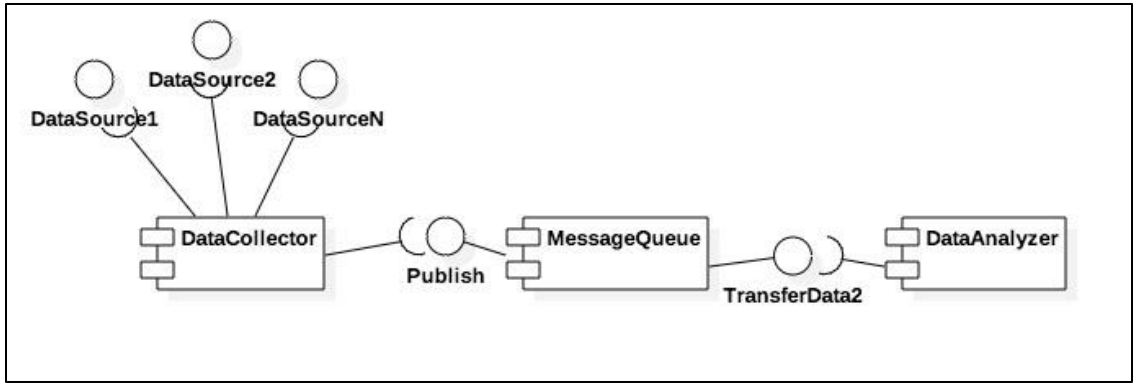
POLITECNICO
MILANO 1863

Exercise Lecture 4

Availability



Consider the system above. Assume that the participating components offer the following availability:

- DataCollector: 99%
- MessageQueue: 99.99%
- DataAnalyzer: 99.5%

Provide an estimation of the total availability of your system (you can provide a raw estimation of the availability without computing it completely).

Assuming that you wanted to improve this total availability by exploiting replication, which component(s) would you replicate? Please provide an argument for your answer.

How would such replication impact on the way the system works and is designed?

Availability - solution

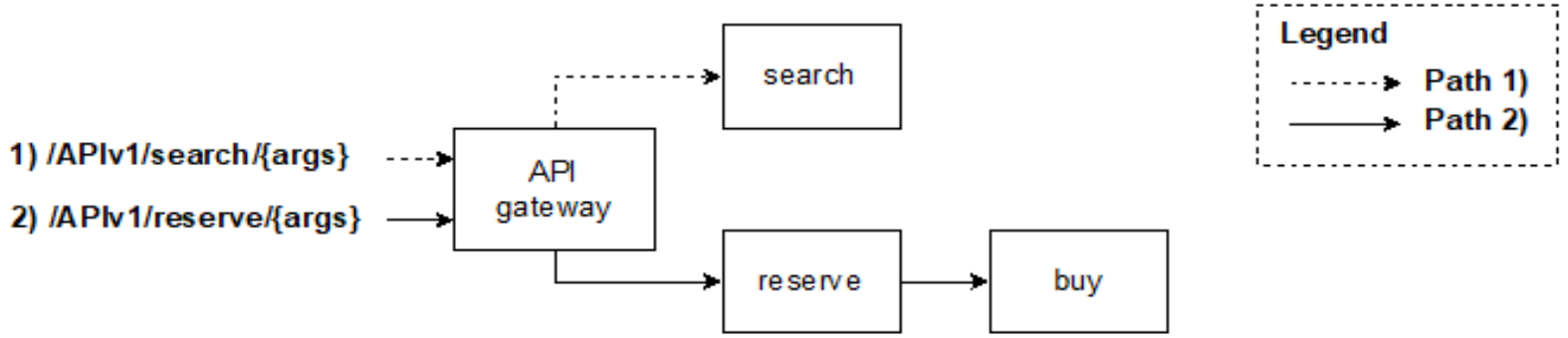
- Data must flow through the whole chain of components to be processed. This implies that we can model the system as a series of components.
- The total availability of the system is determined by the weakest element, that is, the DataCollector.
  - A_{Total} = 0.99*0.9999*0.995 = 0.985
- If we parallelize the data collector adding a new replica, we can achieve the following availability:
  - A_{Total} = (1-(1-0.99)^2) *0.9999*0.995 = 0.995

Availability - solution

- Impact of parallelization:
  - If both DataCollector replicas acquire information from the same sources (hot spare approach), then the other components will see all data duplicated and will have to be developed considering this situation.
  - For instance, the MessageQueue could discard all duplicates.
  - Another aspect to be considered is that both DataSources and MessageQueue have to implement mutual exclusion mechanisms that ensure the communication between them and the two DataCollector replicas does not raise concurrency issues.
  - Another option could be that only one DataCollector replica at a time is available and the other is activated only when needed, for instance, if the first one does not send feedback within a certain timeout (warm or cold spare).

