{m}$$

For  $k$  hash functions, the probability that a bit is not set is

$$\left(1 - \frac{1}{m}\right)^k$$

After  $n$  inserted elements:

$$\left(1 - \frac{1}{m}\right)^{kn}$$

The probability that the bit is set is therefore

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

When querying (i.e. testing  $k$  bits for being set) for an element *not in the set*, the probability that all  $k$  bits will come up set, and the result of the element being in the set is wrong is

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

# Bloom Filters

Minimizing the error rate  $p$  yields

$$k_{opt} = \frac{m}{n} \ln 2$$

Hence,

$$p_{opt} = \left(1 - e^{-\left(\frac{m}{n} \ln 2\right) \frac{n}{m}}\right)^{\frac{m}{n} \ln 2}$$

$$\ln p_{opt} = -\frac{m}{n} (\ln 2)^2$$

$$m_{opt} = -\frac{n \ln p_{opt}}{(\ln 2)^2}$$

$$k_{opt} = -\frac{\ln p}{\ln 2} = -\log_2 p$$

# Transaction Receipts

## Receipt Content

- ▶ Results of a transaction
  - ▶ **Not the return value of a contract method**
- ▶ Tuple of (*state*, *gas\_used*, *logbloom*, *logs*)
- ▶ Each log is a tuple of (*address*, (*topic1*, *topic2*, ...), *data*)
- ▶ *Logbloom* is a bloom filter of addresses and topics
- ▶ A bloom filter in the block header is the OR value of all transaction receipt bloom filters

## Example Use Case

- ▶ Client interested in topic X for address Y
1. Check block header bloom filter. If positive
  2. Check each transaction receipt bloom filter. If positive
  3. Check actual transaction log

# Ethereum Hands-On



# Initializing the Chain

```
geth init ethereum.init.json
```



# Creating an Account

```
geth account new
```

# Starting the Ethereum Node

```
geth --networkid 2019 --nodiscover --rpc --rpcapi \
    eth,shh,web3,net,admin,miner,personal,txpool
```

# Console Access via RPC

- ▶ In a new terminal window:

```
geth attach http://localhost:8545
```

# Start Mining

```
miner.start()
```

```
eth.getBalance(eth.accounts[0])
```

```
eth.blockNumber
```

# My First Transaction

- ▶ Make another account

```
personal.newAccount()
```

- ▶ Unlock the sending account

```
personal.unlockAccount(eth.accounts[0], "password",  
600)
```

- ▶ Send some Ether from your first account to your second

```
eth.sendTransaction({from:eth.accounts[0],  
to:eth.accounts[1], value: web3.toWei("4",  
"ether")})
```

# Check the Results

- ▶ Check balances

```
eth.getBalance(eth.accounts[0])  
eth.getBalance(eth.accounts[1])
```

- ▶ Check the transaction

```
eth.getTransaction("your_transaction_hash")
```

- ▶ Check the transaction receipt

```
eth.getTransactionReceipt("your_transaction_hash")
```

- ▶ Examine the block

```
eth.getBlock(your_block_number, true)
```

# Connect Nodes

- ▶ **Get Node Information**

```
admin.NodeInfo.enode
```

- ▶ **Add peer by "enode" (change to correct the IP address)**

```
admin.addPeer("your_enode_info_here")
```

- ▶ **Check connections**

```
admin.peers
```

# Play Time

- ▶ Play around with basic Ethereum, mining, and transferring Ether



# Ethereum Web3.js Module Functions (Incomplete)

## General Info

- ▶ `admin.nodeInfo`
- ▶ `eth.syncing`
- ▶ `eth.coinbase`
- ▶ `eth.mining`
- ▶ `eth.hashrate`
- ▶ `eth.gasPrice`
- ▶ `eth.accounts`
- ▶ `eth.blockNumber`
- ▶ `eth.getBalance(address, [n|"earliest"|"latest"|"pending"])`
- ▶ `eth.getTransactionCount(address, [n|"earliest"|"latest"|"pending"])`

# Ethereum Web3.js Module Functions (Incomplete)

## Block and Uncles Info

- ▶ `eth.getBlock(block1|"earliest"|"latest"|"pending", txObjects)`
- ▶ `eth.getBlockTransactionCount(block1|"earliest"|"latest"|"pending")`
- ▶ `eth.getBlockUncleCount(block1|"earliest"|"latest"|"pending")`
- ▶ `eth.getUncle(block1|"earliest"|"latest"|"pending", uncleIndex)`

---

<sup>1</sup>May be block number or block hash

# Ethereum Web3.js Module Functions (Incomplete)

## Transaction Info

- ▶ `eth.getTransactionfromBlock(block2|"earliest"|"latest"|"pending", txIndex)`
- ▶ `eth.getTransaction(transactionHash)`
- ▶ `eth.getTransactionReceipt(transactionHash)`
- ▶ `eth.pendingTransactions`

