Architectural Tactics & Analysis

Truong Ho-Quang
truongh@chalmers.se

Schedule

Week		Date	Time	Lecture	Note
3	L1	Wed, 20 Jan	10:15 – 12:00	Introduction & Organization	Truong Ho
3	L2	Thu, 21 Jan	13:15 – 15:00	Architecting Process & Views	Truong Ho
4		Tue, 26 Jan	10:15 – 12:00	**Skip**	
4	S1	Wed, 27 Jan	10:15 – 12:00	<< Supervision: Launch Assignment 1>>	TAs
4	L3	Thu, 28 Jan	13:15 - 15:00	Roles/Responsibilities & Functional Dec	Ho
5	L4	Mon, 1 Feb	13:15 – 15:00	Architectural Styles P1	Ho
5	S2	Wed, 3 Jan	10:15 – 12:00	<< Supervision/As	
5	L5	Thu, 4 Jan	13:15 – 15:00	Architectural Styles P2	para
6	L6	Mon, 8 Feb	13:15 – 15:00	Architectural Styles P3	Ho
6	S3	Wed, 10 Feb	10:15 – 12:00	<< Supervis n/Assignment>>	TAs
6	L7	Thu, 11 Feb	13:15 – 15:00	Design Principles (Maintainabil y, Modifiability)	Truong Ho
7	L8	Mon, 15 Feb	13:15 – 15:00	Architectural Tactics & Analysis	Truong Ho
7	S4	Wed, 17 Feb	10:15 – 12:00	<< Supervision/Assignment>>	TAs
7	L9	Thu, 18 Feb	13:15 – 15:00	Architecture Evaluation	Truong Ho
8	L10	Mon, 22 Feb	13:15 – 15:00	Reverse Engineering & Correspondence	Truong Ho
8	S5	Wed, 24 Feb	10:15 – 12:00	<< Supervision/Assignment>>	TAs
8	L11	Thu, 25 Feb	13:15 – 15:00	Guest Lecture 1	TBD
9	L12	Mon, 1 Mar	13:15 – 15:00	Guest Lecture 2: Architectural Changes in Volvo AB	Anders M.
9	S6	Wed, 3 Mar	10:15 – 12:00	<< Supervision/Assignment>>	TAs
9	L13	Thu, 4 Mar	13:15 – 15:00	To be determined (exam practice?)	Truong Ho
9		Fri, 5 Mar	Whole day	Group presentation of Assignment (TBD)	Teachers
11	Exam				

We are HERE!

Before Lecture

- Peer-Review of A1T1 has started
- Student representatives to be announced today

Outline / Contents

- (Prerequisite) Quality Attribute Scenario
- Architectural Tactics
- Analysis
  - Reliability
  - Performance

Quality Attribute Scenario (QAS)*

QAS appears to solve the untestable and overlapping concerns.

The aim of a QAS is to capture the explicit and testable quality requirements

It does it in the same way the use case scenarios do for functional requirements by initiating a use case instant.

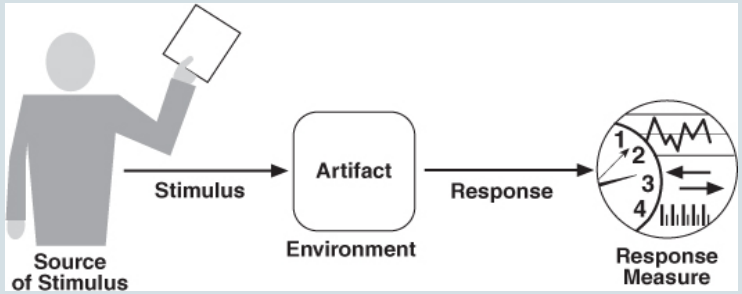
QAS consists of six parts.

* “Software Architecture for Business”, by Lina Khalid, ISBN 978-3-030-13631-4 © Springer Nature Switzerland AG 2020

QA Scenarios

We specify quality attribute requirements, we capture them formally as six parts of QAS:

1. *Source of stimulus.* (a human, or any other actuator) that generated the stimulus.
2. *Stimulus.* A condition that requires a response. For different quality it means something specific.
3. *Environment.* The system may be in an overload condition, test, or in normal operation.
4. *Artifact.* Some artifact is stimulated. This may be a collection or whole system, or pieces of it.
5. *Response.* The response is the activity undertaken as the result of the arrival of the stimulus.
6. *Response measure.* A response should be measurable so that the requirement can be tested.



Parts of a quality attribute scenario (ex. web portal responsiveness).

Quality Attribute Workshop (QAW)

Quality Attribute Workshop is a facilitated method for a few-days workshop.
It connects stakeholders in the early part of the life cycle in order to find quality attributes for the existing system.

The important thing to know about QAW is that:

- It is focused on the stakeholders.
- It is scenario based.
- It is used before the software architecture begins.
- It is focused on the system level concerns and on the role of software in the system.

Architectural Tactics

Tactics

A tactic is a construction pattern that influences the achievement of a quality attribute response

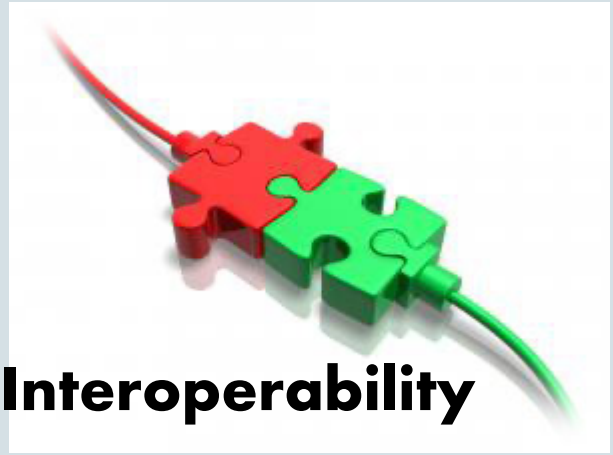
- Different tactics for each quality attribute
- The same tactic could be relevant to many quality attributes

- Could be seen as an ’add-on’/refinement of architectural styles

- Tactics do not change the functionality of the system hence they exist in the ’implementation-refinement’-layers of describing a system.

Example Quality Attributes

Availability



Interoperability

Performance

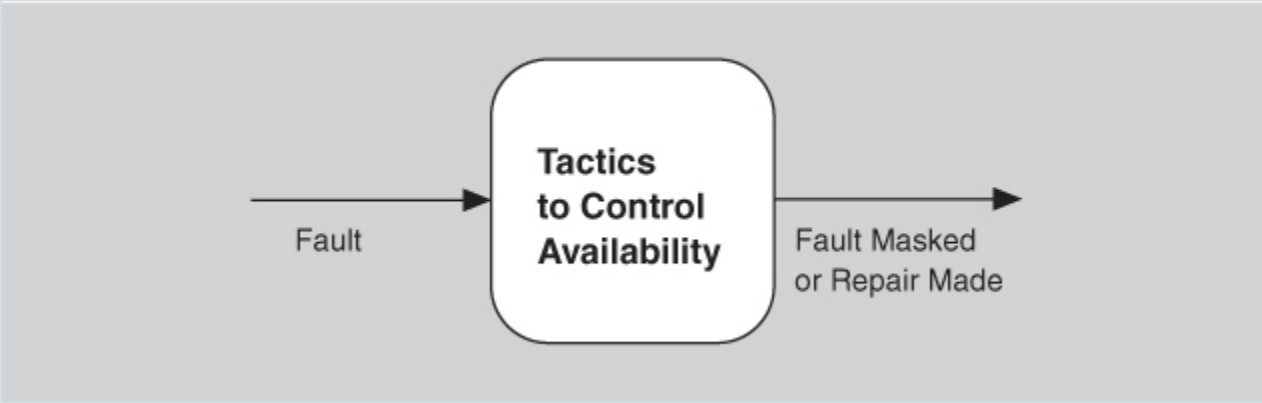
Modifiability

Security

Availability

- **Definition**:

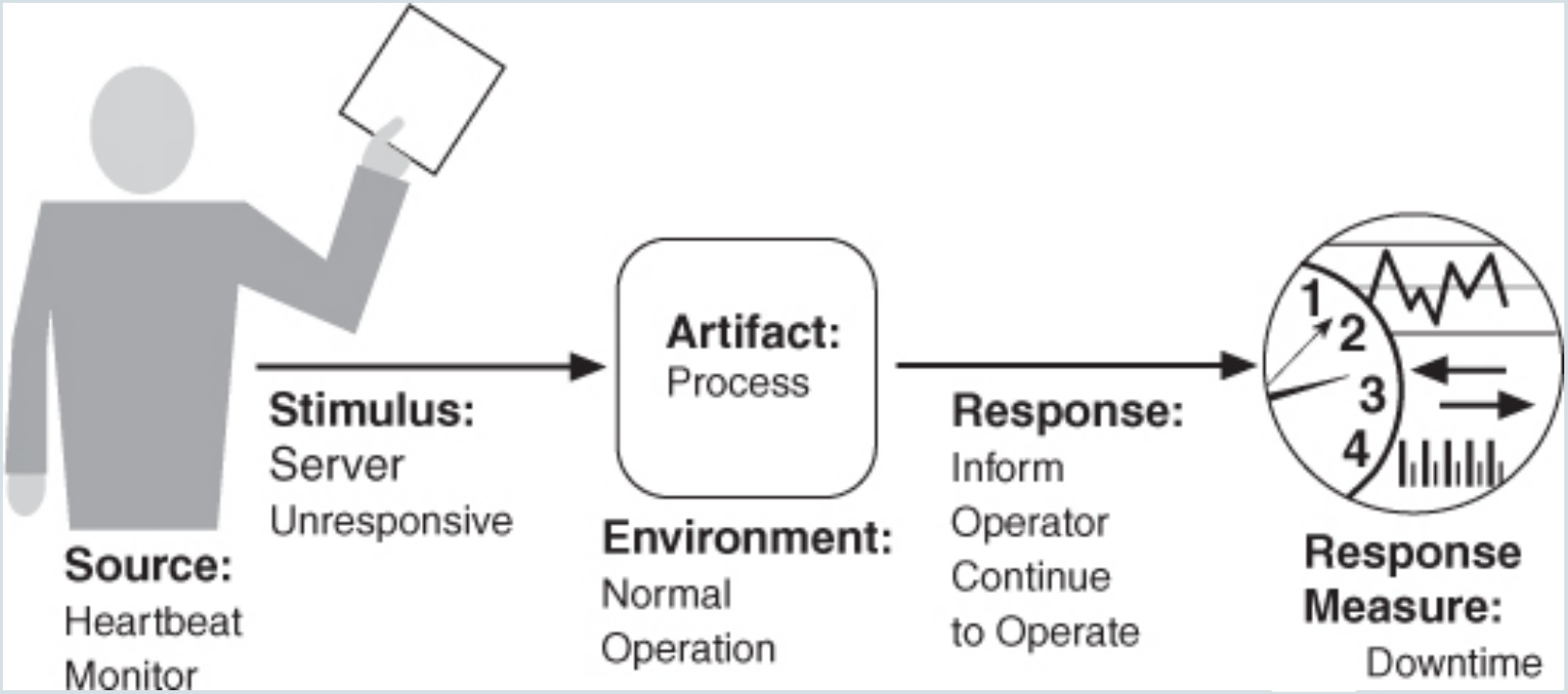
  The ability of a system to mask or repair faults (such as the cumulative service outage period does not exceed a required value over a specified time interval).



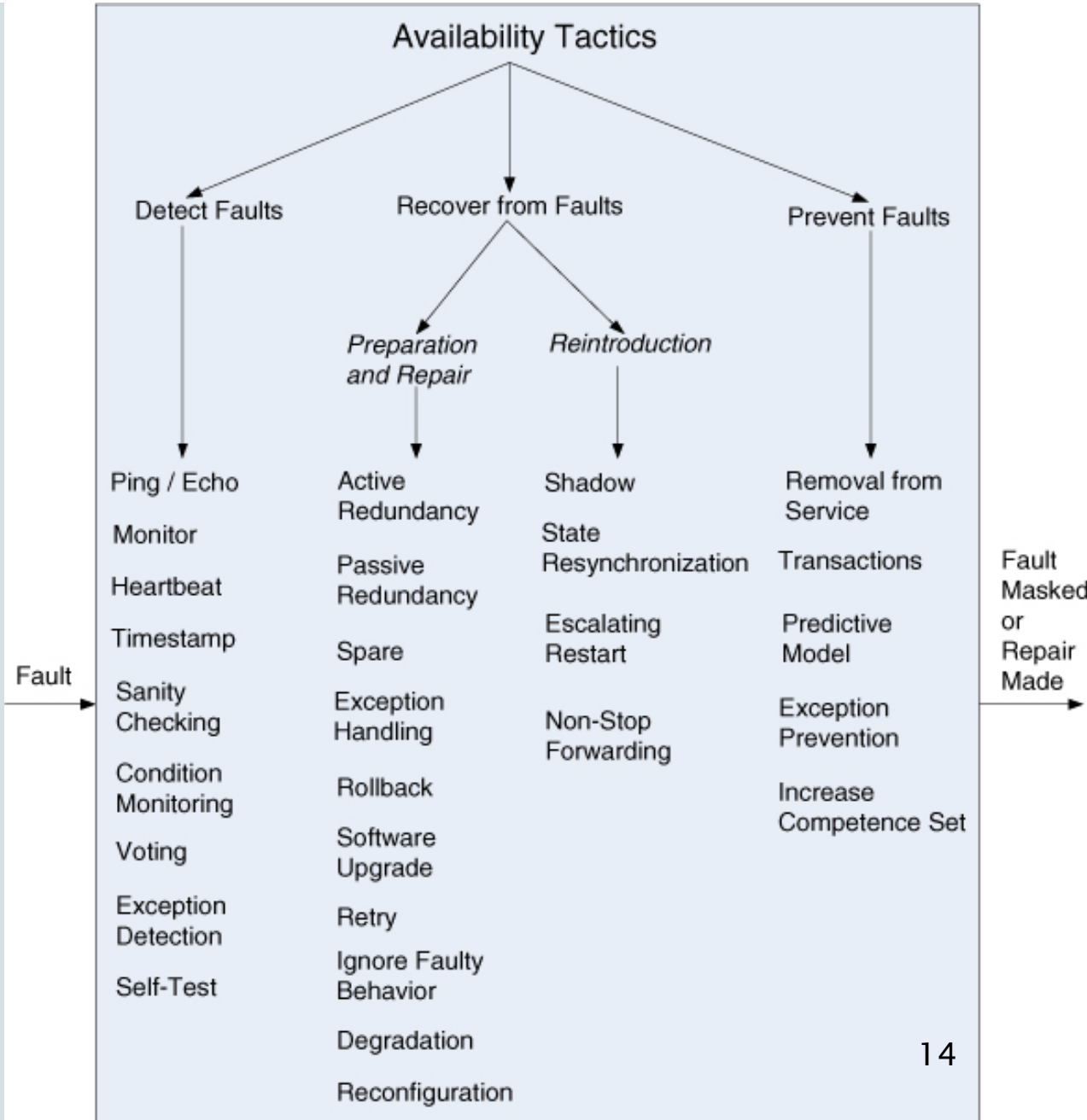
Availability QA-Scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: ▪ Log the fault ▪ Notify appropriate entities (people or systems) Recover from the fault: ▪ Disable source of events causing the fault ▪ Be temporarily unavailable while repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g., 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

