# Verification & Validation

Terminology

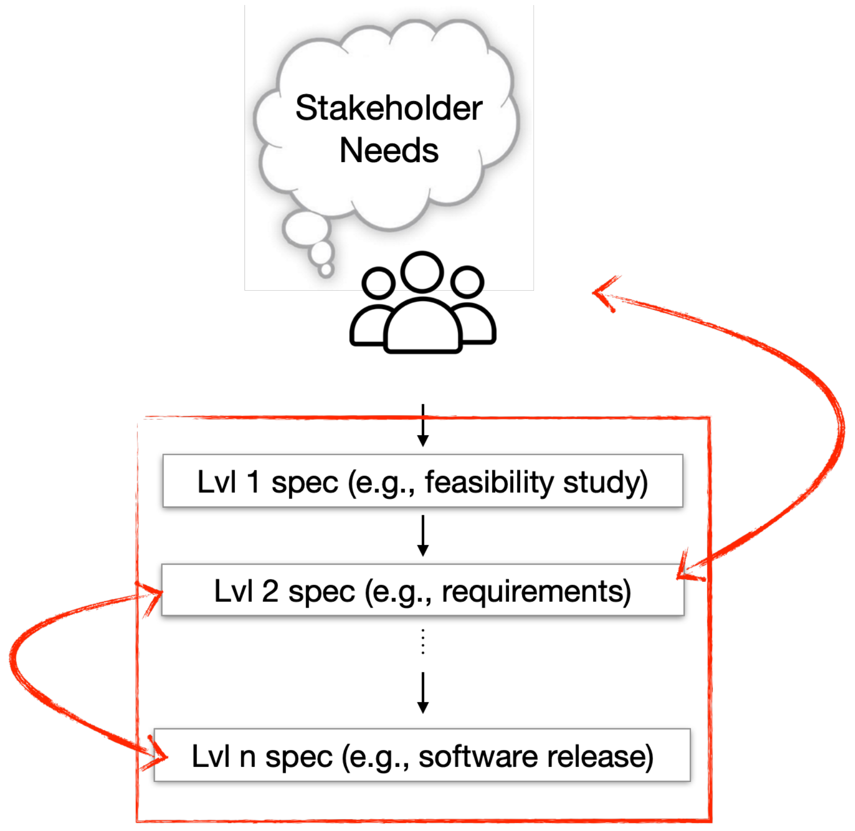
## Verification & Validation

- Verification is internal
  - *Are we building the software right (w.r.t. a specification)?*
- Validation is external
  - *Are we building the right software (w.r.t. stakeholder needs)?*

# Verification & Validation

**Verification**
*"is level i consistent with level i+1?"*

internal consistency check


Stakeholder Needs
Lvl 1 spec (e.g., feasibility study)
Lvl 2 spec (e.g., requirements)
Lvl n spec (e.g., software release)

(abstract) process

**Validation**
*"does level i conform to needs?"*

external consistency check

## Quality Assurance (QA)

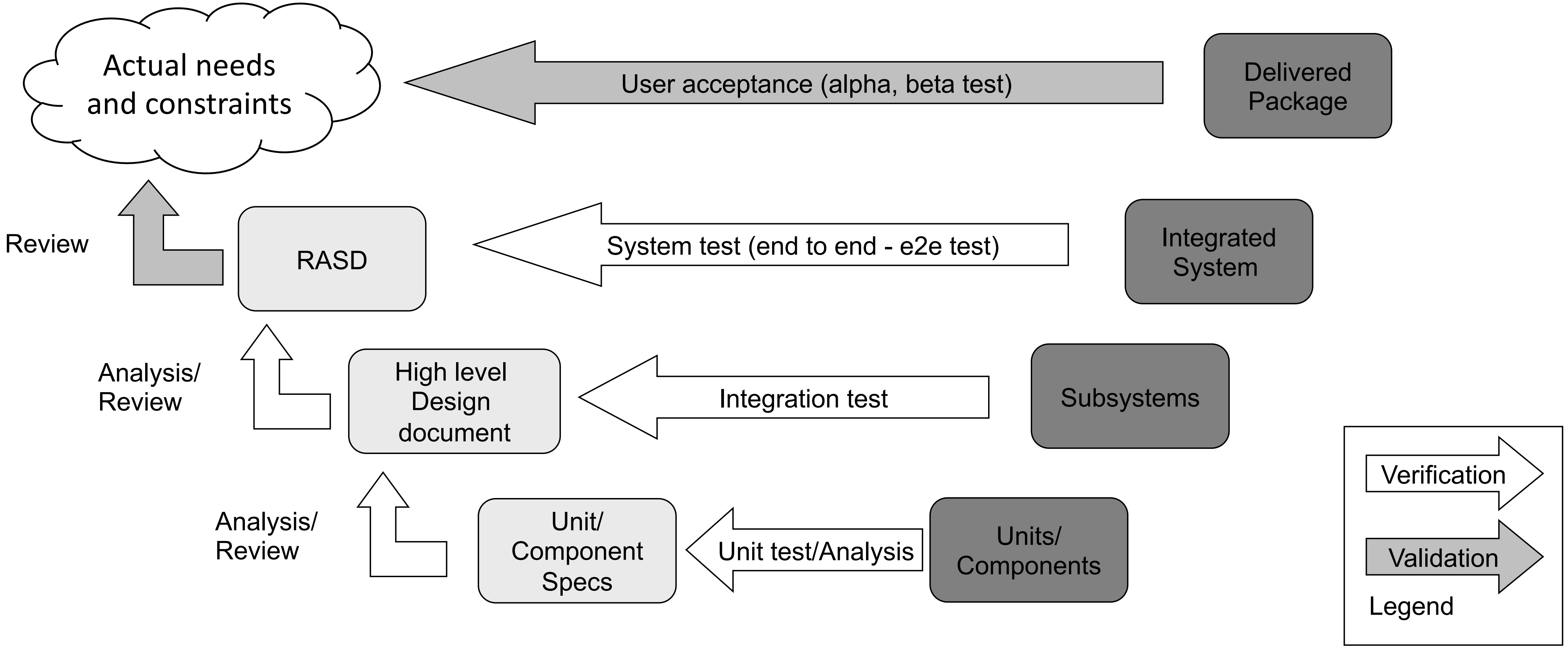
- Define policies and processes to achieve **quality**
- Assess quality and find **defects** (through V&V techniques)
- Improve quality

- Quality = general term, may refer to
  - Absence of defects (or bugs)
  - Absence of other issues that prevent the fulfilment of non-functional requirements or the degradation of some software qualities
    - external qualities (e.g., performance, availability) or internal ones (e.g., maintainability)

# QA challenges

- **Zero defect** software practically impossible to achieve, so…
  - Careful and continuous QA needed
  - Ideally, every artifact shall be subject of QA (spec documents, design documents, test data, …)
    - even the verification artifacts must be verified!
- QA along the entire development process, not just at the end
- … in this course, focus on **verification** and not on validation

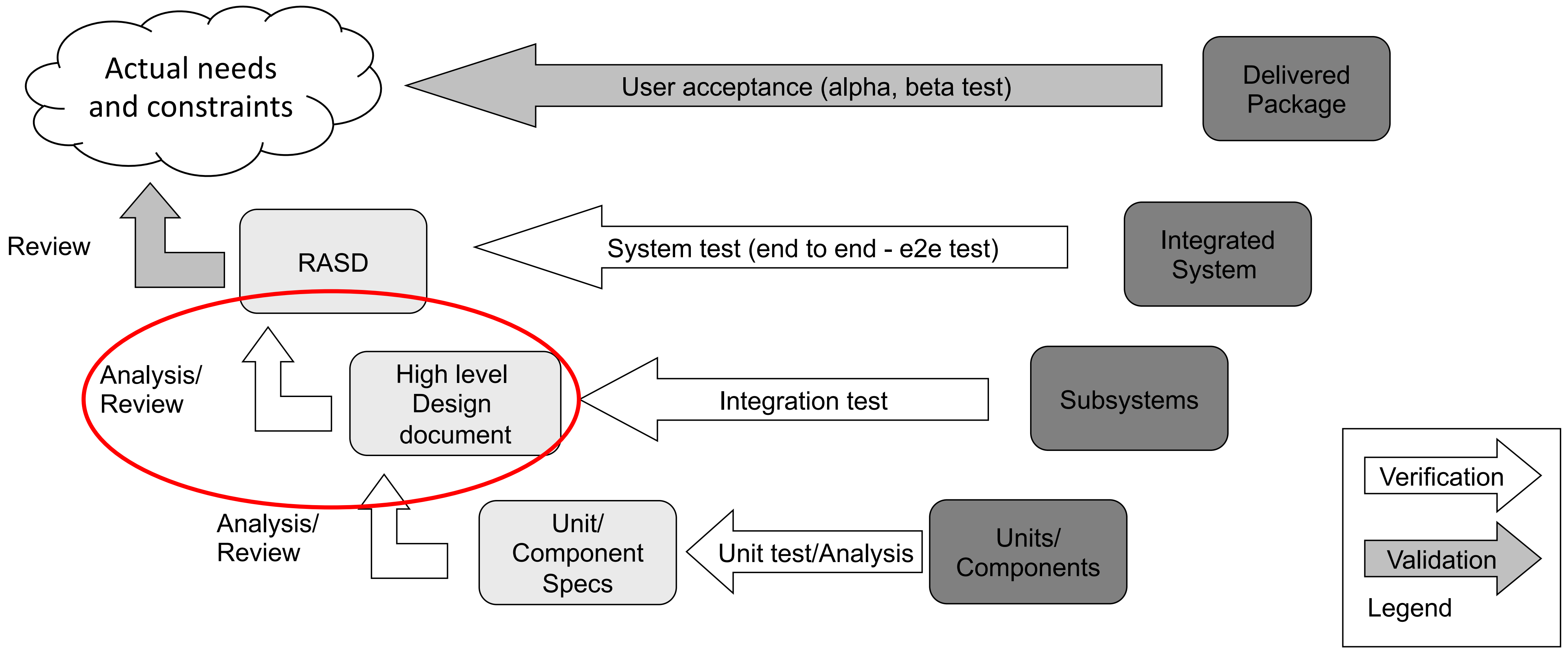
## Verification at which level? (V model)



