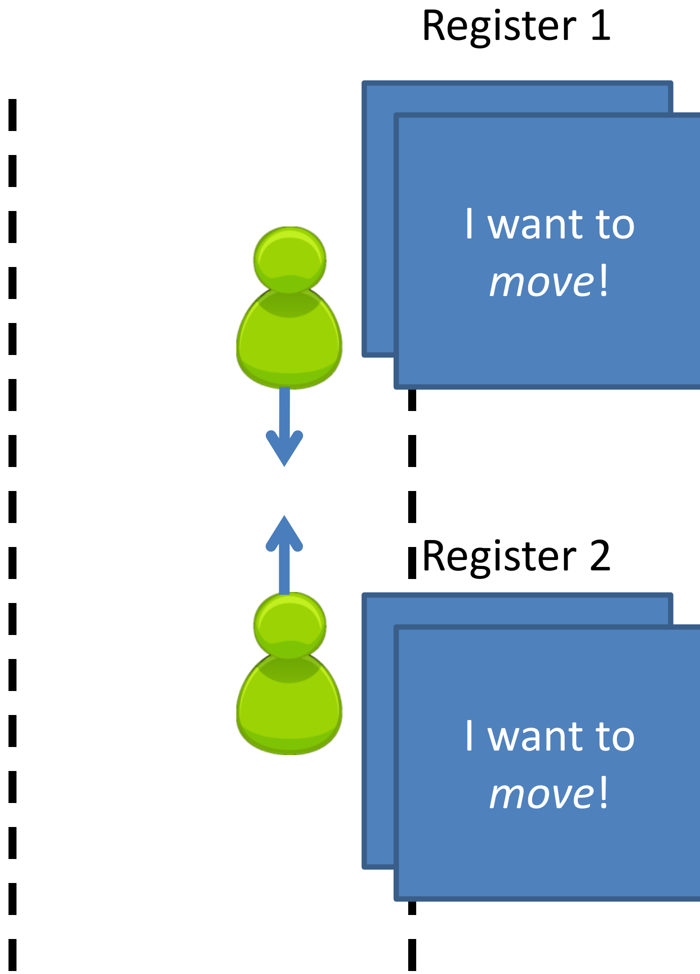
Randomized Concurrent Algorithms

Based on slides by Dan Alistarh
Giuliano Losa

A simple 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!



Register 1

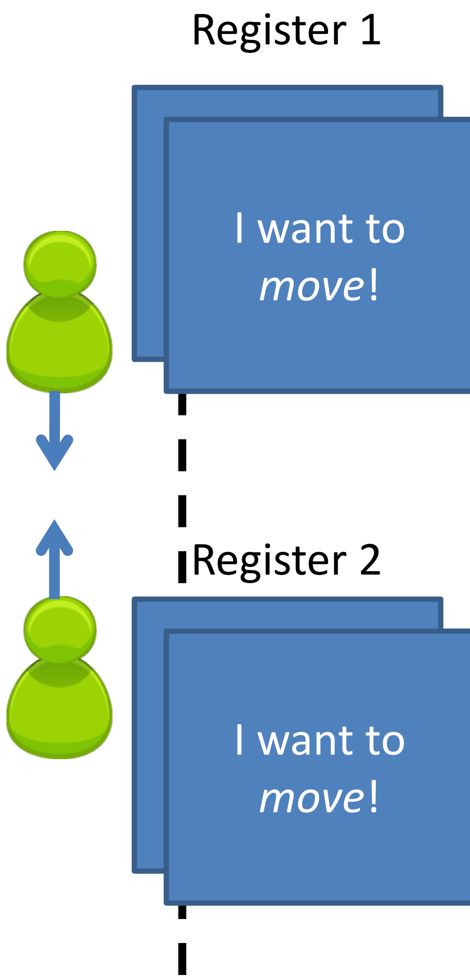
I want to *move*!

Register 2

I want to *move*!

A simple 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?



Register 1

I want to *move*!

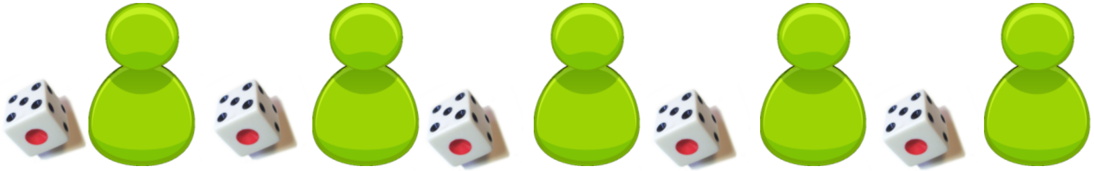
Register 2

I want to *move*!

Analysis

- Always finishes in practice!
- 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?

The problem has changed!



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

The plan for today

- Intro
  - Motivation
- The randomized model
- A Randomized Test-and-Set algorithm
  - From 2 to N processes
- Randomized Consensus
  - Shared Coins
- Randomized Renaming

Semantics of deterministic algorithms, correctness, limitations.

