Online Pseudonym Parties, a proof-of-unique-human system

Johan Nygren, johanngrn@gmail.com

ABSTRACT: Online Pseudonym Parties is based on pseudonym events, global and simultaneous verification events that occur at the exact same time for every person on Earth. In these events, people are randomly paired together, 1-on-1, to verify that the other is a person, in a pseudo-anonymous context, over video chat. The event lasts 15 minutes, and the proof-of-unique-human is that you are with the same person for the whole event. The proofs, valid for a month, can be disposed of once no longer valid, and are untraceable from month to month.

Introduction, or What is it like to be a nym?

Online Pseudonym Parties is a population registry for a new global society that is built on top of the internet. It provides a simple way to give every human on Earth a proof-of-unique-human, and does so in a way that cannot exclude or reject a human being, as long as the average person in the population would recognize them as human. This means it is incapable of shutting anyone out, and that it will inherently form a single global population record, that integrates every single human on Earth. It is incapable of discrimination, because it is incapable of distinguishing one person from another, since it has absolutely zero data about anyone. It's incapable of discriminating people by gender, sexuality, race or belief. The population registry also has infinite scalability. The pairs, dyads, are autonomous units and the control structure of Online Pseudonym Parties. Each pair is concerned only with itself, compartmentalized, operating identically regardless of how many times the population doubles in size.

The "court" system, and the "virtual border"

The key to Online Pseudonym Parties is the "court" system, that subordinates people under a random pair, to be verified in a 2-on-1 way. This is used in two scenarios, both equally important. The first, is when a person is paired with an account that does not follow protocol. In most cases, this would be a computer script, or a person attempting to participate in two or more pairs at the same time. Normally, in the pairs, people have to mutually verify each other to be verified. In case a person is paired with an attacker, they can choose to break up their pair, and subordinate both people in the pair under a random pair each, a "court". The attacker, if they were a bot, will not be verified by the court, or have any ability to coerce it, while the normal person will be verified by their "court". Both people in the pair that "judges" a court have to verify the person being judged.

```
struct Court { uint id; bool[2] verified; }

mapping (uint => mapping (address => Court)) public court;
mapping (uint => mapping (uint => address)) public courtIndex;
mapping (uint => uint) public immigrants;

function dispute(bool _premeet) external {
    uint t = schedule() - (2 + boolToUint(_premeet));
    uint id = nym[t][msg.sender].id;
    require(id != 0);
    if(_premeet == false) require(!pairVerified(t, getPair(id)));
    pair[t][getPair(id)].disputed = true;
}
```

The second scenario for the "court" system, is the "virtual border". The population has a form of "border" or "wall" around it, and anyone not verified in the previous event is on the "outside" of this "border". To register, you need to go through an "immigration process". During this process, you are assigned to a "court", another pair, and they verify you in a 2-on-1 way, so that a bot would have no way to intimidate or pressure this "border police". This "border", together with the dispute mechanism, acts as a filter that prevents any attackers to the system.

2, 4, 8, 16… 1 billion, growth by doubling

The pairs are the control structure of Online Pseudonym Parties, and each pair is concerned only with itself, regardless of how many times the population doubles in size. The population is allowed to double in size each event, made possible by that each person verified during an event is authorized to invite another person to the next event. This allows the population to grow from 2 to 10^3 (1024) in 10 events, 10^3 to 10^6 (1048576) in 20 events, and 10^6 to 10^9 (1073741824) in 30 events. There is no theoretical upper limit to this scalability mechanism, Online Pseudonym Parties has infinite scalability. The constructor allows the contract to intialize from two people, and can also be used when transferring the population from one ledger to another as digital ledgers get more advanced.

```
constructor(uint _population, address _genesis) {
    balanceOf[schedule()][Token.Registration][_genesis] = _population;
    balanceOf[schedule()+1][Token.Registration][_genesis] = _population;
}
```