Availability - TrainTicket

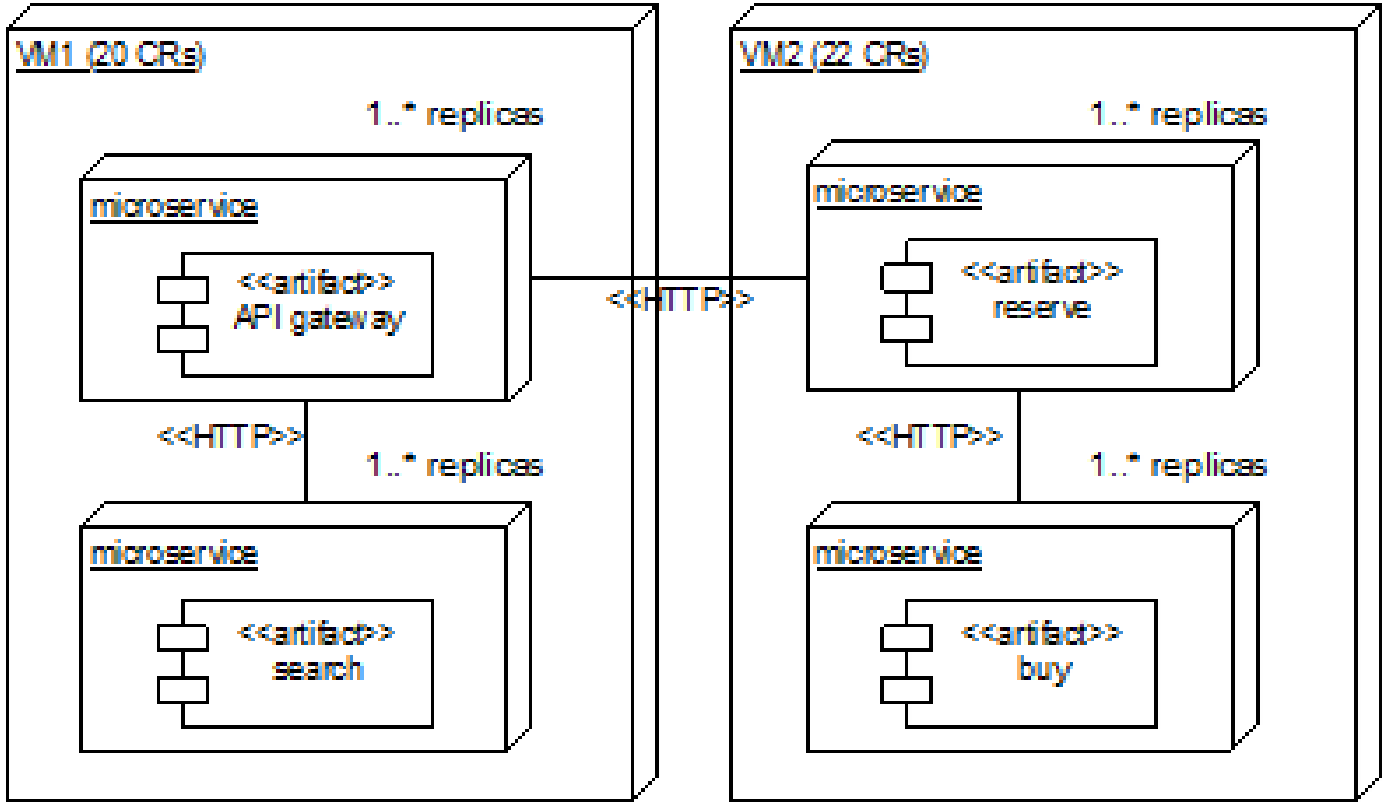
- Consider a microservices application called TrainTicket composed of 3 domain microservices (search, reserve, buy) and 1 additional microservice that acts as the API gateway. TrainTicket supports two basic operations invoked using the exposed RESTFul APIs:
- **search**: /APIv1/search/{args}
- **reserve**: /APIv1/reserve/{args}



- Requests (both search and reserve) are received and then dispatched by the API gateway. In particular, the following high-level schema shows how requests propagate from the gateway to internal microservices. Note that in this example reserve includes also the purchase of reserved items.

