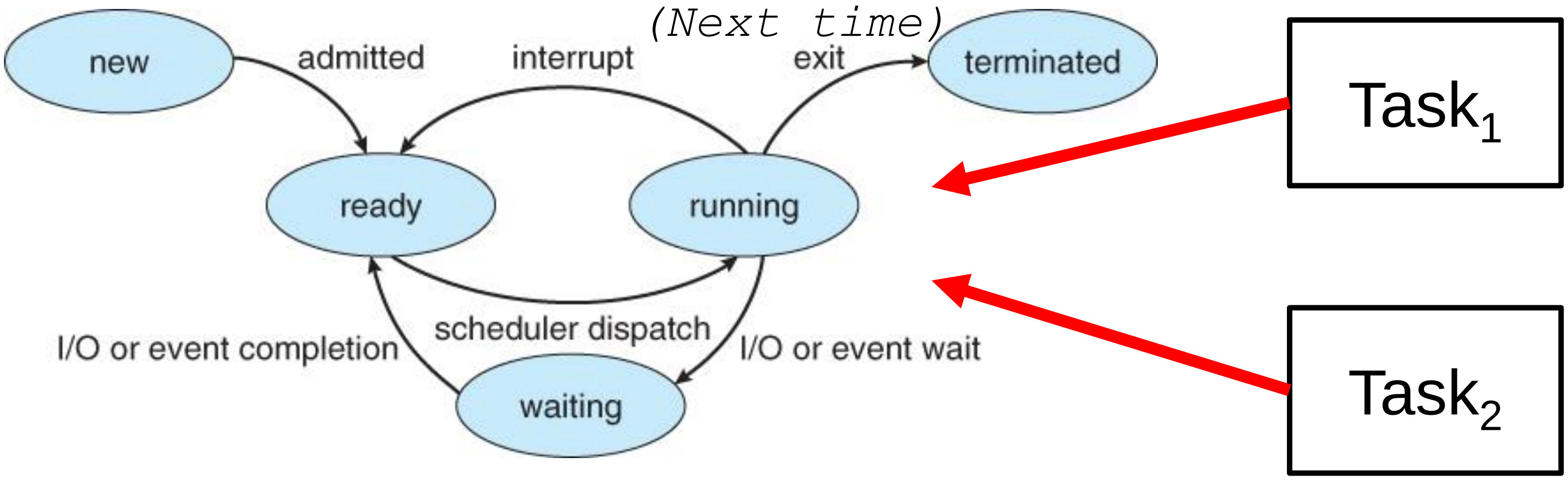
Performance Evaluation
and Applications

# Race, Concurrency, and the Exponential Distribution

# Motivating example

We have seen how to analyze very simple systems. But how can we analyze larger scenario, such as the process execution of an operative system supporting several concurrent tasks?



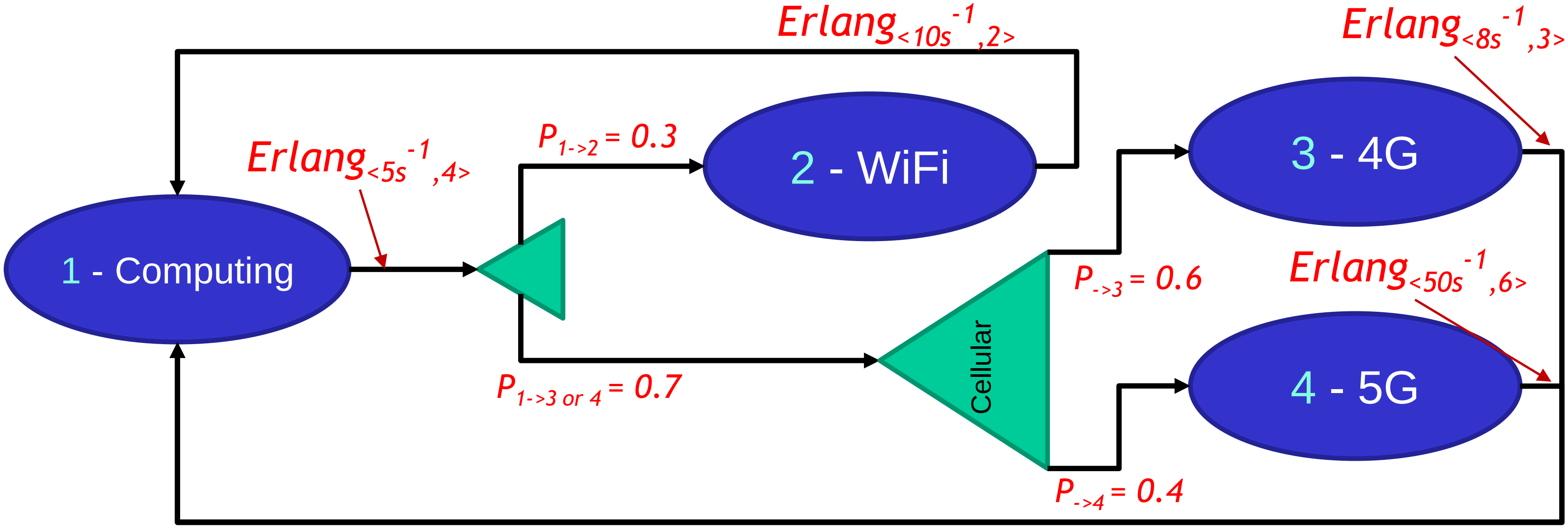
(Next time)

# Mobile Phone App *(Choice)*

Consider a Mobile Phone App that alternates between two phases: computing or sending data. The latter can be done using either WiFi or cellular (either 4G or 5G): depending on the technology this phase will take a different amount of time.

All timing have been measured and modelled with Erlang distributions with different number of stages and rate parameters.

Moreover, the probability of choosing either WiFi or cellular (4G or 5G) has been estimated.