---

<sup>2</sup>May be block number or block hash

# Ethereum Web3.js Module Functions (Incomplete)

## Transaction Functions

- ▶ `eth.sign(address, message)`
- ▶ `eth.sendTransaction(from, to, [value], [gas], [gasPrice], data, [nonce])`
- ▶ `eth.sendRawTransaction(signedTxData)`

## Log Functions

- ▶ `eth.newFilter(fromBlock, toBlock, address|ArrayOfAddress, topic|ArrayOfTopics)`
- ▶ `eth.newBlockFilter()`
- ▶ `eth.newPendingTransactionFilter()`
- ▶ `eth.uninstallFilter(filterId)`
- ▶ `eth.getFilterChanges(filterId)`
- ▶ `eth.getFilterLogs(filterId)`

# Management Modules and Functions (Incomplete)

## admin

- ▶ `admin.addPeer(url)`
- ▶ `admin.datadir()`
- ▶ `admin.peers`

## miner

- ▶ `miner.setExtra(string)`
- ▶ `miner.setGasPrice(hexNumber)`
- ▶ `miner.start()`
- ▶ `miner.stop()`
- ▶ `miner.getHashrate()`
- ▶ `miner.setEtherbase(address)`

# Management Modules and Functions (Incomplete)

## personal

- ▶ `personal.listAccounts`
- ▶ `personal.lockAccount(address)`
- ▶ `personal.newAccount()`
- ▶ `personal.unlockAccount(address, password, duration)`
- ▶ `personal.sendTransaction(tx, password)`
- ▶ `personal.sign(message, account, password)`

## txpool

- ▶ `txpool.content`
- ▶ `txpool.inspect()`
- ▶ `txpool.status`

# Modules and Functions (Incomplete)

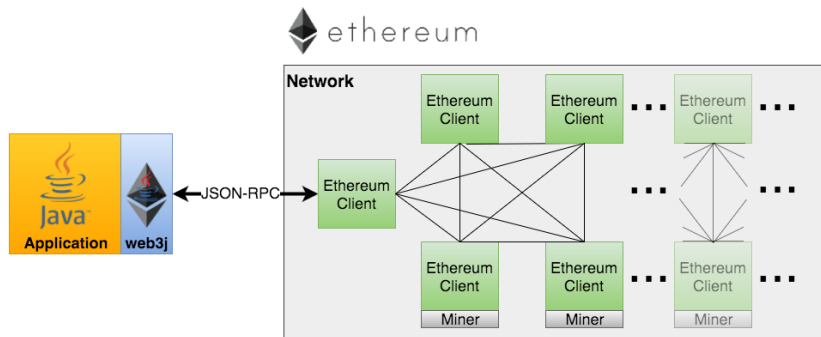
## web3

- ▶ `web3.sha3(string)`
- ▶ `web3.toWei(number, unit)`
- ▶ `web3.fromWei(number, unit)`

## net

- ▶ `net.version`
- ▶ `net.peerCount`
- ▶ `net.listening`

# Access Ethereum from Java Using Web3j



- ▶ Performs same role as Ethereum Web3.js Library
- ▶ Uses same RPC interface
- ▶ Exposes similar API as Web3.js
- ▶ **It is not an Ethereum node implemented in Java!**



# Web3j Example

## Example

- ▶ Open the project "EthereumWeb3j" in your Eclipse workspace
- ▶ Make sure your *geth* node is running and mining (connected or disconnected)
- ▶ Open the "EthereumWeb3jApplication.java" file
- ▶ **Adjust** the "EthereumWeb3jApplication.java" file:
  - ▶ The name of the keystore file for the address you would like to send Ether from, and its password
  - ▶ The address that you would like to send Ether to
- ▶ Run the "EthereumWeb3jApplication" to test it

# Web3j Asynchronous Example

## Asynchronous Operations

- ▶ RPC calls introduce latency until results are returned
- ▶ **Blockchain introduces latency while transactions are mined**
- ▶ We'd like to not wait, but do something useful instead

## Example

- ▶ Open the **"EthereumWeb3jApplicationAsynch.java"** file
- ▶ Adjust the **"EthereumWeb3jApplicationAsynch.java"** file:
  - ▶ The name of the keystore file for the address you would like to send Ether from, and its password
  - ▶ The address that you would like to send Ether to
- ▶ Run the **"EthereumWeb3jApplicationAsynch"** to test it

# Web3j Exercise

- ▶ Extend the example(s) to wait until the transaction has  $n$  confirmed blocks on top of it!

