Randomized Distributed Algorithms

Dan Alistarh

The story so far

- Agreement is sometimes impossible

- Sharing is hard

Good news today:

Randomization can help!

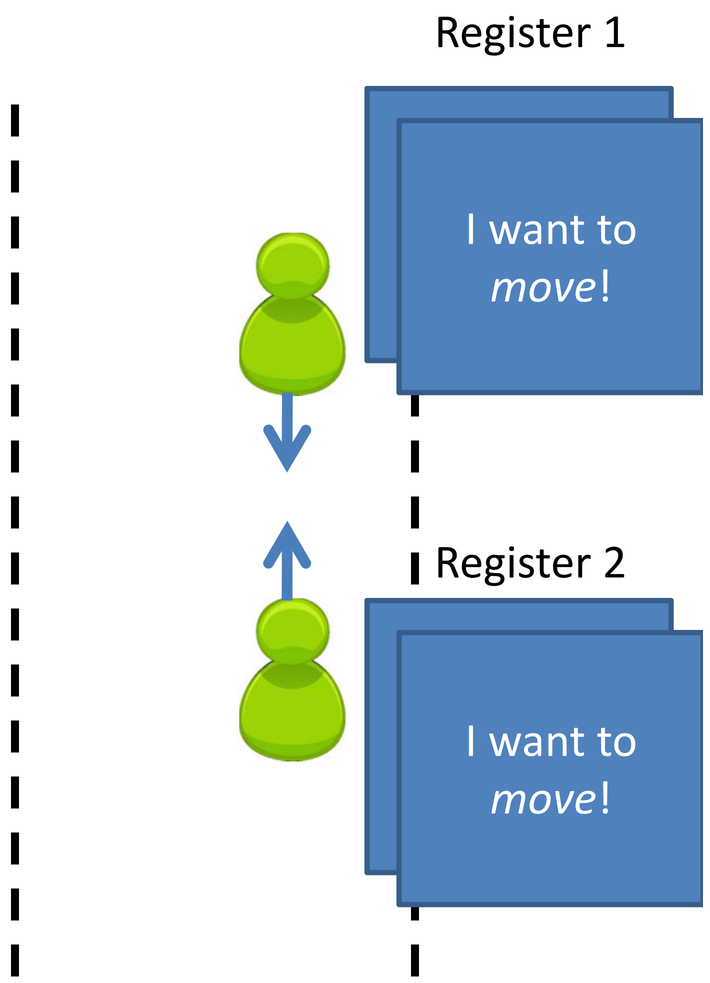
Randomization

- Processes are now allowed to flip coins
- Their actions (reads, writes) may depend on the outcome of the random coin flips

A real-life(?) example

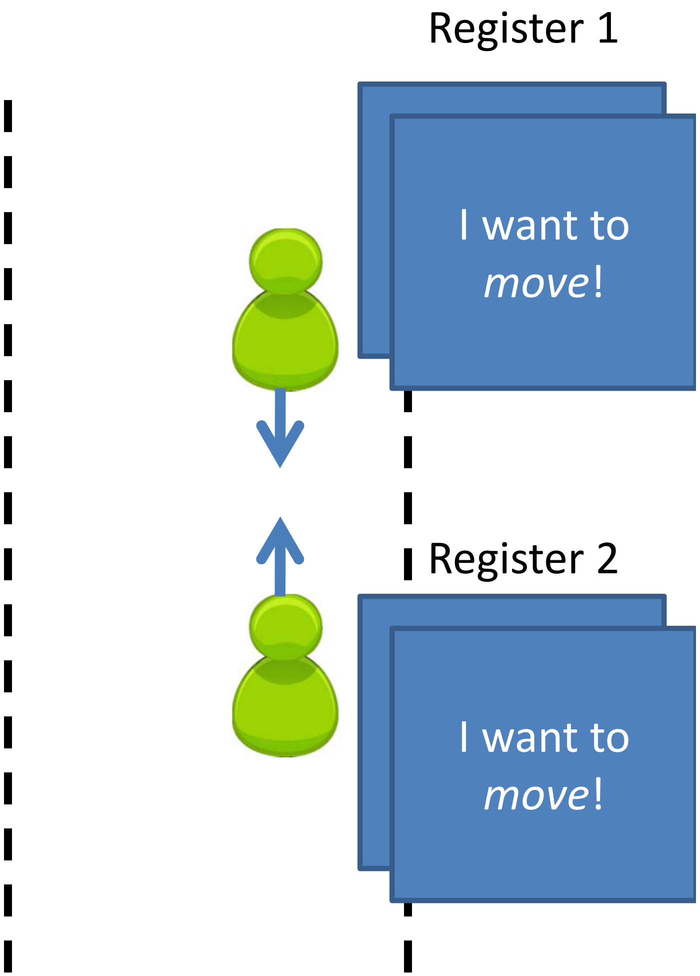
- Two students in a narrow hallway
- To proceed, one of them has to *change direction!*

- Let's allow them to communicate (registers)
  - They will have to solve *consensus* for 2 processes!