Randomization, Vires in Numeris

The randomization of the pairs is a major security assumption, and is very simple. The processes are separated over two phases. In the first phase, people register and are appended to a list. In the second phase, the list is shuffled using a randomly generated seed. The population is able to autonomously generate the random seeds that secure the system, using “random majority vote”, described in the next section.

```
uint entropy;

struct Nym { uint id; bool verified; }

mapping (uint => mapping (address => Nym)) public nym;
mapping (uint => address[]) public registry;

function register(bytes32 _commit) external {
    require(_commit != 0);
    uint t = schedule();
    require(shufflerIndex[t][msg.sender] == 0);
    require(!halftime(t));
    deductToken(Token.Registration, t);
    registry[t].push(msg.sender);
    commit[t][msg.sender] = _commit;
    shufflers[t].push(msg.sender);
    shufflerIndex[t][msg.sender] = shufflers[t].length;
}
function shuffle(address _shuffler) external {
    uint t = schedule()-1;
    uint unshuffled = shufflers[t].length;
    uint shuffled = registered(t)-unshuffled;
    if(shuffled == 0) entropy = generator[t][leader[t]];
    uint randomNumber = shuffled + entropy % unshuffled;
    entropy ^= uint160(registry[t][randomNumber]);
    (registry[t][shuffled], registry[t][randomNumber]) = (registry[t][randomNumber], registry[t][shuffled]);
    nym[t][registry[t][shuffled]].id = shuffled+1;
    clearShuffler(_shuffler, t);
}
```

The population generates random numbers

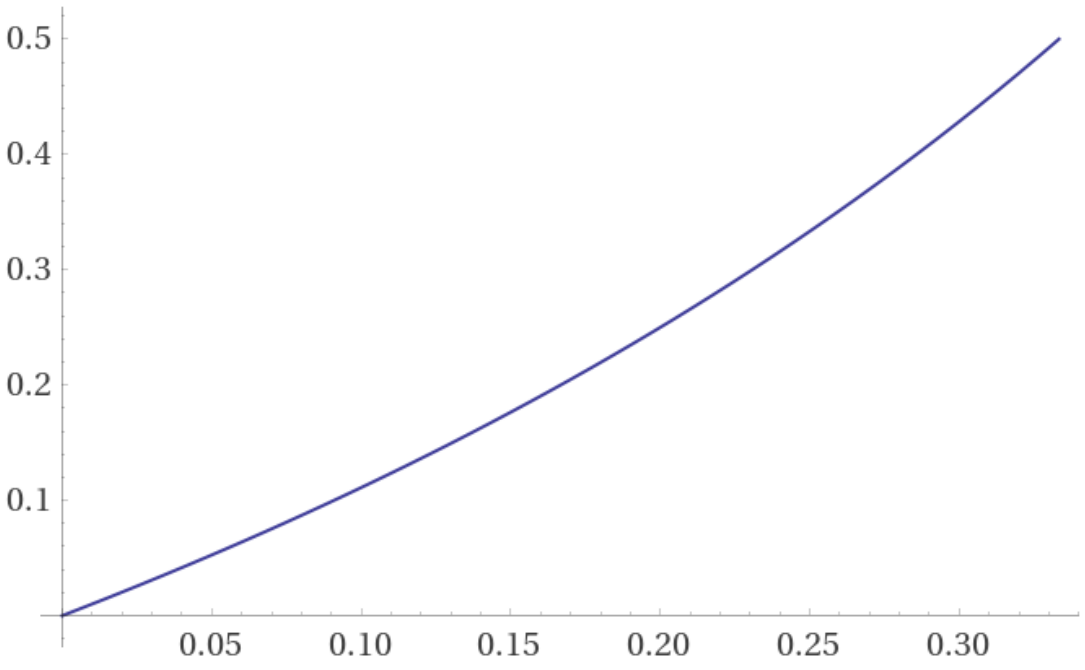
The population has a protocol that allows them to generate a random numbers each period in a way that cannot be manipulated by foreign actors. It relies on a form of majority vote, but, no one can control what they vote for. They can only control that their vote is random, and know that the vote of every other person is random. This is possible by using a commit-reveal scheme, where votes are committed before the votes in the previous period have been revealed. The random number generated by majority vote each period, is used to “mutate” the votes in the next period. Votes contribute randomness to generators. There are as many generators as there are pairs, and the probability that some generator gets k “hits” is 2^k/(e^2 k!). The generator that gets the most “hits”, i.e., a majority vote, wins.

```
mapping (uint => mapping (address => bytes32)) public commit;

mapping (uint => mapping (uint => uint)) public generator;
mapping (uint => mapping (uint => uint)) public points;
mapping (uint => uint) public leader;
```