# Monitoring the Blockchain with Filters

- ▶ Filters for new pending transactions, new mined transactions, new blocks
- ▶ Filters for past blocks, past transactions, past and new blocks ("replay filters")
- ▶ Filters for topics and events in smart contracts

## Example

- ▶ Open the "EthereumWeb3jFilterApplication.java" file
- ▶ Run the "EthereumWeb3jFilterApplication" to test it

## Exercise

- ▶ Modify the filter to print transactions only if they involve the addresses/accounts of your local node

# Smart Contracts in Solidity

# My First Contract

```
1 contract Myfirstcontract {  
    address creator;  
3  
    constructor() public {  
5        creator = msg.sender;  
    }  
7  
    function kill() public {  
9        if (msg.sender == creator) {  
            selfdestruct(creator);  
11       }  
    }  
13 }
```

# Datatypes

- ▶ `bool`
- ▶ `int, uint, int8...int256, uint8...uint256`
- ▶ `fixed, ufixed`
- ▶ `address/address payable`
  - ▶ `balance, transfer, send, call` **members/methods**
- ▶ `byte, bytes1 ... bytes32`
- ▶ `bytes, string`

## Visibility

- ▶ **Public, Internal**

## Location

- ▶ **Storage, Memory, Calldata**

# Data Location and Assignment

```
1 contract C {  
    uint[] x; // the data location of x is storage  
3  
    // the data location of memoryArray is memory  
5 function f(uint[] memory memoryArray) public {  
    x = memoryArray;  
7    // works, copies the whole array to storage  
    uint[] storage y = x;  
9    // works, assigns a pointer, data location of y is storage  
    y[7];  
11    // fine, returns the 8th element  
    y.length = 2;  
13    // fine, modifies x through y  
    delete x;  
15    // fine, clears the array, also modifies y
```



# Data Location and Assignment

```
16 // The following does not work; it would need to create
17 // a new temporary unnamed array in storage, but storage
18 // is "statically" allocated:
19 y = memoryArray;
20 // This does not work either, since it would 'reset' the
21 // pointer, but there is no sensible location it could
22 // point to.
23 delete y;
24 g(x);
25 // calls g, handing over a reference to x
26 h(x);
27 // calls h and creates an independent, temporary copy
28 // in memory
29 }
30 function g(uint[] storage) internal pure {}
31 function h(uint[] memory) public pure {}
32 }
```

# Data Location Rules

- ▶ Assignment between storage and memory always creates a copy
- ▶ Assignment from memory to memory creates only a reference
- ▶ Assignment from storage to local storage variable creates a reference
- ▶ All other assignments to storage create a copy

# Arrays

```
1 contract Myfirstcontract {
   uint[] public a; // location is storage
3
   constructor() public {
5     a = new uint [] (0);
     a.push(1); // Push can be used for dynamic storage arrays
7     a.length = 5; // Or you can (re)set the length

9     uint[5] memory b; // fixed length array
     b[0] = 1;

11    uint[3][5] memory c;
13    c[1][2] = 3;
     c[2] = [uint256(1), 2, 3];
15    // Note: The indices are in reversed order!
     // Array literals are by default uint8, but declared arrays
17    // are by default uint256
   }
19 }
```

# Structs

```
1  struct Funder {
2      address addr;
3      uint amount;
4  }

6  struct Campaign {
7      address payable beneficiary;
8      uint fundingGoal;
9      uint numFunders;
10     uint amount;
11     mapping (uint => Funder) funders;
12 }
```

# Maps

```
1 contract MappingExample {
2     mapping(address => uint) public balances;

4     function update(uint newBalance) public {
5         balances[msg.sender] = newBalance;
6     }
7 }
8
9
10 contract MappingUser {
11     function f() public returns (uint) {
12         MappingExample m = new MappingExample();
13         m.update(100);
14         return m.balances(address(this));
15     }
16 }
```

# Block and Transaction Properties

- ▶ `blockhash(uint blockNumber)` returns (bytes32): **hash of the given block**
- ▶ `block.coinbase (address payable)`: **block miner address**
- ▶ `block.difficulty (uint)`: **current block difficulty**
- ▶ `block.gaslimit (uint)`: **current block gaslimit**
- ▶ `block.number (uint)`: **current block number**
- ▶ `block.timestamp (uint)`: **current block timestamp as seconds**
- ▶ `gasleft()` returns (uint256): **remaining gas**
- ▶ `msg.data (bytes calldata)`: **complete calldata**
- ▶ `msg.sender (address payable)`: **sender of the message (current call)**
- ▶ `msg.sig (bytes4)`: **first four bytes of the calldata (i.e. function identifier)**
- ▶ `msg.value (uint)`: **number of wei sent with the message**
- ▶ `now (uint)`: **current block timestamp (alias for block.timestamp)**
- ▶ `tx.gasprice (uint)`: **gas price of the transaction**
- ▶ `tx.origin (address payable)`: **sender of the transaction (full call chain)**