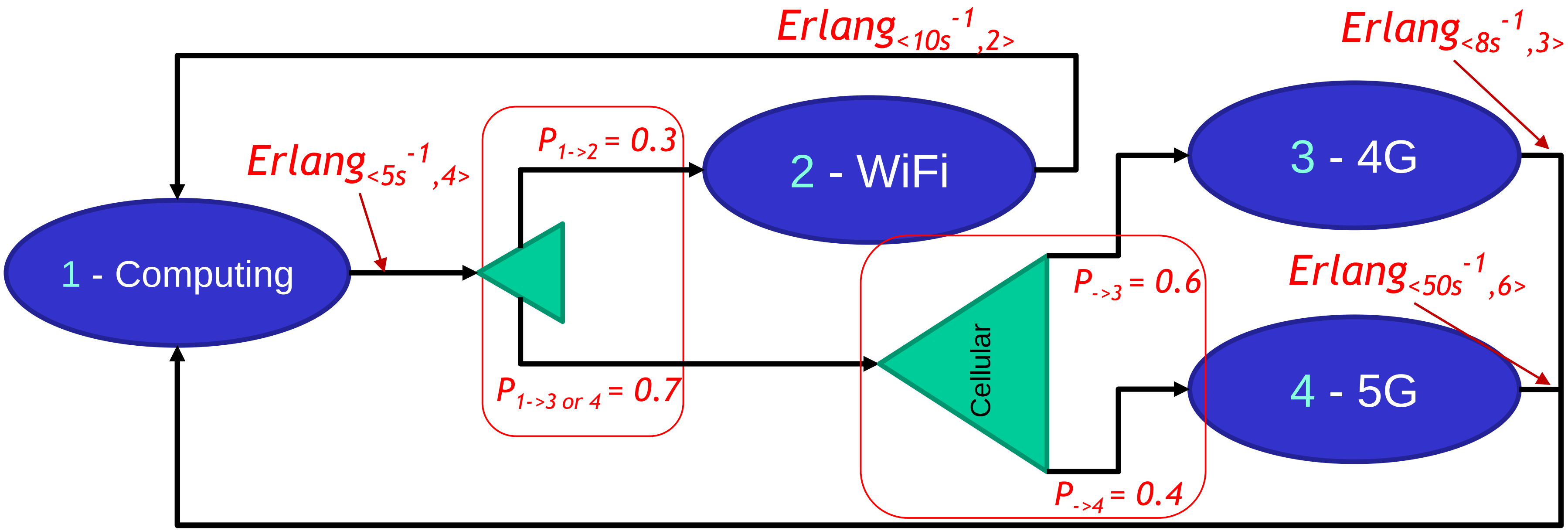
# Mobile Phone App *(Choice)*

We would like to compute:

- State probabilities
- Throughput

In this case, the choice between the transition from state 1 to either state 2, 3 or 4, is guided by a discrete random variable.

$P_{1\text{->}2} = 0.3$ $P_{1\text{->}3} = 0.7 \cdot 0.6 = 0.42$ $P_{1\text{->}4} = 0.7 \cdot 0.4 = 0.28$



# Mobile Phone App *(Choice)*

```
…
    if s = 1 then
            u = rand()
            if u < 0.3
                    ns = 2;
            else if u < 0.72
                    ns = 3;
            else
                    ns = 4;
            end
            dt = GenErlang(5,4);
    end
    if s = 2 then
            ns = 1;
            dt = GenErlang(10,2);
    end
    if s = 3 then
            …
…
```

$F_{1\text{->}2} = 0.3$

$F_{1\text{->}3} = 0.3 + 0.42 = 0.72$

$F_{1\text{->}4} = 0.72 + 0.28 = 1$

# Mobile Phone App *(Choice)*

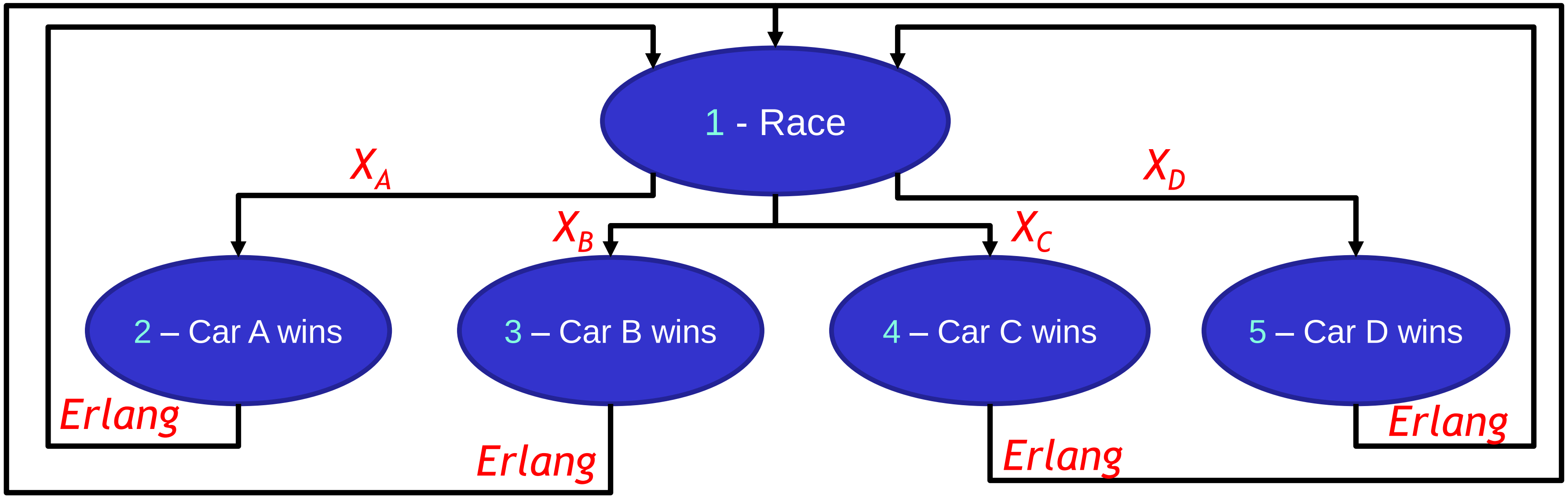
```
Ts1 = Ts2 = Ts3 Ts4 = C = 0;
…
     if s = 1 then
               …
               dt = GenErlang(5,4);
               Ts1 += dt;
     end
     if s = 2 then
               …
               Ts2 += dt;
               C ++;
     end
     if s = 3 then
               …
               Ts3 += dt;
               C ++;
     end
…
```

