The Blockmania Consensus Protocol &
Scaling Distributed Ledgers with Chainspace

A Research Talk
(zero marketing = zero liability)
Who is George Danezis?
A brief introduction

Passion: Decentralization & Privacy

Co-Founder & Head of Research at chainspace.io

Prof. of Security and Privacy Engineering at University College London, London.

Before: Microsoft Research, KU Leuven, Cambridge (academic dad: Ross Anderson)

UCL is actively recruiting faculty, post-docs and PhD students in security. Apply!
Where to go find out more

Our research papers

George Danezis, Dave Hrycyszyn:
**Blockmania: from Block DAGs to Consensus.**
CoRR abs/1809.01620 (2018)

Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, George Danezis:
**Chainspace: A Sharded Smart Contracts Platform.** NDSS 2018

And some sneak previews of unpublished material.
Outline

How to build reliable distributed systems?

What is consensus, and what is it good for?

What is ‘the simplest’ practical form of Byzantine consensus?

How to implement it as efficiently as possible?

How can we scale ‘blockchains’ beyond faster consensus?
Consensus as a primitive has been studied since the 1980s.

Bitcoin proposed “Nakamoto Consensus”. Ethereum uses it.

Pro: open membership through PoW. Con: Weak finality, and energy hungry.

Renewal of interest in “traditional” consensus protocols.

Why care about Consensus?

Smart contracts, distributed ledgers and Blockchains
Set of network nodes, that may be subject to failures.

Consensus is a joint network protocol to make a joint decision.

Agreement (safety) – they want to all take the same decision.

Liveness & finality – they all eventually take a decision, and it is final.

Single decision or sequence of decisions (optimization)

What is consensus?

Building block of reliable distributed systems.
Consensus is key to reliable distributed systems

State machine replication paradigm for secure distributed computing (Fred Schneider, 1990)

All replicas start at State 0 and execute the same sequence of operation resulting in the same state i+1.
Flavors of consensus.

- **Network model**: Synchronous, asynchronous, partial synchrony.
- **Failure model**: crash-fail, crash-recovery, byzantine.
- **Initiator**: Honest or byzantine.

**Blockmania**: asynchronous safety, partial synchrony for liveness.

**Core**: simplification of PBFT protocol (Liskov & Castro, 1999)
Hard Limits
Limits to asynchronous Byzantine consensus

• **FLP theorem**: byzantine consensus is impossible, even with a single faulty node, under full asynchrony for a deterministic protocol.

• Solution: **partial synchrony**. After some period of asynchrony, the system becomes synchronous.

• **Synchrony**: messages from honest nodes are received within a known delay by other honest nodes.

• **Tolerance to faulty nodes**: \(3f+1\) participants are required to tolerate up to \(f\) faulty nodes.
The Blocmania Core Consensus Algorithm
Blockmania / PBFT core consensus (happy path, view 0)

- A participant $n_0$ proposes a block for slot $k$. All need to agree on it, or agree on ‘no block’.
- **Why**: A byzantine participant $n_0$ may propose conflicting blocks, or no blocks.

Bracha’s Reliable Broadcast (1985)

Instance $(n_0, k)$

- **Pre-prepare**: Prepare first proposal
- **Prepare**: Wait for $2f+1$ same propose
- **Commit**: Wait for $2f+1$ same commit

Commit messages must contain $2f+1$ prepare.

Deliver!
Insight: why do we need 2f+1 good nodes?

- Consider **both n₀ and n₁ are byzantine** – N < 3f+1. Example attack: **failed agreement**.

![Diagram](image.png)

- Incorrect operations = equivocation
- But why not wait for the other honest one?
Insight: why have a Commit Phase & View Change.

Why not simply use Bracha's Broadcast (pre-propose & propose Phases)?

Liveness under faulty (or slow) initiator:
- Initiator does not sent a value for \((n, k)\)?
- Initiator sends contradictory values for \((n, k)\)?
- Initiator or network is too slow, and no delivery happens within some timeout?

Solution: view change & new view protocols:
- Nodes time out & broadcast “ViewChange”: 2f+1 messages, new view for the same decision.
- Must not rely on the same initiator -> might be faulted!
- Commit phase: safety across views. Must propose the same value if one was committed.
- How to tune timeouts?
View Change & New View Preserves Liveness (1)

- Consider $n_0$ is byzantine – $N = 3f+1$. Example attack: failed termination for view 0.
Consider $n_0$ is byzantine — $N = 3f + 1$.

- **Pre-prepare**
  - Wait for $2f+1$ same propose
- **Prepare**
  - Must keep promises from previous views.
- **View Change**
  - Wait for $2f+1$ View change
- **New View**
  - Instance $(n_0, k)$

**Diagram Notes:**
- **Bad $n_0$**
- **Instance $(n_0, k)$**
- **Prepare first proposal**
- **Wait for 2f+1 same propose**
- **View Change**
- **New View**
- **Prepare**
- **Timeout!**

**Legend:**
- MUST keep promises from previous views.
Blockmania vs PBFT View Change Simplifications

Traditional PBFT is complex:
• Rotate leader.
• Decide on a sequence of decisions/transactions.
• New leader must propose a value for all previous positions.

Blockmania takes a simpler view:
• No special leader (but initiator for each instance).
• Each instance of the consensus protocol to decide one block per node / position. \((n_i, k) \rightarrow B\).
• On new view either any node propose: (1) “no block” if none of the \(2t+1\) have committed or (2) the one value committed (there can only be one).
• Finality: either decide a block for a position, or “no block”.
From block agreement to full consensus

Order all transaction in decisions \((n_i, k)\)

- Agree on a block, or ‘no block’ for all nodes in round \(k\).
- Apply a deterministic function to all transactions to get a total order.
- Hash of transaction = PoW.
- Order by fee.
- Commit then reveal + shared randomness for unbiasable order.
From Blockmania Instances to Full Consensus

- Run blockmania instance for each node and position
- Determine block $B_{i,k}$ or no block $NB_{i,k}$

Once all blocks in a round are determined, apply any deterministic ordering function to get a total order.