## Addresses

- ▶ `<address>.balance (uint256)`
- ▶ `<address payable>.transfer(uint256 amount)`
- ▶ `<address payable>.send(uint256 amount)` returns `(bool)`

## Error Handling

- ▶ `assert(bool condition):`
- ▶ `require(bool condition):`
- ▶ `require(bool condition, string memory message):`
- ▶ `revert():`
- ▶ `revert(string memory reason):`

# Control Structures

- ▶ `if ... else`
- ▶ `while ... do`
- ▶ `for ...`
- ▶ `break, continue, return`



# Functions

```
1 contract Simple {  
    function arithmetic(uint _a, uint _b)  
3     public pure  
        returns (uint o_sum, uint o_product)  
5     {  
        return (_a + _b, _a * _b);  
7     }  
}
```

## Modifiers

- ▶ *View*: promises not to change state
- ▶ *Pure*: promises to not even look at state

## Visibility

- ▶ *External*: can only be called from transactions
- ▶ *Public*: can be called internally or from transactions/messages
- ▶ *Internal*: can be called internally only, or from derived contracts
- ▶ *Private*: cannot be called from derived contracts

# Contracts

```
1 contract C {
2   uint private data;

4   function f(uint a) private pure returns(uint b) {return a+1;}
   function setData(uint a) public { data = a; }
6   function getData() public view returns(uint) { return data; }
   function compute(uint a, uint b) internal pure returns (uint)
       { return a + b; }

8 }
contract D { // This will not compile
10  function readData() public {
    C c = new C();
12    uint local = c.f(7); // error: 'f' is not visible
    c.setData(3);
14    local = c.getData();
    local = c.compute(3, 5); // error: 'compute' is not visible
16  }
}
18 contract E is C {
    function g() public {
20    C c = new C();
    uint val = compute(3, 5); // access to internal member
22  }
}
```

# Getter and Setter Functions

```
1 contract C {
    uint public data;
3    function x() public returns (uint) {
        data = 3; // internal access
5        return this.data(); // external access
    }
7 }

9 contract arrayExample {
    uint[] public myArray;
11
    /* Getter function generated by the compiler
13    function myArray(uint i) returns (uint) {
        return myArray[i];
15    } */
    // function that returns entire array
17    function getArray() returns (uint[] memory) {
        return myArray;
19    }
}
```

# Function Modifiers

```
1 contract owned {
2   constructor() public { owner = msg.sender; }
3   address payable owner;
4
5   // This contract only defines a modifier but does not use
6   // it: it will be used in derived contracts.
7   // The function body is inserted where the special symbol
8   // `_;` in the definition of a modifier appears.
9   // This means that if the owner calls this function, the
10  // function is executed and otherwise, an exception is
11  // thrown.
12  modifier onlyOwner {
13    require(
14      msg.sender == owner,
15      "Only owner can call this function."
16    );
17    _;
18  }
19 }
```

# Function Modifiers

```
20 contract mortal is owned {  
    // This contract inherits the 'onlyOwner' modifier from  
22    // 'owned' and applies it to the 'close' function, which  
    // causes that calls to 'close' only have an effect if  
24    // they are made by the stored owner.  
    function close() public onlyOwner {  
26        selfdestruct(owner);  
    }  
28 }
```

# Function Modifiers with Arguments

```
29 contract priced {
30     // Modifiers can receive arguments:
31     modifier costs(uint price) {
32         if (msg.value >= price) {
33             _;
34         }
35     }
36 }
contract Register is priced, owned {
37     mapping (address => bool) registeredAddresses;
38     uint price;
39
40     constructor(uint initialPrice) public {price = initialPrice;}
41
42     // It is important to also provide the 'payable' keyword here,
43     // otherwise the function will reject all Ether sent to it.
44     function register() public payable costs(price) {
45         registeredAddresses[msg.sender] = true;
46     }
47
48     function changePrice(uint _price) public onlyOwner {
49         price = _price;
50     }
51 }
52 }
```

# More Function Modifiers

```
1 contract Mutex {
2   bool locked;
3   modifier noReentrancy() {
4     require(
5       !locked,
6       "Reentrant call."
7     );
8     locked = true;
9     _;
10    locked = false;
11  }
12
13  // This function is protected by a mutex, so that reentrant
14  // calls from within `msg.sender.call` cannot call `f` again.
15  // The `return 7` statement assigns 7 to the return value but
16  // still executes the statement `locked=false` in the modifier
17  function f() public noReentrancy returns (uint) {
18    (bool success,) = msg.sender.call("");
19    require(success);
20    return 7;
21  }
22 }
```