Also in this case, performance counters are updated in the code that handles each state.

```
Ps1 = Ts1 / T;
Ps2 = Ts2 / T;
Ps3 = Ts3 / T;
Ps4 = Ts4 / T;
X = C / T;
```

# Car Race Game

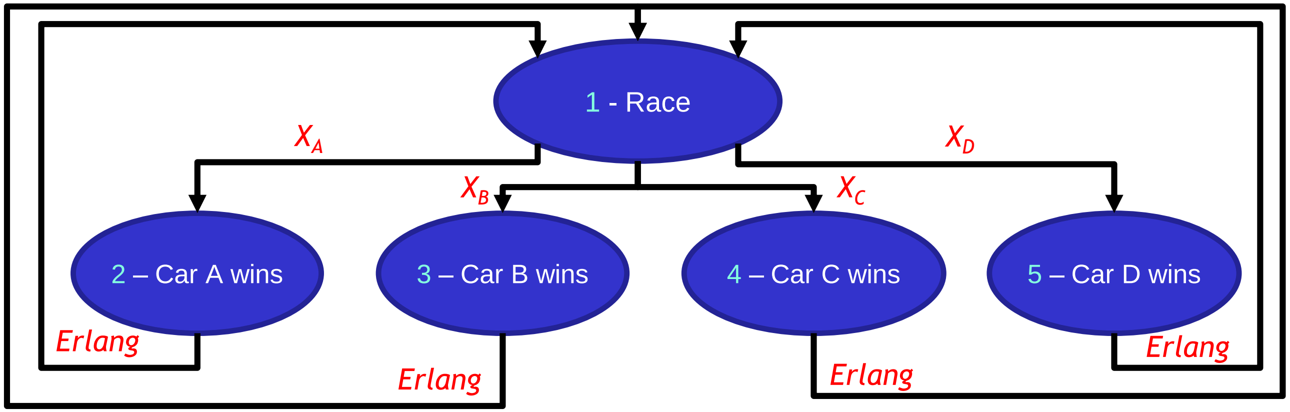
We want simulate a race, where four cars compete against each other. Each car has a different track completion time distribution. After a car wins the race, the game immediately stops and a new race restarts after an *Eralng* distributed amount of time. We can consider five states:



# Car Race Game

We would like to compute

- Victory probability for each car
- Average duration of a match.



# Car Race Game

```
…
    if s = 1 then
            tA = GenXA();
            …
            tD = GenXD();
            if (tA == min(tA, tB, tC, tD)) then
                    ns = 2; dt = tA;
            end
            if (tB == min(tA, tB, tC, tD)) then
                    ns = 3; dt = tB;
            …
    end
    if s = 2 then
            ns = 1;
            dt = GenErlang();
    end
    if s = 3 then
            ns = 1;
    …
…
```

# Car Race Game

```
…
    if s = 1 then
        if (tA == min(tA, tB, tC, tD)) then
            ns = 2; dt = tA;
            WinA ++;
        end
        …
        Races ++;
        RT(CurrRace,1) = dt;
        CurrRace ++;
    end
…
```

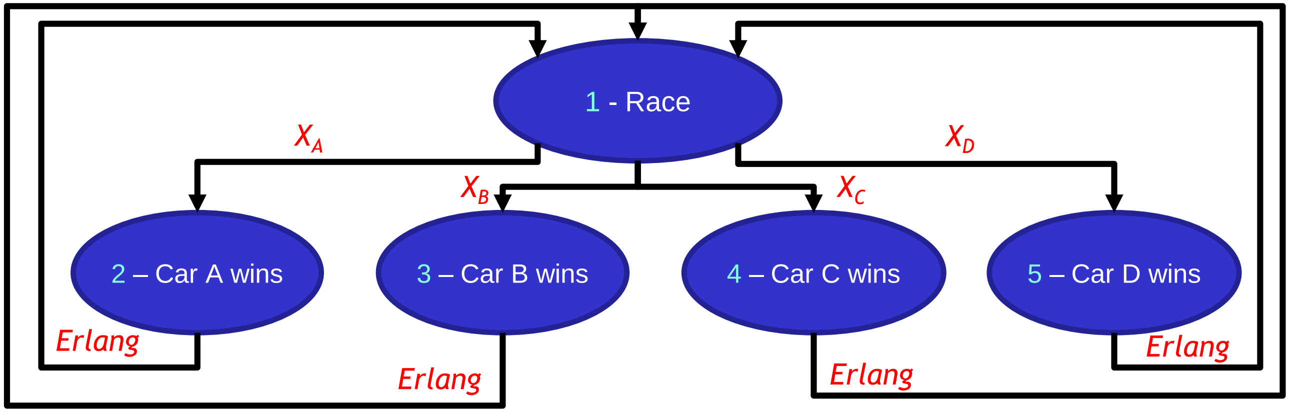
```
PwinA = WinA / Races;
PwinB = WinB / Races;
PwinC = WinC / Races;
PwinD = WinD / Races;
AvgRT = mean(RT);
```

# Car Race Game

In this case, the choice of the next state is determined by the so called *Race policy*:

- the next state is chosen according to the path that has extracted the minimum time