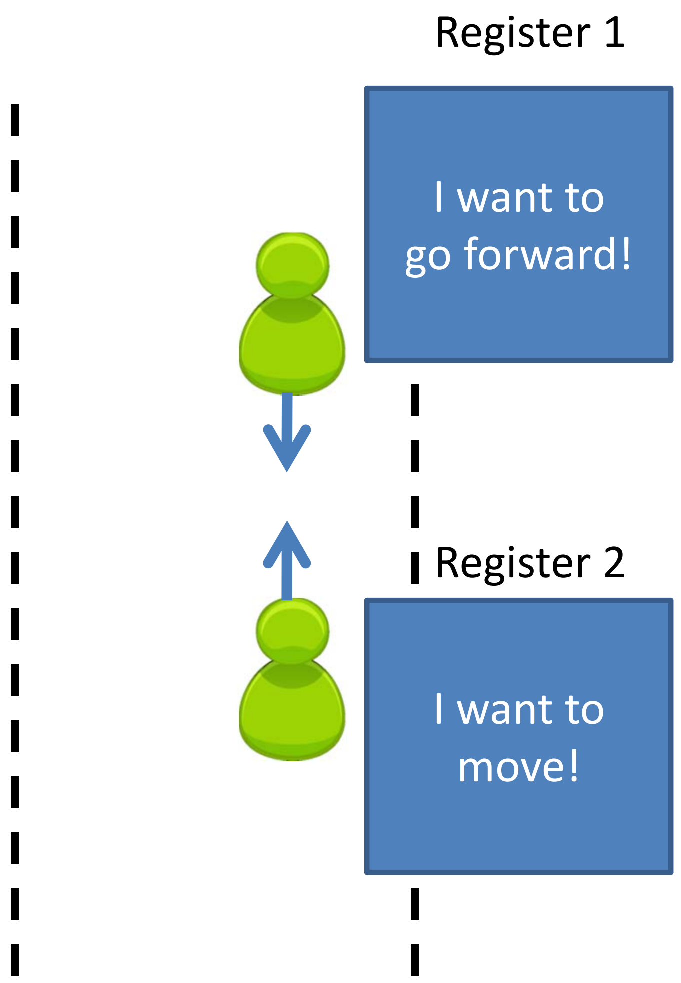
A real-life(?) example

- [FLP] : there exists an execution in which processes *get stuck forever*, or they *run into each other*!
- Does this happen in real life?!
- Is this *possible* in real life?
- It is *unlikely* that two people will continue choosing exactly the same thing!
- What does *unlikely* mean?



Slightly modified example

- Two people in a narrow hallway
- In each "round",
  - choose an option (*go forward* or *move*) with probability 1 / 2
  - write it to the register
- If they chose *different* options, they finish, otherwise continue
- Pr[finish in round 1] = 1 / 2
- Pr[continue after round r] = $(1 / 2)^r$
- For example,
  Pr[continue for > 10 rounds] < 0.001



Status

- Processes definitely finish in less than 100 rounds!
- Does there still exist an execution in which they do not finish?
  - Do we contradict FLP?
- Yes, the *infinite* execution is still there
  - We *do not* contradict FLP!
- What is the probability of that infinite execution?

$$\lim_{r \to \infty} \left(\frac{1}{2} \right)^r = 0$$

The problem has changed!

- By allowing processes to use random coin flips, we give *probability* to executions
- *Bad executions* (like FLP) should happen with extremely low probability (in this case, **0**)
- We ensure *safety* in *all* executions, but termination is ensured *with probability* **1**

Example: Consensus

- **Validity**: if all processes propose the same value *v*, then every correct process decides *v*.
- **Integrity**: every correct process decides at most one value, and if it decides some value *v*, then *v* must have been proposed by some process.
- **Agreement**: if a correct process decides *v*, then every correct process decides *v*.
- **Termination**: every correct process decides some value.

Randomized Consensus

- **Validity**: if all processes propose the same value *v*, then every correct process decides *v*.
- **Integrity**: every correct process decides at most one value, and if it decides some value *v*, then *v* must have been proposed by some process.
- **Agreement**: if a correct process decides *v*, then every correct process decides *v*.
- **(Probabilistic) Termination**: *with probability 1*, every correct process decides some value.

The plan for today

- Intro
  - Motivation
- Some Basic Probability
- A Randomized Test-and-Set algorithm
  - From 2 to N processes
- Randomized Consensus
  - Shared Coins
- Randomized Renaming

Some Basic Probability

- Fix a space Ω of all possible events
- To each event Ev in Ω, we associate a *probability* in [0, 1]
- Two events A, B are *independent* iff Pr[A and B] = Pr [A] Pr[B]
- Random variable f = function from Ω to real numbers
- Expectation

$$E[f] = \sum_{x \in R} x \cdot \Pr[f = x]$$

- Two consecutive independent tosses of a fair coin: Ω = { HH, HT, TH, TT }
- Pr [Ev] = 1 / 4, for all Ev in Ω
- Pr[first coin H, second coin T] = Pr[first coin H] Pr[second T] = 1 / 4
- f = number of heads in two consecutive tosses
- Expected nr. of heads:

$$E[f] = 0 \cdot 1/4 + 1 \cdot 1/2 + 2 \cdot 1/4 = 1$$

Test-and-set specification

Sequential specification:

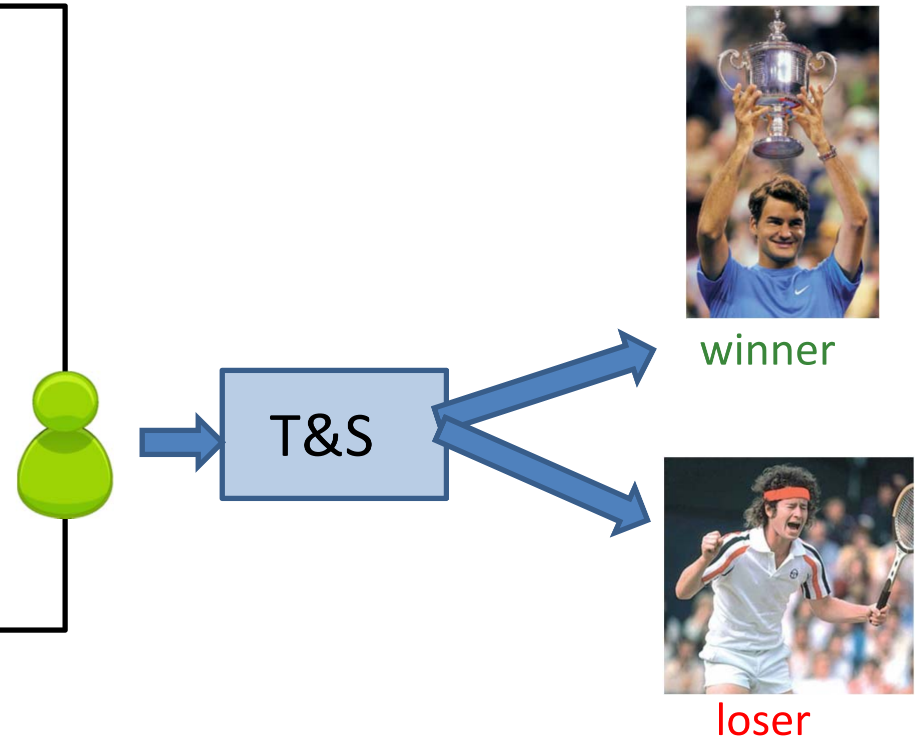
Shared: V, a binary MWMR atomic register, initially 0

procedure Test-and-Set()

