Snowflake

Current Families

1. “Classical consensus” - Leslie Lamport and Barbra Liscoff
2. Quick finality.
3. Quadratic communication.
4. Security Judicious Overlap between quorums of people.
5. Paxos is based on this.

Good basis for Permissions Chained.

1. Nakamoto 2009 - Robust
2. Slow (Bitcoin is 3-10 TPS, Power = 2 times Denmark, Ethereum runs 15+ TPS etc.)
3. High communication - Order n squared.
4. Limited Throughput - bandwidth / communication limited.
5. Expensive - When we are using these systems - we are paying for all that electricity. Either via minting new tokens or via transaction fees.

Snowflake (Avalance) 1. Finality in 2 seconds - 2. Thousands of TPS 3. Scales from 10,000 to millions of nodes 4. No special ecosystem of mining

It is a Gosip or Epidemic Protocol. This is basically a sub-sample voting system. In each round you are getting a sub-sample of the voting for a result and with each round you increase the probability of “A” result.

New Idea has “meta stability”

No stability in the middle. You want it to tip in some direction.

A Vote Process - but not by “counting” the votes.

The process - when you look at the proofs - relies on chaotic instability.

You as a listener can see the votes of the nodes that you talk to. You pick a solution (based on your data) and reveal your solution. Let’s call this solution “left”. You see the solutions of your neighbors (“left” or “right”) and you can choose witch direction to go in. If you see a preponderance of, let’s say, “right” you can choose to switch your solution to “right”. In each round you get this choice.

With a larger and larger set of nodes the rate at which you can “see” the “left” or “right” in a solution rises. More nodes leads to a faster spread of the instability and a faster rate of going from one solution to a common agreed upon solution.

A lying node - one attempting to produce a solution that is incorrect - has to persuade other nodes that it’s lie is more valid than all the other nodes promoting commonly accepted solutions. This makes the protocol robust.

The economy of the protocol means that it is egalitarian - It is runnable from virtually any computing device. A Phone is at a disadvantage because of network latency - not because of lack of compute power. However it is at no more of a disadvantage than a person on a slow network connection.

Evolution and Extra Legality

Bitcoin (Nakomoto) protocol is extra legal and brittle.

Snowflake can be persuaded to fix results via consensus. This makes it manageable by an avalanche of consensus. Centralized authority can then drive corrective actions to the “state” of the chain.

Implications

1. Participant (nodes) need not “participate” - this means that nodes may not affect safety of the system. Nodes are only visible if the “vote”.
2. “Green” System - low costs and low overhead.
3. No liveness grantee - A double spend scenario - you send to A, and B at the same time. There is no grantee to resolve this. The tokens can be stuck in the middle. There is no grantee that the chain will ever “choose” one over the other. In practical terms this is improbable - but - it puts the burden of responsibility on the “sender” who tried to defraud the receivers.

Avalanche (AVA) tokens based on this.

1. This uses a “staking” mechanism with traditional Solidity/Viper contracts (Ethereum) or via a off-chain based contract with Bitcoin.
2. The “stake” is a non-paying stake that prevents the creation of accounts for free. This stops the ability to have Sybil (51%) attacks on the system.
3. No “risk” - the stake is not a grantee of accuracy. There is no risk of loosing your stake - but you can not participate if you withdraw your stake. “Free” creates a commons that can be abused - “free” email created spam - this removes “free” - you have to have some “skin” in the game - to participate. The real “risk” or “cost” is opportunity cost risk.
4. Large enough staking allows the “minting” of new AVA tokens. This is a builtin interest rate that is payed on the “stake” in AVA tokens. It is low but it can change.

Governance

1. Corrections to the chain can take place. Via “consensus” you can change the minting rate - inflation - or the rate of return can be increased to “grow” the network. This is “dental banking” via chain consensus.

Stock v.s. Bonds

An example business with an ICO or Stock raise.

You want to start a business but you don’t have the capital to do it. SO you create a company - Then sell some stock to some investors.

You need to invest in hardware / fixed asset. This asset depreciates.

You need to pay for labor / this is a liability - the value can’t be counted.

Now you need more money - but you don’t want to sell more of the business.

The alternative is to get a “loan”.

There are 2 forms of loans - bank loans and bonds.