- An algorithm denotes a set of histories
- Solving consensus means solving it in all possible histories.
- FLP: consensus is impossible in an asynchronous systems if a single process may crash.

Solving problems with good probability

- We need to assign probabilities to histories
  - Probability distribution on scheduling?
  - Probability distribution on inputs?
- No, we don't have control over these in practice
- Instead:
  - Processes make independent random choices (they flip coins).
  - Schedule and inputs determined by a deterministic adversary (a function of the history so far).
  - We look at the worst possible adversary

Semantics of randomized protocols

- **A protocol and an adversary** (P,A) denote a set of histories and an associated probability distribution.
- A pair (P,A) and a sequence of bits s uniquely determine a history of the algorithm P.
- The probability of a given execution is the probability of the sequence of bits that corresponds to it.

Correctness Properties for Randomized Algorithms

- We define the class of adversaries we consider.
- Usually, we keep the safety condition of the deterministic problem and relax liveness:

"The algorithm should terminate with probability 1 for all adversary in the class"

Example: Consensus

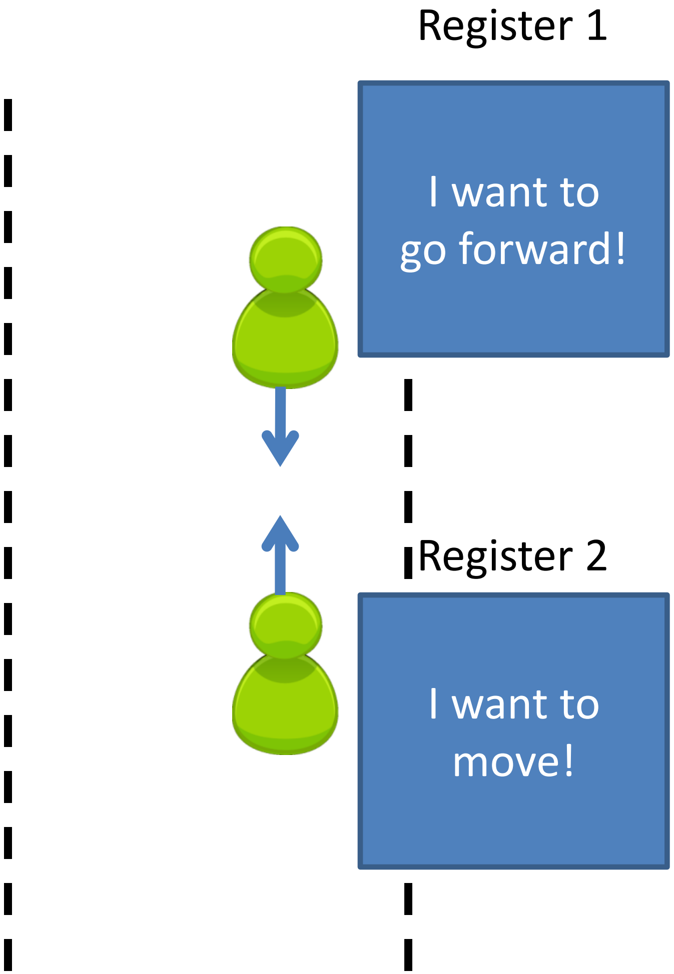
- **Validity**: if all processes propose the same value *v*, then every correct process decides *v*.
- **Integrity**: every correct process decides at most one value, and if it decides some value *v*, then *v* must have been proposed by some process.
- **Agreement**: if a correct process decides *v*,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

Randomized Consensus

- **Adversary:** any deterministic adversary.
- **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.

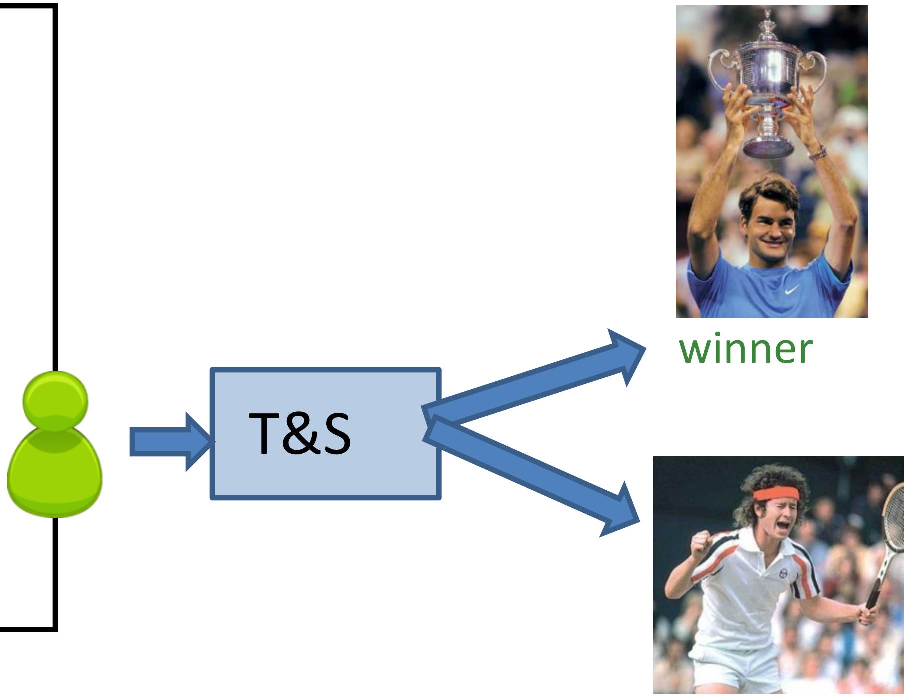
The simple example

- 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
- What is the worst case adversary? Arriving on the same side and scheduled in lock-steps.
- What is the probability of finishing in less than 2 rounds? ½ * ½ + ½ = ¾