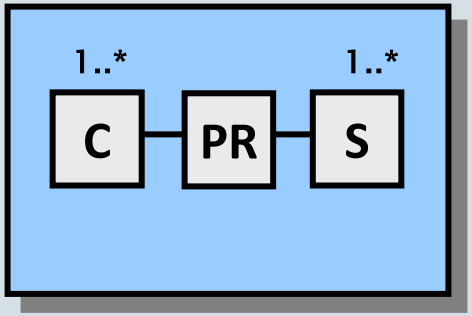
Concrete Availability Scenario

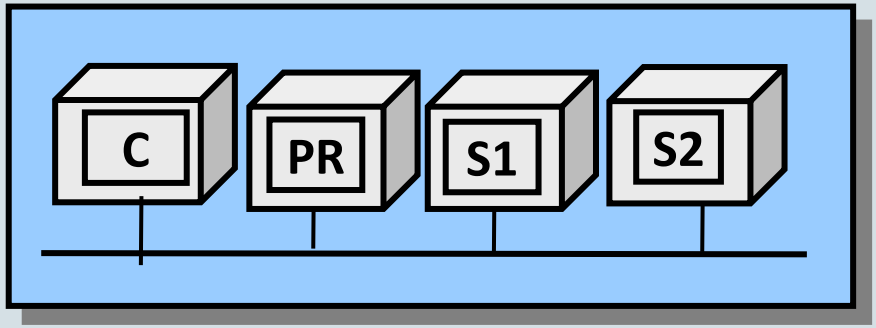


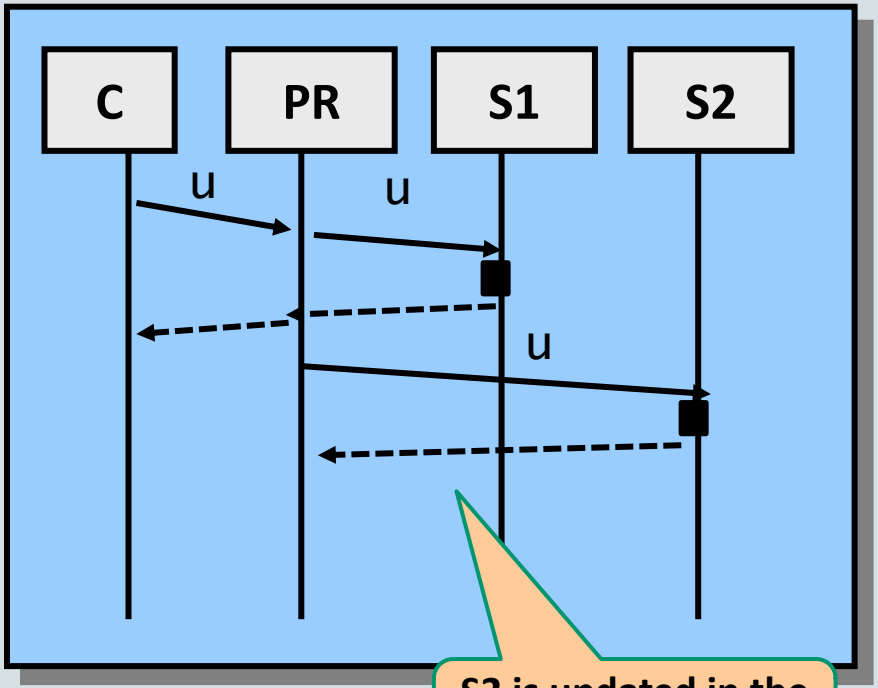
Availability Tactics

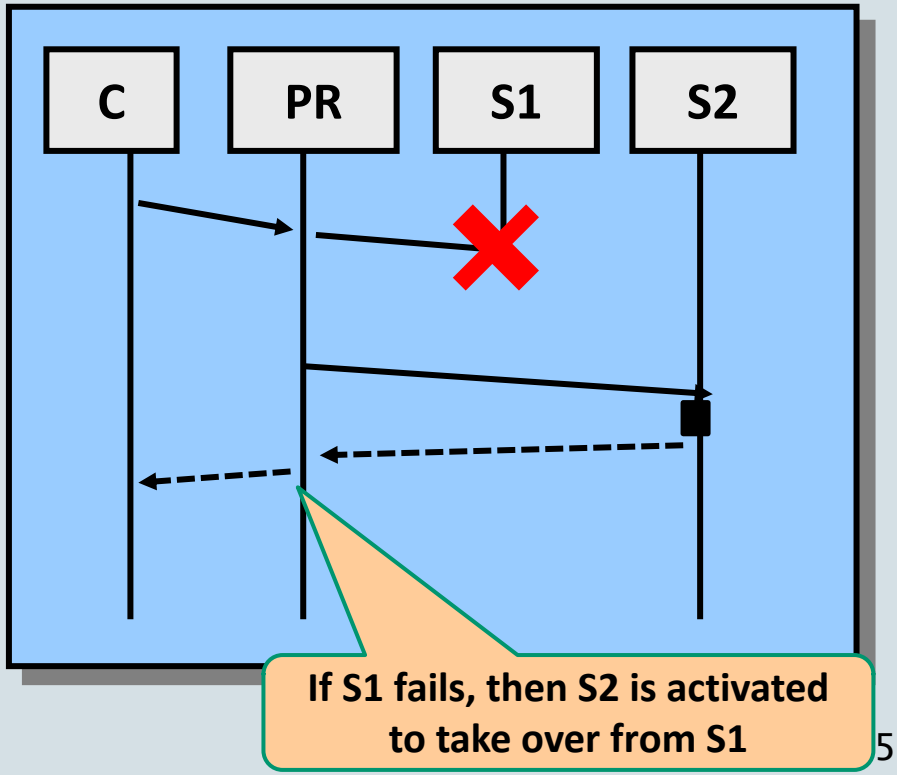


Client-Server Passive Redundancy

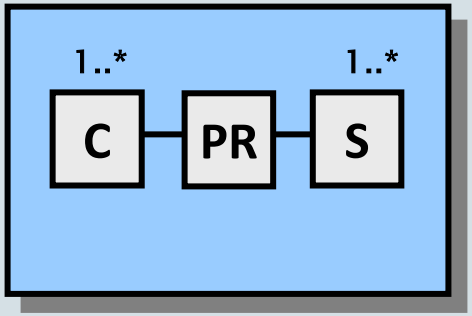


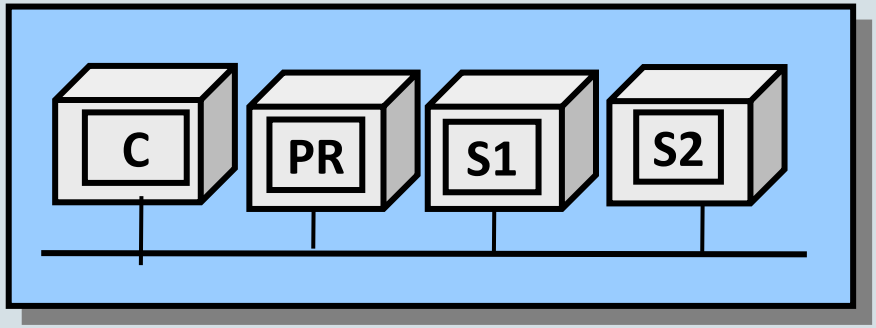


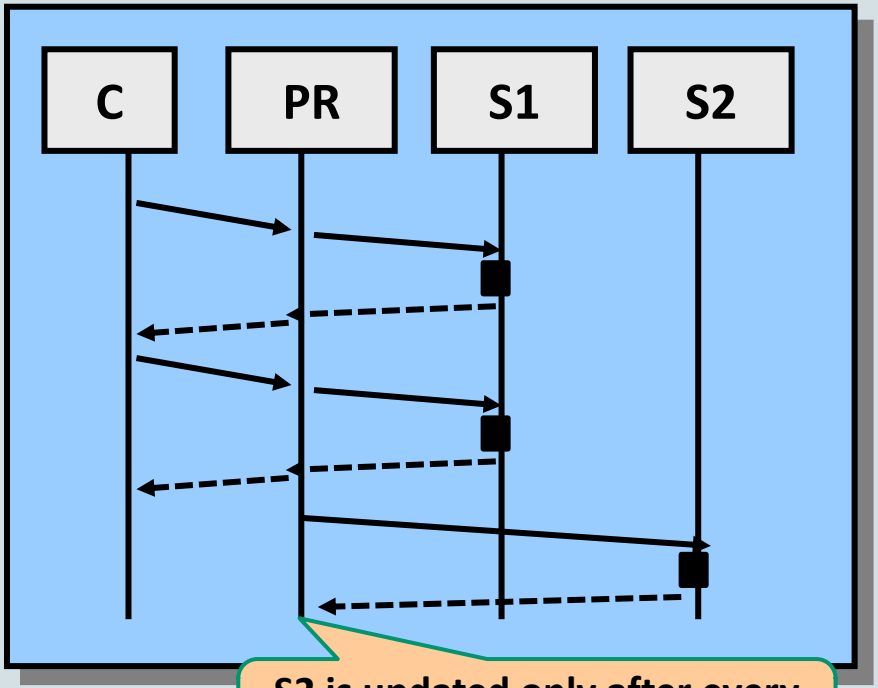




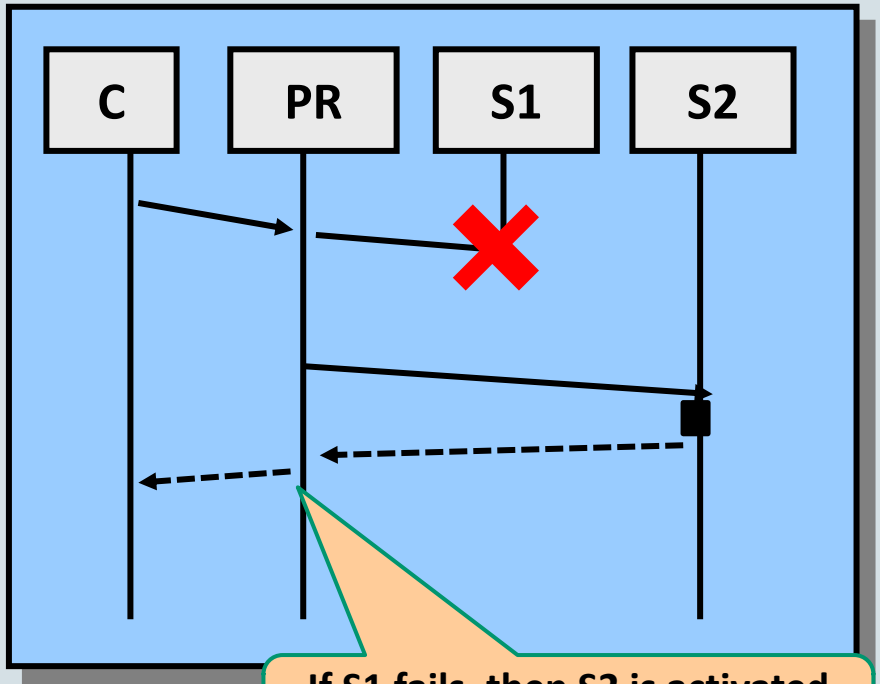
Client-Server Passive Redundancy





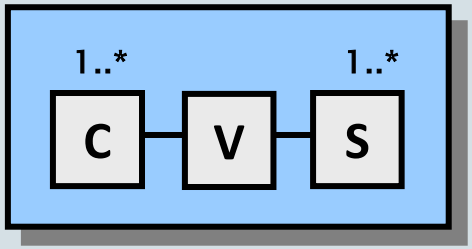


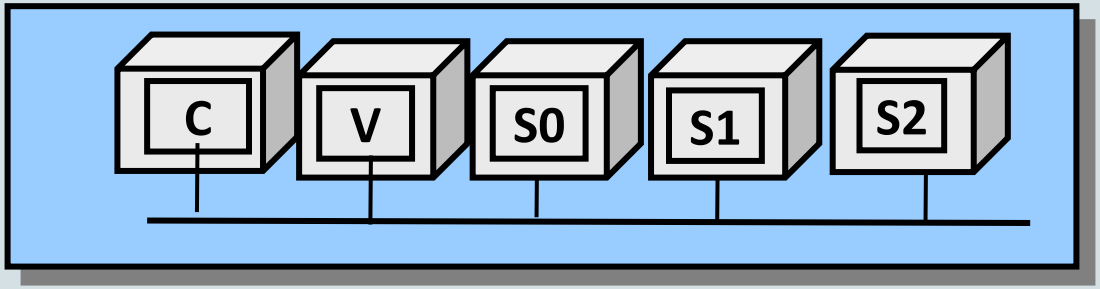
S2 is updated only after every X updates from S1