## Main approaches: static vs dynamic analysis

- Static Analysis
  - Done on source code or on other software artifacts, without execution
  - Analysis is static but properties are dynamic
- Testing (dynamic analysis)
  - Done by executing the sources (usually by sampling)
  - Analysis of the actual behavior compared to an expected one

# Verification and software architectures



# What to verify at the architectural level?

- Structural consistency, e.g.:
  - For every required interface a corresponding provided interface exists
  - Sequence diagrams are consistent with component diagrams and with the defined interfaces
  - For each component there is one or more modules that implement it
- All functional requirements can be fulfilled, e.g.:
  - Each requirement is mapped on one or more components
  - Each use case event flow is detailed in terms of one or more sequence diagrams
- Concurrent use of resources is correctly defined (see next)
- Non-functional requirements can be fulfilled (see next)

## Concurrent use of resources

- Possible problems [Lu et al 2008]
  - Non-deadlock, e.g.
    - Atomicity violation
    - Order violation
  - Deadlock
- How to study these problems at the architectural level?
  - Define and analyse suitable concurrency models

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII). Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

## Nets

- A triple N = (P, T, F)
  - P set of places
  - T set of transitions
  - F flow relation
  - P, T are finite
- Properties:

  (1) $P \cap T = \emptyset$

  (2) $P \cup T \neq \emptyset$

  (3) $F \subseteq (P \times T) \cup (T \times P)$

## Place-transition nets (Petri nets)

- P/T = (P, T, F, W, $M_0$)

  P, T, F like in nets

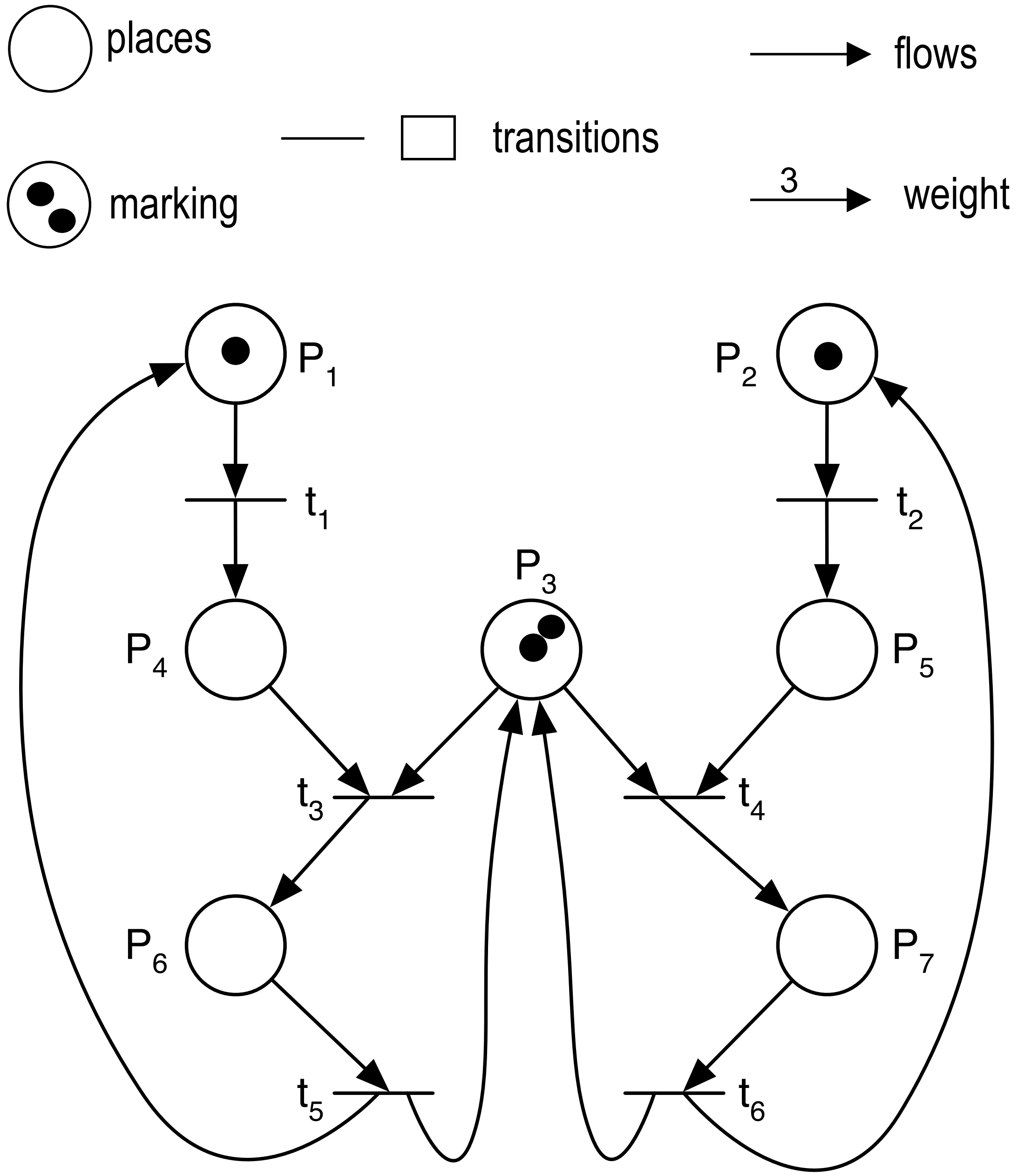
  (4) W: F $\rightarrow$ N-{0}

    - W = weight function
    - total function, defined for *all* elements of F
    - default value is 1

  - *marking* M: P $\rightarrow$ N

  (5) $M_0$: P $\rightarrow$ N

    - $M_0$ = initial marking
    - total function

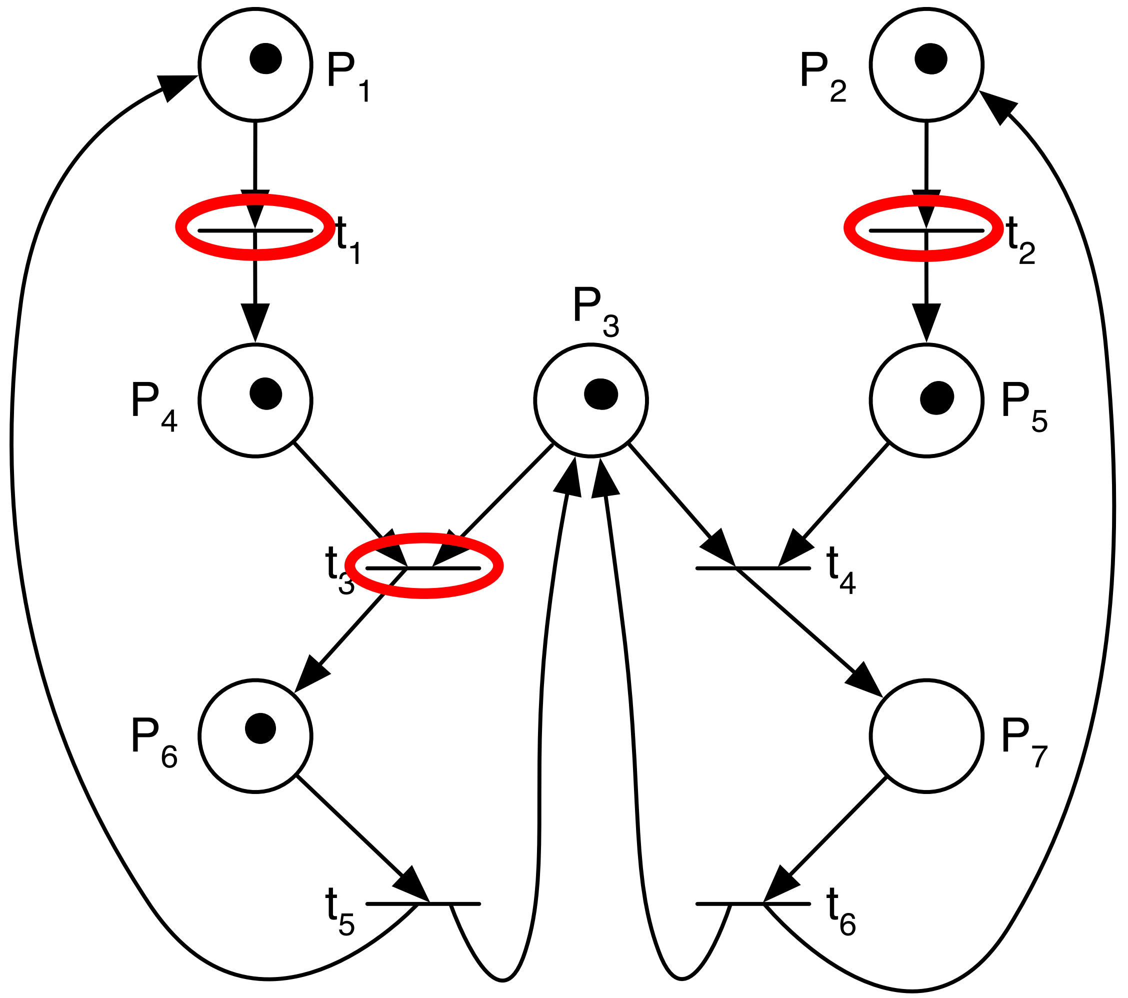

places
flows
transitions
3
weight
marking
P1
P2
t1
t2
P3
P4
P5
t3
t4
P6
P7
t5
t6

## Semantics: dynamic evolution

- State of a Petri Net = marking M
  - M(p) = marking of place p
- Given a transition t:
  - Pre(t) = input places of t
    - Pre(t) $\subseteq$ P
    - Pre(t) = {p $\in$ P | <p, t> $\in$ F}
  - Post(t) = output places of t
    - Post(t) $\subseteq$ P
    - Post(t) = {p $\in$ P | <t, p> $\in$ F}
- Given an element (arrow) <p, t> of F:
  - W(<p, t>) = weight of element <p, t>
    - Similarly for elements <t, p> of F