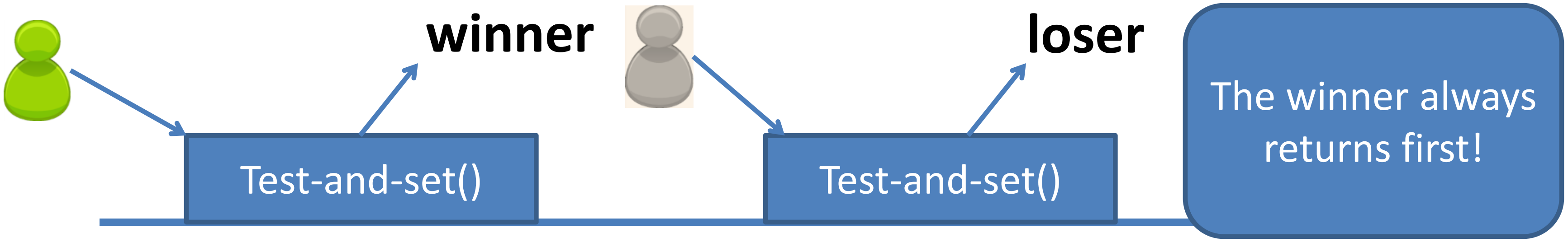


Test-and-set specification

- *Sequential specification:*
- V, a binary register, initially 0
- **procedure** Test-and-Set()
    - if V = 0 then V ← 1
        - **return** winner
    - else **return** loser



Linearization:



The winner always returns first!

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[p] can have one of 2 possible values:
  - Mine, Yours
- Processes express their choices through the registers
- Adapted from an algorithm by Tromp and Vitanyi (see references at the end).

2-process test-and-set

Shared: Registers R[p], R[p’], initially Yours

Local: Registers last_read[p], last_read[p’]

```
procedure test-and-set$_p$()    //at process p
R[p] <- Mine
Loop
    last_read[p] <- R[p’]
    If (R[p] = last_read[p])
        R[p] <- Random(Yours,Mine)
    Else break;
EndLoop
If R[p] = Mine then return 1
Else return 0
```

Correctness (rough sketch)

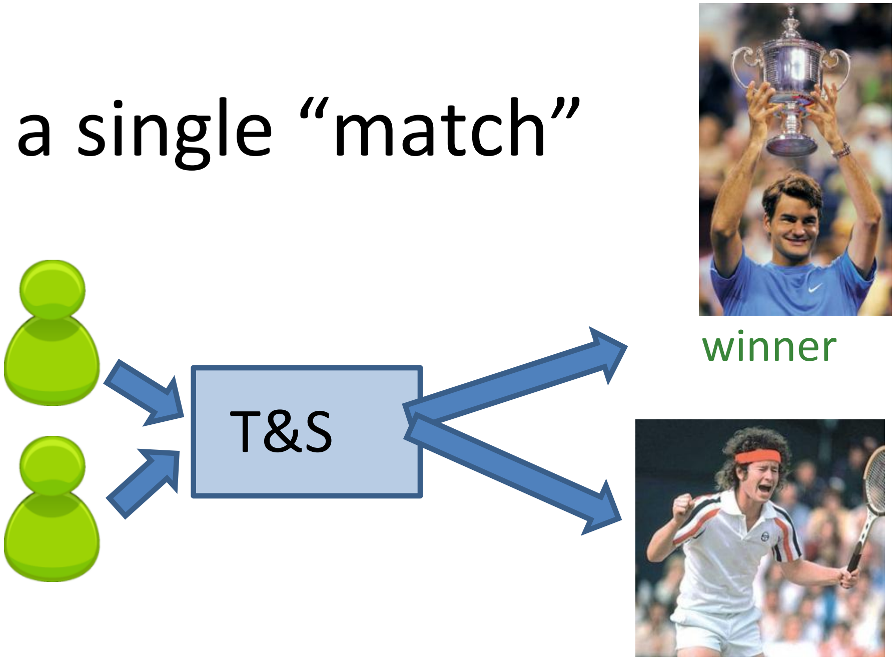
- **Worst case adversary:** lock-step scheduling.
- **Unique Winner**: Inductive invariants:
  - If both processes are at lines 4, 8, or 9, then one of them has a accurate last_read value.
  - If process p is at lines 8 or 9, then last_read[p] is different from R[p].
- **Termination**:
  - Every time processes execute the coin flip in line 5, 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 0.