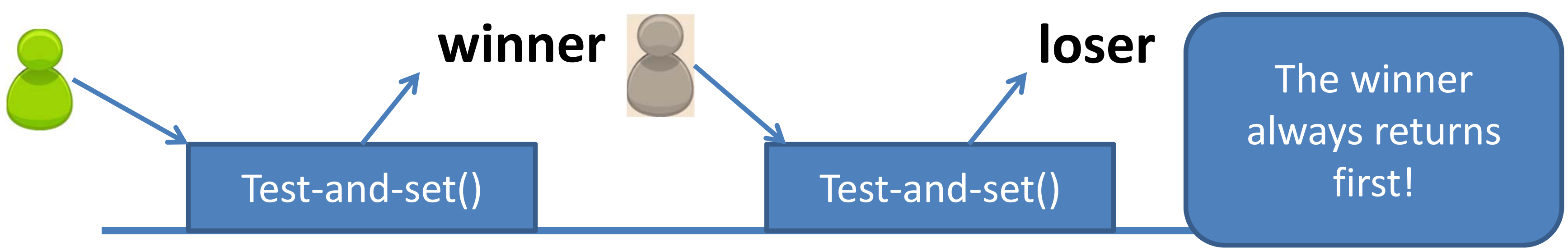
  if V = 0 then V ← 1

    return winner

  else **return** loser



Linearization:



2-process test-and-set

- Based on the previous “hallway” example
- Two SWMR registers R_1, R_2
  - Each owned by a process
- A register R_i can have one of 4 possible values:
  - NULL, Mine, His, Choosing
- Processes express their choices through registers
- Algorithm by Tromp and Vitanyi

The main idea

```
//general structure:
Registers R1, R2
procedure test-and-set() //at process i
  R_i = present
  while(true)
       value = flip local coin
       if both present AND flipped the same
           continue
       else
           one of them wins
```

2-process test-and-set

Shared: Registers R1, R2, initially NULL

procedure test-and-set$_i$() //at process i

```
R_i = Mine
while( R_i == R_{1 – i} )
    R_i = Choosing
    if( R_{1 – i} == His )     //if the other guy gave up
        R_i = Mine
        continue
    if ( R_{1 – i} == Choosing  AND CoinFlip() == Heads )
        R_i = Mine
    else R_i = His
//loop finished
if ( R_i == Mine ) return WINNER
else return LOSER
```

Flip *a local* coin to decide who gets the object. If both flip Heads, then it's a draw and we repeat

Eventually (with prob. 1) processes return from the loop

Correctness (rough sketch)

- **Unique Winner**:

Assume for contradiction that the two processes both return 1 (winner). Then both processes had R_i = Mine at line 13. It is easy to check that this is impossible, by case analysis.

- **Termination**:

Every time processes execute the coin flip in line 9, the probability that the while loop terminates in the next iteration is ½. Hence, the probability that the algorithm executes more than r coin flips is $(1/2)^r$. Therefore, the probability that the algorithm goes on forever is

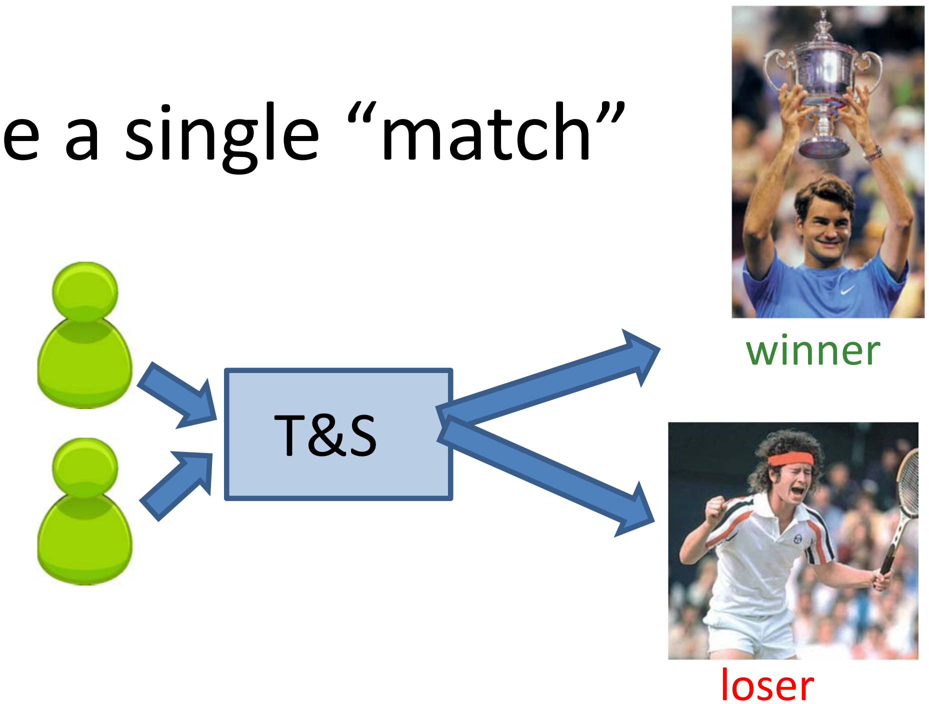
$$\lim_{r \to \infty} \left(\frac{1}{2} \right)^r = 0$$

Performance

- What is the *expected* number of steps that a process performs in an execution?
- The probability that they finish in an iteration is 1 / 2
- The expected number of iterations is 2!
  - Try it at home!
  - Geometric distribution