# Semantics: dynamic evolution (2)

- Transition t is *enabled* (in marking M):
  - $\forall p \in Pre(t), \quad M(p) \geq W(<p,t>)$
- t *fires*: produces a new marking (state) M'
  - $\forall p \in Pre(t) - Post(t)$: $M'(p) = M(p) - W(<p,t>)$
  - $\forall p \in Post(t) - Pre(t)$: $M'(p) = M(p) + W(<t,p>)$
  - $\forall p \in Post(t) \cap Pre(t)$: $M'(p) = M(p) - W(<p,t>) + W(<t,p>)$
  - $\forall p \in P - (Post(t) \cup Pre(t))$: $M'(p) = M(p)$

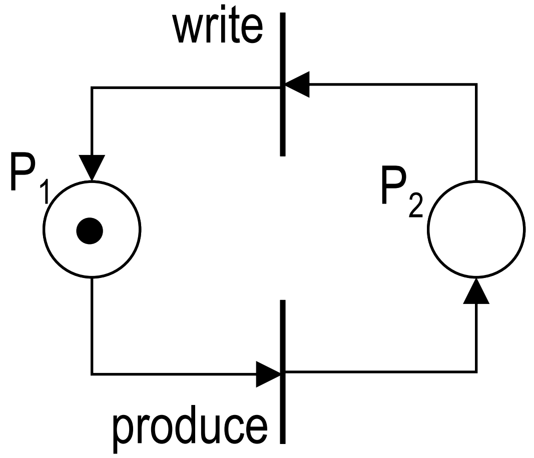


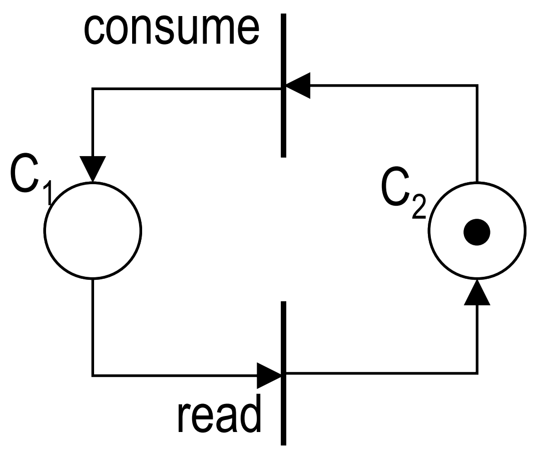
P1
P2
t1
t2
P3
P4
P5
t3
t4
P6
P7
t5
t6

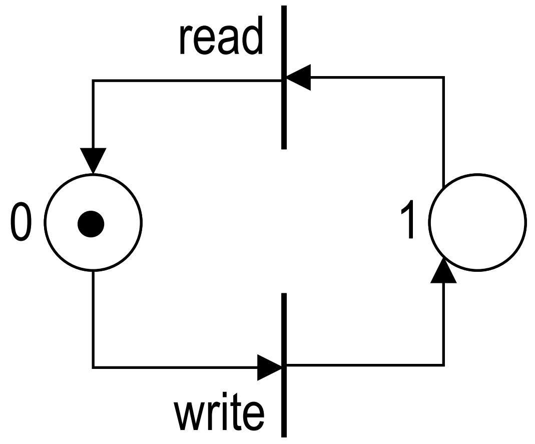
## Observations

- In a given marking M, a transition t can fire only if it is enabled
- An enabled transition not necessarily fires
- More than one transition can be enabled in a marking
- If two transitions are enabled at the same time:
  - which one fires first is not determined
  - Petri Nets are an intrinsically *nondeterministic model*
  - the firing of a transition might disable another enabled transition
- In fact, if two transitions are enabled at the same time:
  - they  can fire simultaneously …
  - … unless the firing of one transition disables the other
  - Petri Nets are suitable for modeling *concurrent systems*

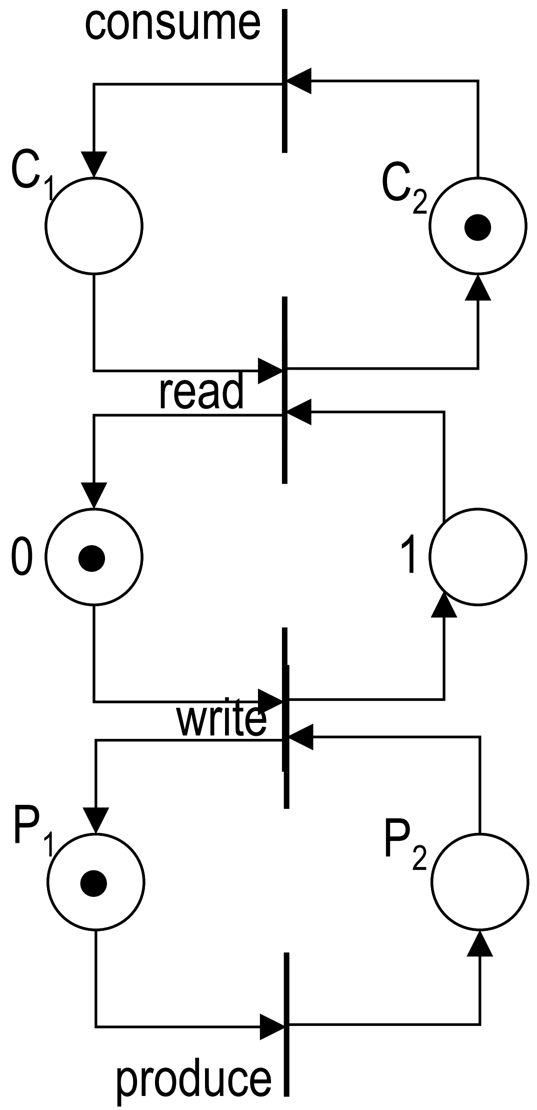
## Producer-consumer example





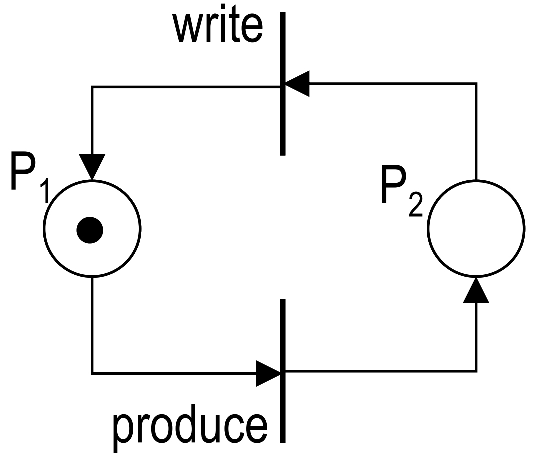


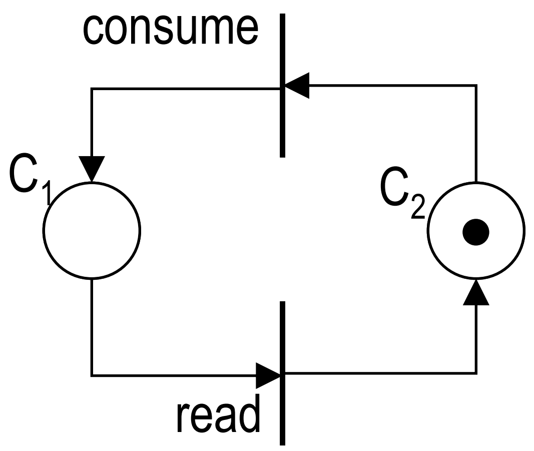
## Producer-consumer example

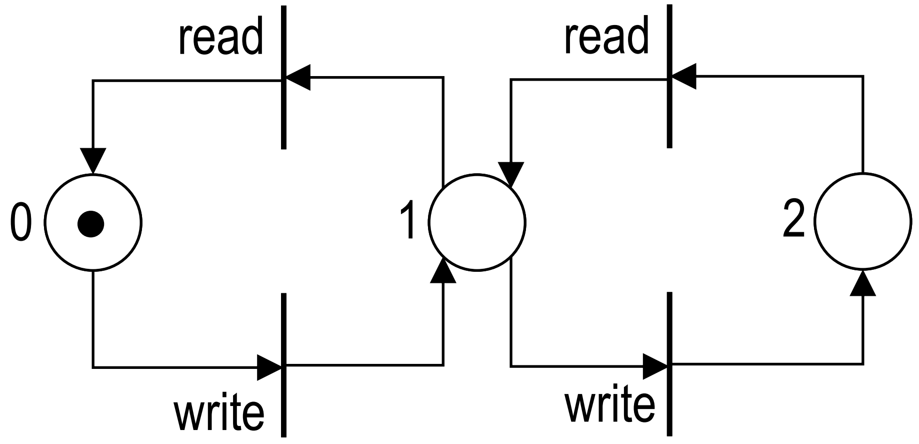


- Note: In this model we cannot have atomicity violations between the consumer and producer