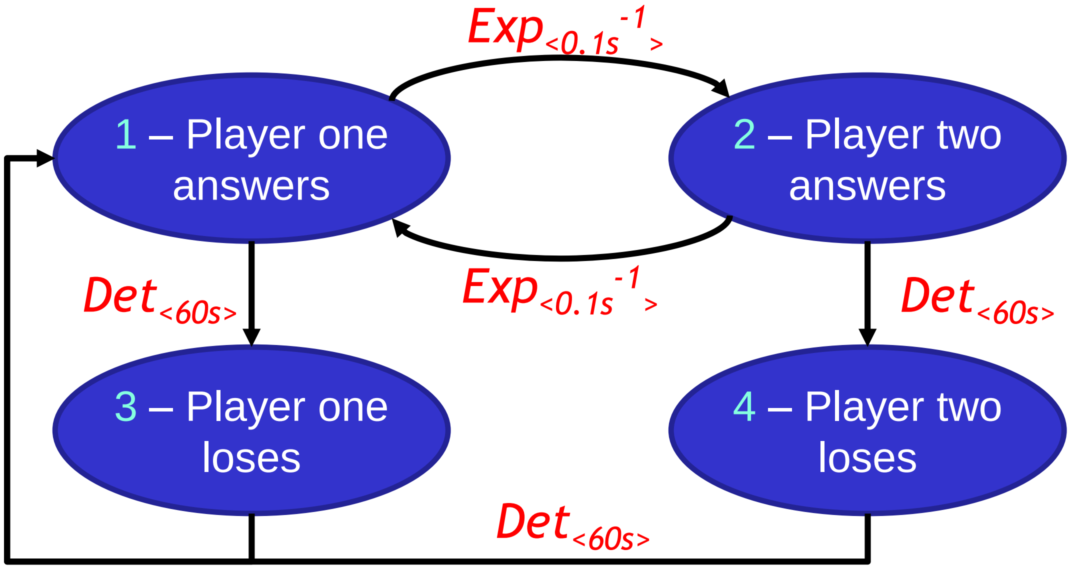
This is possible only in very few cases, in which all the transitions starts at the same time (such in this example), or when we have some special firing time distribution for the event.



# “L’Eredita”

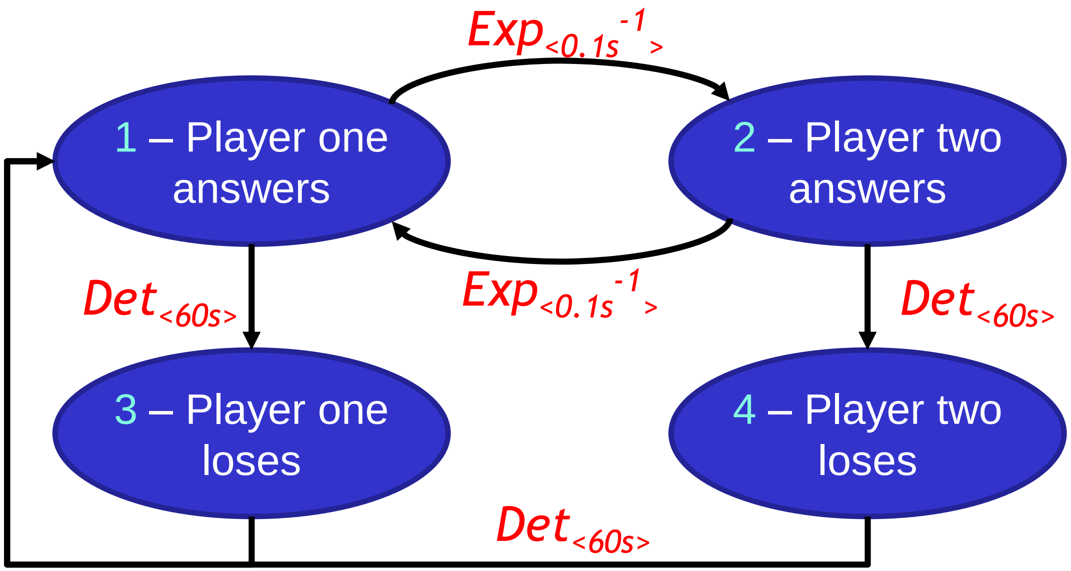
We want to simulate a two-contestant game: two people have *60 s.* each, in which they have to answer a set of questions. Each time a player gives the correct answer, the turn passes to the other and hers timer stops. The player for which the timer expires first loses the game. Each player finds the correct answer in an exponentially distributed amount of time, with an average of *10 s*.

Player one starts, and when a contestant loses, the game restarts after a deterministic time of *1 min*.



# “L’Eredita”

- Verify that the game is *fair*: each player has the same probability of winning.
- Compute the average length of one play.