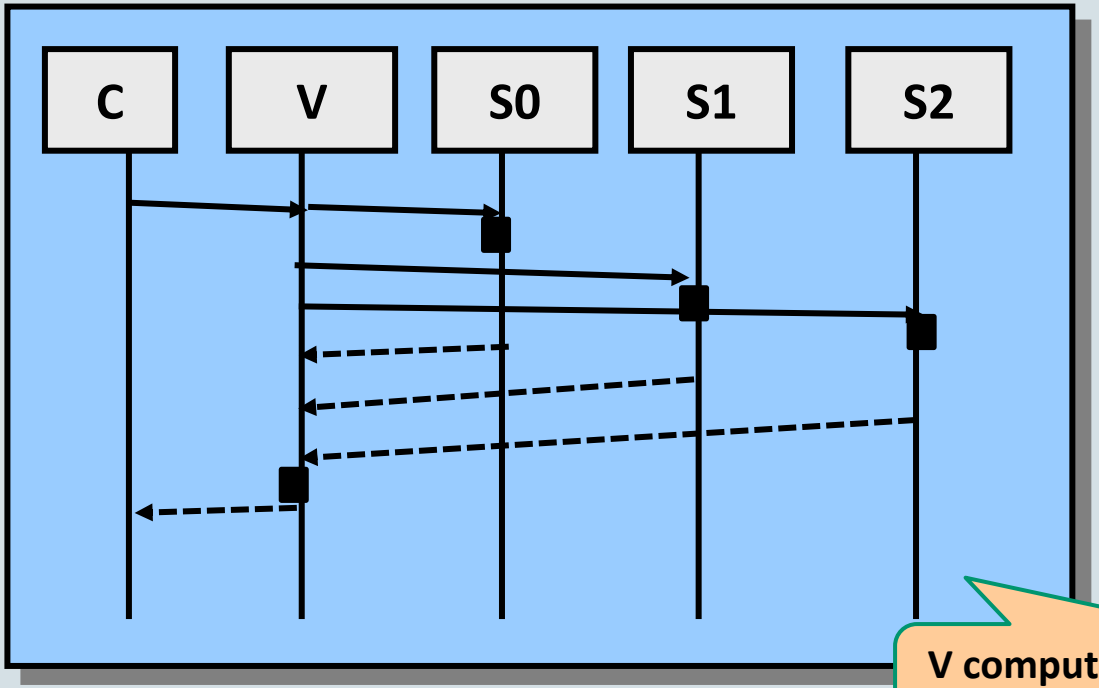


If S1 fails, then S2 is activated to take over from S1

Voting

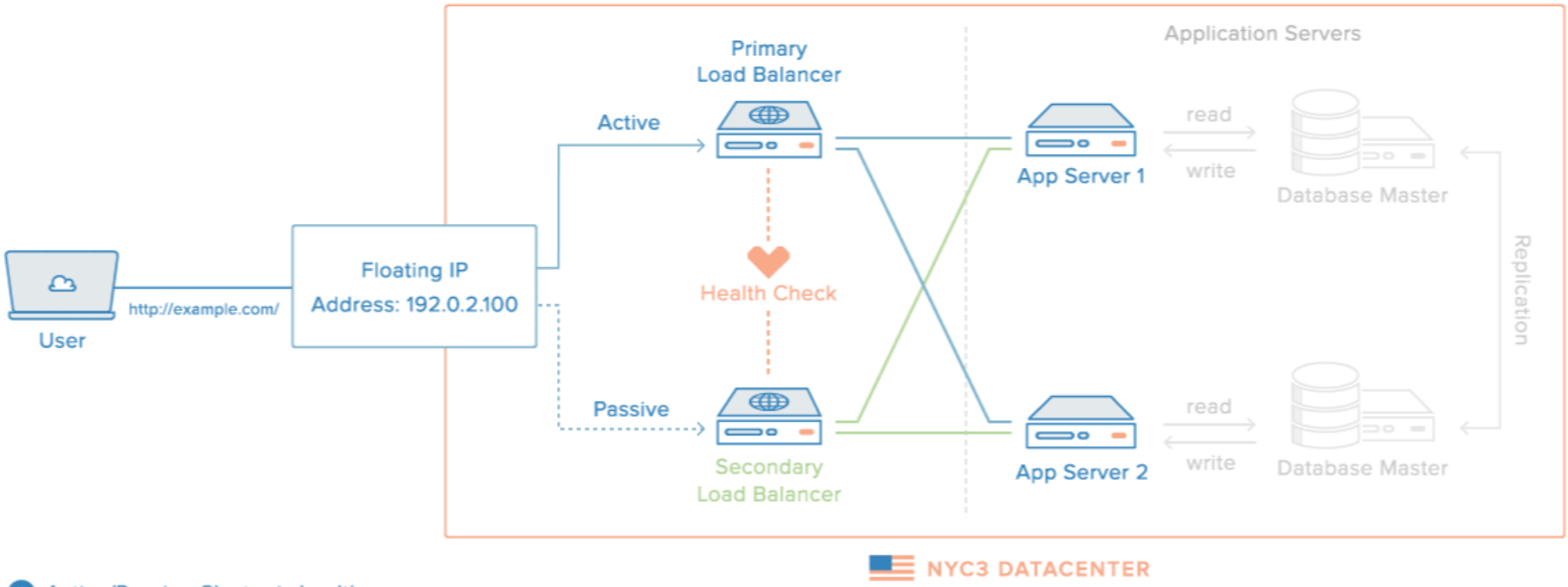






V computes the majority vote from the answers of S0, S1 and S2

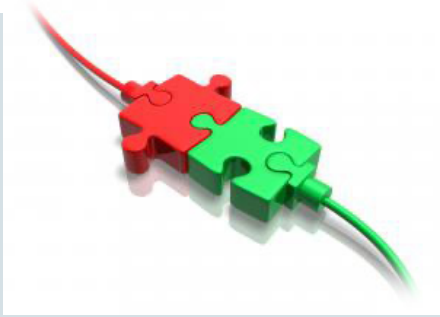
Example Availability Tactic



1. Active/Passive Cluster is healthy
2. Primary node fails
3. Floating IP is assigned to Secondary node

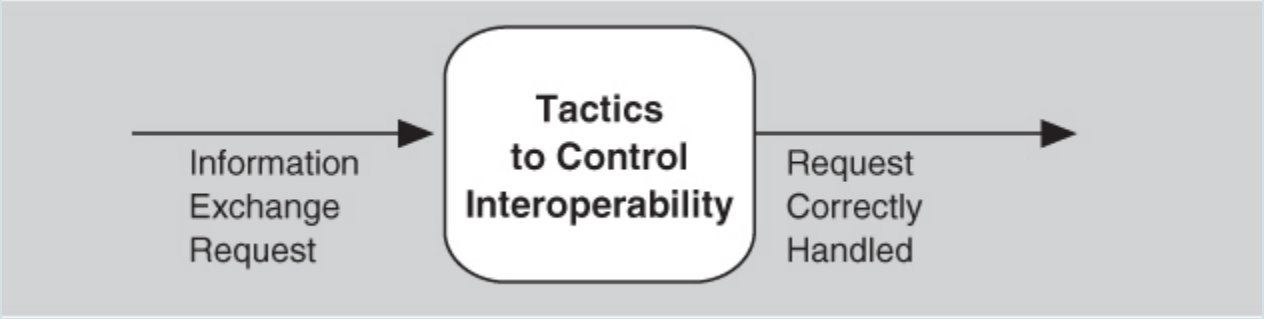
[https://www.digitalocean.com]

Interoperability



- Definition:

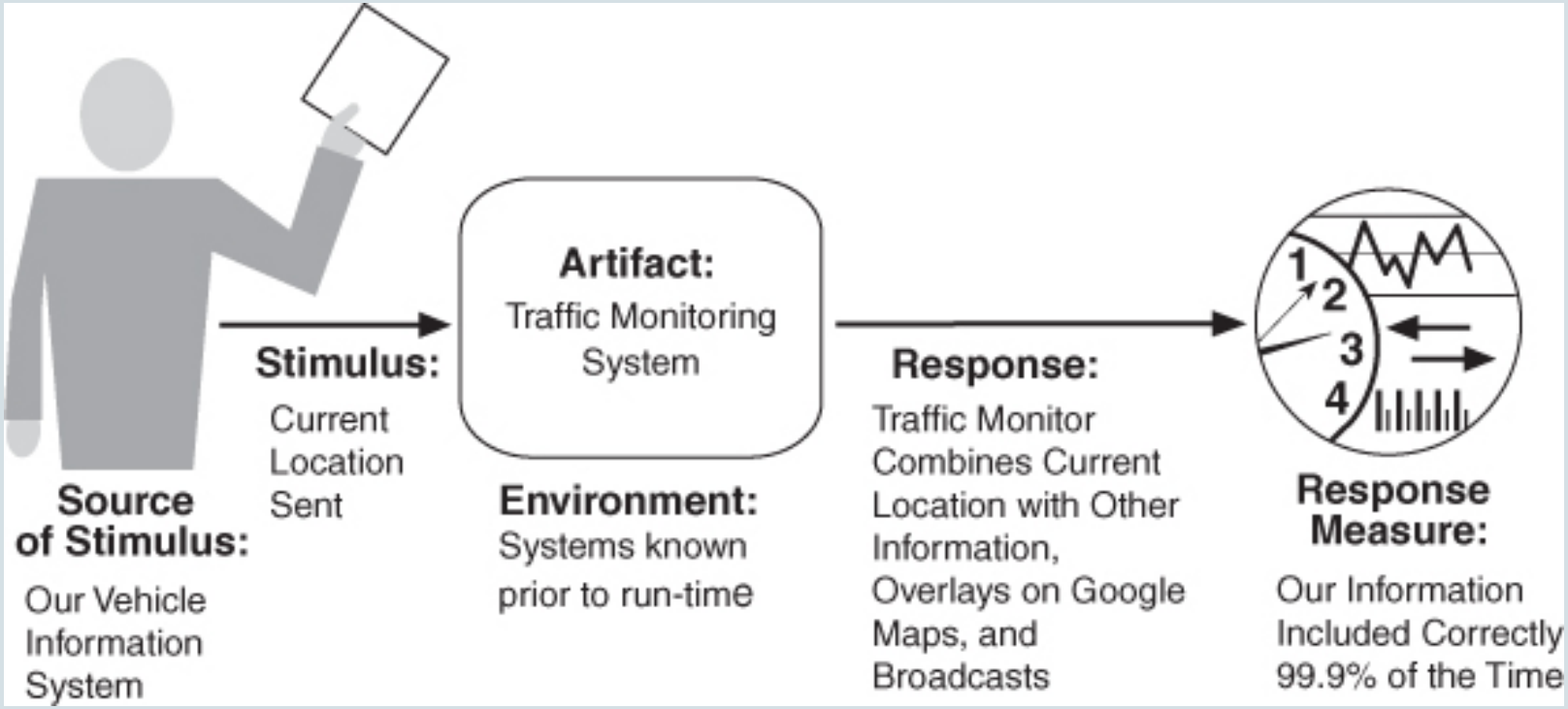
The degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context.