Availability - TrainTicket

- Microservices run in units deployed onto 2 different Virtual Machines, VM1 and VM2 as shown in the following UML deployment diagram.



Availability - TrainTicket

POLITECNICO MILANO 1863

The available VMs have Computational Resources (CRs) that can be allocated to run microservices. Each VM has a maximum number of CRs and each microservice requires a certain number of CRs, according to the executed artifact. As shown in the schema, available CRs are as follows:

- VM1: 20 CRs
- VM2: 22 CRs

The mapping between microservices and required CRs is as follows:

- API gateway: 2 CRs
- search: 5 CRs
- reserve: 4 CRs
- buy: 5 CRs

Availability - TrainTicket

- The deployment diagram shows that each microservice can be replicated to have redundant business-critical components. In the latter case, requests are directed to all the replicas rather than to an individual instance and the first answer received from a replica is returned to the caller, while the others are simply ignored. The number of replicas for each microservice shall be defined so that the following nonfunctional requirement is satisfied and the deployment constraints defined in the deployment diagram and above are fulfilled.

- **R1: “Both search and reserve services exposed through API gateway shall have availability greater than or equal to 0.99”.**

Availability – TrainTicket: R1

Considering the constraints of the execution environment represented above, determine whether requirement R1 can be satisfied or not assuming the following availability estimates for each microservice:

- API gateway: 0.99
- search: 0.98
- reserve: 0.95
- buy: 0.91

Availability – TrainTicket: R1 Solution

Considering the execution environment, we can derive the following inequations constraining the number of replicas:

Constraints extracted from environment:

- 2x + 5y <= 20
- 4u + 5z <= 22

Constraints extracted from requirement R1:

- (1 - (1 - 0.99)^x) * (1 - (1 - 0.98)^y) >= 0.99
- (1 - (1 - 0.99)^x) * (1 - (1 - 0.95)^u) * (1 - (1 - 0.91)^z) >= 0.99

Where variables x, y, u, and z represent the number of replicas for the microservices API gateway, search, reserve, and buy, respectively.

The requirement R1 can be satisfied since there exists a valid assignment to variables that satisfies all constraints. For instance:

- x = 2, y = 2, u = 3, z = 2

Availability – TrainTicket

Consider the problem of resource allocation taking into account the operational profile, that is, the behavior of the users. Assume the following workload in terms of average number of concurrent users for each request:

- search: 50 users
- reserve: 90 users

Assume also that for reserve, only 20% of users complete the purchase at reservation time. This means that 20% of reserve requests get through and reach the buy microservice, while 80% of them terminate the execution without calling buy.

Availability – TrainTicket

After a preliminary analysis, we realize that availability depends on the workload according to the following new estimates:

LOW workload: 0-60 concurrent users

Microservice	**Availability**
API gateway	0.99
search	0.98
reserve	0.95
buy	0.91

HIGH workload: 60-150 concurrent users

Microservice	**Availability**
API gateway	0.98
search	0.95
reserve	0.93
buy	0.90

Availability – TrainTicket: R2

- Does the execution environment have enough computational resources to support the workload defined above still fulfilling requirement R1 and the defined constraints?

Availability – TrainTicket: R2 Solution

The expected workload for each microservice is as follows:

- API gateway: 140 users (HIGH)
- search: 50 users (LOW)
- reserve: 90 users (HIGH)
- buy: 18 users (LOW)

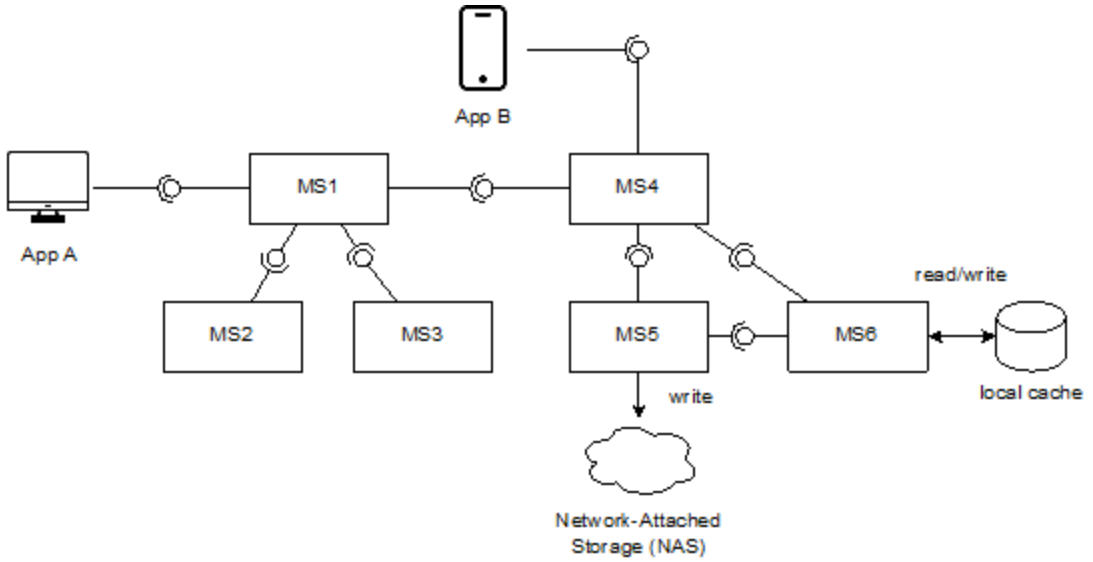
The constraints extracted from requirement R1 become as follows:

- (1 - (1 - 0.98)^x) * (1 - (1 - 0.98)^u) >= 0.99
- (1 - (1 - 0.98)^x) * (1 - (1 - 0.93)^u) * (1 - (1 - 0.91)^z) >= 0.99

An optimal resource allocation is represented by the assignment x = 2, y = 2, u = 3, z = 2 that is again feasible according to environment constraints.

Software Architecture

- Consider the following high-level component structure that is a static description of a possible microservices-based system implementing the business logic for two applications (App A, and App B). The UML Component Diagram below highlights the interfaces (both provided and required ones) of microservices MS1, …, MS6. Moreover, the diagram shows that MS5 writes data on a Network-Attached Storage (NAS) device and MS6 reads and write data on a local cache.



Software Architecture - R1

- Based exclusively on the static information presented in the Component Diagram, enumerate all possible execution paths (in terms of sequences of microservices) triggered by requests generated by App A and App B, respectively. Notice that we are interested only in paths that terminate with a microservice that does not require any further interface.
- Then, describe the ripple effect possibly generated by a sudden slowdown of the NAS on each of the identified paths. Assume every request to MS5 causes interactions with the NAS. Identify the application(s) affected by this disruptive event. Justify your answer.

Software Architecture – R1: Solution

POLITECNICO MILANO 1863

According to static information only, the possible paths triggered by either App A or App B are as follows:

- App A -> MS1 -> MS3
- App A -> MS1 -> MS4 -> MS5 -> MS6
- App A -> MS1 -> MS4 -> MS6
- App B -> MS4 -> MS5 -> MS6
- App B -> MS4 -> MS6

- A sudden slowdown of the NAS could cause a growing number of pending requests to MS5 due to high number of concurrent (slow) operations of the NAS. This could cause a saturation of the resources of MS5. Therefore, pending requests to MS4 also grow potentially saturating its available resources. Essentially the ripple effect follows in a backward manner all the paths listed above that include MS5 or MS4. Thus, according to the paths the effect potentially propagates back to both App A and App B.

Software Architecture – R2

Assume you have the following additional information about the runtime behavior of the system. Microservice MS5 writes data into the NAS and then propagates the same pieces of data to MS6, which writes them on the local cache. Remember that MS5 does not read data from the NAS. All reads are carried out by MS6 using the local cache only.

Describe how you can use the circuit breaker pattern to mitigate the disruption described above (SA_Q1), taking into account the following constraints:

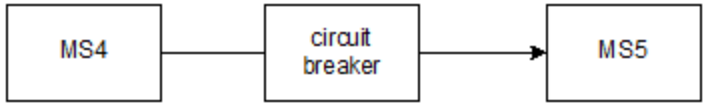
- NAS must be avoided during the disruption.
- All write operations shall eventually take place, either on the NAS or on the local cache, or on both. Do not consider possible data reconciliation issues.

In your description, explicitly state:

- how many circuit breakers you would use;
- where you would place them;
- their behavior.

Software Architecture – R2: Solution

- Only one circuit breaker is necessary to avoid interactions with the slow NAS. We can place the circuit breaker in between MS4 and MS5.
- As soon as the number of failed requests grows above the threshold, the circuit breaker goes into state open. At this point the mechanism drops all requests directed to MS5 and runs a fallback procedure that interacts directly with MS6, thus writing the fresh data into the local cache (rather than the NAS). When the circuit breaker changes from half-open to closed, write requests can follow again the nominal flow (… -> MS4 -> MS5 -> MS6).