# Events

```
1 contract ClientReceipt {
2     event Deposit(
3         address indexed _from,
4         bytes32 indexed _id,
5         uint _value
6     );

7
8     function deposit(bytes32 _id) public payable {
9         // Events are emitted using `emit`, followed by
10        // the name of the event and the arguments
11        // (if any) in parentheses. Any such invocation
12        // (even deeply nested) can be detected from
13        // the JavaScript API by filtering for `Deposit`.
14        emit Deposit(msg.sender, _id, msg.value);
15    }
16 }
```



# Logs

```
1 contract C {
2     function f() public payable {
3         uint256 _id = 0x420042;
4         log3(
5             bytes32(msg.value),
6             bytes32(0
7                 x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83
8                 ),
9             bytes32(uint256(msg.sender)),
10            bytes32(_id)
11        );
12    }
13 }
```

- ▶ log0, log1, log2, log3, log4 available (for 1, 2, 3, 4, 5 parameters)

# Abstract Contracts and Interfaces

## Abstract Contracts

- ▶ A contract that does not implement all its functions is abstract

## Interface

- ▶ A contract that does not implement any of its functions is an interface
- ▶ No constructor,
- ▶ No state variables
- ▶ All functions must be "external"
- ▶ Cannot inherit other contracts or interfaces

```
1 interface Token {  
    enum TokenType { Fungible, NonFungible }  
3     struct Coin { string obverse; string reverse; }  
    function transfer(address recipient, uint amount) external;  
5 }
```

# Contracts Hands-On

# Contracts in Web3.js

## Compiling using Solc

- ▶ Your first Solidity contract in this folder

```
cd /eclipse-workspace/EthereumWeb3jSolidity/src
```

- ▶ Compile with the Solidity compiler, targeting the "byzantium" Ethereum virtual machine (EVM), and produce the bytecode (binary) and an ABI description file:

```
solc --bin --abi --evm-version byzantium \  
  --overwrite -o . Contract1.sol
```

- ▶ This generates the EVM bytecode and the ABI description files in the current directory.

# Deployment

## Deployment

- ▶ Copy/paste the ABI description into the following contract definition:

```
def = eth.contract(...ABI Description...)
```

- ▶ Create a new contract instance by copy/pasting the EVM bytecode into the contract creation:

```
inst = def.new({from: eth.accounts[0], data:  
'0x...EVM Bytecode...', gas: '4700000'}, function  
(e, contract) { console.log('Err: '+e+' Tx:  
'+contract.transactionHash+' Addr:  
'+contract.address)})
```

- ▶ It may take a short while for this to be mined into the blockchain. You will get two notifications, the first one with just the transaction hash, the second one with the contract address as well.

```
inst
```

# Interacting with the Contract

- ▶ Use `call()` to call a view or a pure function.
- ▶ Doesn't cost any gas
- ▶ Example:

```
inst.getCounter.call()
```

- ▶ Use `sendTransaction()` to call a state-changing function,
- ▶ Requires "gas" to make it go.
- ▶ Example:

```
inst.inc.sendTransaction({from: eth.accounts[0], \  
  gas: '50000'})
```

# Solidity from Web3j

## Eclipse Java Project

- ▶ Open project "EthereumWeb3jSolidity" in your Eclipse workspace
- ▶ Contains a basic application, and the solidity contract "Contract1.sol"

## Compiling with Solc

- ▶ Same as above, only required if you made changes

```
cd /eclipse-workspace/EthereumWeb3jSolidity/src
```

```
solc --bin --abi --evm-version byzantium \  
--overwrite -o . Contract1.sol
```

# Solidity in Web3J

## Create Java Wrapper

```
web3j solidity generate -b Contract1.bin \  
-a Contract1.abi -p Default -o .
```

- ▶ Creates a Java wrapper class for the contract in package "Default".
- ▶ Wrapper class provides methods deploy and interact with the contract.
- ▶ Refresh your Eclipse project and open "Contract1.java"
- ▶ Run the "EthereumSolidityApplication" to see your contract in action



# Events in Web3J

## Project

- ▶ Open project "EthereumWebjSolidityFilter" in your Eclipse workspace
- ▶ Contains a basic application, and the solidity contract "Contract2.sol"

## Compile with Solc

```
cd /eclipse-workspace/EthereumWeb3jSolidityFilter/src  
  
solc --bin --abi --evm-version byzantium \  
    --overwrite -o . Contract2.sol
```