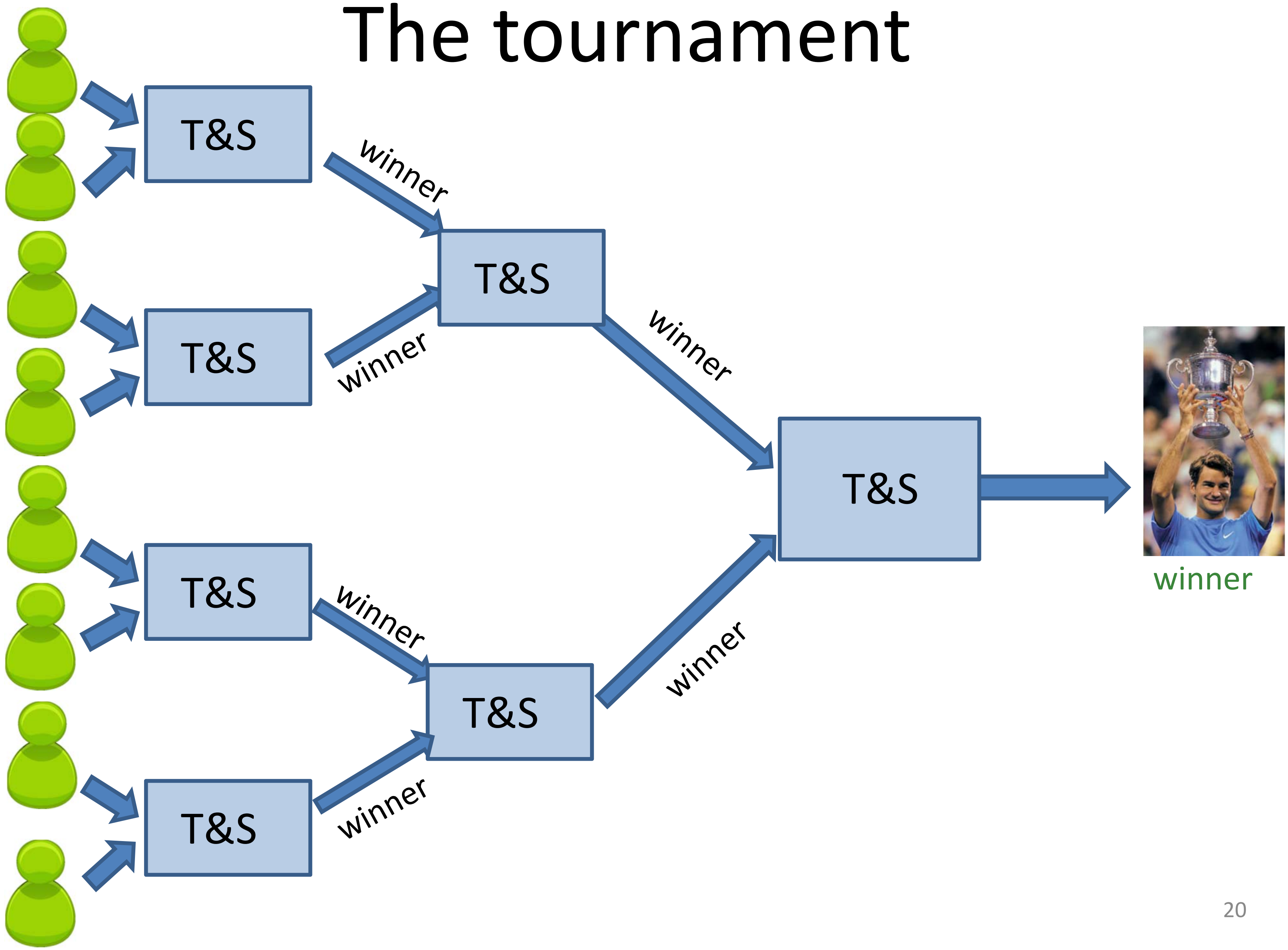
From 2 to N processes

- We know how to decide a single “match”



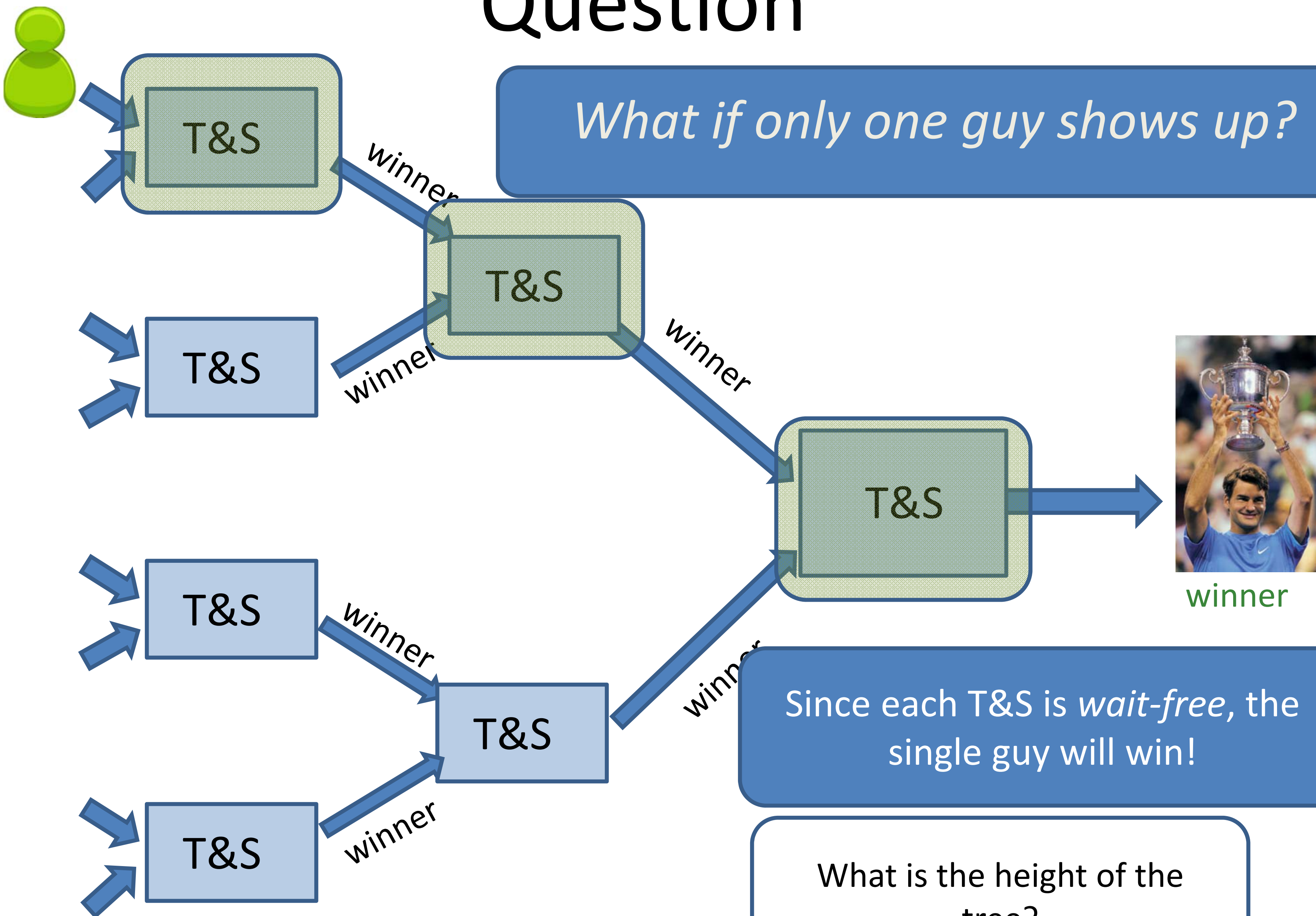
- How do we get a single winner out of a set of N processes?