Software Architecture – R3

Assume now that the expected number of requests per second (workload) for the two applications is as follows:

Application	**Workload**
App A	100
App B	150

You can also assume the following behavior when all circuit breakers are closed:

- Requests directed to MS1 are always distributed evenly among all microservices used by MS1.
- 25% of the requests to MS4 are write operations and are transferred to MS5, which, in turn, executes them into the NAS and transfers them to MS6 for local cache updates.
- 75% of the requests to MS4 are read-only operations and are transferred to MS6.
- The number of requests generated by MS2 is always negligible.

Software Architecture – R3

Address the following points:

- Determine the number of requests per second that reach each microservice.
- Use the numbers you have computed to determine the availability of each individual microservice replica according to the following rules: if the total number of requests a microservice receives is less than 60, the availability estimate is 0.99, if the number of requests is between 60 and 80 the estimate is 0.98, 0.97 otherwise. These estimates apply to each of the considered microservices.
- Focus now on the write operations generated by App B and on the corresponding path. What is the availability shown by App B for write operations when a single replica of each microservice is used?
- If we want to satisfy the following nonfunctional requirement: “The total availability of App B for write operations, shall be at least 0.999, under the condition that any circuit breaker is closed”, what is the required minimum number of replicas for each of the involved microservices?

Software Architecture – R3: Solution

When the circuit is *closed,* given the defined workload, we have the following concurrent requests per each microservice:

- MS1 = 100 requests
- MS2 negligible
- MS3 = 50 requests
- MS4 = 150 + 50 = 200 requests
- MS5 = 200 * 0.25 = 50 requests
- MS6 = 200 * 0.75 + 50 = 200 requests

Under such conditions, individual availability estimates are as follows:

- MS1 = 0.97
- MS2 = 0.99
- MS3 = 0.99
- MS4 = 0.97
- MS5 = 0.99
- MS6 = 0.97

Software Architecture – R3: Solution

The path followed by write operations incoming from App B is MS4 -> MS5 -> MS6, which, for this purpose, are organized in a series. The availability is then: 0.97*0.99*0.97 = 0.93

To satisfy the non-functional requirement we need to introduce additional replicas for all the three involved microservices. In general, we should satisfy the following constraint:

- $(1 – (1 – 0.97)^x) * (1 – (1 – 0.99)^y) * (1 – (1 – 0.97)^z) > 0.999$

With x, y, z, number of replicas for MS4, MS5, MS6, respectively.

Assignments satisfying the previous condition are:

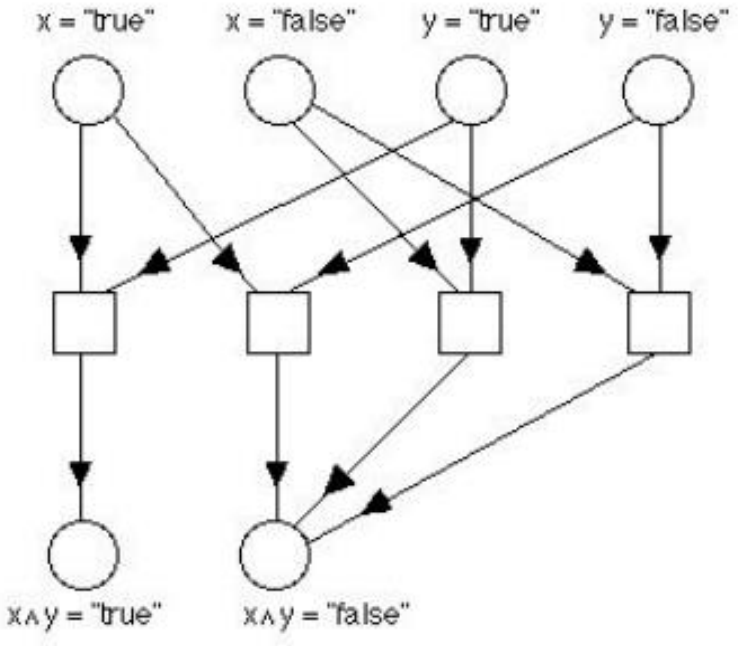
- x=2, y=3, z=3 and x=3, y=3, z=2 that lead to Avail = 0,9991
- x=3, y=2, z=3 that leads to Avail = 0,9998

Instead, instantiating only two replicas per service leads to Avail = 0,9981, while two replicas for MS5 and MS6 or MS4 and three replicas for the remaining service leads to Avail = 0,99897312, which is still slightly lower than 0,999.