Collusion attacks

Online Pseudonym Parties is vulnerable to one type of attack vector only, collusion attacks. The success of collusion attacks increases quadratically, as x^2, where x is the percentage colluding. Repeated attacks conform to the recursive sequence a[n] == (x + a[n—1])²/(1 + a[n—1]), and can be seen to approach n as x -> 1. It plateaus at the limit $a[\infty] = x^2/(1-2x)$ for $0<x<0.5$. Colluders reach 50% control when $(a[\infty] + x) == (1+a[\infty])/2$, this happens at x = 1/3, i.e., Online Pseudonym Parties is a 66% majority controlled system.



plot (x^2/(1−2x)+x)/(1+x^2/(1−2x)) from 0 to 1/3 | Computed by Wolfram|Alpha

Scheduling the pseudonym event

Scheduling is trivial. The current month is calculated using a timestamp for the genesis event, the periodicity in seconds, and the current time.

```
uint constant period = 4 weeks;
uint constant genesis = 198000;

function schedule() public view returns(uint) { return ((block.timestamp - genesis) / period); }
function toSeconds(uint _t) public pure returns (uint) { return genesis + _t * period; }
```

The event is scheduled to always happen on the weekend, for all time zones. The exact hour of the event varies, to be fair to all time zones.

```
mapping (uint => uint) public hour;
function computeHour(uint _t) public { hour[_t] =  1 + uint(keccak256(abi.encode(_t)))%24; }

function pseudonymEvent(uint _t) public returns (uint) {
   if(hour[_t] == 0) computeHour(_t); return toSeconds(_t) + hour[_t]*1 hours;
}
```

An anonymous population registry

Since “who a person is” is not a factor in the proof, mixing of the proof-of-unique-human does not reduce the reliability of the protocol in any way. It is therefore allowed, and encouraged. This is practically achieved with “tokens” that are intermediary between verification in one pseudonym event and registration for the next event, authorizing mixer contracts to handle your “token” using the approve() function.

```
enum Token { Personhood, Registration, Immigration }

mapping (uint => mapping (Token => mapping (address => uint))) public balanceOf;
mapping (uint => mapping (Token => mapping (address => mapping (address => uint)))) public allowed;

function _transfer(uint _t, address _from, address _to, uint _value, Token _token) internal {
   require(balanceOf[_t][_token][_from] >= _value);
   balanceOf[_t][_token][_from] -= _value;
   balanceOf[_t][_token][_to] += _value;
}
function transfer(address _to, uint _value, Token _token) external {
   _transfer(schedule(), msg.sender, _to, _value, _token);
}
function approve(address _spender, uint _value, Token _token) external {
   allowed[schedule()][_token][msg.sender][_spender] = _value;
}
function transferFrom(address _from, address _to, uint _value, Token _token) external {
   uint t = schedule();
   require(allowed[t][_token][_from][msg.sender] >= _value);
   _transfer(t, _from, _to, _value, _token);
   allowed[t][_token][_from][msg.sender] -= _value;
}
```

Proof-of-unique-human as a commodity

The proof-of-unique-human is only valid for one month, untraceable from month to month, and disposable once expired. The population registry has no concept of “who you are”. It cannot distinguish one person from another. Any other contract can build applications based on this proof-of-unique-human just by referencing proofOfUniqueHuman[_period][_account]. Applications that use some kind of majority vote, can reference population[_period] to know how many votes are needed to be >50% of the population.

```
mapping (uint => uint) public population;
mapping (uint => mapping (address => bool)) public proofOfUniqueHuman;

function claimPersonhood() external {
    uint t = schedule();
    deductToken(Token.Personhood, t);
    proofOfUniqueHuman[t][msg.sender] = true;
    population[t]++;
}
```

“I’m doing my part” to shuffle()