The tournament



Question

What if only one guy shows up?



Since each T&S is *wait-free*, the single guy will win!

What is the height of the tree?

Code: Variant #1

- **procedure** test-and-set() // at process i
  - current = leaf-test-and-set[i]
  - **while** (true)
    - result = current.test-and-set ()
    - **if** (result == winner)

      if (current == root) return winner

      else current = current.parent()
    - **else** return loser

Start at the leaf corresponding to your ID i

As long as you keep winning, you go up the tree!

If you lose a test-and-set, you have to leave

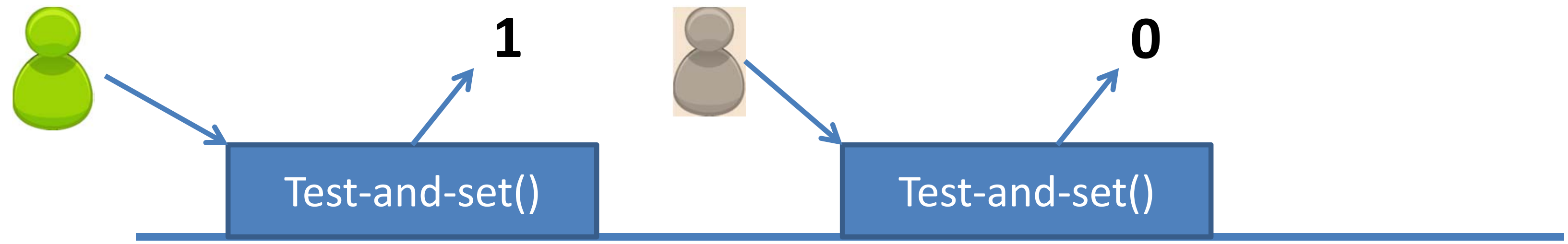
Correctness

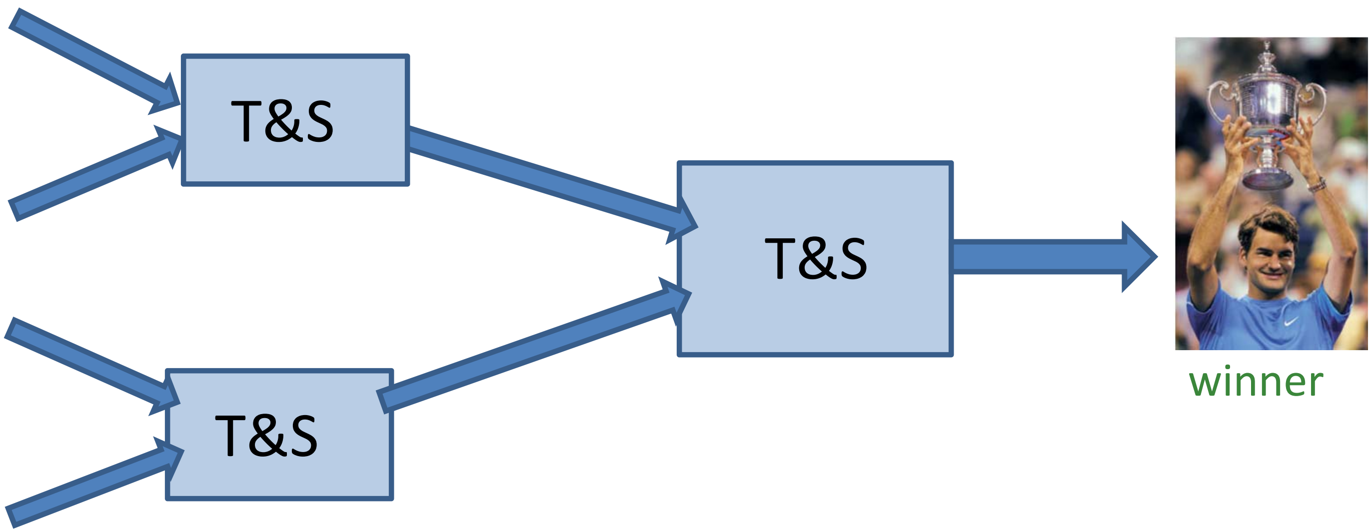
- **Unique winner**: Suppose there are two winners. Then both would have to win the root test-and-set, contradiction
- **Termination (with probability 1!)**: Follows from the termination of 2-process test-and-set
- **Winner**: Either there exists a process that returns winner, or there is at least a failure

Is this it?