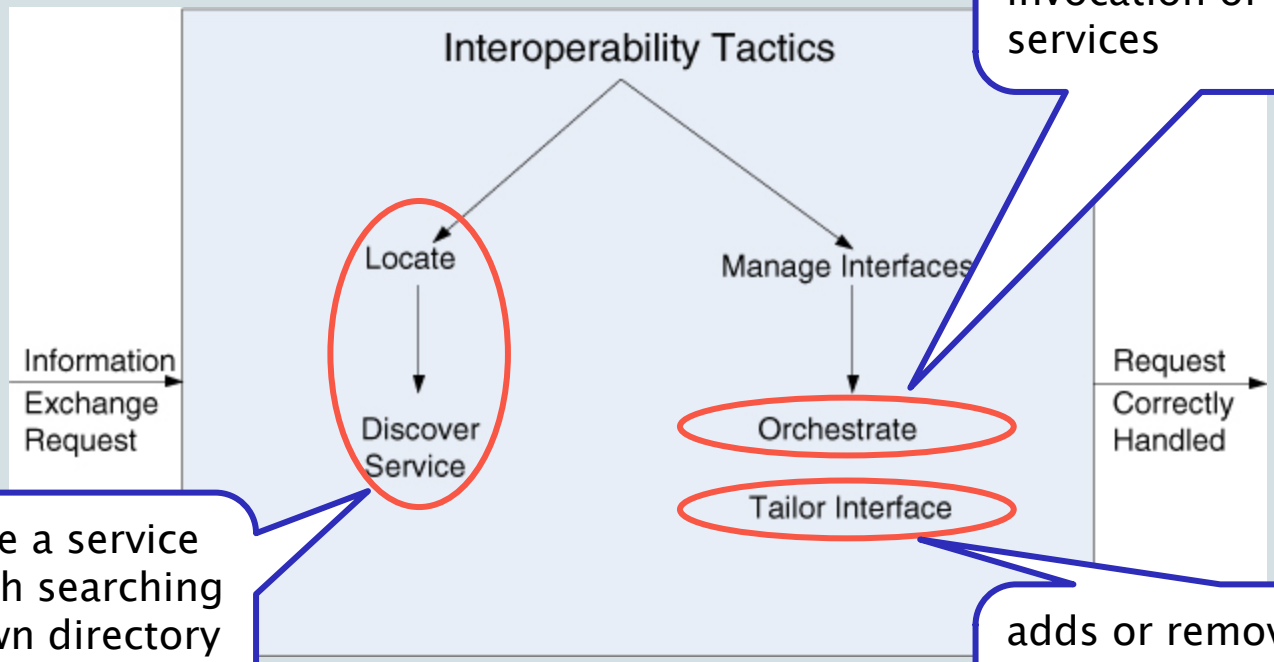
Interoperability Scenario

Portion of Scenario	Possible Values
Source	A system initiates a request to interoperate with another system.
Stimulus	A request to exchange information among system(s).
Artifact	The systems that wish to interoperate.
Environment	System(s) wishing to interoperate are discovered at runtime or known prior to runtime.
Response	One or more of the following: ▪ The request is (appropriately) rejected and appropriate entities (people or systems) are notified. ▪ The request is (appropriately) accepted and information is exchanged successfully. ▪ The request is logged by one or more of the involved systems.
Response Measure	One or more of the following: ▪ Percentage of information exchanges correctly processed ▪ Percentage of information exchanges correctly rejected

Concrete Interoperability Scenario



Interoperability Tactics



coordinate and manage and sequence the invocation of particular services

Locate a service through searching a known directory service.

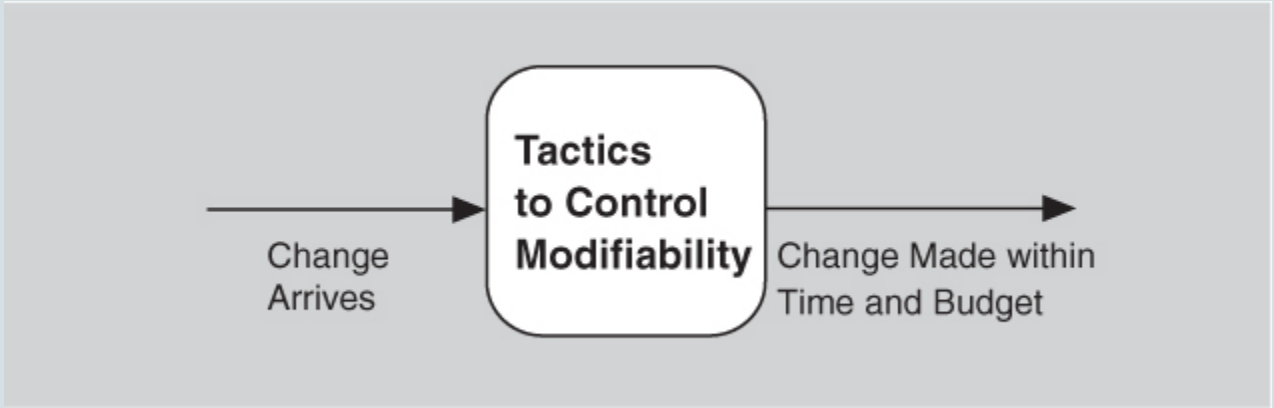
adds or removes capabilities to an interface (e.g. to hide information from untrusted users)

In which view(s) could we see interoperability tactics?

Modifiability

- Definition:

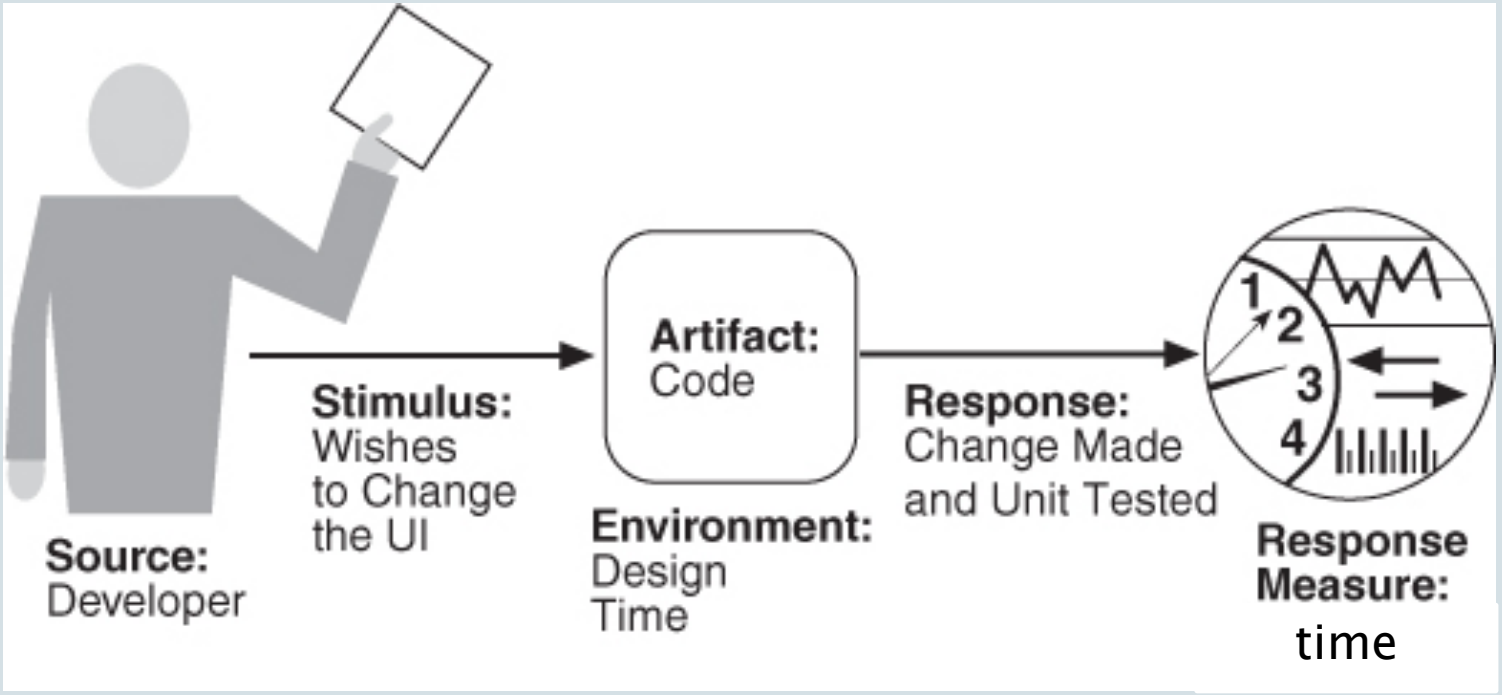
Modifiability is about change, and our interest in it centers on the cost and risk of making changes.



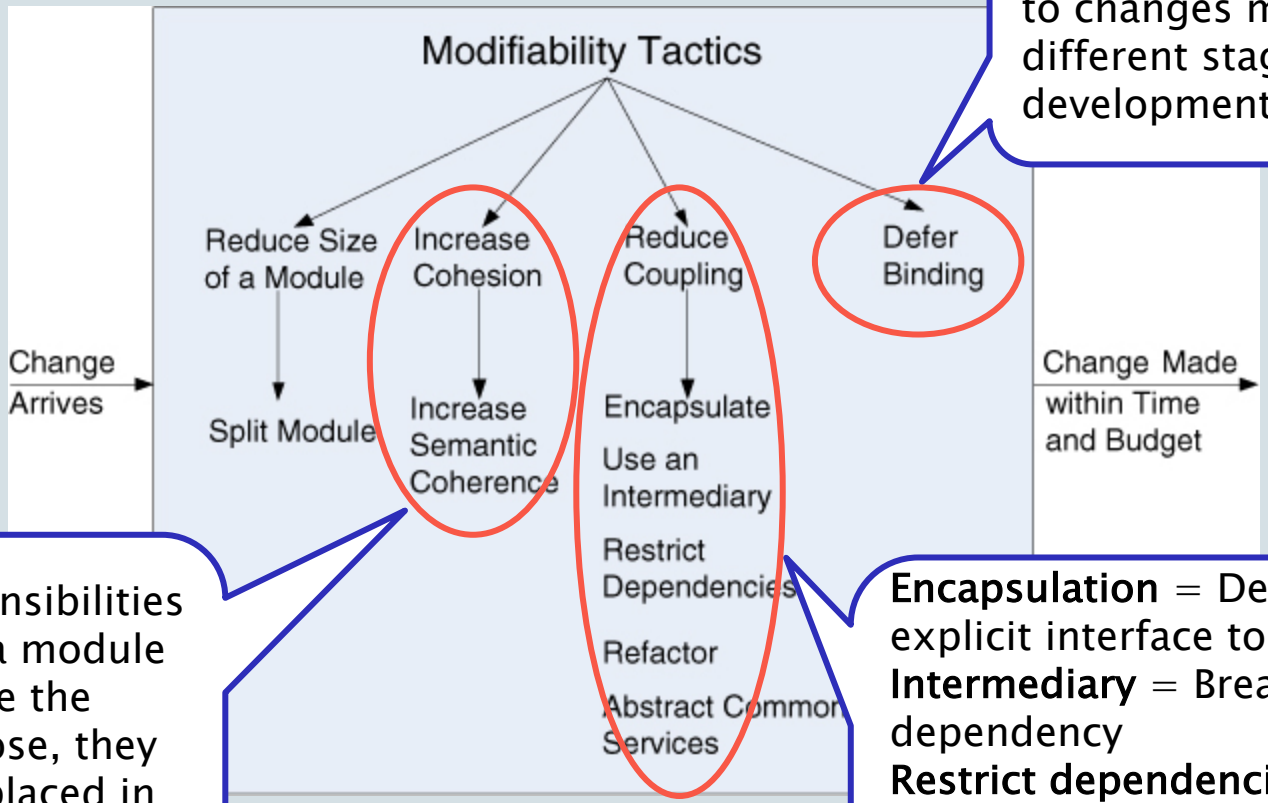
Modifiability General Scenario

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	A directive to add/delete/modify functionality, or change a quality attribute, capacity, or technology
Artifacts	Code, data, interfaces, components, resources, configurations, . . .
Environment	Runtime, compile time, build time, initiation time, design time
Response	One or more of the following: ▪ Make modification ▪ Test modification ▪ Deploy modification
Response Measure	Cost in terms of the following: ▪ Number, size, complexity of affected artifacts ▪ Effort ▪ Calendar time ▪ Money (direct outlay or opportunity cost) ▪ Extent to which this modification affects other functions or quality attributes ▪ New defects introduced

Concrete Modifiability Scenario



Modifiability Tactics



Flexibility of the system to changes made in different stages of sw development/deployment

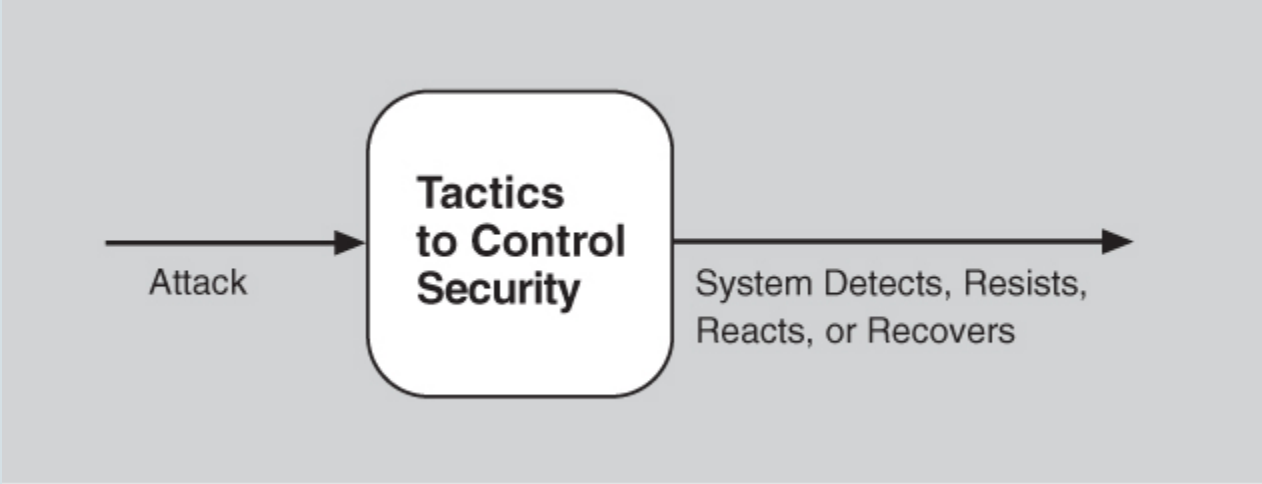
If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules

Encapsulation = Define an explicit interface to a module.
Intermediary = Break a dependency
Restrict dependencies = control 'visibility' (layered architecture)
Refactor = reducing duplicate code/responsibilities

Security

Definition:

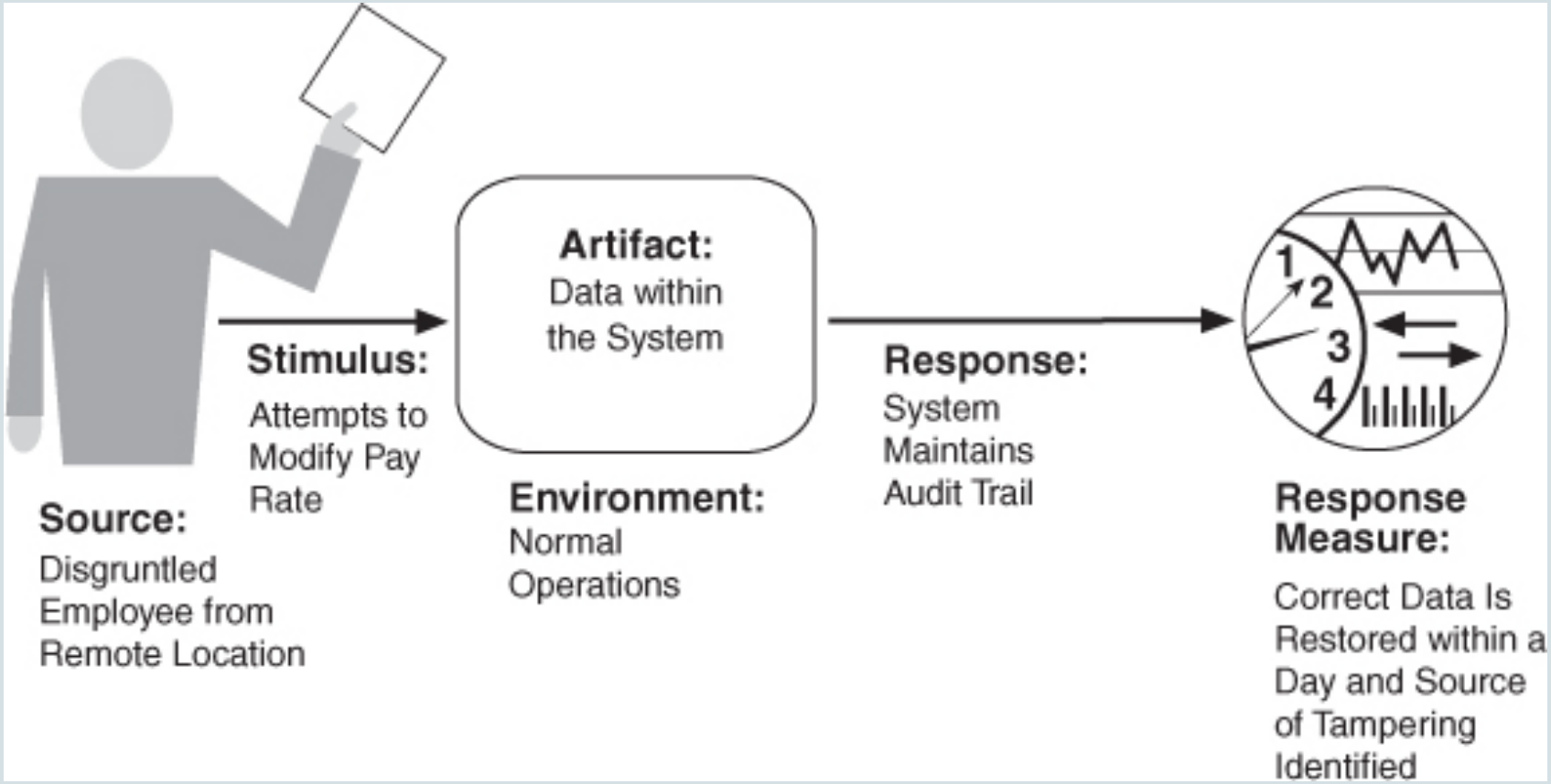
Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.



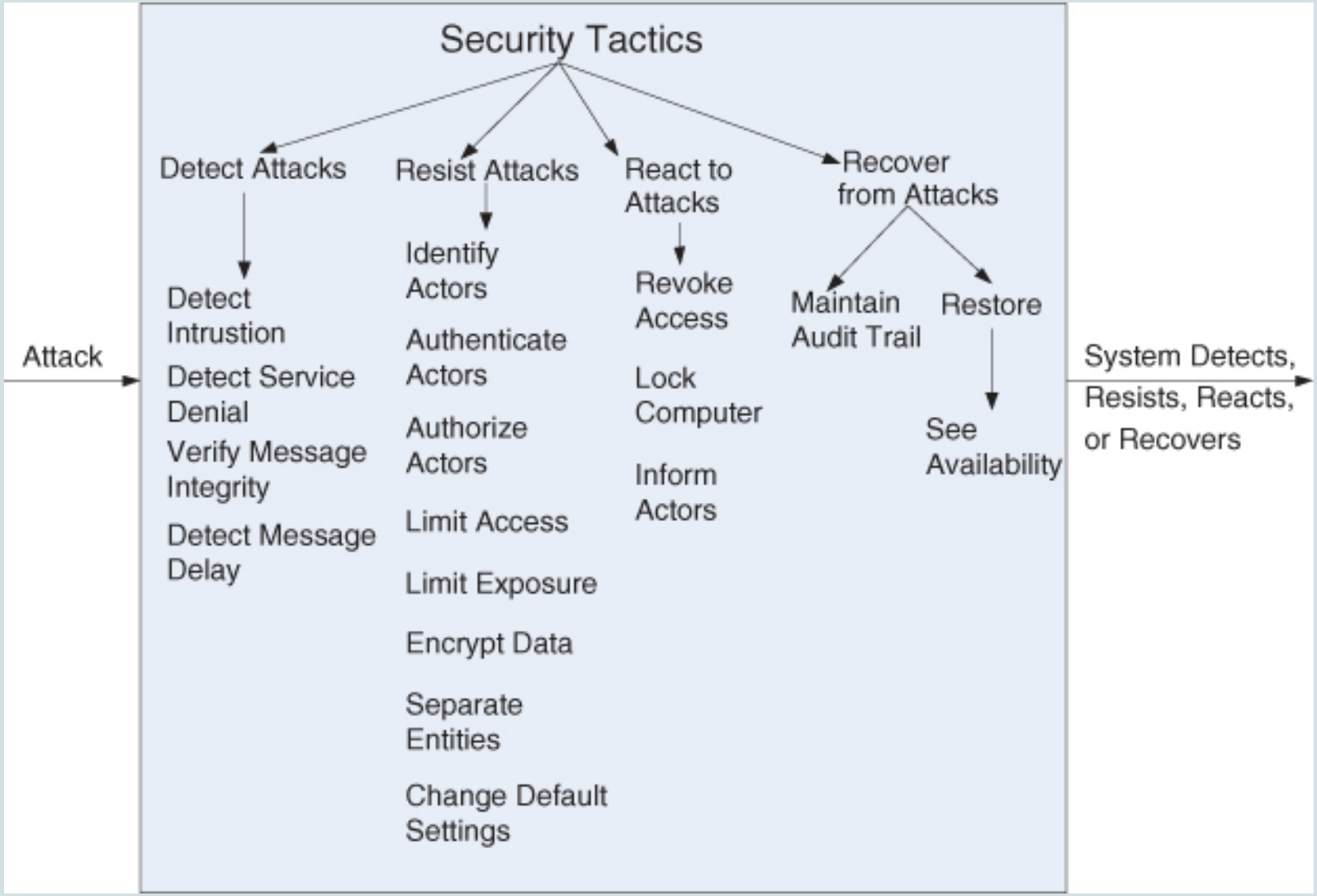
Security General Scenario

Portion of Scenario	Possible Values
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services, data within the system, a component or resources of the system, data produced or consumed by the system
Environment	The system is either online or offline; either connected to or disconnected from a network; either behind a firewall or open to a network; fully operational, partially operational, or not operational.
Response	Transactions are carried out in a fashion such that ▪ Data or services are protected from unauthorized access. ▪ Data or services are not being manipulated without authorization. ▪ Parties to a transaction are identified with assurance. ▪ The parties to the transaction cannot repudiate their involvements. ▪ The data, resources, and system services will be available for legitimate use. The system tracks activities within it by ▪ Recording access or modification ▪ Recording attempts to access data, resources, or services ▪ Notifying appropriate entities (people or systems) when an apparent attack is occurring
Response Measure	One or more of the following: ▪ How much of a system is compromised when a particular component or data value is compromised ▪ How much time passed before an attack was detected ▪ How many attacks were resisted ▪ How long does it take to recover from a successful attack ▪ How much data is vulnerable to a particular attack

Concrete Security Scenario



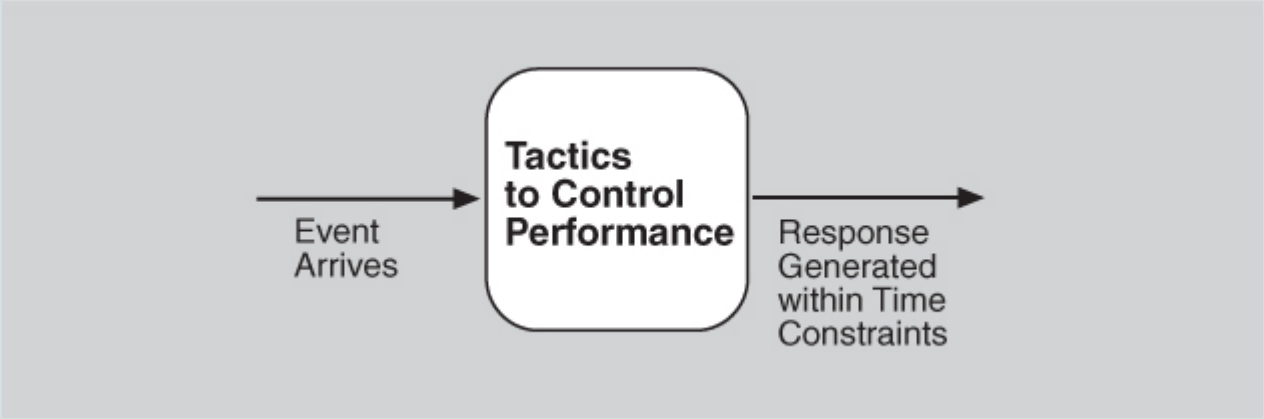
Security Tactics



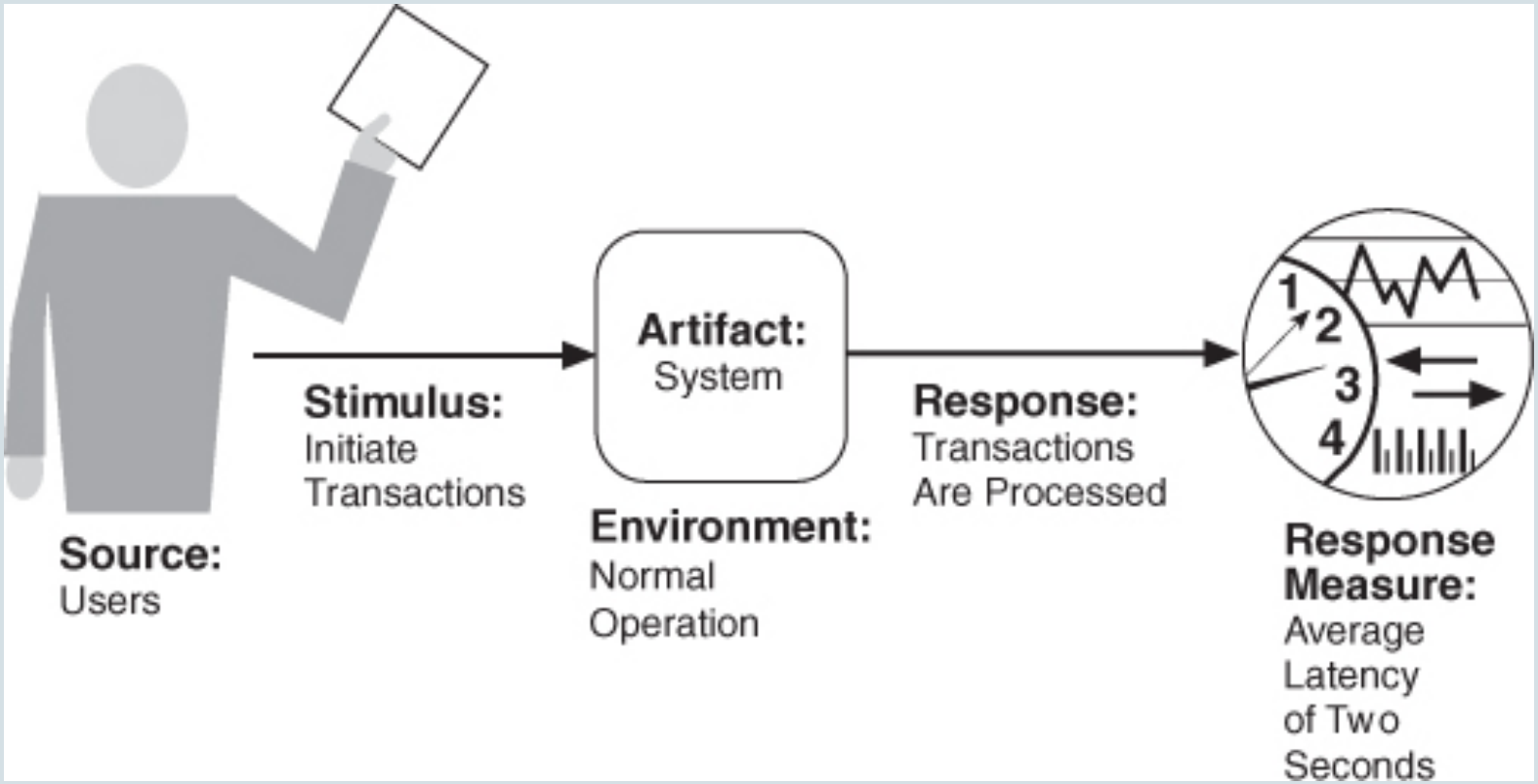
Performance

- Definition:

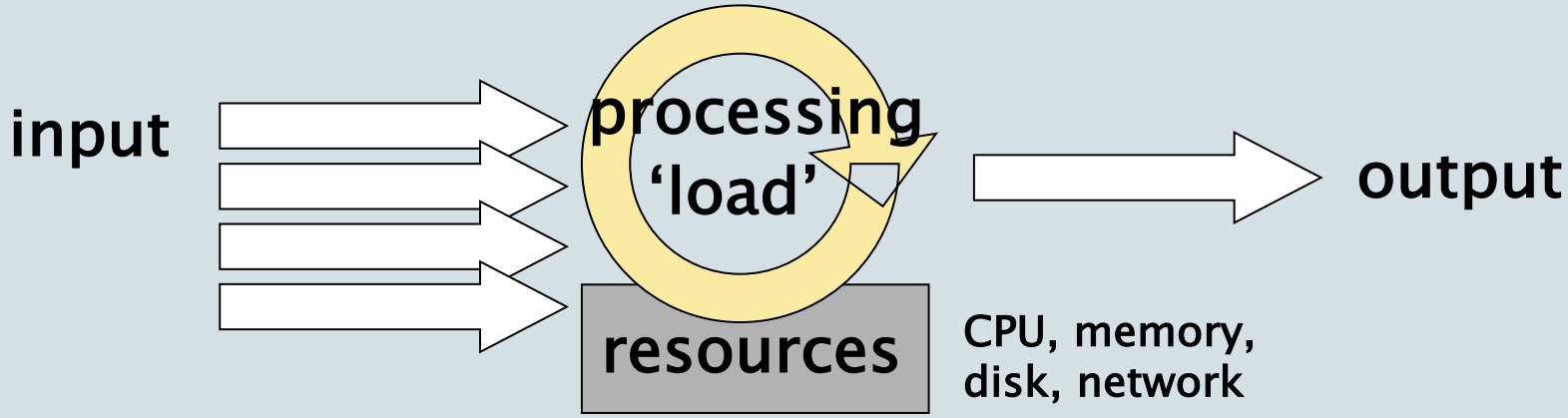
Performance is about time and the software system's ability to meet timing requirements.



Performance QA-Scenario



Resource Demand



Reduce load

- Manage event rate (e.g. drop requests, queue requests)
- Control frequency of sampling
- Prioritize events

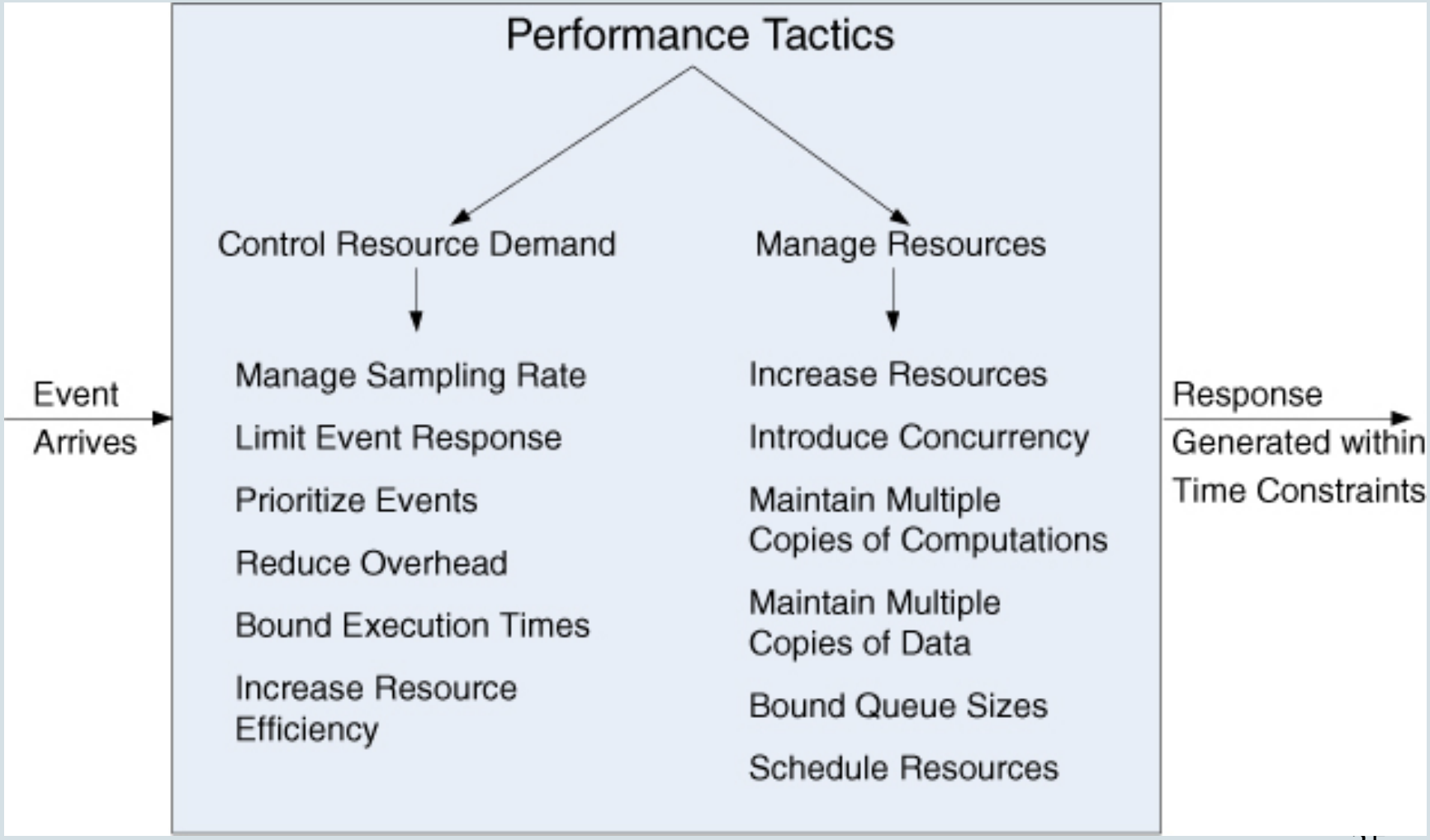
Increase computational efficiency

- Reduce computational overhead
- Reduce computation (e.g. scale quality of output)
- Bound execution times
- Bound queue sizes

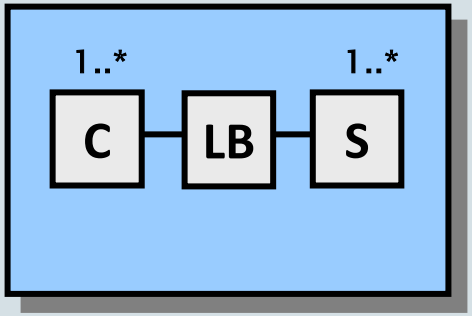
Increase Resources

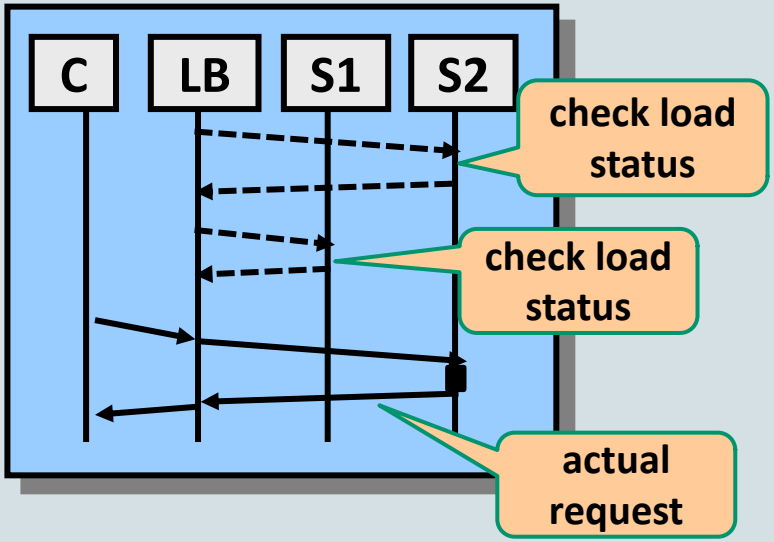
- Replicate server (=CPU's)