To enforce that each person contributes to invoking shuffle(), to ensure the computational resources needed for it, while still allowing anyone else to invoke shuffle() as many times as they want, the function clearShuffler() is used. When people register, their account is also required to participate in the shuffle process, to be allowed to participate in the Pseudonym event. At the same time, people are also required to have been shuffled, and have been assigned their id. And, it makes no sense to not allow anyone to call shuffle(), since the faster people are shuffled, the faster the pairs are recorded on the ledger, allowing more time for “pre-meetings”. These factors are resolved by that anyone calling “shuffle()” has to include the address of an account that is required to call it. If people are rude and refuse to do their “part”, the incentive to call shuffle() is divided between those people, and, the people who has not yet been shuffled, equally.

```
mapping (uint => address[]) public shufflers;
mapping (uint => mapping (address => uint)) public shufflerIndex;

function clearShuffler(address _shuffler, uint _t) internal {
    uint index = shufflerIndex[_t][_shuffler];
    require(index != 0);
    shufflerIndex[_t][shufflers[_t][shufflers[_t].length-1]] = index;
    delete shufflerIndex[_t][_shuffler];
    shufflers[_t][index-1] = shufflers[_t][shufflers[_t].length-1];
    shufflers[_t].pop();
}
```

“Border attack” component of collusion attacks

The “border attack” is when x% of the population collude, and get pairs with two colluders for x% of them, can get x% of the immigrants they can invite into those pairs, giving them $p*x\%^3$ bots, on top of $p*x\%^2$ for the collusion attack, in total $(x\%^3 + x\%^2)*p$, 1+x% times than with just collusion attack. This attack shifts the curve for when colluders gain 50% control, 50% of all proof-of-unique-human, to 22% instead of 33%. 2/3rds majority requirement is already a bit high, so this would not be good.

To defend against the “border attack”, in a way that does not remove anonymity, an “anonymous endorsement mechanism” is used. People who invite another person publicly record that they did so, but not who they invited. After the event, every immigrant who is verified will clear a randomly selected “endorser” from the record. Endorsers who are not cleared are forced to register as immigrants the next event. This prevents larger attacks, leaving only negligible attacks with up to roughly x% of the immigrants (and these attacks publicly reveal how large x% is.)

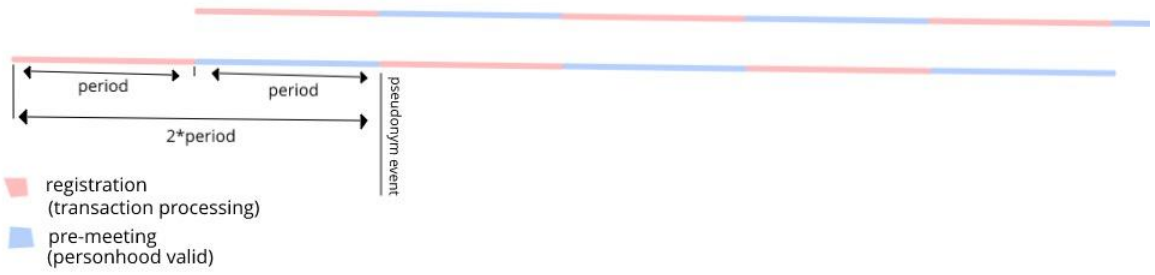
```
function invite() external {
    uint t = schedule();
    require(!halftime(t));
    require(shufflerIndex[t][msg.sender] != 0);
    require(endorserIndex[t][msg.sender] == 0);
    balanceOf[t][Token.Immigration][msg.sender]++;
    endorsers[t].push(msg.sender);
    endorserIndex[t][msg.sender] = endorsers[t].length;
}
```

Man-in-the-middle attacks, and “pre-meetings”

Man in the middle attacks are when two fake accounts relay the communication between the two real humans the fake accounts are chosen to verify. Then the two real humans each verified a bot. These are defended against simply by the real people asking each other what pair they are in. To have extra security, they can be defended in a way that is as “Turing safe” (same difficulty for breaking Turing test) as the actual event itself, but it requires an extra step.

The ideal defense adds a "handshake" to secure the video channel. The mechanism for this, the pair schedules a “pre-meeting” at a random time before the event, by agreeing on a random number using a commit-reveal scheme. The time for the pre-meeting is the sum of both numbers. Public keys are exchanged along with the numbers, as part of the encrypted message. Once they have proven encrypted numbers have been exchanged (either by meeting over video while doing the exchange, or, having a deadline), they then meet at the agreed time. This “pre-meeting” can only take place if they got the same number.
This proves that their channel is secure by validating that they both got the same number, and that the public key of the other person (that was also exchanged) is the public key they are paired with. The probability of the man in the middle attacking this in a way that both peers get the same number is 1/numberSize. This defense is as "Turing safe" as the actual event itself.