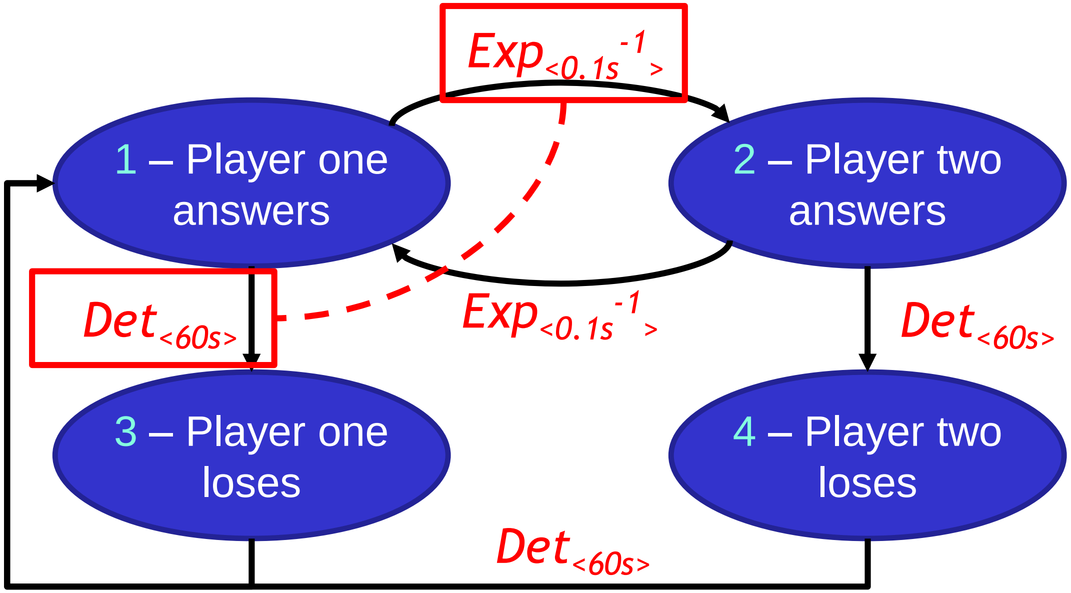


# "L'Eredita"

In this case we cannot use the "race policy" shown in the previous example to handle states with more than an output arc:

- Race policy would result in each player always having one full minute to answer the question, without the time running out.

Timers and answers are *concurrent* event, and must be handled separately.



# “L’Eredita”

```
timerA = 60; timerB = 60;
…
    if s = 1 then
            tA = GenExp(0.1);
            if (tA < timerA) then
                    ns = 2; dt = tA;
                    timerA -= tA;
            else
                    ns = 3; dt = timerA;
                    timerA = 60;
            end
    end
    if s = 2 then
            …
    end
    if s = 3 or s = 4 then
            ns = 1; dt = 60;
    …
…
```

# “L’Eredita”

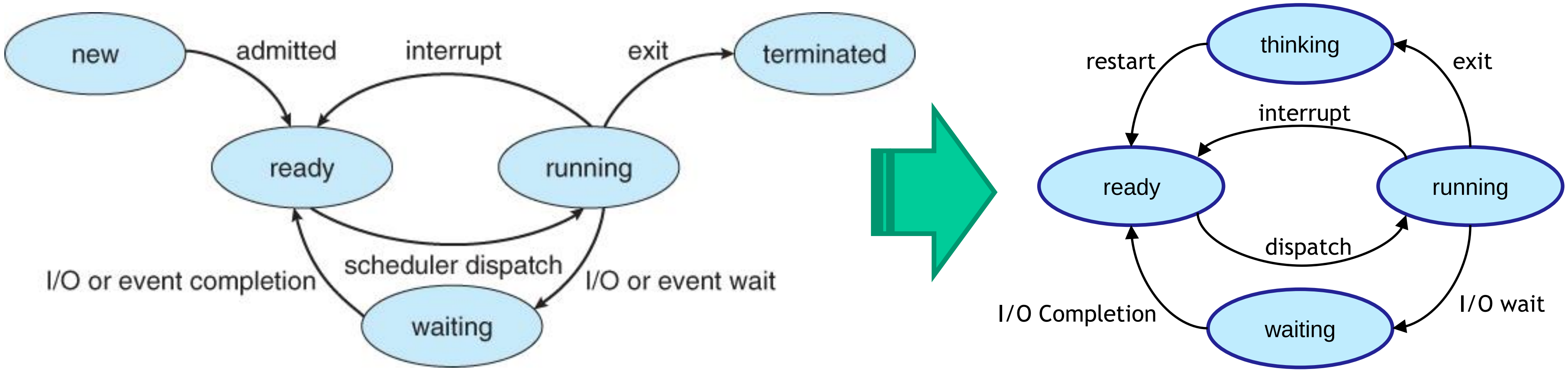
```
timerA = 60; timerB = 60; WinA = 0; WinB = 0; Match = 0;
…
    if s = 1 then
            tA = GenExp(0.1);
            if (tA < timerA) then
                    ns = 2; dt = tA;
                    timerA -= tA;
            else
                    ns = 3; dt = timerA;
                    timerA = 60; WinB ++;
            end
            RT(CurrMatch,1) += dt;
    end
    if s = 2 then
            …
    end
    if s = 3 or s = 4 then
            ns = 1; dt = 60;
            CurrMatch ++; Match ++;
    …
…
```

```
PWinA = WinA / Match;
PWinB = WinB / Match;
AvgRT = mean(RT);
```

# Concurrency

*Concurrency* can occur in many forms, and in general it cannot be handled with simple solutions like the ones previously seen.

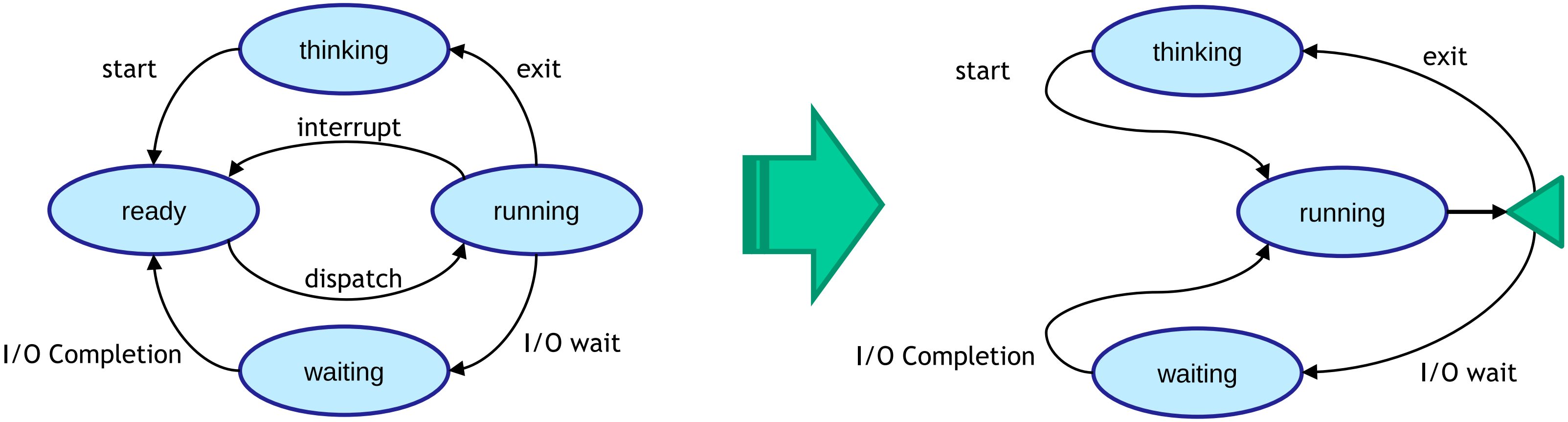
Consider for example the state machine for the operative system process that we saw last time, applied to a batch process (a *closed model*): whenever a job ends, it restarts after a “think time”.