Performance

- What is the *expected* number of steps that a process performs in an execution?
- We need to consider the worst case adversary: the lock-steps schedule.
- Consider the random var T counting the number of rounds before termination.
- T counts the number of trials before first success in a series of independent binary trials with probability $p = ½$.
- T has geometric distribution.
- The expected number of rounds is $1/p = 2$!

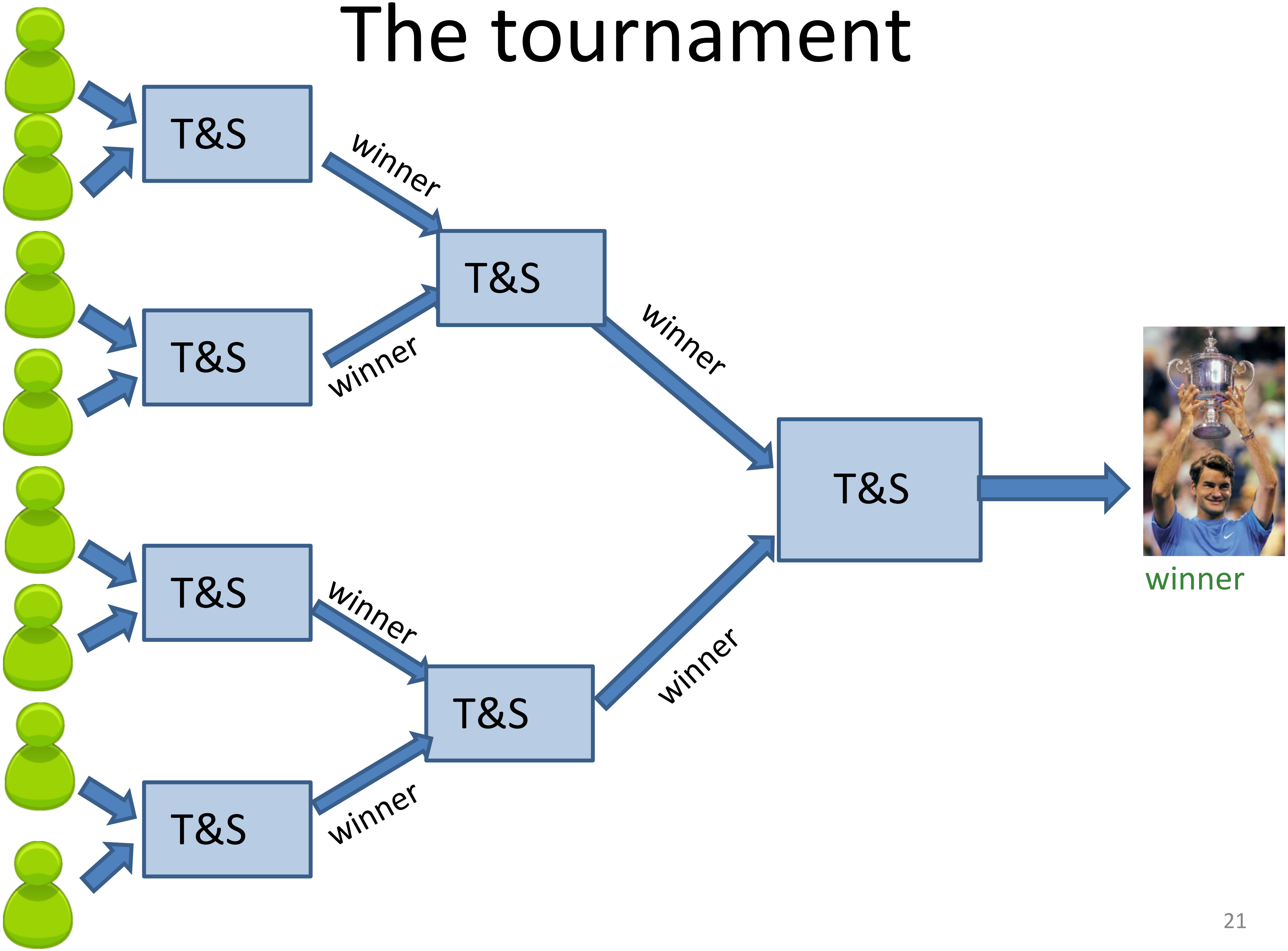
From 2 to N processes

- We know how to decide a single “match”