How about this property?

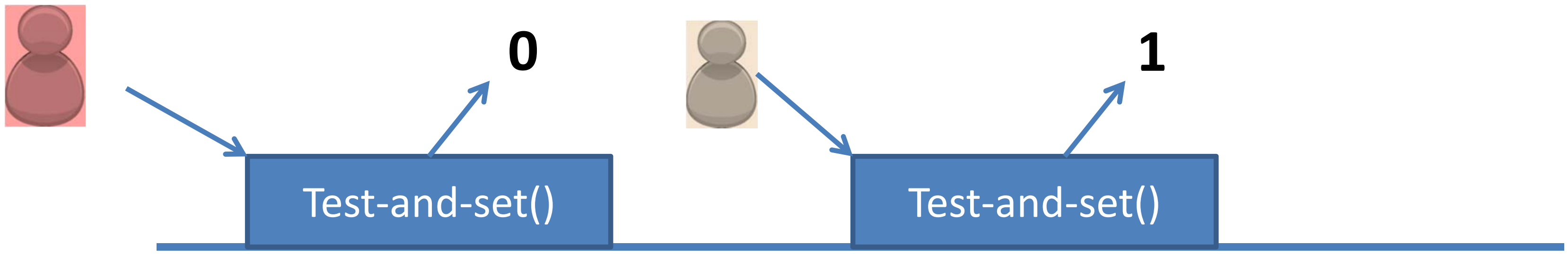
Linearization:

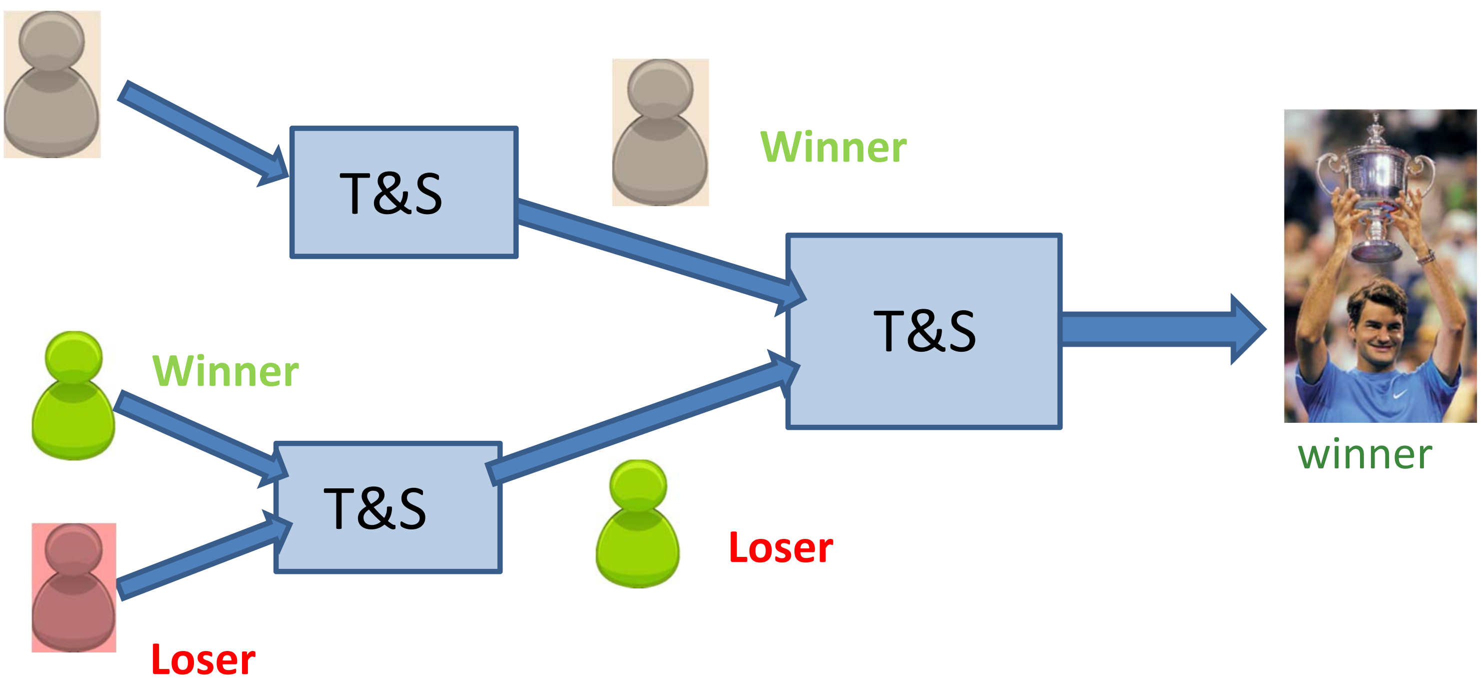




How about this?

Linearization:





Homework

- Fix the N-process test-and-set implementation so that it is ***linearizable***
- Hint: you only need to add *one* register

Wrap up

- We have a test-and-set algorithm for N processes
- Always safe
- Terminates with probability 1
- Worst-case local cost O(log N) per process
- Expected total cost O(N)

The plan for today

- Intro
  - Motivation
- Some Basic Probability
- A Randomized Test-and-Set algorithm
  - From 2 to N processes
- Randomized Consensus
  - Shared Coins
- Randomized Renaming

Randomized Consensus

- Can we implement Consensus with the same properties?