# Concurrency

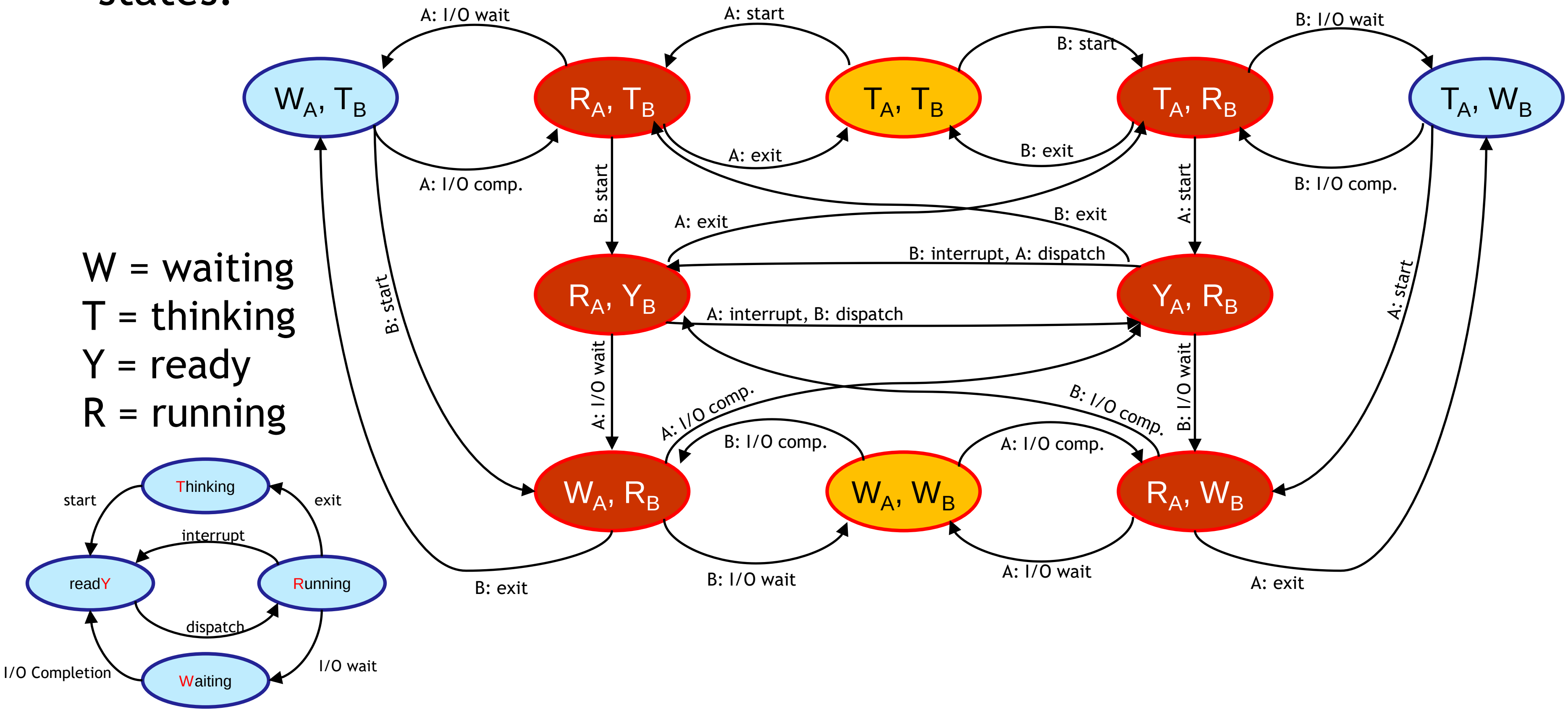
In case of a single batch process, jobs can be dispatched immediately, and the system would be in the “ready” state for a negligible amount of time. Such state could be removed.

Selection between “I/O wait” and “exit” in the running state can be performed using choice: a distribution of running before an event, and couple of probabilities for either ending or doing I/O.



# Concurrency

In case of more jobs, different types of concurrency occur in almost all the states, and *choice* cannot be used to solve the problem in all states.