An example bond contract:

```
1: // SPDX-License-Identifier: MIT
2: pragma solidity >=0.4.22 <0.9.0;
```

```
 3:
 4: import "@openzeppelin/contracts/access/Ownable.sol";
 5: import "@openzeppelin/contracts/utils/math/SafeMath.sol";
 6: import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
 7:
 8: contract ZeroCouponBond is Ownable {
 9:
10:     using SafeMath for uint256;
11:
12:     string name;
13:     address tokenToRedeem;
14:     uint256 totalDebt;
15:     uint256 parDecimals;
16:     uint256 bondsNumber;    // How many bonds to issue
17:     uint256 cap;
18:     uint256 parValue;        // value of bond at maturity - face value.
19:     uint256 couponRate;        // discount rate that will be payed (interest)
20:     uint256 term;            // how long the bond last for
21:     uint256 timesToRedeem;
22:     uint256 loopLimit;        // stay under "gas" limit for actions in pay out.
23:     uint256 nonce = 0;
24:
25:     uint256 couponThreshold = 0;
26:
27:     ERC20 token;        // Use an ERC-20 as the "holder" of the tokens.
28:
29:     event MintedBond(address buyer, uint256 bondAmount);
30:         // emit MintedBond(buyer, _bondsAmount);
31:     event RedeemedCoupons(address sender, uint256[] bonds);
32:         // emit RedeemedCoupons(msg.sender, _bonds);
33:     event Transferred(address sender, address receiver, uint256[] bonds);
34:         // emit Transferred(msg.sender, receiver, _bonds);
35:
36:     mapping(uint256 => address) bonds;
37:     mapping(uint256 => uint256) maturities;
38:     mapping(uint256 => uint256) couponsRedeemed;
39:     mapping(address => uint256) bondsAmount;
40:
41:     constructor(string memory _name, uint256 _par, uint256 _parDecimals, uint256 _coupon,
42:             uint256 _term, uint256 _cap, uint256 _timesToRedeem, address _tokenToRedeem,
43:             uint256 _loopLimit) {
44:
45:         require(bytes(_name).length > 0);
46:         require(_coupon > 0);
47:         require(_par > 0);
48:         require(_term > 0);
49:         require(_loopLimit > 0);
50:         require(_timesToRedeem >= 1);
51:
52:         name = _name;
53:         parValue = _par;
54:         cap = _cap;
55:         loopLimit = _loopLimit;
56:         parDecimals = _parDecimals;
57:         timesToRedeem = _timesToRedeem;
58:         couponRate = _coupon;
59:         term = _term;
60:         couponThreshold = term.div(timesToRedeem);
61:
62:         if (_tokenToRedeem == address(0)) {
63:             tokenToRedeem = _tokenToRedeem;
64:         }
65:         else {
66:             token = ERC20(_tokenToRedeem);
67:         }
68:     }
```

```
69:
70:     /**
71:     * @notice Change the number of elements you can loop through in this contract
72:     * @param _loopLimit The new loop limit
73:     */
74:     function changeLoopLimit(uint256 _loopLimit) public onlyOwner {
75:         require(_loopLimit > 0);
76:         loopLimit = _loopLimit;
77:     }
78:
79:     /**
80:     * @notice Mint bonds to a new buyer - this is the bond "issue"
81:     * @param buyer The buyer of the bonds
82:     * @param _bondsAmount How many bonds to mint
83:     */
84:     function mintBond(address buyer, uint256 _bondsAmount) public onlyOwner {
85:         require(buyer != address(0));
86:         require(_bondsAmount >= 1);
87:         require(_bondsAmount <= loopLimit);
88:
89:         if (cap > 0) {
90:             require(bondsNumber.add(_bondsAmount) <= cap);
91:         }
92:         bondsNumber = bondsNumber.add(_bondsAmount);
93:         nonce = nonce.add(_bondsAmount);
94:         for (uint256 i = 0; i < _bondsAmount; i++) {
95:             maturities[nonce.sub(i)] = block.timestamp.add(term);
96:             bonds[nonce.sub(i)] = buyer;
97:             couponsRedeemed[nonce.sub(i)] = 0;
98:             bondsAmount[buyer] = bondsAmount[buyer].add(_bondsAmount);
99:         }
100:        totalDebt = totalDebt.add(parValue.mul(_bondsAmount))
101:                             .add((parValue.mul(couponRate)
102:                             .div(100)).mul(timesToRedeem.mul(_bondsAmount)));
103:
104:        emit MintedBond(buyer, _bondsAmount);
105:    }
106:
107:    /**
108:    * @notice Redeem coupons on your bonds
109:    * @param _bonds An array of bond ids corresponding to the bonds you want to redeem apon
110:    */
111:    function redeemCoupons(uint256[] memory _bonds) public {
112:
113:        require(_bonds.length > 0);