Petri Nets - R1

- Construct a Petri net model for calculating the logical conjunction of two variables x and y, each of which takes the values “true” or “false”. Each is assigned a value independently of the other.

Petri Nets - R1: Solution

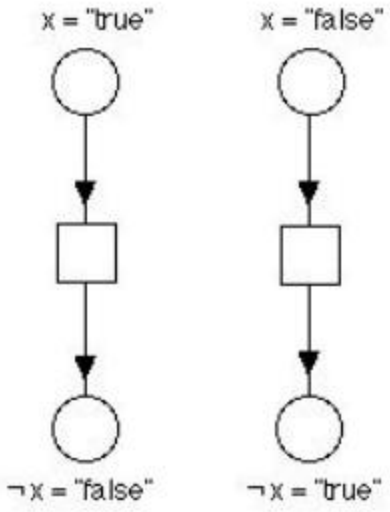


To see how this net works, we walk through the firing sequence corresponding to x = “true” and y = “false”. We start with one token in the places corresponding to x =“true” and y = “false”, the leftmost and rightmost places on the top line of Figure 9. Now only one transition is enabled for firing, the second one from the left. Firing that transition puts one token into the place labeled x ∧ y = “false”. No further transition firings are possible and the desired result is obtained.

Petri Nets - R2 easy

- Construct a Petri net that computes the negation of x, which again takes only the values “true” and “false”.

Petri Nets - R2 easy: solution

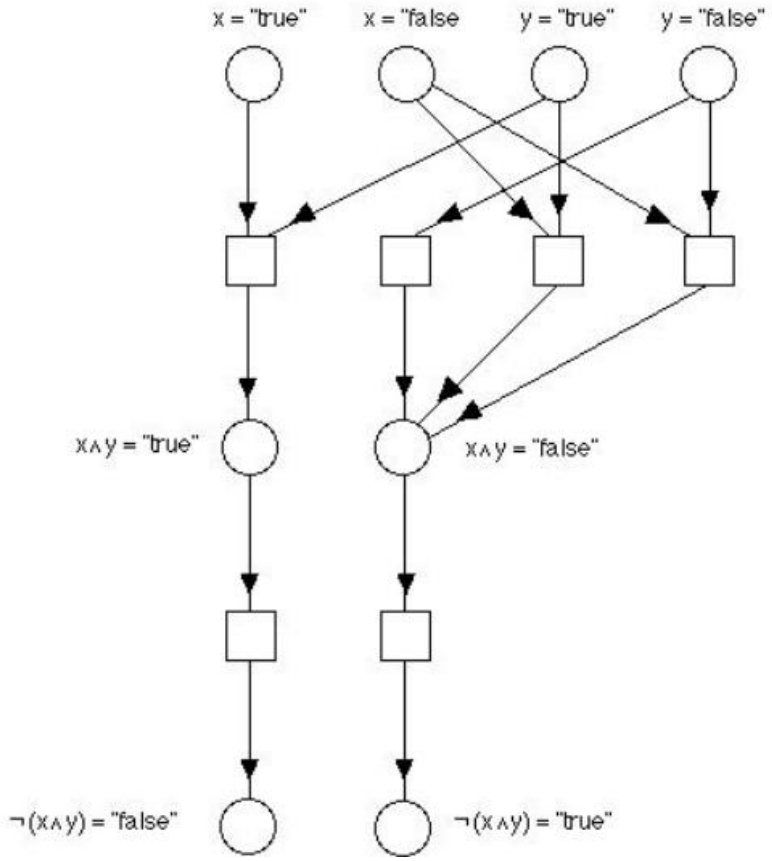


If we want to know the value of ¬x when x has a certain value, we put one token in the place for that value, carry out the firing, and observe which place has a token in it.

Petri Nets - R3 easy

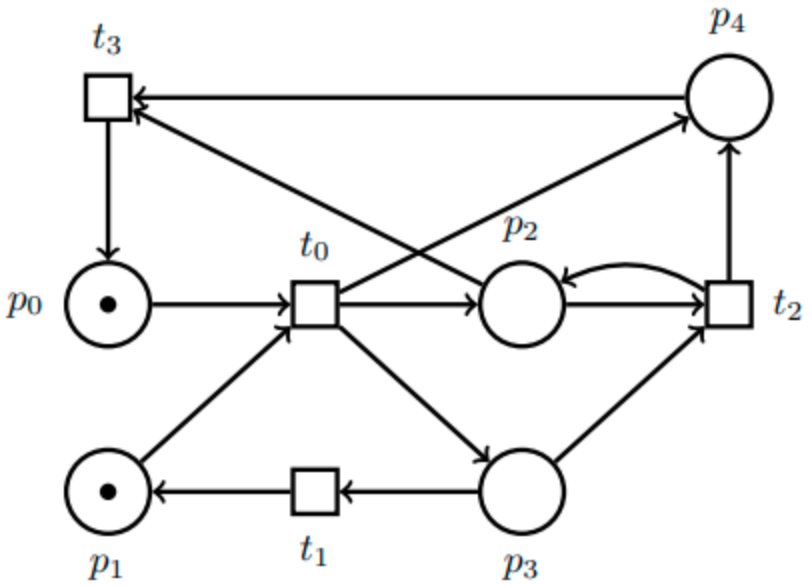
- Construct a Petri net for computing ¬(x ∧ y) by composing the previous nets.