To allow time for “pre-meetings” while also having enough time to process transactions for registration, “random majority vote”, and shuffling, two separate pseudonym event “timelines” can exist, allowing an entire period for “pre-meetings”. See image below.



The two timelines below show the “pre-meetings” integrated with the registration phase, shuffle phase, and random majority vote phase.

A very minor attack vector is “man-in-the-middle-attack” on immigrants. The colluders can get honest immigrants in pairs the colluders control for up to x^2 percent of the immigrants, x being the fraction colluding. This is a very small number of immigrants. They can then relay the communication from those immigrants to fake immigrants the colluders control, getting them verified. To avoid this attack, immigrants could also “pre-meet” their pair.

Very short summary, Overview

Everyone on Earth meets 1-on-1, as strangers, at the exact same time, for 15 minutes. Proof is valid a month, then new global event. Has a "virtual border" around it. If verified, can register again. If not, have to "immigrate", assigned under random pair, 2-on-1 verification. If problem in pair, split it, be assigned under random other pair for 2-on-1 verification. Fully anonymous. The proof-of-unique-human is mixed. Untraceable from month to month.

References

Pseudonym Parties: An Offline Foundation for Online Accountable Pseudonyms, https://pdos.csail.mit.edu/papers/accountable-pseudonyms-socialnets08.pdf (2008)