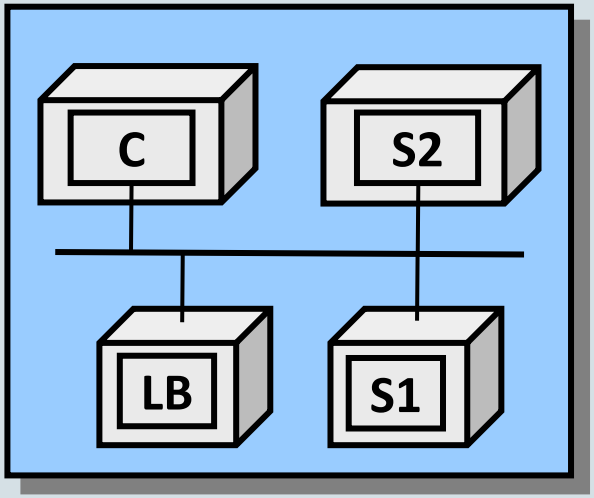
Performance Tactics



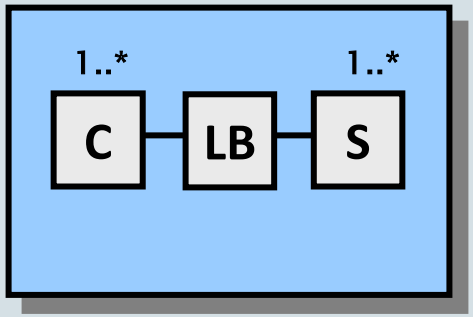
Client-Server with load balancer

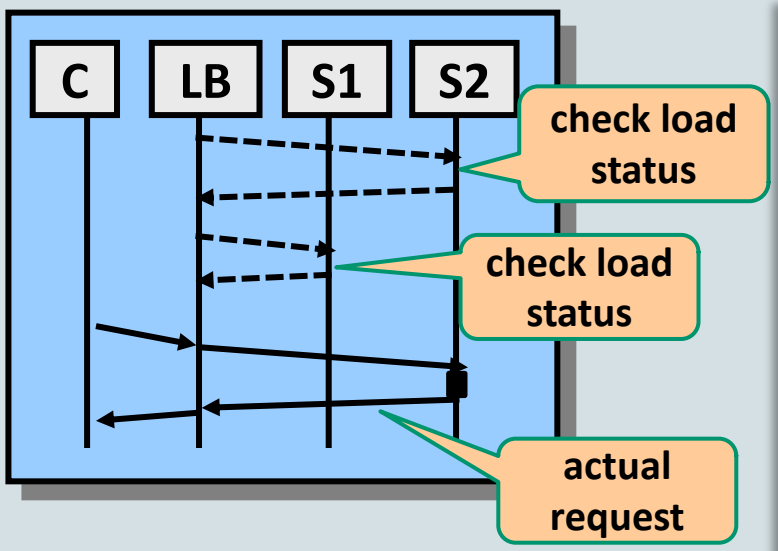


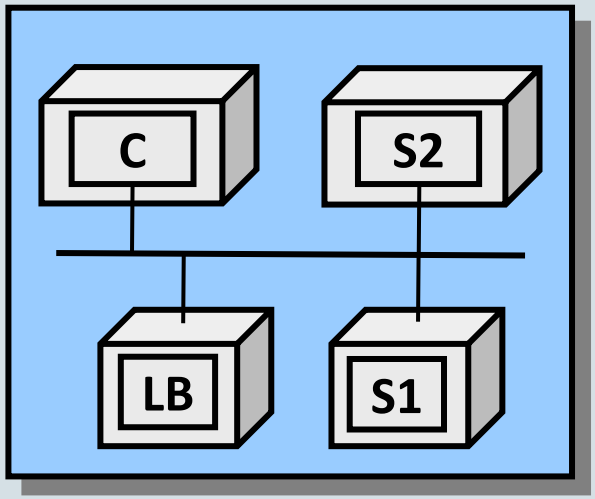


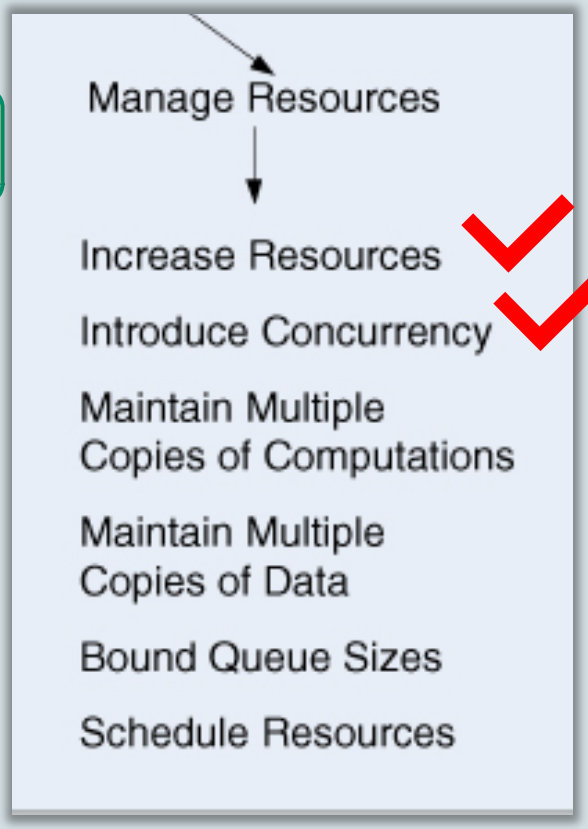


Client-Server with load balancer









Reliability Block Diagrams
Analysis and Tactics

No model is correct, but some are useful
– Albert Einstein

ISO standard on Software Product Quality

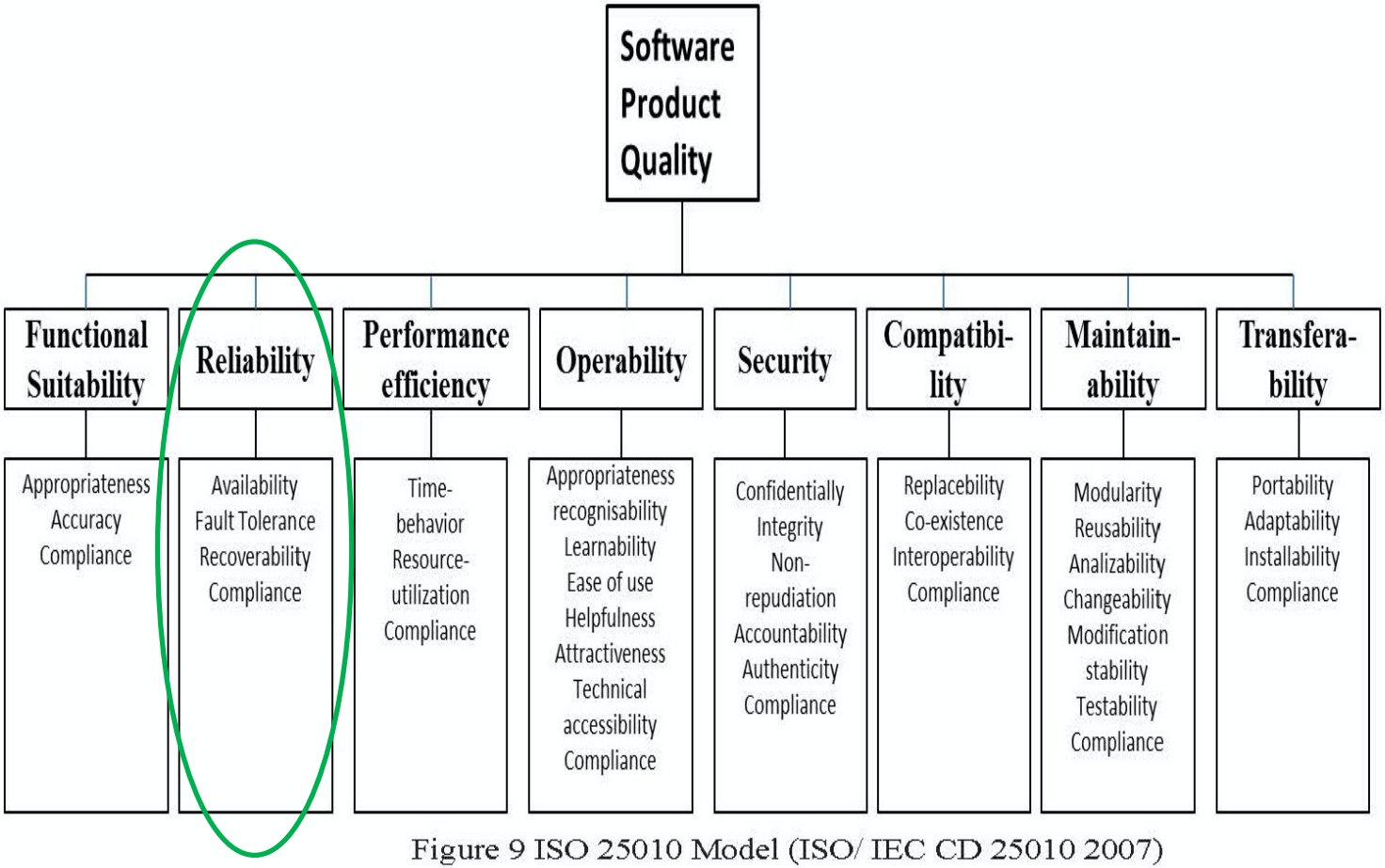


Figure 9 ISO 25010 Model (ISO/ IEC CD 25010 2007)

What is Reliability?

Reliability is a measure of the continuous delivery of correct service (Laprie)

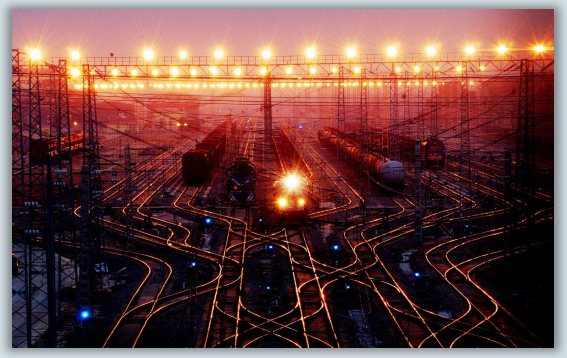
Reliability is defined as the probability that an item will perform its intended function for a specified interval under stated conditions.

Reliability is a property over time:
'*a **probability** that something will work when you want it to.*'

Expressing Reliability Quantitatively

MTBF - Mean Time Between Failures
is the expected Time between Failures

Typically used for: repairable systems



Train control system

MTTF - Mean Time to Failure
is the expected Time to Failure

Typically used: non-repairable systems

Mars rover

MTTR - Mean Time to Repair
is the expected Time to Repair/Recover Failure

Examples of Reliability Measurement

Typically expressed in terms of

for repairable systems

- **Mean Time Between Failures (MTBF)**
  - Number of hours that pass before a component fails
  - E.g. 2 failures per million hours:
    - MTBF = 10^6 / 2 = 0,5 * 10^6 hr

For non-repairable systems

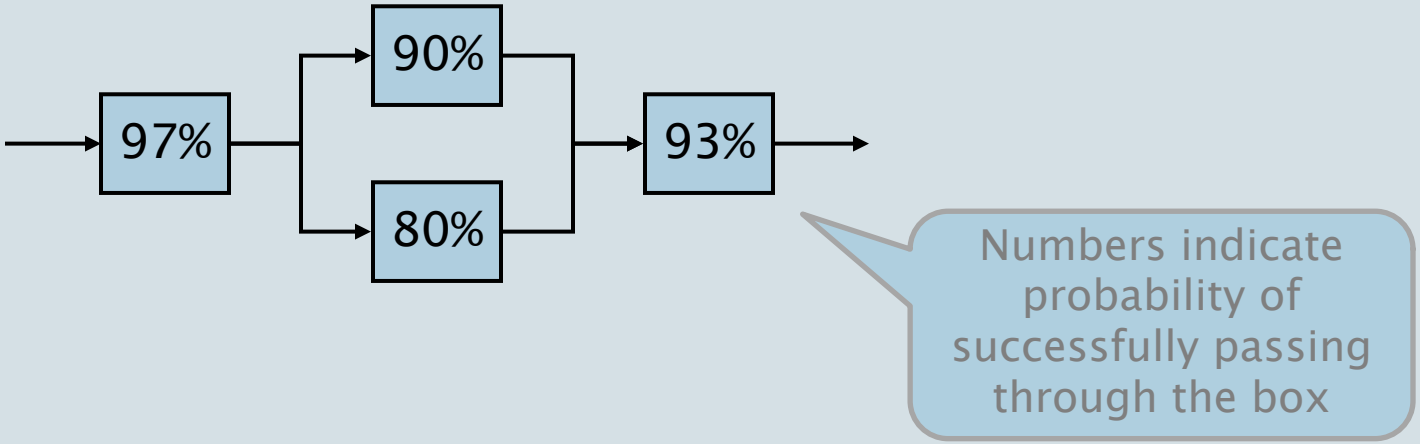
- **Mean Time To Failure (MTTF)**
  - Mean time expectec until the first failure of a system
  - Is a statistical value over a long period of time

- **Mean Time To Repair (MTTR)**

Availability

Reliability Block Diagrams (RBD)

A RBD is a graphical depiction of the system's **components** and **connectors** which can be used to determine the overall system reliability



If any path through the system is successful, then the system succeeds, otherwise it fails.

RBD 2: Assumptions

- Lines have reliability 1
- Failures of blocks are statistically independent
- Blocks are bi-modal:

  either their operate correctly or
  they fail and do nothing

RBD 3: How to express reliability?

Assuming a homogeneous failure rate,
a failure rate of λ (per unit of time t)
constitutes a reliability over a period T of

$$R = e^{-\lambda \cdot T/t}$$

If a component has a failure rate of 10 failures per 1000 hours, then its reliability over a 24 hour period is (approx.). 79%.

A simple example

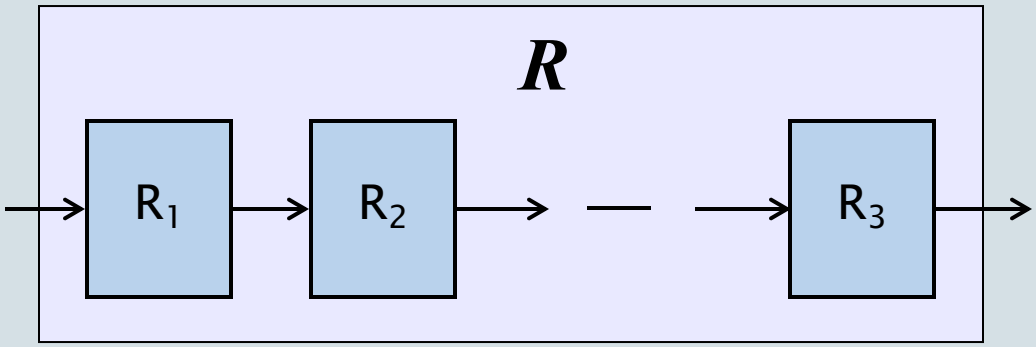
- A system has 4000 components with a failure rate of 0.02% per 1000 hours. Calculate λ and MTBF.

- $\lambda = (0.02 / 100) * (1 / 1000) * 4000 = 8 * 10^{-4}$ failures/hour

- $MTBF = 1 / (8 * 10^{-4}) = 1250$ hours

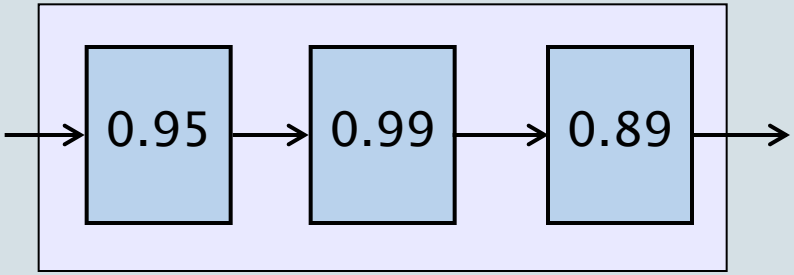
51

Common RBD Patterns: Chain of Components



$$R = \prod_{i=1}^{n} R_i$$

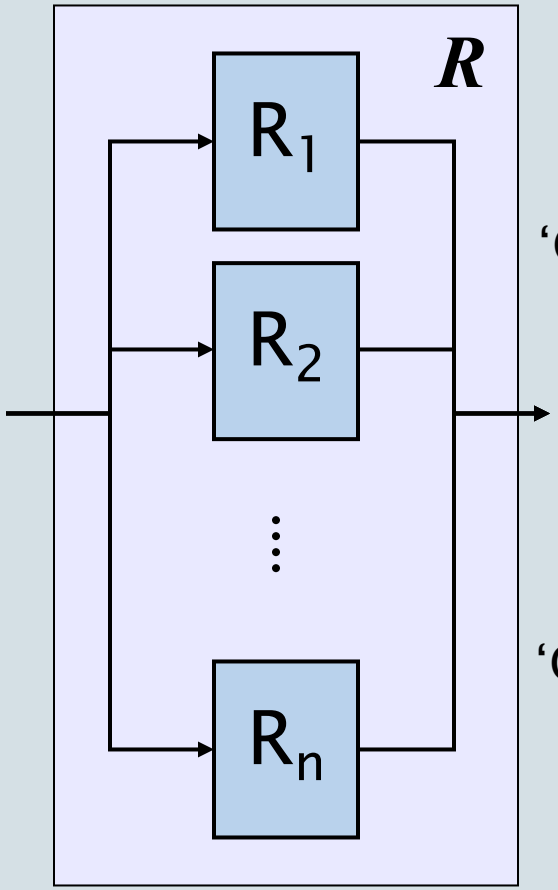
For example



'AND' 'AND'