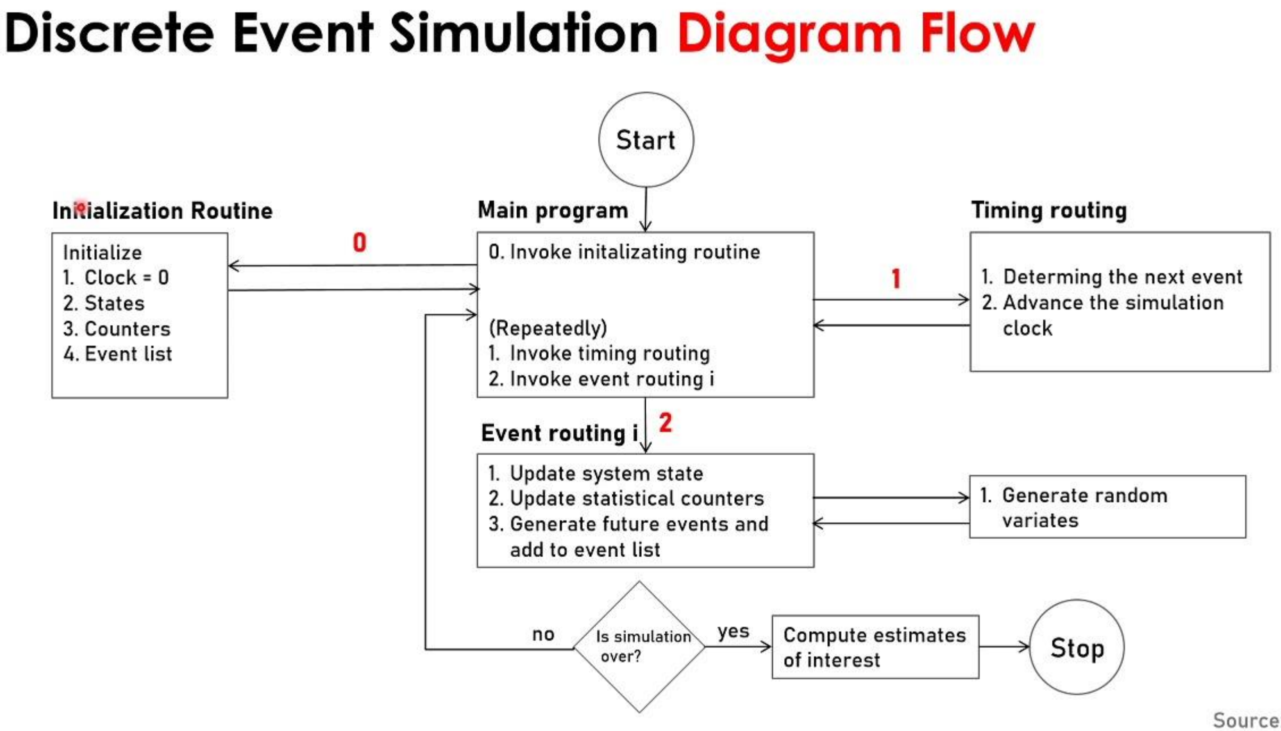


W = waiting
T = thinking
Y = ready
R = running

# Discrete Event Simulation

Concurrency in general can be handled using *Discrete Event Simulation*.

A complete description on how to perform Discrete Event Simulation, is however outside the scope of this course.



Discrete Event Simulation Diagram Flow

Start

Main program

0. Invoke initalizating routine

(Repeatedly)
1. Invoke timing routing
2. Invoke event routing i

Initialization Routine

Initialize
1. Clock = 0
2. States
3. Counters
4. Event list

Timing routing

1. Determing the next event
2. Advance the simulation clock

Event routing i

1. Update system state
2. Update statistical counters
3. Generate future events and add to event list

1. Generate random variates

Compute estimates of interest

Stop

Source: Law

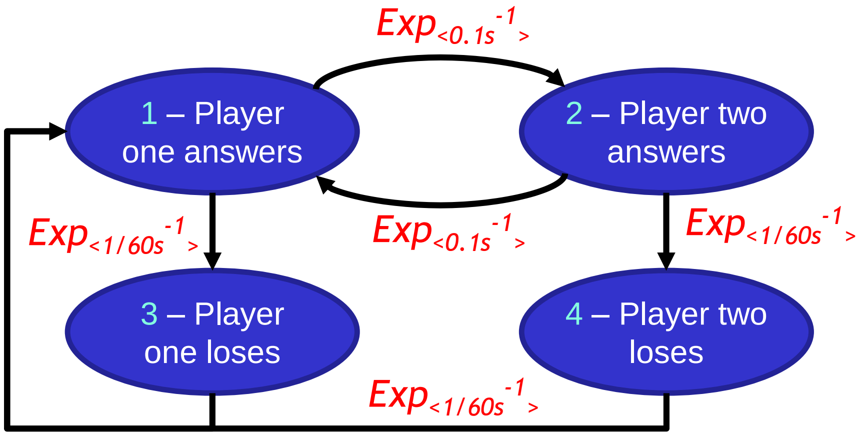
# Exponential distribution and concurrency

When ALL random timing follows an exponential distribution, analyzing systems become simpler thanks to its memory less property:

- Race policy becomes identical to concurrency
- Choice can be implemented with race policy, by embedding selection probability into the rates.

# Exponential distribution and concurrency

Race policy becomes identical to concurrency: for example, if in the "l'Eredità" game the timer would be exponentially distributed with an average of one minute, it could be analyzed in the following way:

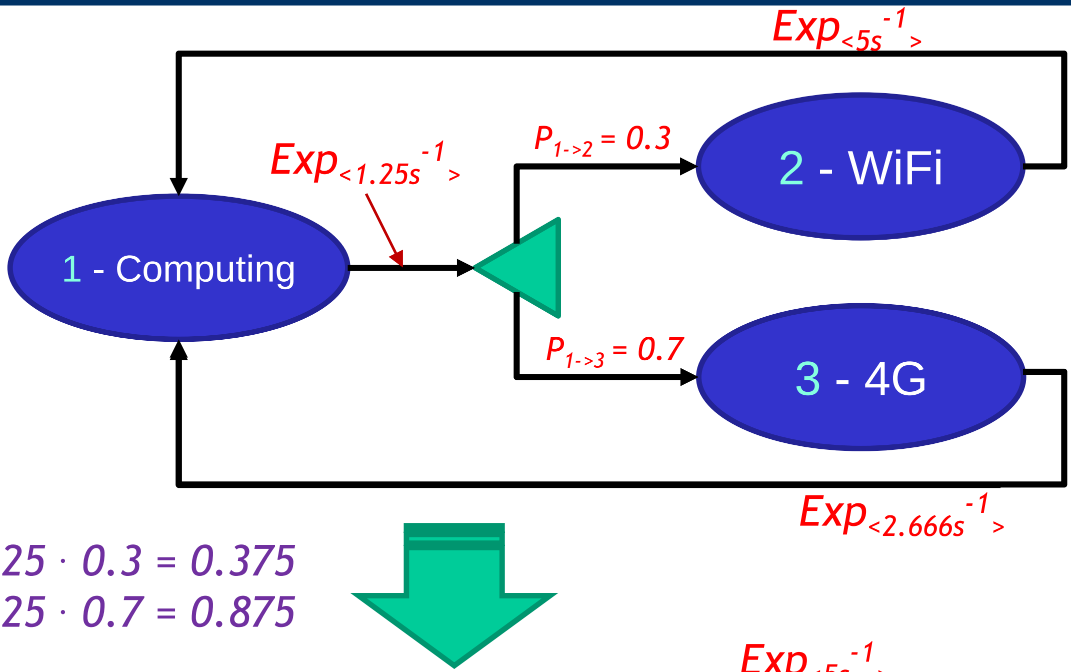


```
…
  if s = 1 then
       tA = GenExp(0.1);
       timer = GenExp(1/60);
       if (tA == min(tA, timer)) then
            ns = 2; dt = tA;
       else
            ns = 3; dt = timer;
       end
       …
  end
  if s = 2 then
       …
  end
  if s = 3 then
       ns = 1; dt = GenExp(1/60);
  …
…
```

# Exponential distribution and concurrency

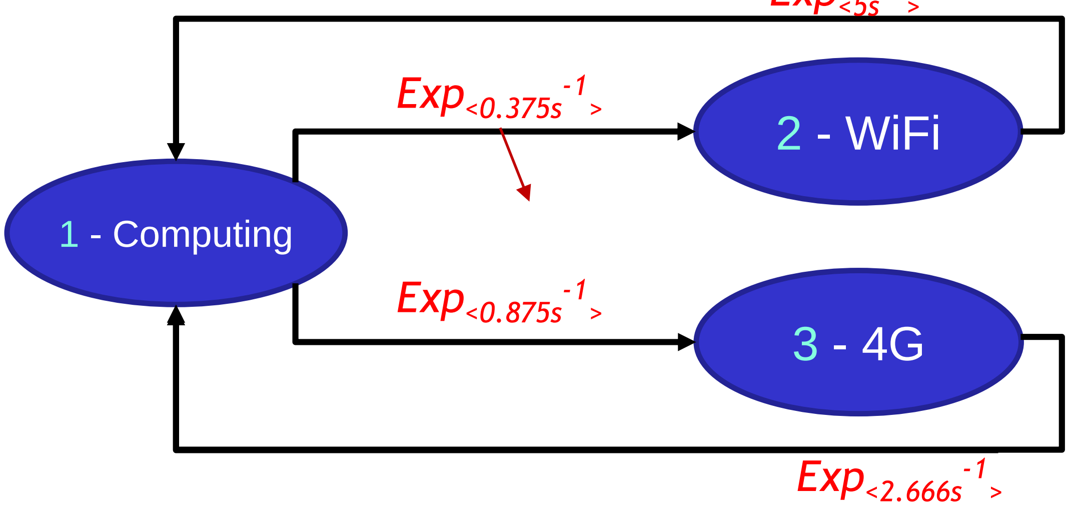
Choice can be implemented with race policy, using the product of the original exponential rate parameter with the selection probability.

For example, a 4G only version of the remote app model become:



*1.25 · 0.3 = 0.375*

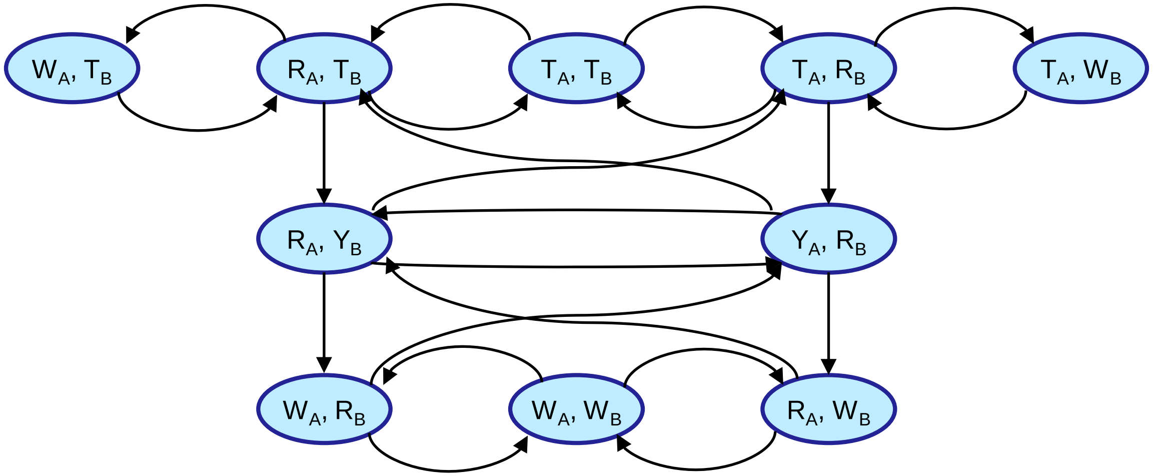
*1.25 · 0.7 = 0.875*



# Analysis of Motivating Example

If we can assume exponential distributed timing, we can create a state machine that includes all the possible evolutions, and analyze it using race policy.

W = waiting
T = thinking
Y = ready
R = running


$W_A$, $T_B$
$R_A$, $T_B$
$T_A$, $T_B$
$T_A$, $R_B$
$T_A$, $W_B$
$R_A$, $Y_B$
$Y_A$, $R_B$
$W_A$, $R_B$
$W_A$, $W_B$
$R_A$, $W_B$