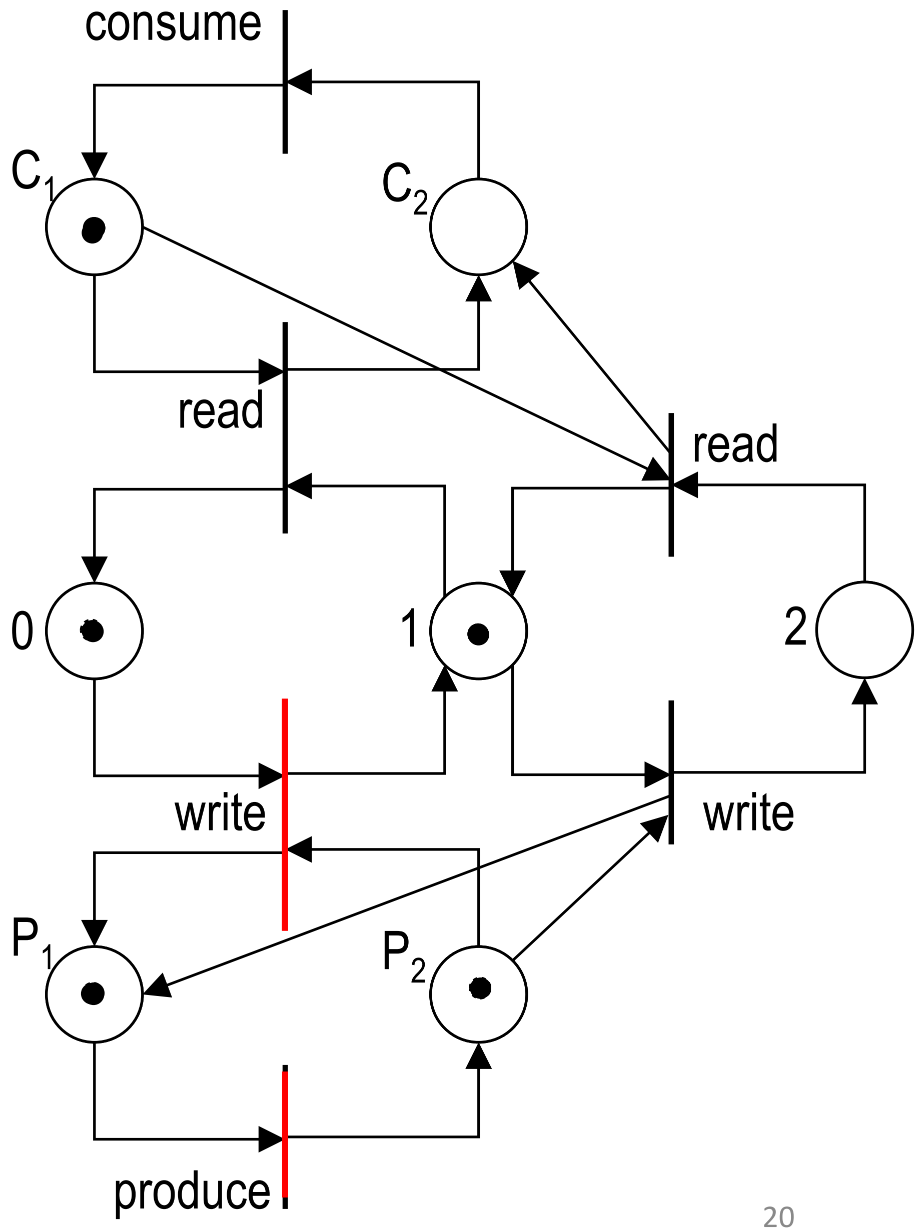
# Two positions buffer



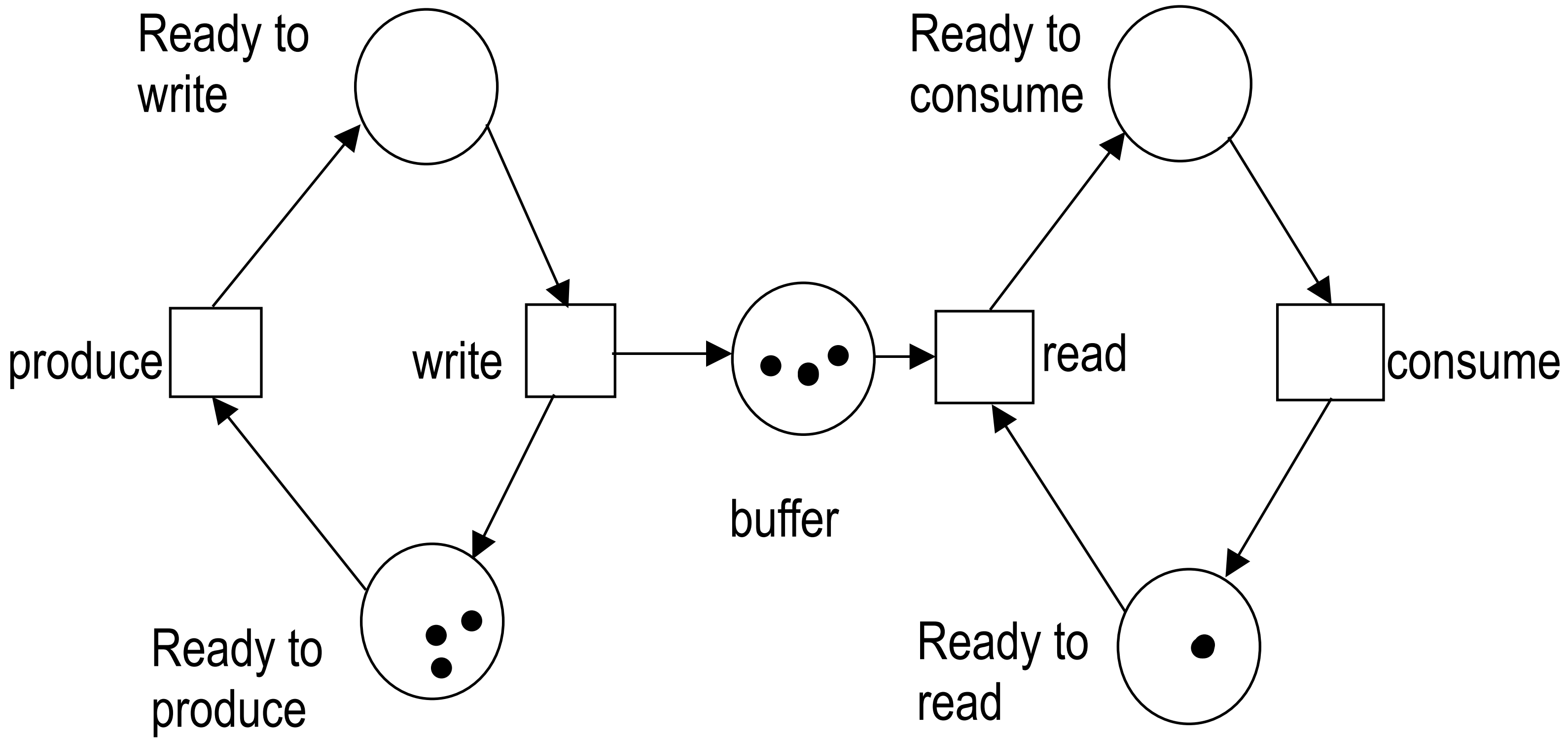




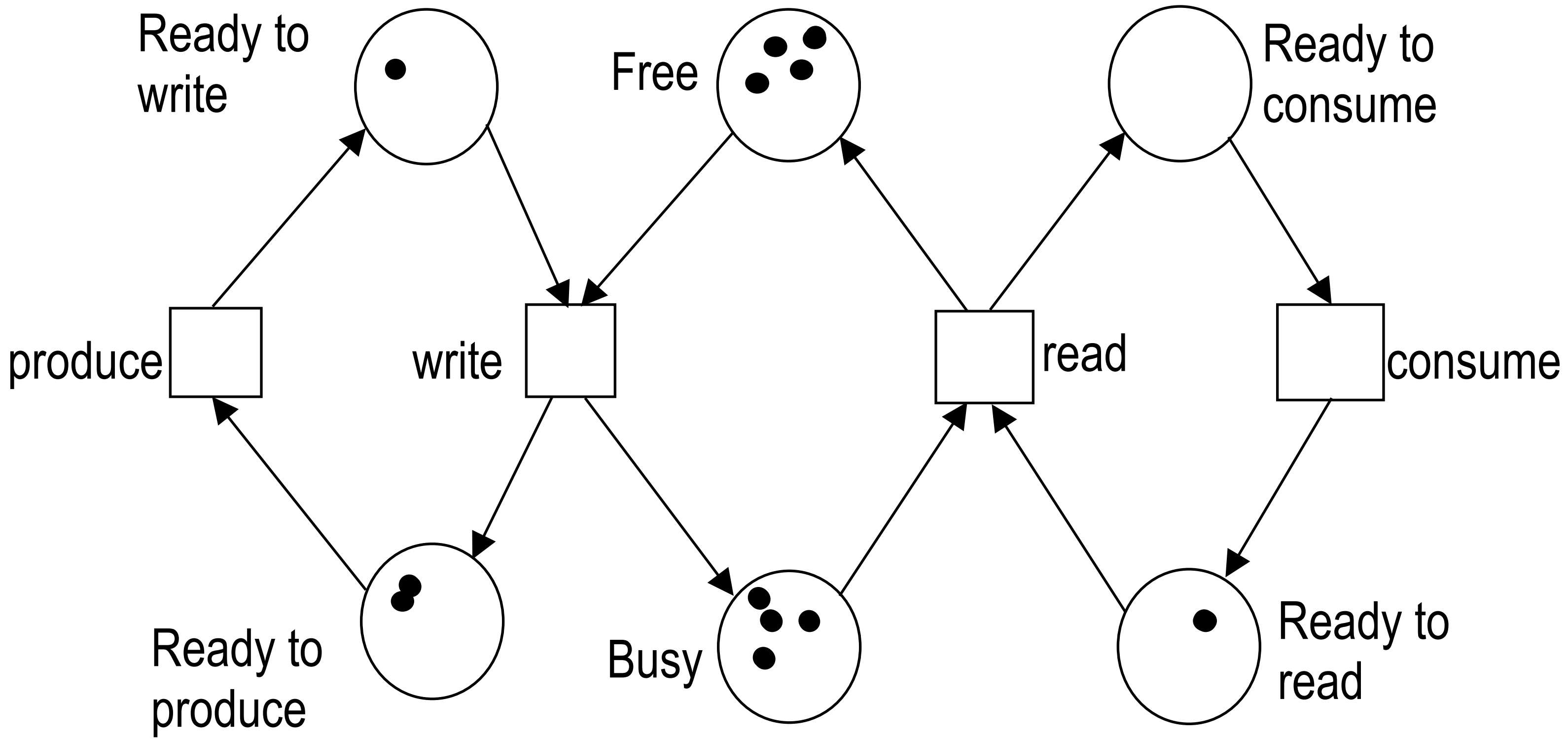
## Two positions buffer



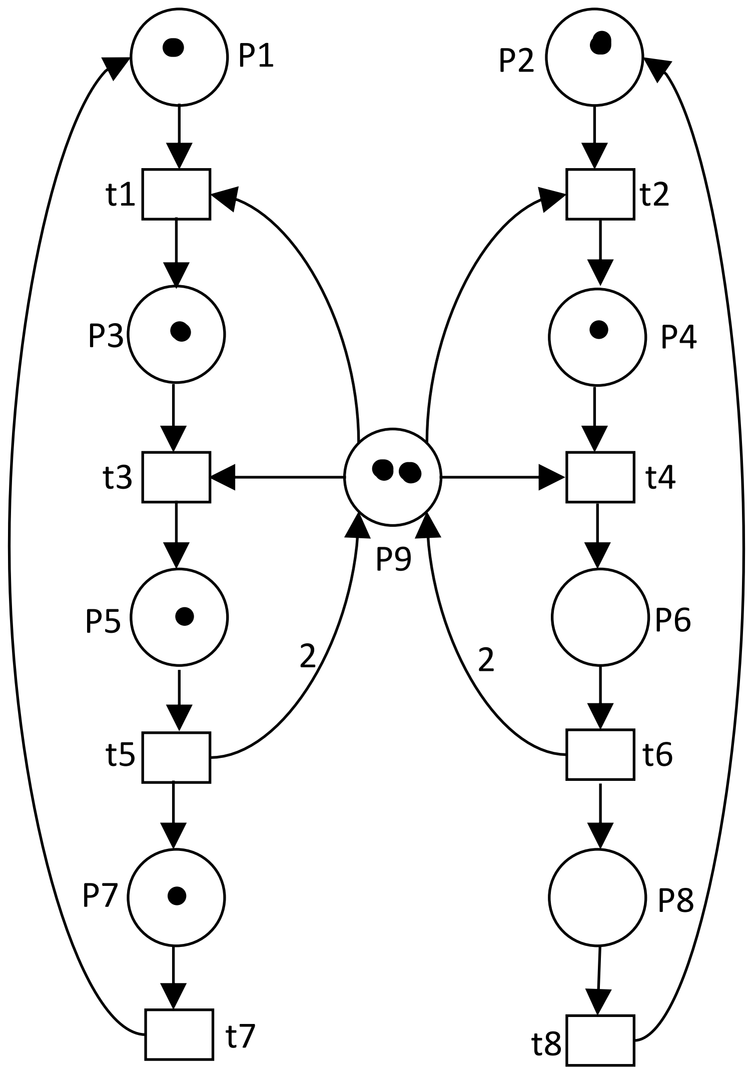
## Unbounded buffer



# Finite buffer with a parametric number of positions



## Deadlock



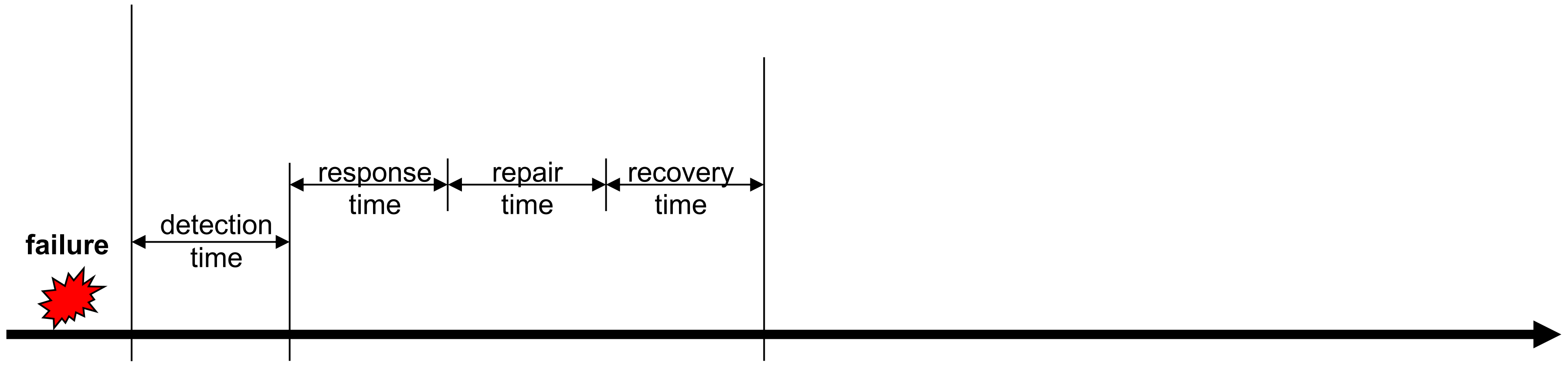
## Software qualities and architectures

- Several software qualities are directly influenced by architectural choices
  - Scalability
  - Reliability
  - Availability
  - Usability
  - …
- How do we cope with this?
  - We need **metrics** to quantify qualities and specific **methodologies to analyze** the quantitative impact of architectural choices on these qualities
  - **Tactics** to address the issues

# Availability

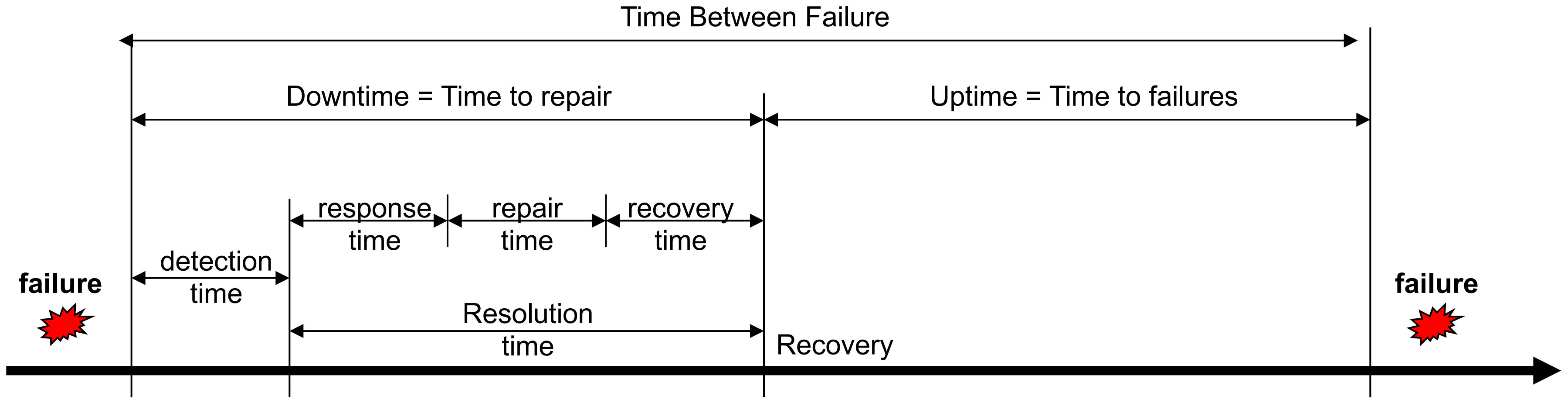
- A service shall be continuously available to the user
  - **If it fails, then little downtime** and **rapid** service **recovery**
- The availability of a service depends on:
  - Complexity of the IT **infrastructure** architecture
  - **Reliability** of the individual components