114:        require(_bonds.length <= loopLimit);
115:        require(_bonds.length <= getBalance(msg.sender));
116:
117:        uint256 issueDate = 0;
118:        uint256 lastThresholdRedeemed = 0;
119:        uint256 toRedeem = 0;
120:
121:        for (uint256 i = 0; i < _bonds.length; i++) {
122:
123:            if (bonds[_bonds[i]] != msg.sender || couponsRedeemed[_bonds[i]] == timesToRedeem) {
124:                continue;
125:            }
126:            issueDate = maturities[_bonds[i]].sub(term);
127:            lastThresholdRedeemed = issueDate.add(couponsRedeemed[_bonds[i]].mul(couponThreshold));
128:            if (lastThresholdRedeemed.add(couponThreshold) >= maturities[_bonds[i]] ||
129:                block.timestamp < lastThresholdRedeemed.add(couponThreshold)) {
130:                continue;
131:            }
132:
133:            toRedeem = (block.timestamp.sub(lastThresholdRedeemed)).div(couponThreshold);
134:
```

```
135:             if (toRedeem == 0) {
136:                 continue;
137:             }
138: 
139:             couponsRedeemed[_bonds[i]] = couponsRedeemed[_bonds[i]].add(toRedeem);
140:             getMoney( toRedeem.mul(parValue.mul(couponRate).div( 10 ** (parDecimals.add(2)) ) ), msg.sender );
141:             if (couponsRedeemed[_bonds[i]] == timesToRedeem) {
142:                 bonds[_bonds[i]] = address(0);
143:                 maturities[_bonds[i]] = 0;
144:                 bondsAmount[msg.sender]--;
145:                 getMoney(parValue.div( (10 ** parDecimals) ), msg.sender );
146:             }
147: 
148:         }
149: 
150:         emit RedeemedCoupons(msg.sender, _bonds);
151:     }
152: 
153:     /**
154:     * @notice Transfer bonds to another address
155:     * @param receiver The receiver of the bonds
156:     * @param _bonds The ids of the bonds that you want to transfer
157:     */
158:     function transfer(address receiver, uint256[] memory _bonds) public {
159:         require(_bonds.length > 0);
160:         require(receiver != address(0));
161:         require(_bonds.length <= getBalance(msg.sender));
162: 
163:         for (uint256 i = 0; i < _bonds.length; i++) {
164:             if (bonds[_bonds[i]] != msg.sender || couponsRedeemed[_bonds[i]] == timesToRedeem) {
165:             continue;
166:         }
167:             bonds[_bonds[i]] = receiver;
168:             bondsAmount[msg.sender] = bondsAmount[msg.sender].sub(1);
169:             bondsAmount[receiver] = bondsAmount[receiver].add(1);
170:         }
171: 
172:         emit Transferred(msg.sender, receiver, _bonds);
173:     }
174: 
175:     /**
176:     * @notice Donate money to this contract
177:     */
178:     function donate() public payable {
179:         require(address(token) == address(0));
180:     }
181: 
182:     receive() external payable {
183:         revert();
184:     }
185: 
186:     // ============================================================================================
187:     // Private Functions
188:     // ============================================================================================
189: 
190:     /**
191:     * @notice Transfer coupon money to an address
192:     * @param amount The amount of money to be transferred
193:     * @param receiver The address which will receive the money
194:     */
195:     function getMoney(uint256 amount, address receiver) private {
196:         if (address(token) == address(0)) {
197:             payable(receiver).transfer(amount);
198:         }
199:         else {
200:             token.transfer(msg.sender, amount);
```

```
201:         }
202:         totalDebt = totalDebt.sub(amount);
203:     }
204:
205:     // ============================================================================================
206:     // Getters
207:     // ============================================================================================
208:
209:     /**
210:     * @dev Get the last time coupons for a particular bond were redeemed
211:     * @param bond The bond id to analyze
212:     */
213:     function getLastTimeRedeemed(uint256 bond) public view returns (uint256) {
214:         uint256 issueDate = maturities[bond].sub(term);
215:         uint256 lastThresholdRedeemed = issueDate.add(couponsRedeemed[bond].mul(couponThreshold));
216:         return lastThresholdRedeemed;
217:     }
218:
219:     /**
220:     * @dev Get the owner of a specific bond