Pseudonym Pairs: A foundation for proof-of-personhood in the web 3.0 jurisdiction, https://panarchy.app/PseudonymPairs.pdf (2018)

```
contract Scheduler {

    uint constant public genesis = 198000;
    uint constant public period = 4 weeks;
    function schedule() public view returns(uint) { return ((block.timestamp - genesis) / period); }

}

contract Mixer is Scheduler {

    enum Token { Personhood, Registration, Immigration }

    mapping (uint => mapping (Token => mapping (address => uint))) public balanceOf;
    mapping (uint => mapping (Token => mapping (address => mapping (address => uint)))) public allowed;

    function _transfer(uint _t, address _from, address _to, uint _value, Token _token) internal {
        require(balanceOf[_t][_token][_from] >= _value);
        balanceOf[_t][_token][_from] -= _value;
        balanceOf[_t][_token][_to] += _value;
    }
    function transfer(address _to, uint _value, Token _token) external {
        _transfer(schedule(), msg.sender, _to, _value, _token);
    }
    function approve(address _spender, uint _value, Token _token) external {
        allowed[schedule()][_token][msg.sender][_spender] = _value;
    }
    function transferFrom(address _from, address _to, uint _value, Token _token) external {
        uint t = schedule();
        require(allowed[t][_token][_from][msg.sender] >= _value);
        _transfer(t, _from, _to, _value, _token);
        allowed[t][_token][_from][msg.sender] -= _value;
    }
}

contract RandomMajorityVote {

    mapping (uint => mapping (address => bytes32)) public commit;
    mapping (uint => mapping (uint => uint)) public generator;
    mapping (uint => mapping (uint => uint)) public points;
    mapping (uint => uint) public leader;

    function _reveal(uint _t, uint _entropy, uint _mutator, uint _generators) internal {
        require(keccak256(abi.encode(_entropy)) == commit[_t][msg.sender]);
        uint vote = ((_entropy%_generators) + _mutator)%_generators;
        generator[_t][vote] ^= _entropy;
        points[_t][vote]++;
        if(points[_t][vote] > points[_t][leader[_t]]) leader[_t] = vote;
    }
}

contract OnlinePseudonymParties is Mixer, RandomMajorityVote {

    mapping (uint => uint) public hour;

    function toSeconds(uint _t) public pure returns (uint) { return genesis + _t * period; }
    function halftime(uint _t) public view returns (bool) { return((block.timestamp > toSeconds(_t)+period/2)); }
    function computeHour(uint _t) public { hour[_t] =  1 + uint(keccak256(abi.encode(_t)))%24; }
    function pseudonymEvent(uint _t) public returns (uint) { if(hour[_t] == 0) computeHour(_t); return toSeconds(_t) + hour[_t]*1 hours; }

    uint entropy;

    struct Nym { uint id; bool verified; }
    struct Pair { bool[2] verified; bool disputed; }
    struct Court { uint id; bool[2] verified; }

    mapping (uint => mapping (address => Nym)) public nym;
    mapping (uint => address[]) public registry;
    mapping (uint => mapping (uint => Pair)) public pair;
    mapping (uint => mapping (address => Court)) public court;
    mapping (uint => mapping (uint => address)) public courtIndex;
    mapping (uint => uint) public immigrants;

    mapping (uint => uint) public population;
    mapping (uint => mapping (address => bool)) public proofOfUniqueHuman;
```

```
mapping (uint => address[]) public shufflers;
mapping (uint => mapping (address => uint)) public shufflerIndex;

mapping (uint => address[]) public endorsers;
mapping (uint => mapping (address => uint)) public endorserIndex;

constructor(uint _population, address _genesis) {
    balanceOf[schedule()][Token.Registration][_genesis] = _population;
    balanceOf[schedule()+1][Token.Registration][_genesis] = _population;
}

function registered(uint _t) public view returns (uint) { return registry[_t].length; }
function pairs(uint _t) public view returns (uint) { return (registered(_t)/2); }
function deductToken(Token _token, uint _t) internal { require(balanceOf[_t][_token][msg.sender] >= 1); balanceOf[_t][_token][msg.sender]--; }
function pairVerified(uint _t, uint _pair) public view returns (bool) { return (pair[_t][_pair].verified[0] == true && pair[_t][_pair].verified[1] == true); }
function getPair(uint _id) public pure returns (uint) { return (_id+1)/2; }
function getCourt(uint _t, uint _court) public view returns (uint) { require(_court != 0); return 1+(_court-1)%pairs(_t); }
function boolToUint(bool _bool) internal pure returns (uint) { return _bool ? 1 : 0; }

function register(bytes32 _commit) external {
    require(_commit != 0);
    uint t = schedule();
    require(shufflerIndex[t][msg.sender] == 0);
    require(!halftime(t));
    deductToken(Token.Registration, t);
    registry[t].push(msg.sender);
    commit[t][msg.sender] = _commit;
    shufflers[t].push(msg.sender);
    shufflerIndex[t][msg.sender] = shufflers[t].length;
}
function immigrate() external {
    uint t = schedule();
    require(!halftime(t));
    deductToken(Token.Immigration, t);
    immigrants[t]++;