Petri Nets - R3 easy: Solution



Petri Notes - R4

- For Petri net (N , M0) below:



Construct the reachability graph of (N , M0).

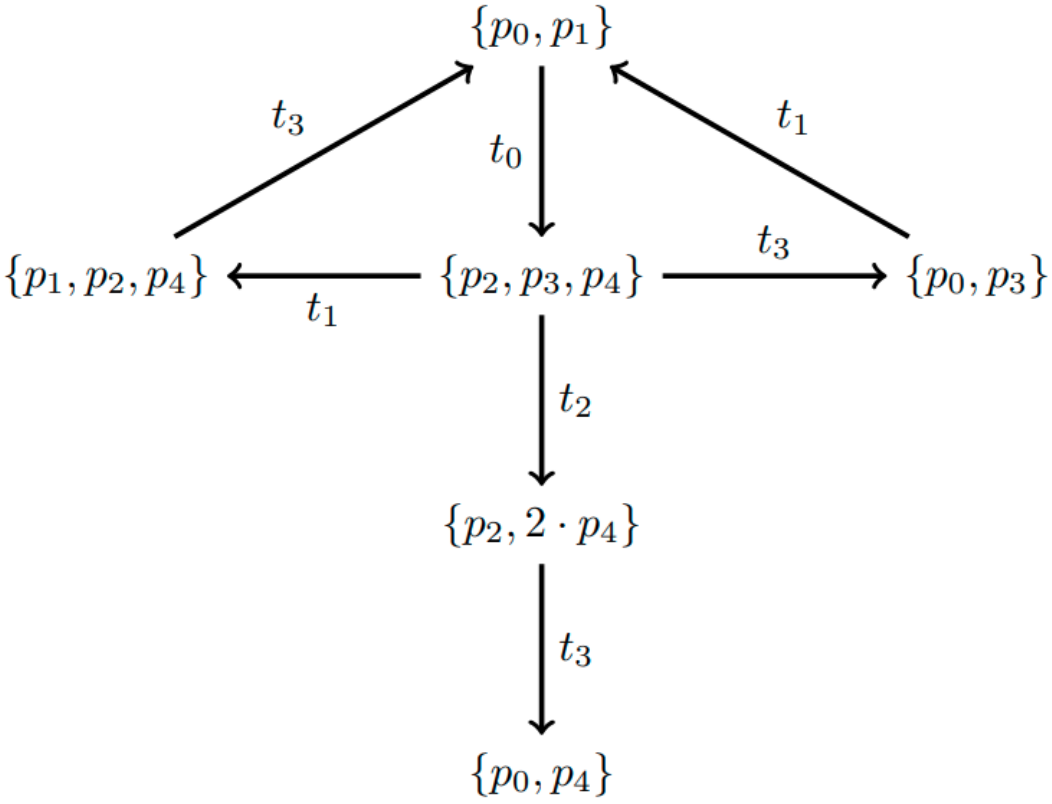
Reachability Graph Algorithm

POLITECNICO MILANO 1863

The reachability graph of a Petri net is a directed graph, G=(V, E), where each node, v ∈ V, represents a reachable marking and each edge, e ∈ E, represents a transition between two reachable markings.
The main steps for the construction of the reachability graph of a marked net (N,m0) are summarized in the following algorithm.