  - **Ability to respond** quickly and effectively to faults
  - **Quality of the maintenance** by support organizations and suppliers
  - **Quality** and scope of the **operational management** processes

# System life-cycle



- **Time of occurrence:** Time at which the user becomes aware of the failure
- **Detection time:** Time at which operators become aware of the failure
- **Response time:** Time required by operators to diagnose the issue and respond to users
- **Repair time:** Time required to fix the service/components that caused the failure
- **Recovery time:** Time required to restore the system (re-configuration, re-initialization,…)

# System life-cycle



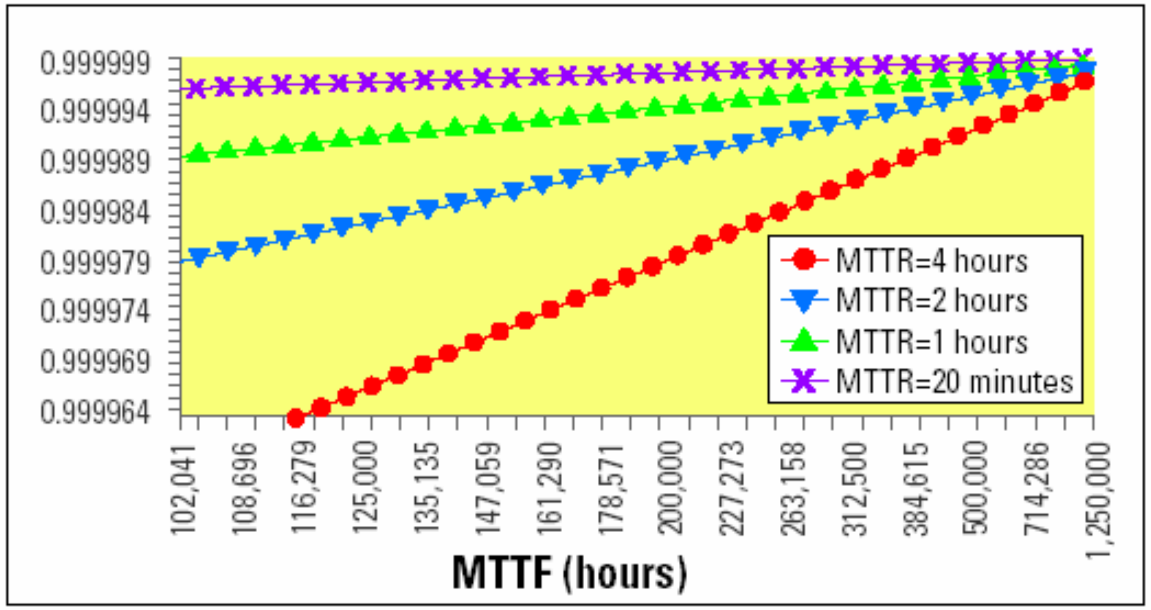
- **Mean Time to Repair (MTTR):** Average time between the occurrence of a failure and service recovery, also known as the downtime
- **Mean Time To Failures (MTTF):** Mean time between the recovery from one failure and the occurrence of the next failure, also known as uptime
- **Mean Time Between Failures (MTBF):** Mean time between the occurrences of two consecutive failures