221:     * @param bond The bond id to analyze
222:     */
223:     function getBondOwner(uint256 bond) public view returns (address) {
224:         return bonds[bond];
225:     }
226:
227:     /**
228:     * @dev Get how many coupons remain to be redeemed for a specific bond
229:     * @param bond The bond id to analyze
230:     */
231:     function getRemainingCoupons(uint256 bond) public view returns (int256) {
232:         address owner = getBondOwner(bond);
233:         if (owner == address(0)) {
234:         return -1;
235:     }
236:         uint256 redeemed = getCouponsRedeemed(bond);
237:         return int256(timesToRedeem - redeemed);
238:     }
239:
240:     /**
241:     * @dev Get how many coupons were redeemed for a specific bond
242:     * @param bond The bond id to analyze
243:     */
244:     function getCouponsRedeemed(uint256 bond) public view returns (uint256) {
245:         return couponsRedeemed[bond];
246:     }
247:
248:     /**
249:     * @dev Get the address of the token that is redeemed for coupons
250:     */
251:     function getTokenAddress() public view returns (address) {
252:         return (address(token));
253:     }
254:
255:     /**
256:     * @dev Get how many times coupons can be redeemed for bonds
257:     */
258:     function getTimesToRedeem() public view returns (uint256) {
259:         return timesToRedeem;
260:     }
261:
262:     /**
263:     * @dev Get how much time it takes for a bond to mature
264:     */
265:     function getTerm() public view returns (uint256) {
266:         return term;
```

```
267:     }
268:
269:     /**
270:     * @dev Get the maturity date for a specific bond
271:     * @param bond The bond id to analyze
272:     */
273:     function getMaturity(uint256 bond) public view returns (uint256) {
274:         return maturities[bond];
275:     }
276:
277:     /**
278:     * @dev Get how much money is redeemed on a coupon
279:     */
280:     function getSimpleInterest() public view returns (uint256) {
281:         uint256 rate = getCouponRate();
282:         uint256 par = getParValue();
283:         return par.mul(rate).div(100);
284:     }
285:
286:     /**
287:     * @dev Get the yield of a bond
288:     */
289:     function getCouponRate() public view returns (uint256) {
290:         return couponRate;
291:     }
292:
293:     /**
294:     * @dev Get the par value for these bonds
295:     */
296:     function getParValue() public view returns (uint256) {
297:         return parValue;
298:     }
299:
300:     /**
301:     * @dev Get the cap amount for these bonds
302:     */
303:     function getCap() public view returns (uint256) {
304:         return cap;
305:     }
306:
307:     /**
308:     * @dev Get amount of bonds that an address has
309:     * @param who The address to analyze
310:     */
311:     function getBalance(address who) public view returns (uint256) {
312:         return bondsAmount[who];
313:     }
314:
315:     /**
316:     * @dev If the par value is a real number, it might have decimals. Get the amount of decimals the par value
317:     */
318:     function getParDecimals() public view returns (uint256) {
319:         return parDecimals;
320:     }
321:
322:     /**
323:     * @dev Get the address of the token redeemed for coupons
324:     */
325:     function getTokenToRedeem() public view returns (address) {
326:         return tokenToRedeem;
327:     }
328:
329:     /**
330:     * @dev Get the name of this smart bond contract
331:     */
332:     function getName() public view returns (string memory) {
```

```
333:         return name;
334:     }
335:
336:     /**
337:     * @dev Get the current unpaid debt
338:     */
339:     function getTotalDebt() public view returns (uint256) {
340:         return totalDebt;
341:     }
342:
343:     /**
344:     * @dev Get the total amount of bonds issued
345:     */
346:     function getTotalBonds() public view returns (uint256) {
347:         return bondsNumber;
348:     }
349:
350:     /**
351:     * @dev Get the latest nonce
352:     */
353:     function getNonce() public view returns (uint256) {
354:         return nonce;
355:     }
356:
357:     /**
358:     * @dev Get the amount of time that needs to pass between the dates when you can redeem coupons
359:     */
360:     function getCouponThreshold() public view returns (uint256) {
361:         return couponThreshold;
362:     }
363:
364: }
```