    court[t][msg.sender].id = immigrants[t];
    courtIndex[t][immigrants[t]] = msg.sender;
}
function invite() external {
    uint t = schedule();
    require(!halftime(t));
    require(shufflerIndex[t][msg.sender] != 0);
    require(endorserIndex[t][msg.sender] == 0);
    balanceOf[t][Token.Immigration][msg.sender]++;
    endorsers[t].push(msg.sender);
    endorserIndex[t][msg.sender] = endorsers[t].length;
}
function clearShuffler(address _shuffler, uint _t) internal {
    uint index = shufflerIndex[_t][_shuffler];
    require(index != 0);
    shufflerIndex[_t][shufflers[_t][shufflers[_t].length-1]] = index;
    delete shufflerIndex[_t][_shuffler];
    shufflers[_t][index-1] = shufflers[_t][shufflers[_t].length-1];
    shufflers[_t].pop();
}
function shuffle(address _shuffler) external {
    uint t = schedule()-1;
    uint unshuffled = shufflers[t].length;
    uint shuffled = registered(t)-unshuffled;
    if(shuffled == 0) entropy = generator[t][leader[t]];
    uint randomNumber = shuffled + entropy % unshuffled;
    entropy ^= uint160(registry[t][randomNumber]);
    (registry[t][shuffled], registry[t][randomNumber]) = (registry[t][randomNumber], registry[t][shuffled]);
    nym[t][registry[t][shuffled]].id = shuffled+1;
    clearShuffler(_shuffler, t);
}
```

```
function reveal(uint _entropy) external {
    uint t = schedule()-1;
    require(commit[t][msg.sender] != bytes32(0));
    require(halftime(t+1));
    uint id = nym[t][msg.sender].id;
    require(id != 0);
    require(shufflerIndex[t][msg.sender] == 0);
    _reveal(t, _entropy, id, pairs(t));
    delete commit[t][msg.sender];
}

function verify() external {
    uint t = schedule()-2;
    require(block.timestamp > pseudonymEvent(t+2));
    require(commit[t][msg.sender] == bytes32(0));
    uint id = nym[t][msg.sender].id;
    require(id != 0);
    require(pair[t][getPair(id)].disputed == false);
    pair[t][getPair(id)].verified[id%2] = true;
}
function judge(address _court) external {
    uint t = schedule()-2;
    require(block.timestamp > pseudonymEvent(t+2));
    uint signer = nym[t][msg.sender].id;
    require(getCourt(t, court[t][_court].id) == getPair(signer));
    court[t][_court].verified[signer%2] = true;
}

function allocateTokens(uint _t, uint _pair) internal {
    require(pairVerified(_t-2, _pair));
    balanceOf[_t][Token.Personhood][msg.sender]++;
    if(endorserIndex[_t][msg.sender] == 0) balanceOf[_t][Token.Registration][msg.sender]++;
    else balanceOf[_t][Token.Immigration][msg.sender]++;
}
function nymVerified() external {
    uint t = schedule()-2;
    require(nym[t][msg.sender].verified != true);
    uint id = nym[t][msg.sender].id;
    allocateTokens(t+2, getPair(id));
    nym[t][msg.sender].verified = true;
}
function clearEndorser(address _endorser, uint _t) internal {
    uint index = endorserIndex[_t][_endorser];
    require(index != 0);
    endorserIndex[_t][endorsers[_t][endorsers[_t].length-1]] = index;
    delete endorserIndex[_t][_endorser];
    endorsers[_t][index-1] = endorsers[_t][endorsers[_t].length-1];
    endorsers[_t].pop();
}
function courtVerified(address _endorser) external {
    uint t = schedule()-2;
    require(court[t][msg.sender].verified[0] == true && court[t][msg.sender].verified[1] == true);
    allocateTokens(t+2, getCourt(t, court[t][msg.sender].id));
    clearEndorser(_endorser, t);
    delete court[t][msg.sender];
}

function claimPersonhood() external {
    uint t = schedule();
    deductToken(Token.Personhood, t);
    proofOfUniqueHuman[t][msg.sender] = true;
    population[t]++;
}

function dispute(bool _premeet) external {
    uint t = schedule() - (2 + boolToUint(_premeet));
    uint id = nym[t][msg.sender].id;
    require(id != 0);
    if(_premeet == false) require(!pairVerified(t, getPair(id)));
    pair[t][getPair(id)].disputed = true;
}
```

```
    function _reassign(uint _court, uint _t) internal {
        uint _pairs = pairs(_t);
        _court = 1+(_court-1)%_pairs;
        uint i = 0;
        while(courtIndex[_t][_court + _pairs*i] != address(0)) i++;
        court[_t][msg.sender].id = _court + _pairs*i;
        courtIndex[_t][_court + _pairs*i] = msg.sender;
    }
    function reassignNym(bool _premeet) external {
        uint t = schedule() - (2 + boolToUint(_premeet));
        require(commit[t][msg.sender] == bytes32(0));
        uint id = nym[t][msg.sender].id;
        require(pair[t][getPair(id)].disputed == true);
        _reassign(uint256(uint160(msg.sender)) + id, t);
        delete nym[t][msg.sender];
    }
    function reassignCourt(bool _premeet) external {
        uint t = schedule() - (2 + boolToUint(_premeet));
        uint id = court[t][msg.sender].id;
        uint _pair = getCourt(id, t);
        require(pair[t][_pair].disputed == true);
        if(_premeet == true) require(_pair*2 <= registered(t)-shufflers[t].length);
        delete court[t][msg.sender];
        _reassign(1 + uint256(uint160(msg.sender)^uint160(registry[t][_pair*2-1])^uint160(registry[t][_pair*2-2])), t);
    }
}
```