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

The tournament



T&S

winner

T&S

winner

T&S

winner

T&S

winner

T&S

winner

T&S

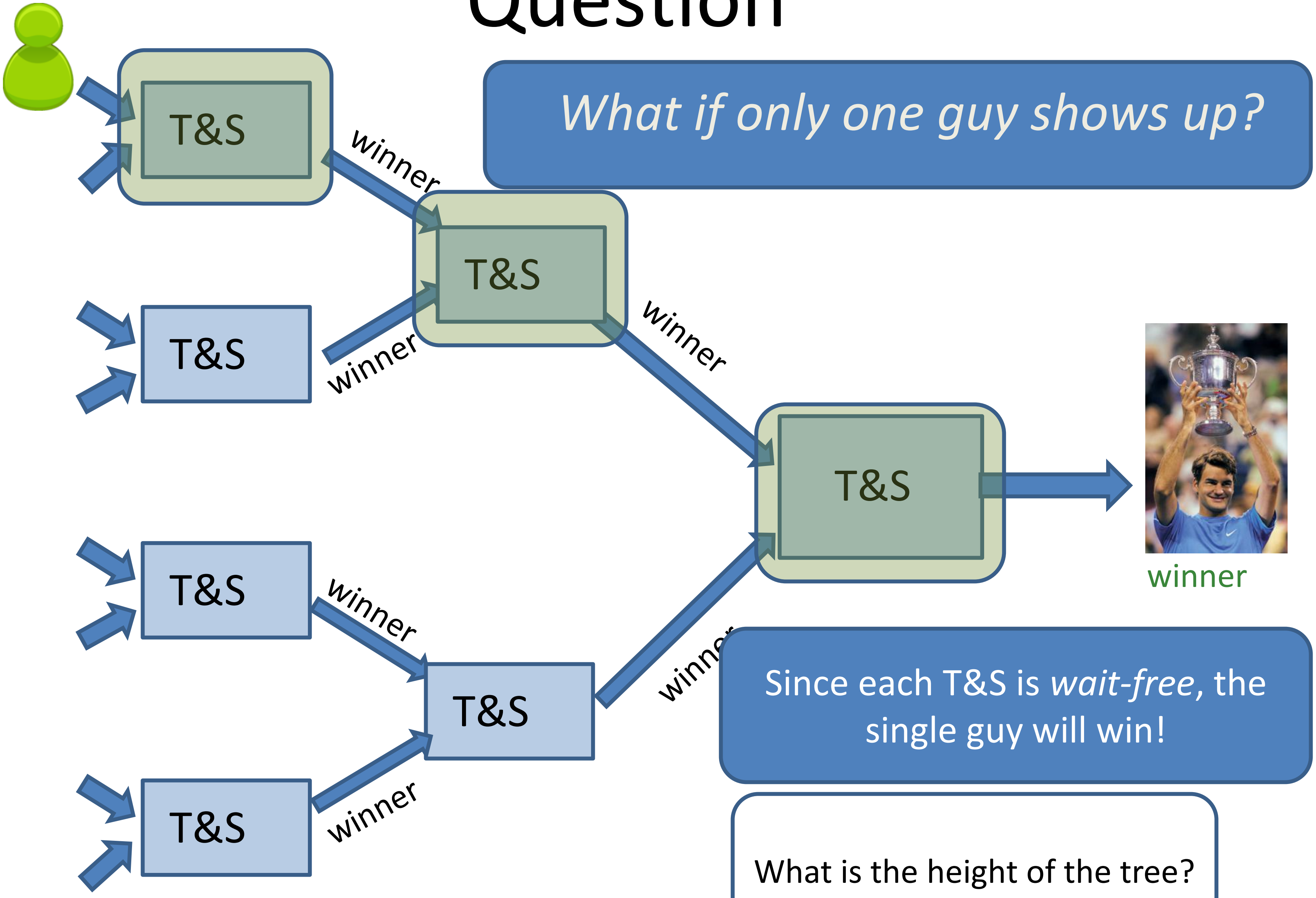
winner

T&S

winner

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?

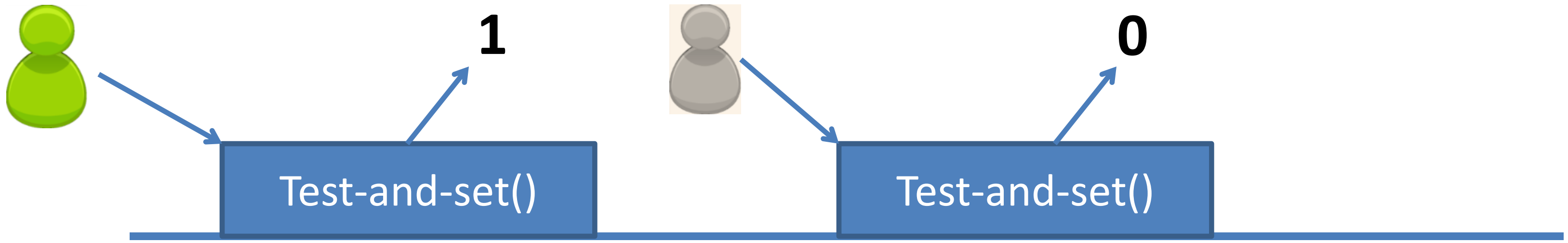
Correctness

- **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:

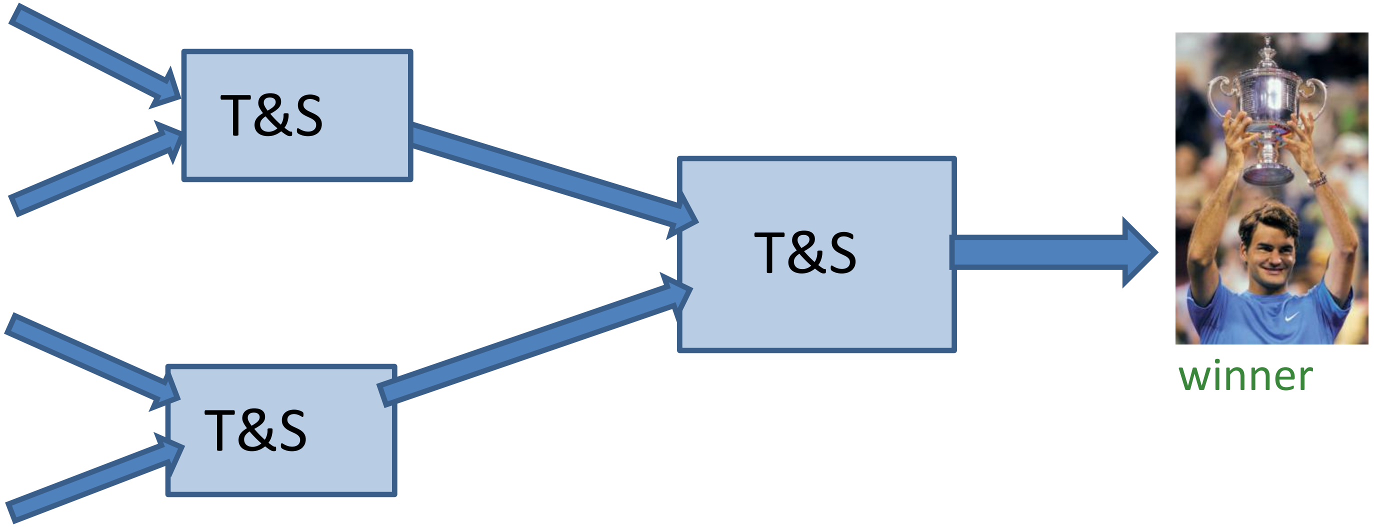


1

0

Test-and-set()

Test-and-set()



T&S

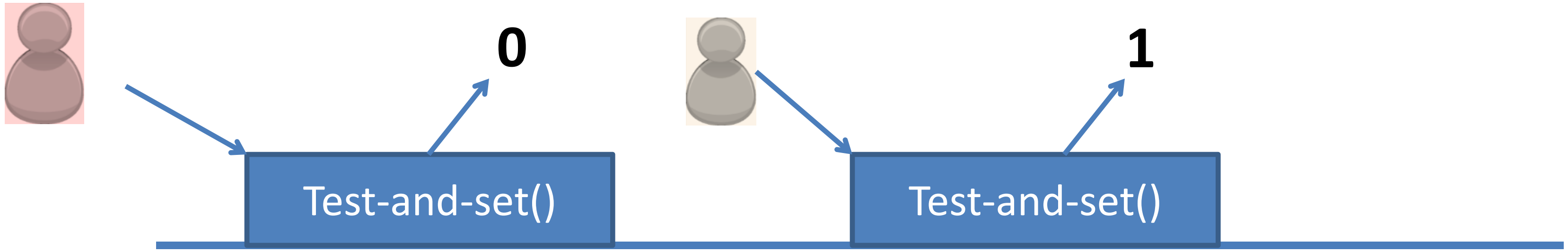
T&S

T&S

winner

How about this?

Linearization:

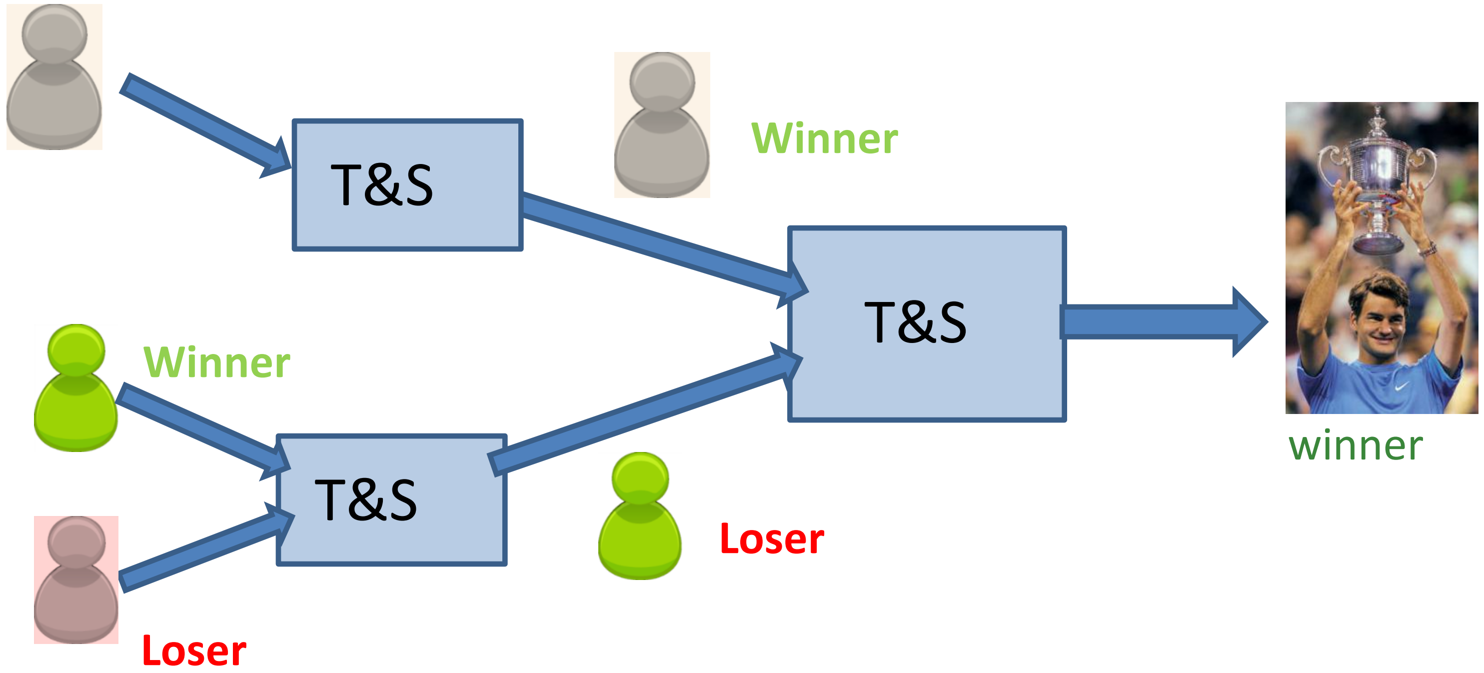


0

1

Test-and-set()

Test-and-set()



T&S

Winner

T&S

Winner

T&S

Loser

winner

Loser

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], init. to null.
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 )
        if (for some j, check-view[j] = agree)
            v = Proposed[r][j]
        else v = flip_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!

Termination

- Worst case adversary: lock-steps schedule.
- Processes have probability at least **2ρ** of flipping the same value at every round **r**
- If all processes have the same value at round r then they decide in round r.
- What is the probability that they go on *forever*?
- $(1 – 2p)x(1-2p)x(1-2p)x\ldots = 0$

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*!

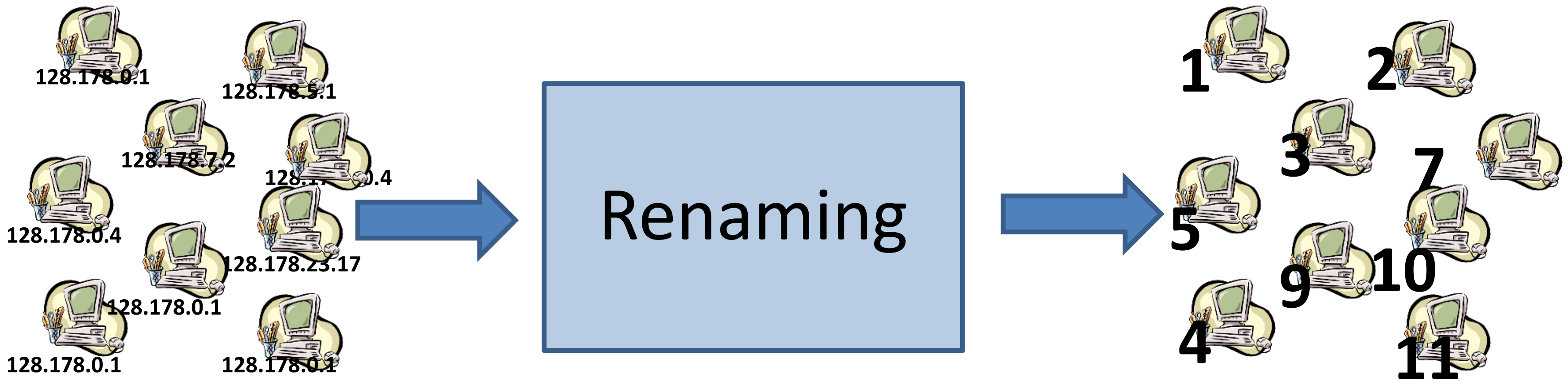
Homework 2: Performance

- 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
  $\rho = (1/2)^n$?
  - Termination time T is a random variable mapping a history to the number of steps before termination
  - Each round is akin to an independent binary trial with success probability **ρ**, hence T has geometric distribution.
  - The expectation of T is 1/ **ρ** = 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

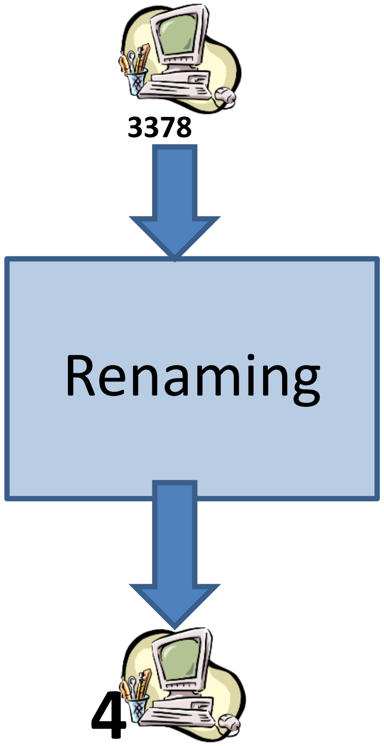
The Renaming Problem



- 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?