Randomized Consensus

- Algorithms based on a *Shared Coin*
- A **Shared coin** with parameter **ρ**, **SC(ρ)** is an algorithm without inputs, which has probability **ρ** that all outputs are 0, and probability **ρ** that all outputs are 1.
- **Example:**
  - Every process flips a local coin, and returns 1 for Heads, 0 for Tails
  - **ρ =** Pr[all outputs are 1] = Pr[all outputs are 0] = $(1/2)^N$
  - Usually, we look for higher output parameters The higher the parameter, the faster the algorithm

Shared Coin -> Binary Consensus

- The algorithm will progress in rounds
- Processes share a doubly-indexed vectors Proposed[r][i], Check[r][i]
  (r = round number, i = process id)
- Proposed[][] stores values, Check[][] indicates whether a process finished
- At each round r > 0, process p_i places its vote (0 or 1) in Proposed[r][i]

Shared Coin->Binary Consensus

```
Shared: Matrices Proposed[r][i]; Check[r][i]
procedure propose_i( v )    //at process i
    decide = false, r = 0
    While( decide == false )
            r = r + 1
            Proposed[r][i] = v
            view = Collect( Proposed[r] [...])
            if (both 0 and 1 appear in view )
                Check[r][i] = disagree
            else Check[r][i] = agree
            check-view = Collect( Check[r] [...])
            if( disagree appears in check-view )
                coin = SharedCoin(r)
                if (for some j, check-view[j] = agree)
                    v = Proposed[r][j]
                else v = coin
            else decide = true
    return v
```

In each round r, the process writes its value in Proposed[r][i]

It then checks to see if there is *disagreement*, and marks it to Check[r][i]

If there is disagreement, then processes flip a shared coin to agree, and post the results

If no-one disagrees, then return!

Correctness

Shared: Matrices Proposed[r][i]; Check[r][i]

```
procedure propose_i( v )    //at process i
1.  decide = false, r = 0
2.  While( decide == false )
3.        r = r + 1
4.        Proposed[r][i] = v
5.        view = Collect( Proposed[r] […])
6.        if (both 0 and 1 appear in view )
7.            Check[r][i] = disagree
8.        else Check[r][i] = agree
9.        check-view = Collect( Check[r] […])
10.       if( disagree appears in check-view )
11.           coin = SharedCoin(r)
12.           if (for some j, check-view[j] = agree)
13.               v = Proposed[r][j]
14.           else v = coin
15.       else decide = true
16. return v
```

- **Validity:** If everyone proposes the same *v*, then Check=*agree,* so they decide on *v*
- **Agreement:** If process p decides *v*, then either all processes wrote *v*, or slower processes will adopt *v* in line 13
- **Termination?**

Termination

- If everyone proposes the same thing, then we're done within a round
- Otherwise, processes have probability at least **2ρ** of flipping the same value at every round **r**
- What is the probability that they go on *forever*?

$$(1-2\rho)\cdot(1-2\rho)\cdot(1-2\rho)\cdot(1-2\rho)\cdot\ldots=$$

$$\lim_{r\to\infty}(1-2\rho)^{r}=0$$

What does this mean?

- We can implement consensus ensuring
  - safety in all executions
  - termination with probability 1.
- By the universal construction, we can implement *anything* with these properties
- So…are we done with this class?
- The limit is no longer impossibility, but *performance*!

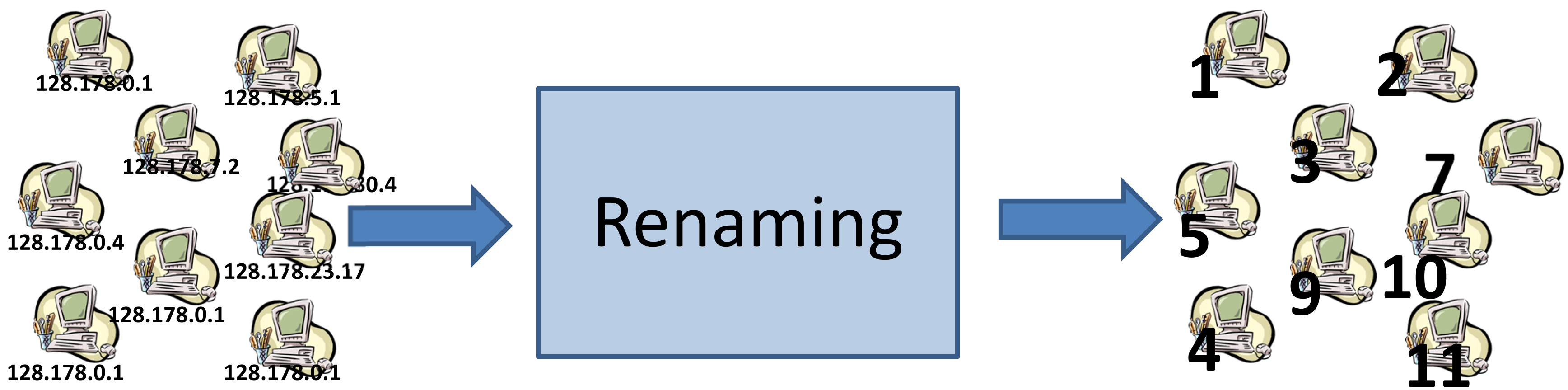
Homework 2: Performance

- What is the *expected* number of rounds that the algorithm runs for, if the Shared coin has parameter **ρ**?
- In particular, what is the *expected* running time for the example shared coin, having **$\rho = (1/2)^n$**?
- Can you come up with a ***better shared coin***?

The plan for today

- Intro
  - Motivation
- Some Basic Probability
- A Randomized Test-and-Set algorithm
  - From 2 to N processes
- Randomized Consensus
  - Shared Coins
- Randomized Renaming

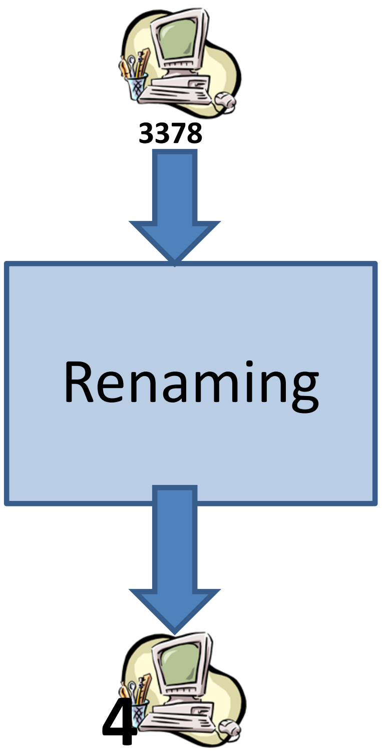
The Renaming Problem



- N processes, t < N might fail by crashing
- Huge initial ID’s (think IP Addresses)
- Need to get new unique ID’s from a small namespace (e.g., from 1 to N)
- The *opposite of consensus*

Why is this useful?