# Availability metric — definition

- Probability that a component is **working properly** at **time *t***

- $A = \frac{MTTF}{MTTF+MTTR}$

- if MTTR small, MTBF ≅ MTTF



## Nines notation

- Availability is typically specified in **"nines notation"**
- For example, 3-nines availability corresponds to 99.9%, 5-nines availability corresponds to 99.999% availability

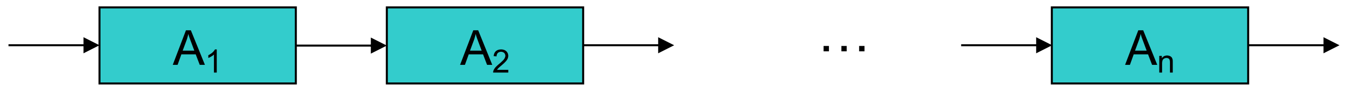
| Availability | Downtime |
|---|---|
| 90% (1-nine) | 36.5 days/year |
| 99% (2-nines) | 3.65 days/year |
| 99.9% (3-nines) | 8.76 hours/year |
| 99.99% (4-nines) | 52 minutes/year |
| 99.999% (5-nines) | 5 minutes/year |

# Availability — analysis methodology

- Availability is calculated by **modeling the system** as an interconnection of **elements in series** and **parallel**
- Elements operating in series
  - Failure of an element in the series leads to a failure of the whole combination
- Elements operating in parallel
  - Failure of an element leads to the other elements taking over the operations of the failed element

# Availability in series

- The combined system is operational only if every part is available
- The combined availability is the **product** of the availability of the component parts



$$A = \prod_{i=1}^{n} A_i$$

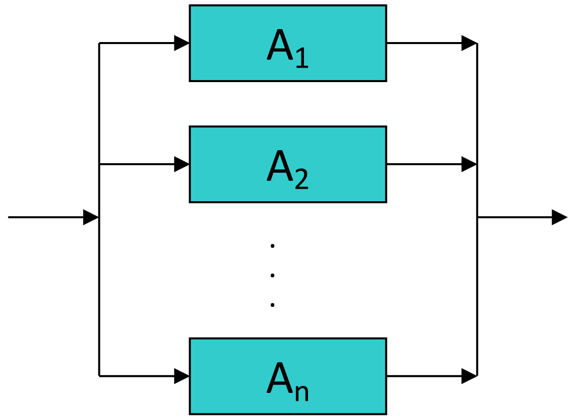
## Availability in series – A numerical example

| | Availability | Downtime |
|---|---|---|
| Component 1 | 99% (2-nines) | 3.65 days/year |
| Component 2 | 99.999% (5-nines) | 5 minutes/year |
| Combined | **98.999%** | **3.65 days/year** |

- **Downtime = (1-A)*365 days/year**
- The availability of the entire system is negatively affected by the low availability of Component 1
- **A chain is as strong as the weakest link!**

# Availability in parallel

- The combined system is operational if at least one part is available
- The combined availability is **1 - P(all parts are not available)**



$$A = 1 - \prod_{i=1}^{n}(1 - A_i)$$

## Availability in parallel – A numerical example

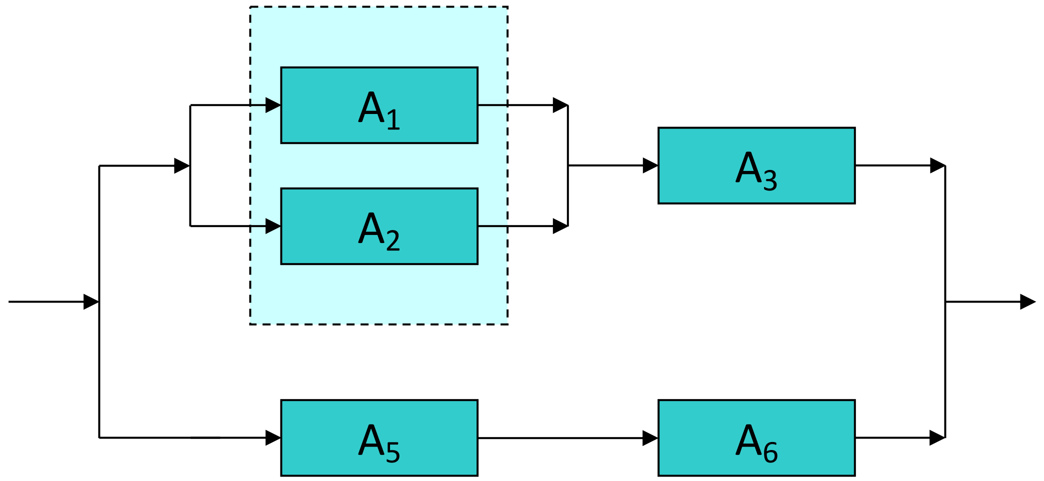
| | Availability | Downtime |
|---|---|---|
| **Component 1** | 99% (2-nines) | 3.65 days/year |
| **Component 2** | 99% (2-nines) | 3.65 days/year |
| **Combined** | **99.99% (4-nines)** | **52 minutes/year** |

- Downtime = (1-A)*365 days/year
- Even though components with very low availability are used, the overall availability of the system is much higher