## Create Java Wrapper

```
web3j solidity generate -b Contract1.bin \  
    -a Contract1.abi -p Default -o .
```

- ▶ Run the "EthereumSolidityApplicationFilter" to see your contract in action

# Events in Web3J

## Topics

- ▶ Process events raised from each transaction in transaction receipt
- ▶ subscribe to a filter for certain contract address and topics
- ▶ Topics are created for signature and value of events:
  - ▶ Example for the event "ReceiveMessage" declared in "Contract2":

```
web3.sha3('ReceiveMessage(address,string,uint256)')  
"0x3274ee9181d28c0968a16da3b8db237fb6f8e7c6e056ab3d0638e34bcbd43e4b"
```

# Events in Web3J

## Log Data

- ▶ Log data can be decoded based on the parameters of the event. It is given in chunks of 32 bytes (64 hex digits):

```
00000000000000000000000000000000a1ffbe52c67aa5a9b45a15af63501a9e690c06ad  
0000000000000000000000000000000000000000000000000000000000000000000060  
000000000000000000000000000000000000000000000000000000000000000000001  
00000000000000000000000000000000000000000000000000000000000000000000b  
48656c6c6f20576f726c64000000000000000000000000000000000000000000000000
```

- ▶ The first 32 bytes are the sender address (left padded with zeros)
- ▶ The second 32 bytes are the string. For variable length parameters, this is given as an offset into the data:  $0x60 = 96$ .

```
web3.toAscii("0x48656c6c6f20576f726c64")
```

- ▶ The third 32 bytes are the uint256:  $0x01 = 1$

Decoding the data requires the ABI!

# Solidity in Remix

## Remix

- ▶ Point your browser to `https://remix.ethereum.org`
- ▶ Provides its own Ethereum node
- ▶ Offers different compiler versions
- ▶ Compile, run, and debug your contracts
- ▶ Creates Web3.js scripts for contract deployment

# Solidity in Remix

The image shows the Remix Solidity IDE interface in a Mozilla Firefox browser. The browser address bar shows the URL `https://remix.ethereum.org/#optimize=false&version=` at 120% zoom. The IDE title is "Remix - Solidity IDE - Mozilla Firefox".

The main editor displays a Solidity contract named `SimpleStorage` in `browser/Untitled.sol`. The code is as follows:

```
1 | pragma solidity ^0.5.0;
2 |
3 | contract SimpleStorage {
4 |
5 |     uint storedData;
6 |
7 |     function set(uint x) public {
8 |         storedData = x;
9 |     }
10 |
11 |     function get() view public returns (uint retVal) {
12 |         return storedData;
13 |     }
14 |
15 | }
```

The left sidebar shows a file explorer with a "browser" folder containing `ballot_test.sol`, `ballot.sol`, and `Untitled.sol`, and a "config" folder.

The right sidebar contains the compiler settings panel. It shows the current version as `version:0.5.1+commit.c8a2cb62.Emscripten.clang`. There is a dropdown menu for "Select new compiler version". Below this are checkboxes for  "Auto compile",  "Enable Optimization", and  "Hide warnings". A "Start to compile (Ctrl-S)" button is present.

Below the compiler settings, there is a dropdown menu for the selected contract, currently showing `SimpleStorage`. Below that are buttons for "Details", "ABI", and "Bytecode".

The bottom console area shows a list of installed dependencies:

- web3\_version 1.0.0
- ethers.js
- swarmgw
- compilers - contains currently loaded compiler

Below the dependencies, there are instructions:

- Executing common command to interact with the Remix interface (see list of commands above). Note that these commands can also be included and run from a JavaScript script.
- Use `exports.register(key, obj).remove(key).clear()` to register and reuse object across script executions.

The console prompt is `>`.

# Solidity Exercise

## Create a contract for auctions

- ▶ Contract owner starts a new auction with item description, minimum bid and reserve price
- ▶ External accounts can bid on an auction
- ▶ Bids require deposits
- ▶ Contract owner manually ends auctions
- ▶ Deposit of highest bidder is sent to contract owner
- ▶ Deposits of losing bidders are refunded

## Tips

- ▶ Develop and test in Remix
- ▶ Search the internet for help, especially on funds management

# Byzantine Fault Tolerance

# The Bigger Picture

- ▶ Blockchains perform:
  - ▶ Validation of blocks
  - ▶ Ordering of blocks
- ▶ Blockchains achieve **consensus** of validity and order



# Ordering Consensus

## Requirements

- ▶ All honest nodes agree on the same order
- ▶ All transactions/blocks must be ordered
- ▶ Nodes must be able to leave and join the network
- ▶ Resilient against dishonest nodes

## Options

- ▶ Proof of work
- ▶ Proof of stake voting
- ▶ Byzantine fault tolerance

# Byzantine Fault Tolerance

## Problem

- ▶ Actors must agree on strategy
- ▶ Some actors may be unreliable or malicious
- ▶ Actors have unreliable communication
- ▶ Actors may be inconsistent
- ▶ Actors may present different states to different observers

## Solution (Pease, Shostak, Lamport, 1980)

- ▶ Minimum of  $3n + 1$  nodes required to survive  $n$  failed/malicious nodes
- ▶ Practical BFT algorithm (Castro, Liskov, 1999, 2002)
- ▶ Multi-stage protocols requiring fully-connected nodes
- ▶ Is blockchain proof-of-work a BFT method? If it is, is it an efficient one?