- Getting a small unique name is important
  - Smaller reads and writes/messages
  - Overall performance
  - Names are a natural prerequisite
- Renaming is related to:
  - Mutual exclusion
  - Test-and-set
  - Counting
  - Resource allocation



What is known

Theorem [HS, RC] In an asynchronous system with $t < N$ crashes, Deterministic Renaming is impossible in $N + t - 1$ or less names.

- Both Shared-Memory and Message-Passing
- Analogous to FLP, much more complicated
- Uses Algebraic Topology!
- Gödel Prize 2004

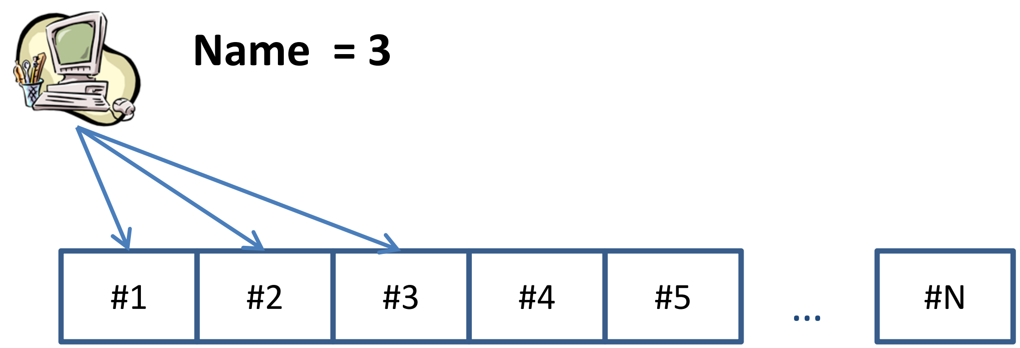

How can randomization help?

- It will allow us to get a tight namespace (of N names), even in an asynchronous system
- It will give us better performance
- Idea: derive *tight renaming* from *test-and-set*
- We now know how to implement test-and-set in an asynchronous system
- What's the catch?

Adaptive Tight Renaming from Test-and-Set

Shared: *V*, an infinite vector of *randomized* test-and-set objects

procedure getName(i)
  j ← 1
  while(true)
    res ← V[j].Test-and-set$_i$ ()
    if res = winner then
      return j
    else j ← j + 1



Performance

Shared: *V*, an infinite vector of test-and-set objects

procedure getName(i)
  j ← 1
  while(true)
    res ← V[j].Test-and-set$_i$ ()
    if res = winner then
      return j
    else j ← j + 1

- What is the worst-case *local complexity*?
- O(N)

- What is the worst-case *total complexity*?
- O(N^2)

Where is the randomization?

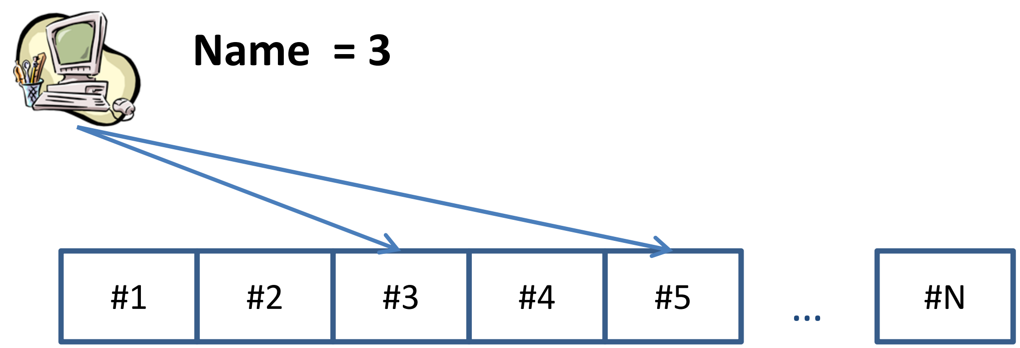
Can we do better using randomization?

Randomized Tight Renaming

Shared: *V*, an infinite vector of test-and-set objects

procedure getName(i)

while(true)
  j = Random(1, N)
  res ← V[j].Test-and-set$_i$ ()
  if res = winner then
    return j



Randomized Tight Renaming

Shared: *V*, an infinite vector of
test-and-set objects

procedure getName(i)

```
while( true )
  j = Random(1, N)
  res ← V[j].Test-and-set_i ()
  if res = winner then
    return j
```

- **Claim**: The expected total number of tries is O(N log N)!
- Sketch of Proof (not for the exam):

1. A process will win at most one test-and-set
2. Hence it is enough to count the time until each test-and-set is accessed at least once!
3. N items, we access one at random every time; how many accesses until we cover all N of them?
4. *Coupon collector:* we need < 2N log N total accesses, with probability $1 – 1 / N^3$

Wrap-up

- We get *tight* renaming
  in asynchronous shared memory
- **Termination** ensured with probability 1
- **Total complexity**:
  O(N log N) total operations in expectation

Conclusion

- Randomization "avoids" the deterministic impossibility results (FLP, HS)
  - The results still hold, the bad executions still exist
  - We give bad executions vanishing probability, ensuring termination with probability 1
- The algorithms always preserve *safety*
- Usually we can get better performance by using randomization

References (use Google Scholar)

- For test-and-set:
  - "Randomized two-process wait-free test-and-set" by John Tromp and Paul Vitányi
  - "Wait-free test-and-set" by Afek et al.
- For randomized consensus:
  - http://pine.cs.yale.edu/pinewiki/RandomizedConsensus
  - You can use the same wiki for other topics as well
- For renaming:
  - "Fast Randomized Test-and-Set and Renaming" by Alistarh, Guerraoui et al.