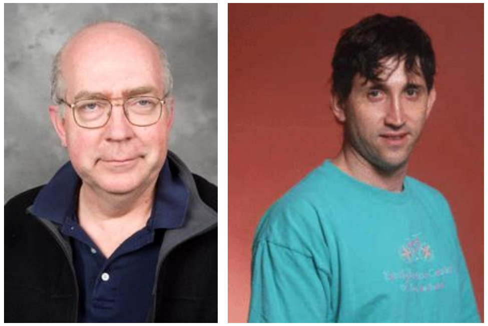
- Getting a small unique name is important
  - Smaller reads and writes/messages
  - Overall performance
  - Names are a natural prerequisite
- Renaming is related to:
  - Mutual exclusion
  - Test-and-set
  - Counting
  - Resource allocation



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



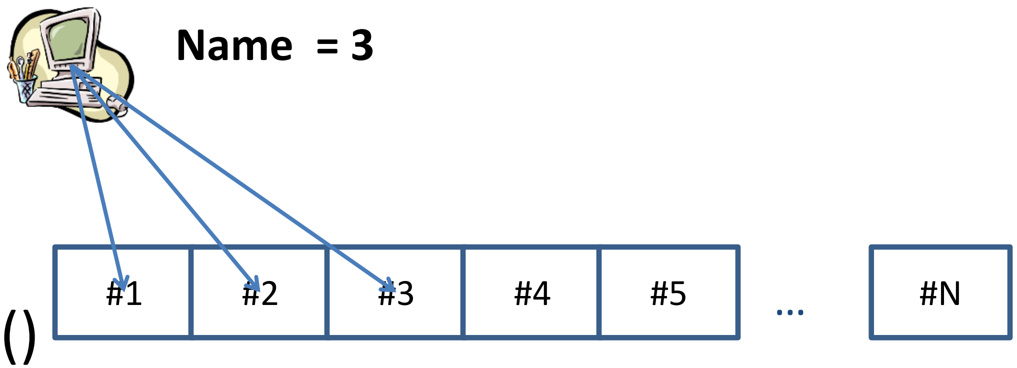
How can randomization help?

- 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 *renaming* from *test-and-set*
- We now know how to implement test-and-set in an asynchronous system

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
```



Name = 3

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?

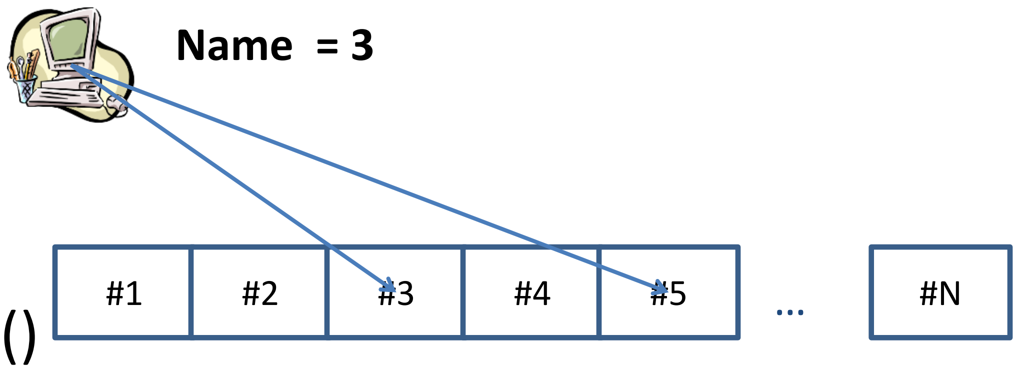
Randomized Renaming

- 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 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
```

#1	#2	#3	#4	#5	…	#N

1. **Claim**: The expected total number of tries is O(N log N)!

- Sketch of Proof (not for the exam):
- A process will win at most one test-and-set
- Hence it is enough to count the time until each test-and-set is accessed at least once!
- N items, we access one at random every time; how many accesses until we cover all N of them?
- *Coupon collector:* we need < 2N log N total accesses, with probability $1 – 1 / N^3$

Wrap-up

- **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 randomization in general:
  - Chapter 14 of “Distributed Computing: Fundamentals, Simulations, and Advanced Topics”, by Hagit Attiya and Jennifer Welch
- For the test-and-set example:
  - “Randomized two-process wait-free test-and-set” by John Tromp and Paul Vitányi.
  - “Wait-free test-and-set” by Afek et al.
- For the randomized consensus example:
  - “Optimal time randomized consensus”by Saks, Shavit, Woll.
  - “Randomized Protocols for Asynchronous Consensus” by Aspnes.
- For the renaming example:
  - “Fast Randomized Test-and-Set and Renaming” by Alistarh, Guerraoui et al.