- **Mission critical systems are designed with redundant components!**

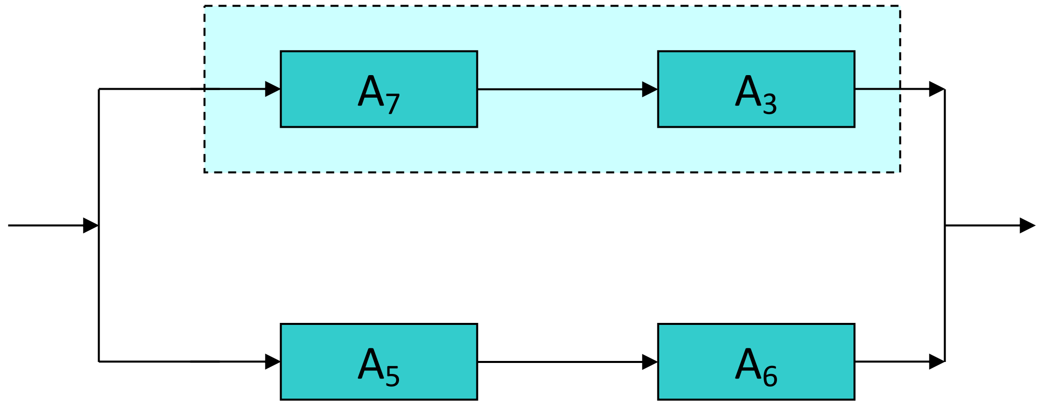
# Availability of complex systems

$$A_7=1-(1-A_1)(1-A_2)$$

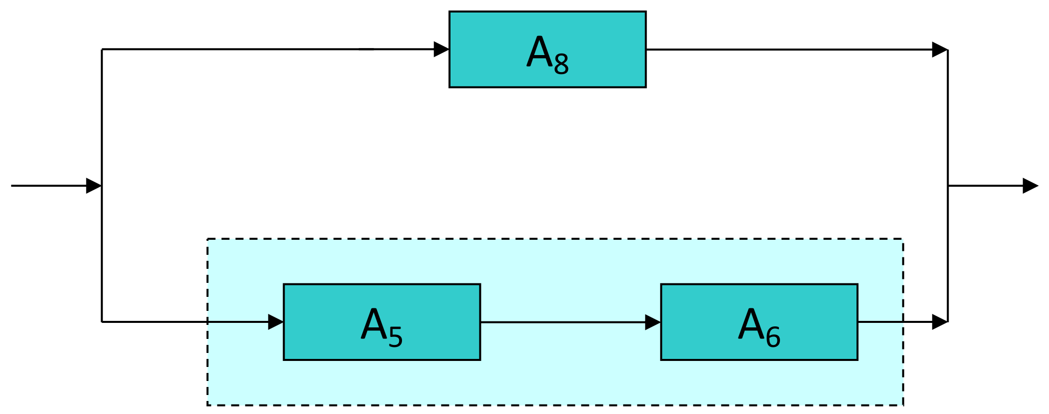


# Availability of complex systems

$$A_8 = A_7 A_3$$



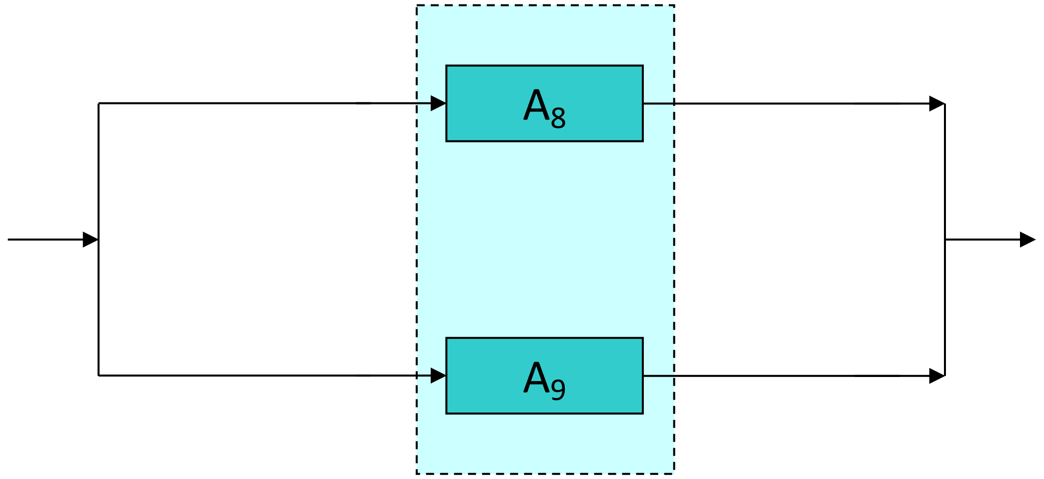
## Availability of complex systems



$A_9 = A_5 A_6$

## Availability of complex systems

$$A=1-(1-A_8)(1-A_9)$$

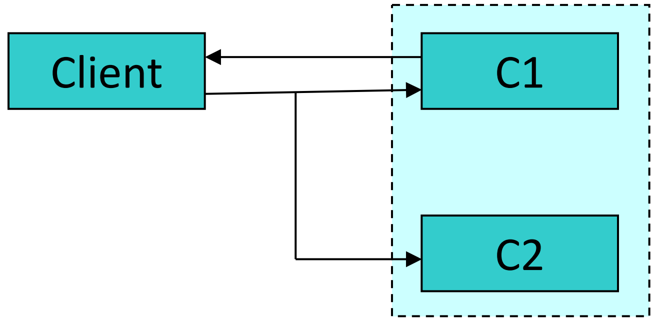


# Tactics for Availability

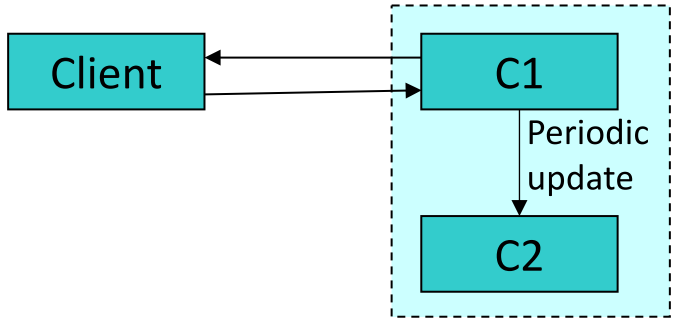
- **Tactic** = Design decisions that influence the control of one or more quality attributes

- **Replication**
  - Very simple to manage in case of stateless components
  - Various strategies in case of statefull components
- **Forward error recovery**
- **Circuit breaker**

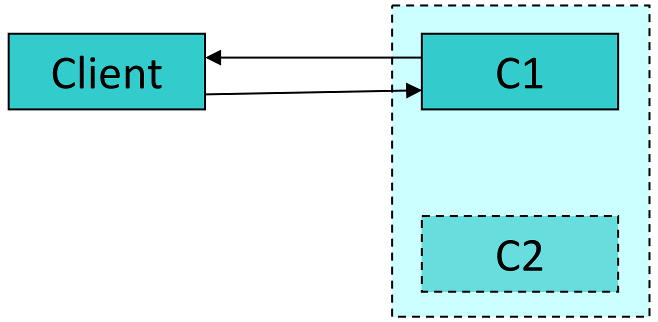
# Replication approaches



Hot spare: C1 is leading, C2 is always ready to take over

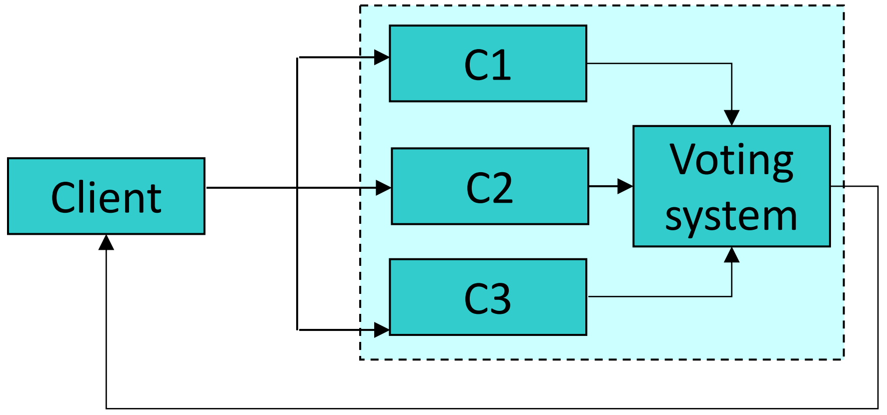


Warm spare: C1 is leading and periodically updating C2.
If C1 fails, some time might be needed to fully update C2

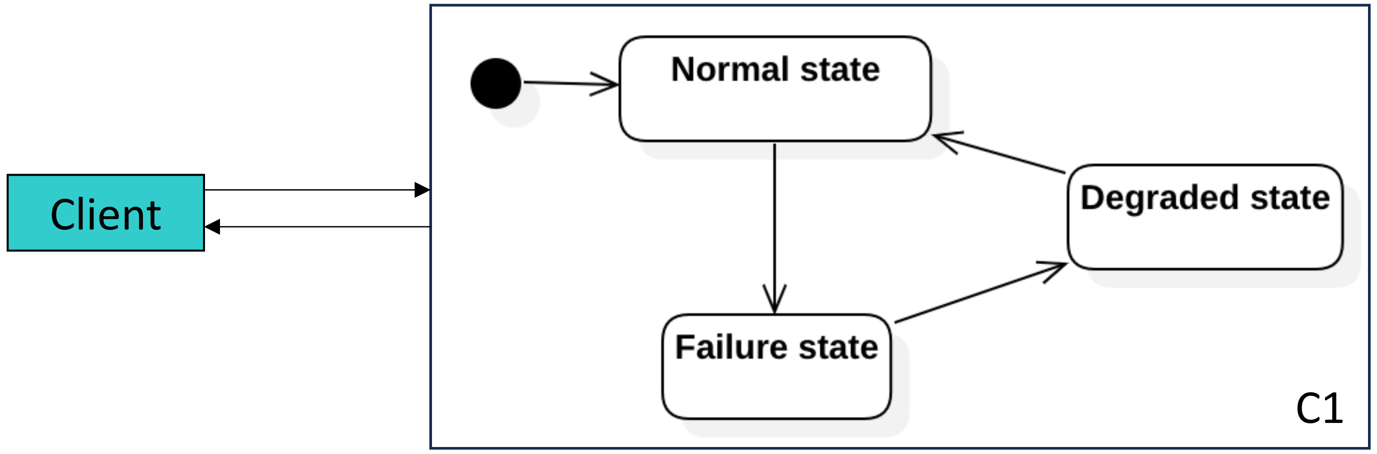


Cold spare: C2 is dormant and started and updated only when needed

Triple modular redundancy: C1, C2, and C3 are all active.
The produced result is the one produced by the majority.
Good when reliability is also important