1. *The initial node of the graph is the initial marking* $\mathbf{m}_0$. *This node is initially unlabeled.*
2. *Consider an unlabeled node* $\mathbf{m}$ *of the graph.*
   a *For each transition t enabled at* $\mathbf{m}$, *i.e., such that* $\mathbf{m} \geq \mathbf{Pre}[\cdot, t]$:
      i. *Compute the marking* $\mathbf{m}' = \mathbf{m} + \mathbf{C}[\cdot, t]$ *reached from* $\mathbf{m}$ *firing t.*
      ii. *If no node* $\mathbf{m}'$ *is on the graph, add a new node* $\mathbf{m}'$ *to the graph.*
      iii. *Add an arc t from* $\mathbf{m}$ *to node* $\mathbf{m}'$.
   b *Label node* $\mathbf{m}$ *"old".*
3. *If there exist nodes with no label, goto Step 2.*

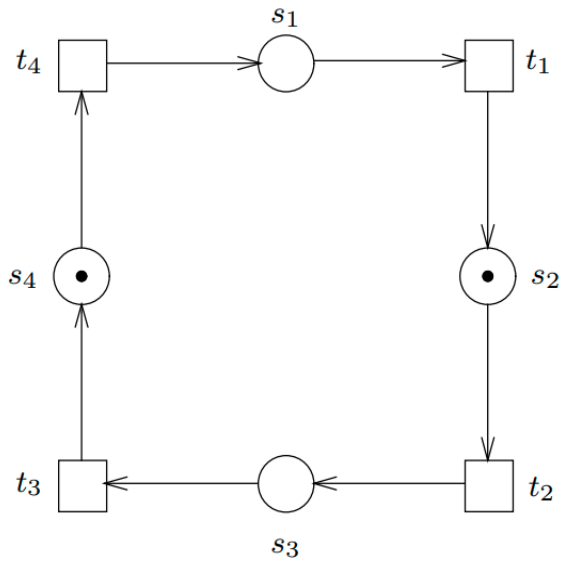
Petri Notes - R4: Solution



Petri Nets – R5

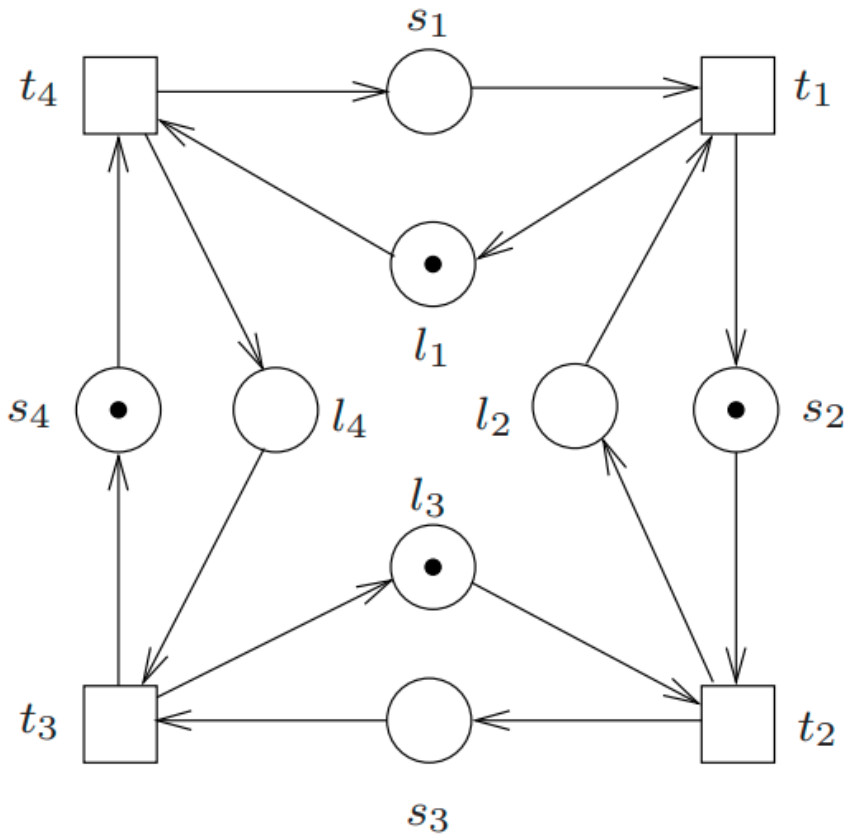
Four cities are connected by unidirectional train tracks building a circle. Two trains circulate on the tracks. We model the four tracks by places

s1, . . . , s4. A token on si means that there is train in the i-th track. Our task is to ensure that it will never be the case that two trains occupy the same track.



Petri Nets – R5: Solution

The four control places l1, . . . , l4 guarantee that no reachable marking puts more than one token on si .



Petri Nets – R5

- How can we prove that everything works?

Petri Nets – R5

- This property can be proven by means of the reachability graph. Since every reachable marking puts at most one token on a place, we denote a marking by the set of places marked by it.