# Byzantine Fault Tolerance

## Comparison

- ▶ More efficient than proof-of-work
- ▶ Higher communication overhead than proof-of-work
- ▶ Potentially faster than proof-of-work
- ▶ Not as resilient as proof-of-work
- ▶ Finality of order consensus

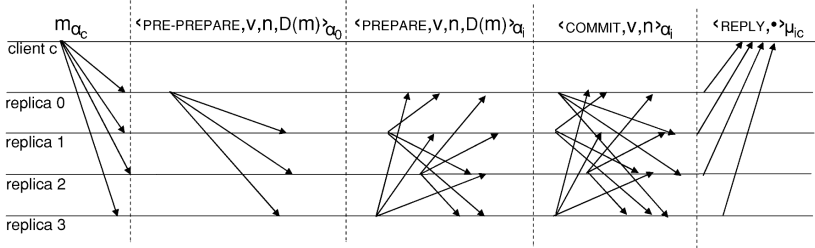
# PBFT Algorithm

## Clients

1. Client multicasts (REQUEST,  $o$ ,  $t$ ,  $c$ ) messages to nodes
  - ▶  $c$  is the client,
  - ▶  $t$  is the timestamp,
  - ▶  $o$  is the operation to be executed
2. Nodes accept request, send (REPLY,  $v$ ,  $t$ ,  $c$ ,  $i$ ,  $r$ ) to client
  - ▶  $v$  is the current view,
  - ▶  $i$  is the node number,
  - ▶  $r$  is the result of executing operation
3. Client waits for  $f + 1$  replies with the same  $t$  and  $r$  before accepting result  $r$  ("reply certificate")
4. Clients may retransmit request when reply certificate not received
  - ▶ Nodes may re-send replies if request already processed
  - ▶ If primary node does not assign valid sequence number, it will be suspected faulty and lead to view change

# PBFT Algorithm

## Ordering Protocol



Source: Castro and Liskov (2002), ACM

# PBFT Algorithm

3-phase protocol to multicasts requests and replies:  
*pre-prepare, prepare, commit*

## Pre-Prepare

1. When primary node receives request  $m$  from client, it assigns a sequence number  $n$
2. Primary node multicasts (PRE-PREPARE,  $v$ ,  $n$ , hash( $m$ ))
  - ▶  $v$  is the view,
  - ▶  $n$  is the sequence number,
  - ▶  $m$  is the request

# PBFT Algorithm

## Prepare

1. Node  $i$  accepts PRE-PREPARE for view  $v$  and sequence  $n$  only if it has not already done for a different  $\text{hash}(m)$ , and that  $n$  is between a low and high boundary ( $h, H$ )
2. If node  $i$  has received corresponding request  $m$ , and has accepted PRE-PREPARE, it multicasts  $(\text{PREPARE}, v, n, \text{hash}(m), i)$ 
  - ▶ "Node accepts sequence #  $n$ ", "request is pre-prepared at  $i$ "
3. Every node collects messages until it has a quorum certificate with the PRE-PREPARE message and  $2f$  matching PREPARE messages for sequence  $n$ , view  $v$  and request  $m$ .
  - ▶ "prepare certificate", "request is prepared at node  $i$ "

# PBFT Algorithm

## Commit

1. Each node  $i$  multicasts (COMMIT,  $v$ ,  $n$ ,  $i$ ) indicating it has accepted the prepare certificate
2. Each node collects messages until it has a quorum certificate with  $2f + 1$  matching COMMIT messages for sequence  $n$ , view  $v$  and request  $m$ .
  - ▶ "commit certificate", "request is committed at node  $i$ "
3. Each node executes the requested operation when  $m$  is committed and the node has executed all previous requests.
4. After executing the request, nodes send REPLY to the client



# PBFT Algorithm

Nodes periodically discard information about past requests

## Checkpointing

1. When node  $i$  produces or fetches a checkpoint request number, it multicasts a (CHECKPOINT,  $n$ ,  $d$ ,  $i$ ) message
  - ▶  $n$  is the sequence number of the last request,
  - ▶  $d$  is the hash of the state
2. Each node collects messages until it has a quorum certificate with  $2f + 1$  CHECKPOINT messages with the same  $n$  and  $d$ 
  - ▶ "stable certificate"
3. Nodes discard log entries with sequence numbers less than  $n$  and earlier checkpoints