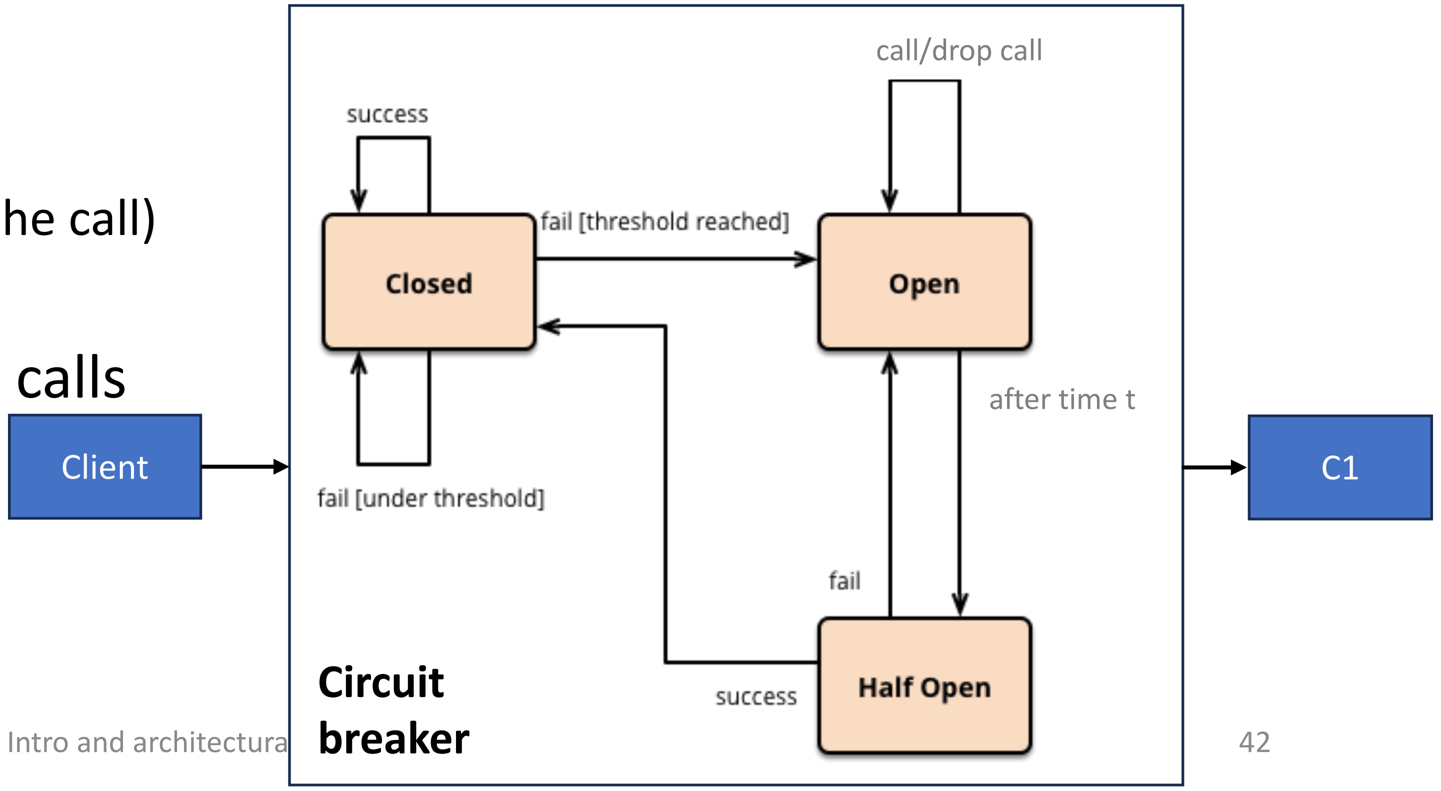
# Forward error recovery



- When C1 is in the failure state, a recovery mechanism moves it to the degraded state
- In the degraded state, C1 continues to be available even if not fully functional

# Circuit breaker

- Circuit breaker (CB) — **Client-side** resiliency pattern
  - CB acts as a **proxy** for a remote component
  - When C1 is called, the CB **monitors** the call
  - **Possible failures**:
    - CB receives an error
    - Call takes «too long» (CB kills the call)
  - **«too many» failures →** circuit breaker inhibits future calls by moving to the Open state

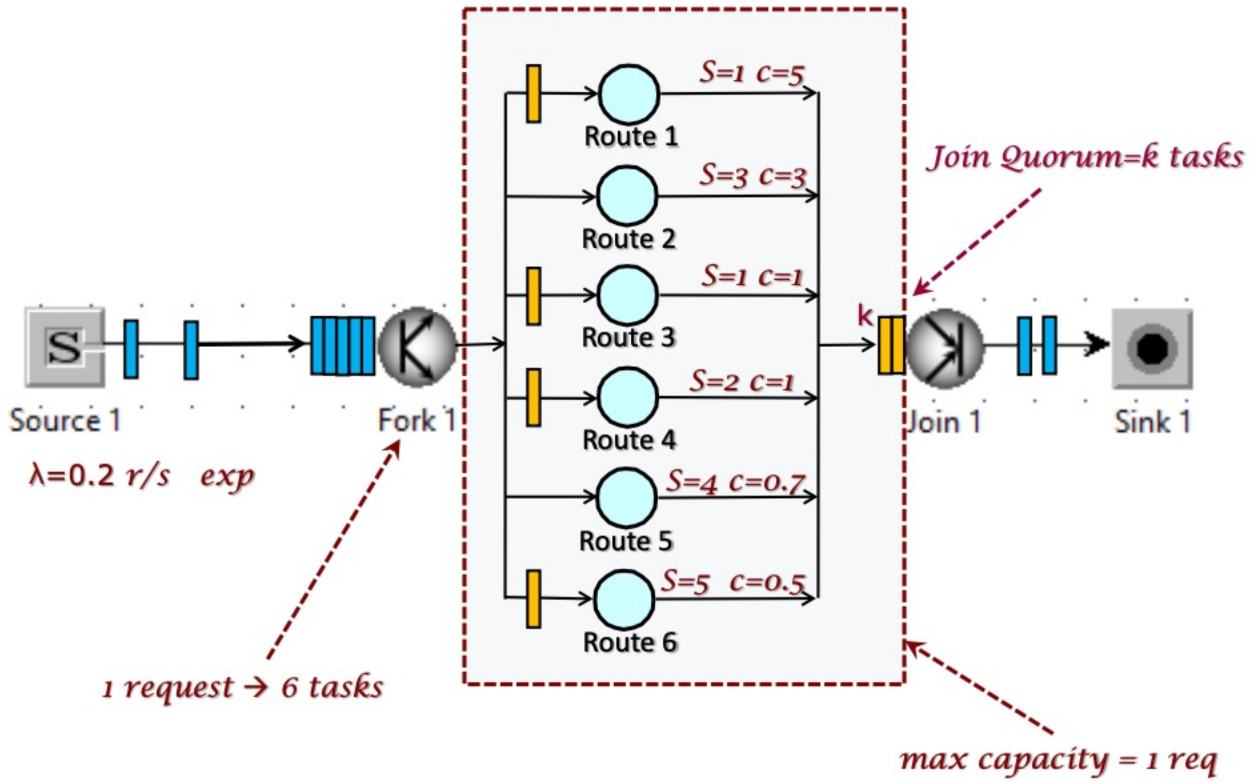


# Performance

- *Performance is an indicator of how well a software system or component meets its requirements for timeliness.*
  - Connie U. Smith, Lloyd G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, 2001, Addison-Wesley Professional.
- Also defined as efficient use of resources
- Performance is connected to scalability: ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases
- Multiple metrics available:
  - Response time
  - Throughput
  - CPU utilization
  - Memory utilization
  - I/O operations

# Performance analysis

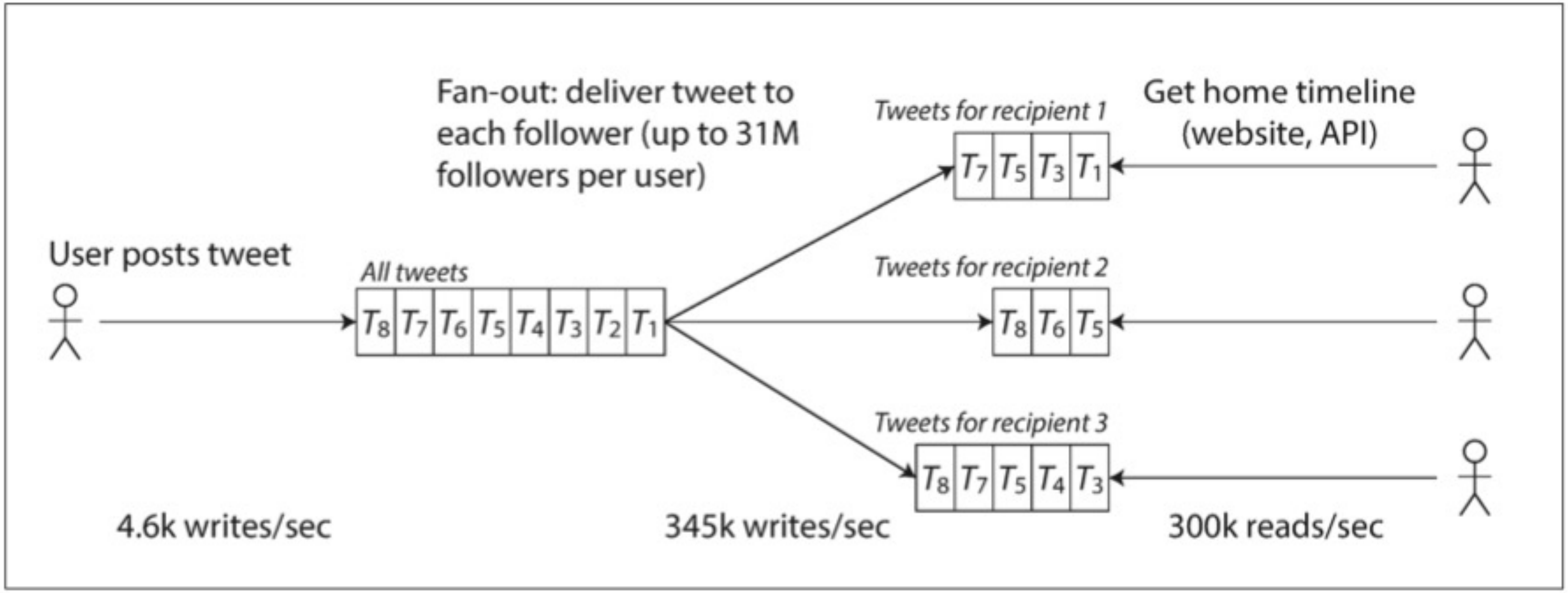
- Queuing networks
  - A queuing networks simulator: JMT [https://jmt.sourceforge.net/](https://jmt.sourceforge.net/)



## Tactics for performance

- Control resource demand
  - Control input
    - Manage event arrival
    - Manage sampling rate
    - Bound event queues' size
    - Prioritize events
  - Improve efficiency of software
    - Reduce indirection
    - Co-locate interacting resources
    - Bound execution time
    - Improve algorithm efficiency
- Manage resources including computation and data
  - Increase resources
  - Introduce concurrency
  - Add multiple replicas and a load balancer
  - Add data replication and/or caching
  - Schedule resources
  - Split input and handle  it

# Data replication: do you remember the Twitter example?


Fan-out: deliver tweet to each follower (up to 31M followers per user)
Tweets for recipient 1
Get home timeline (website, API)
$T_7$ $T_5$ $T_3$ $T_1$
User posts tweet
All tweets
$T_8$ $T_7$ $T_6$ $T_5$ $T_4$ $T_3$ $T_2$ $T_1$
Tweets for recipient 2
$T_8$ $T_6$ $T_5$
Tweets for recipient 3
$T_8$ $T_7$ $T_5$ $T_4$ $T_3$
4.6k writes/sec
345k writes/sec
300k reads/sec

Martin Kleppmann, Designing Data-Intensive Applications : The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, O'Reilly Media, Incorporated, Ed: 2017, ISBN: 9781449373320