1. By Hash = PoW
2. By fee is what we do.
Efficient Network Instantiation
Sending explicit messages for all decisions is Naive.

Inefficiencies and complexities:

- **Mixing code** for networking (efficient asynchronous IO) & protocol logic are intermixed (correctness).
- **Explicit evidence** for all Commit, View Change, New View messages. Increase in size $O(N^3)$ to $O(N^4)$.
- **Full separate 3-rounds** for each decision.

Result: few PBFT quality implementations.

Problems with naïve implementations

Costs & complexities
Blockmania Architecture

Block DAG + Finalization

Consensus and finality layer (correctness)

Finality Layer

Consensus Decisions

Block DAG

Gossip Protocol (high perf. IO)
The Block DAG networking layer

- List of Transactions
- List of other block hashes.
- Previous block hash
- Signature

Node n at time k

Broadcast

Include Blocks with fully known history.

Include Valid Transactions

Clients

Other Nodes

(n, k-1)

Block (n, k)
The finalization layer (interpreting core consensus)

Decide for \((n_0, k)\)

Interpret as pre-prepare

Interpret as prepare

Interpret as commit

Interpret as deliver

Insight:

Since core protocol is deterministic can “simulate” the state of others through messages received and sent.
BLOCKMANIA

PERFORMANCE
Concrete WAN performance (Tx/sec) for different quorum sizes:

- 10 nodes (f=2) – 430K tx/sec
- 13 nodes (f=3) – 440K tx/sec
- 16 nodes (f=4) – 520K tx/sec

(not stat. different, Network bound).

Theory: $O(N^2)$ communication cost:
- Blocks are broadcast to all $O(N)$.
- Blocks are $O(N)$ (hashes)
- However, low constants:
  20 bytes * $N^2$ + transaction bytes * $N$
More blockmania topics:
- Byzantine clock sync.
- Encouraging partial synchrony.
- $O(n)$ variant of Blockmania.
- Sequential variants.
- Reliable Broadcast variants.
- Statistical variants (‘AvalanceMania’).
- Integration with Proof of Stake.

Questions so far?

And more topics for subsequent discussion
SHARDING FOR BETTER SCALABILITY

Chainspace & SBAC

- The world needs more than 500K tx/second
Scalability is not the same as a high number of tx/sec.

**Scalability:** the more resources you invest in the system the more tx/sec you can process.

PBFT/Blockmania: not scalable by that definition (cost $O(N) / O(N^2)$) in $N$ resources.

**Sharding is a generic solution.**

**Sharding:** ensure that a transaction only uses $O(1)$ to $O(\log N)$ resources to be processed.
Sharding Challenges

Not easy even in theory

Naïve sharding: just partition all state, and have the many different shards not interact with each other.

Problem: How to ensure atomicity for operations? Eg. I want a booking for a flight, hotel and conference to be “all-or-nothing”.

Naïve solution 2: No cross shard transactions (poor functionality); or super-shards deals with those (poor scalability).

Chainspace: Shards need to coordinate a little bit!
Chainspace execution model

**Objects**: Objects contain state within the system.

**Object status**: Objects may be active, inactive or locked. (Shard shared state!)

**Procedures**: Take one or more objects as inputs, and produce one or more object outputs.

Object status: to succeed a procedure should use “active” objects, and turns them inactive.

**Transaction**: A trace of execution of one or more transactions, including all the input and output objects for one or more procedures.

Why many? To allow subroutine calls and cross contact calls.

**Checkers**: Code that takes the trace of execution of a single procedure and returns true if it conforms to the contract.

Note: clients execute procedures, and pack transactions for checkers to check in shards.
Sharded Byzantine Atomic Commit Protocol (SBAC)

SBAC guarantees either all process transaction (eventually) or none does. (Safety)
Liveness follows from the liveness of consensus within each shard.
Performance (Summer 2017)

Validation: the more shards, the more transactions per second – linearly.
SBAC in the real-world
Security Under Composition: an attack

Client

T(x,y) -> z

Shard 0
(x)

s1: Commit(T)

Shard 1
(y)

s1: Abort(T)

Inconsistency: s0 aborts T, and s1 commits!

Solution: Associate with objects and transactions sequence numbers. Increment Those wisely. And use them to discard replays. (See manuscript soon).
Performance improvements.

Problem: opening a lot of sockets is expensive. \(O(S^2)\) in the number of shards per transaction.
Solution: Anyone can “drive this protocol” (thanks to Omniledger crew for this!)

Book\((h_i, f_i, c_i) \rightarrow r_i\)

What is the client dies? No problem: anyone else can continue the protocol. Nodes in Shards; third parties; other clients that want to make progress …
SBAC for fun, but mostly for profit.

Problem: SBAC is an expensive protocol. Only execute for a fee!
Solution: make SBAC steps conditional on commit for fee shard.

Book(x, y, fee) -> z, fee’

Design choice: consume fee if commit, or always?
The missing details

The joys of building real systems …

Why procedures vs checkers?
Privacy?
How to support light clients?
What if one or more shards do not have an honest supermajority?
How to shard audit and verification?
How to assign nodes to shards?
Smart contract lifetime management?
Separate checkers from nodes?
Updating smart contracts?
Non-deterministic contracts?
Sybil attack resistant open system?
Proof of stake economics?
Dynamic fees according to congestion?
…
Thanks for listening

george@chainspace.io