# PBFT Algorithm

## View

- ▶ Set of nodes with a designated primary or leader node

## View Change

1. When a node  $i$  suspects the primary to be faulty, it multicasts as (VIEW-CHANGE,  $v + 1$ ,  $h$ ,  $C$ ,  $P$ ,  $Q$ ,  $i$ ) message
  - ▶  $h$  is the sequence number of the last stable checkpoint;
  - ▶  $C$  is a set of pairs with the sequence number and hash of each checkpoint,
  - ▶  $P$  are requests that have been prepared in previous views,
  - ▶  $Q$  are requests that have been pre-prepared in previous views
2. Node  $i$  discards PRE-PREPARE, PREPARE, and COMMIT messages

# PBFT Algorithm

## View Change Acknowledgment

1. Each nodes collects VIEW-CHANGE messages for  $v + 1$  and sends acknowledgements to  $v + 1$ 's leader  $p$
2. Nodes accept VIEW-CHANGE messages only if all information in  $P$  and  $Q$  is for view numbers less than or equal  $v$
3. Nodes multicast (VIEW-CHANGE-ACK,  $v + 1$ ,  $i$ ,  $j$ ,  $d$ ) messages
  - ▶  $i$  is the sender node,
  - ▶  $d$  is the hash of the VIEW-CHANGE message being acknowledged,
  - ▶  $j$  is the node that sent the VIEW-CHANGE message

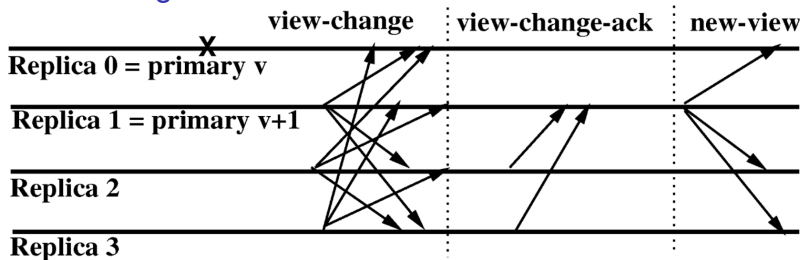
# PBFT Algorithm

## New View

1. New primary collects VIEW-CHANGE and VIEW-CHANGE-ACK messages
2. Add VIEW-CHANGE message from  $i$  to set  $S$  only after receiving  $2f - 1$  VIEW-CHANGE-ACK messages for  $i$ 's VIEW-CHANGE message
  - ▶ "view change certificate"
3. New primary chooses a checkpoint and set of requests when a message is added to  $S$
4. Primary multicasts (NEW-VIEW,  $v + 1$ ,  $V$ ,  $X$ )
  - ▶  $V$  contains the sending node  $i$  and hash of VIEW-CHANGE message for every VIEW-CHANGE message in  $S$  ("new view certificate"),
  - ▶  $X$  identifies the checkpoint and request value selected
5. New primary gets all requests in  $X$  and checkpoint  $h$
6. Each node in  $v + 1$  collects messages until they have correct NEW-VIEW message and matching VIEW-CHANGE messages for each pair in  $V$ , then verifies  $X$  and  $h$

# PBFT Algorithm

## View Change Protocol



Source: Castro and Liskov (2002), ACM

# BFTSmart

- ▶ Java library
- ▶ Ensures all nodes receive messages in same order
- ▶ Ensures nodes can survive crashes, leave and re-join node set

# BFTSmart

## Server

- ▶ Connects to other servers and clients via network sockets
- ▶ Accepts *ordered* and *unordered* requests
  - ▶ `appExecuteOrdered()`
  - ▶ `appExecuteUnordered()`
- ▶ Exchange state with new or recovering servers
  - ▶ `getSnapshot()`
  - ▶ `installSnapshot()`
- ▶ Serialization using Java serialization into byte arrays

## Client (Proxy)

- ▶ Connects to servers via network sockets
- ▶ Exposes functionality that is backed by the ordering servers
- ▶ Invokes/requests functions on server
  - ▶ `invokeOrdered()`
  - ▶ `invokeUnordered()`

# BFTSmart

## Configuration

- ▶ Directory "config"
- ▶ Network addresses for servers in "hosts.conf"
- ▶ Network configuration in "system.conf"
- ▶ Current view in "currentView (should delete before new start)

## Example

- ▶ Demo from BFTSmart in Eclipse project "BFTSmartExample"
- ▶ Implements a distributed map to store objects



# Exercise

Use BFTSmart to Implement a Simple Blockchain  
Ordering Service