$$R = 0.95*0.99*0.89 = 0.84$$

Common RBD Patterns: Alternative (Parallel) Components

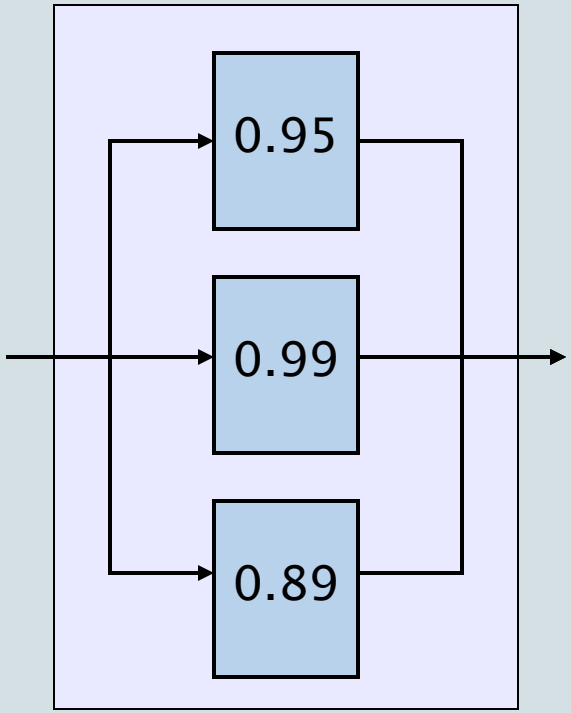


$$R = 1 - \prod_{i=1}^{n} (1 - R_i)$$

R fails only if ***all*** component fail

Prob of fail = *1- success*

Example Parallel Components

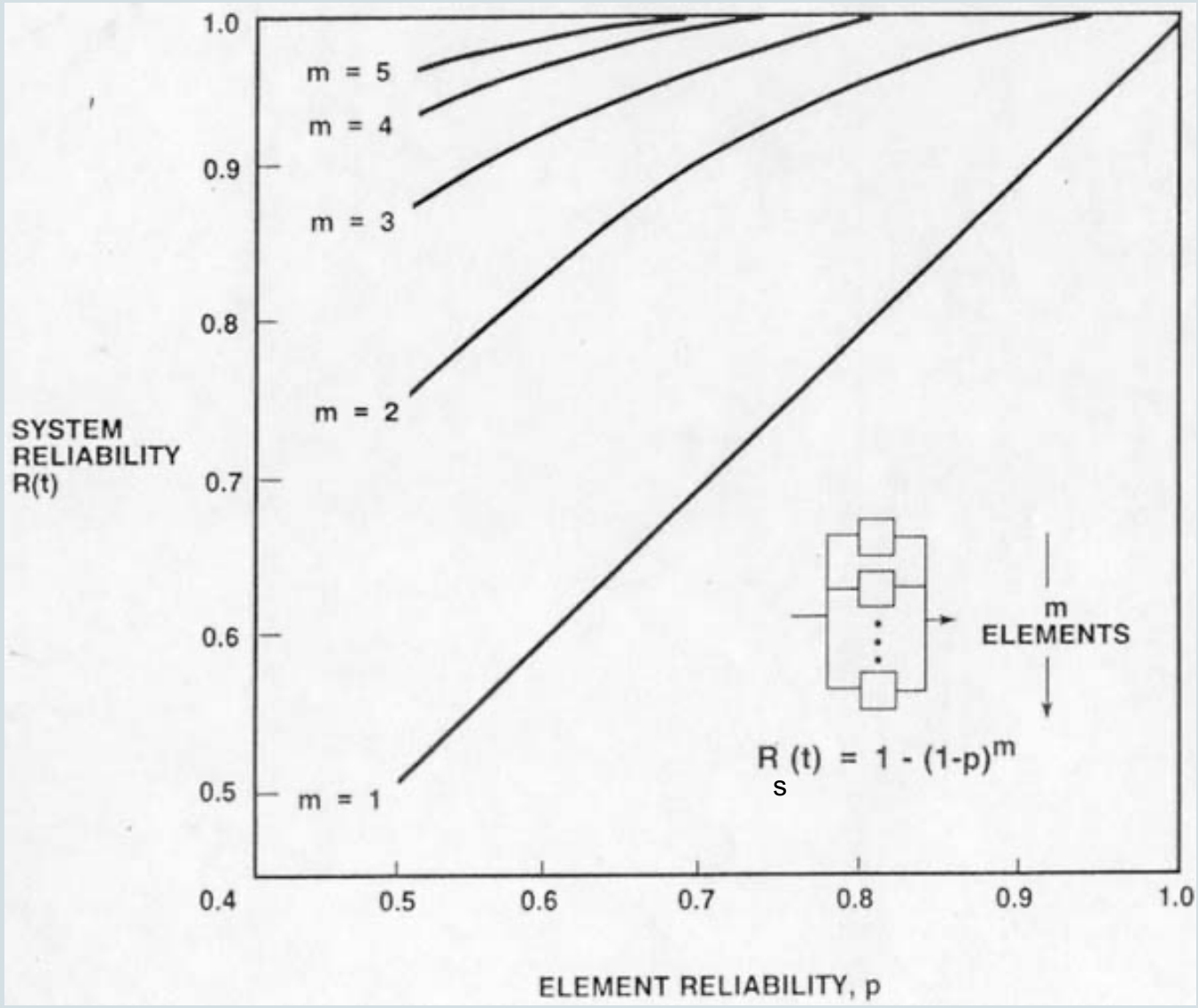


$$R = 1 - \prod_{i=1}^{n} (1 - R_i)$$

$$\begin{aligned} \boldsymbol{R} &= 1 - (1\text{-}0.95)*(1\text{-}0.99)*(1\text{-}0.89) \\ &= 1 - 0.05*0.01*0.11 \\ &= 1 - 5.5*10^{-5} \\ &= 0.99 \end{aligned}$$

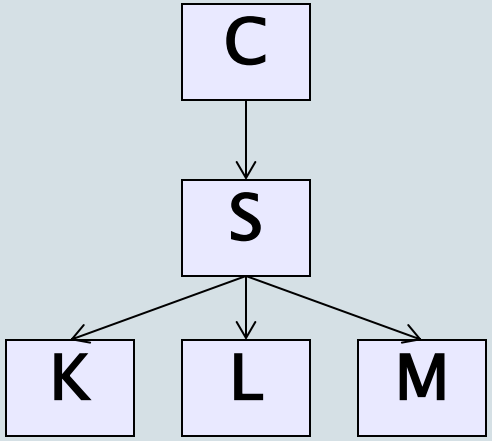
Reliability of system is higher than that of its parts!

Parallel Reliability Configuration

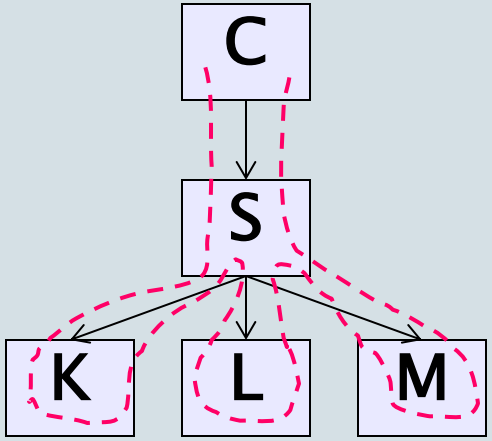


Example Reliability: Structural View

component-diagram
with uses-relations

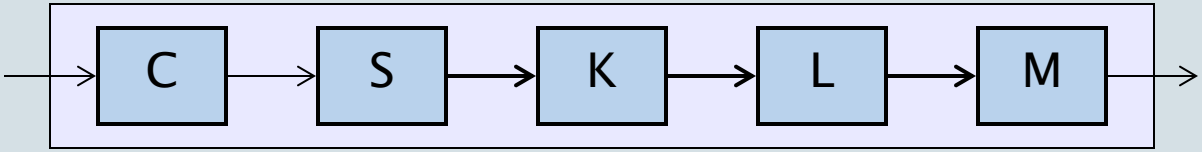


typical flow of control
for procedure-call style



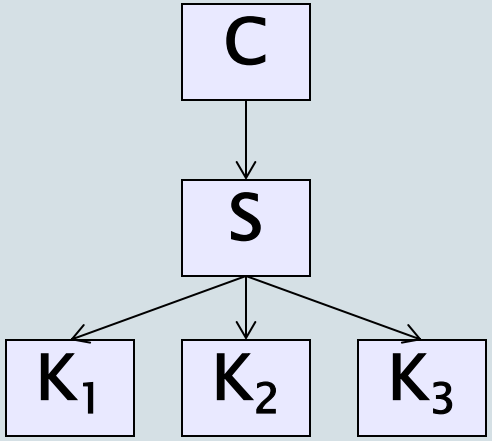
All components are needed

reliability block diagram ?

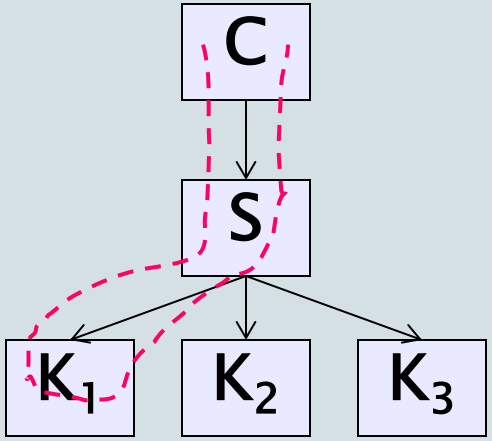


Example Reliability: Structural View

component-diagram
with uses-relations

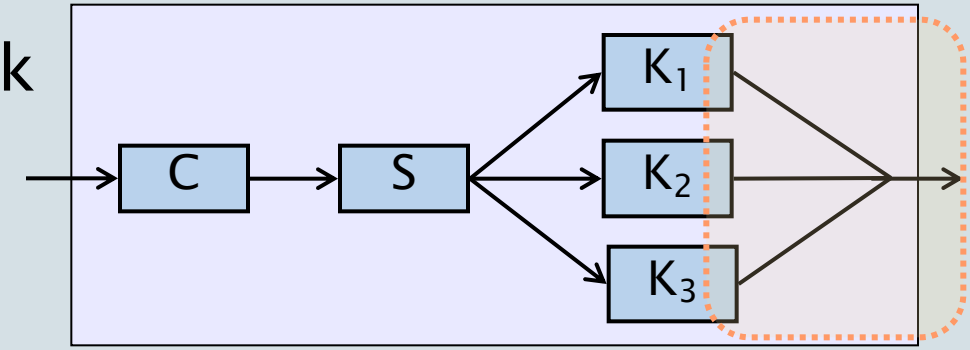


typical flow of control
for procedure-call style



components are alternatives

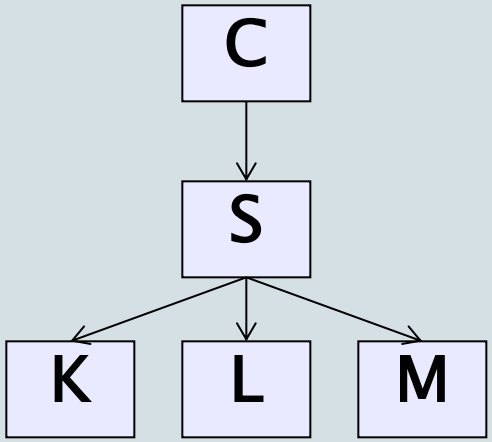
reliability block
diagram ?



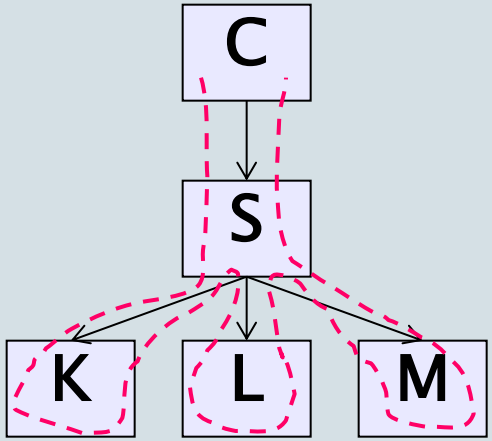
Don't forget the
outgoing arrow

Example Reliability: Deployment View

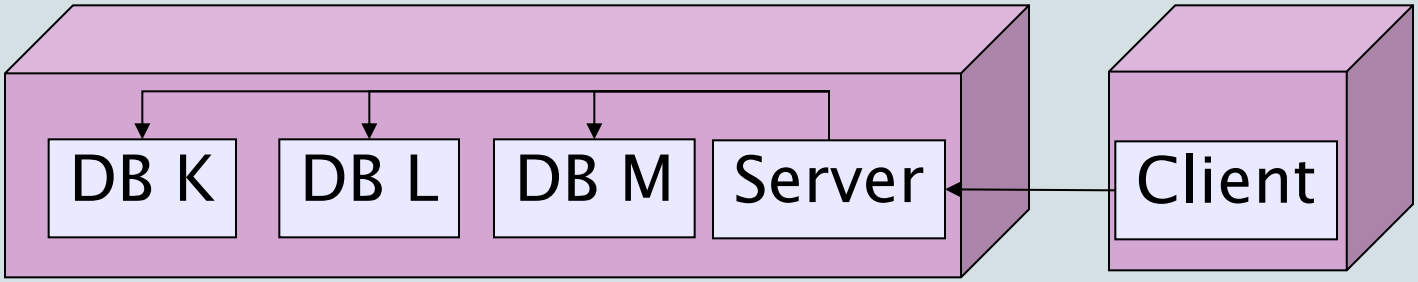
component-diagram
with uses-relations



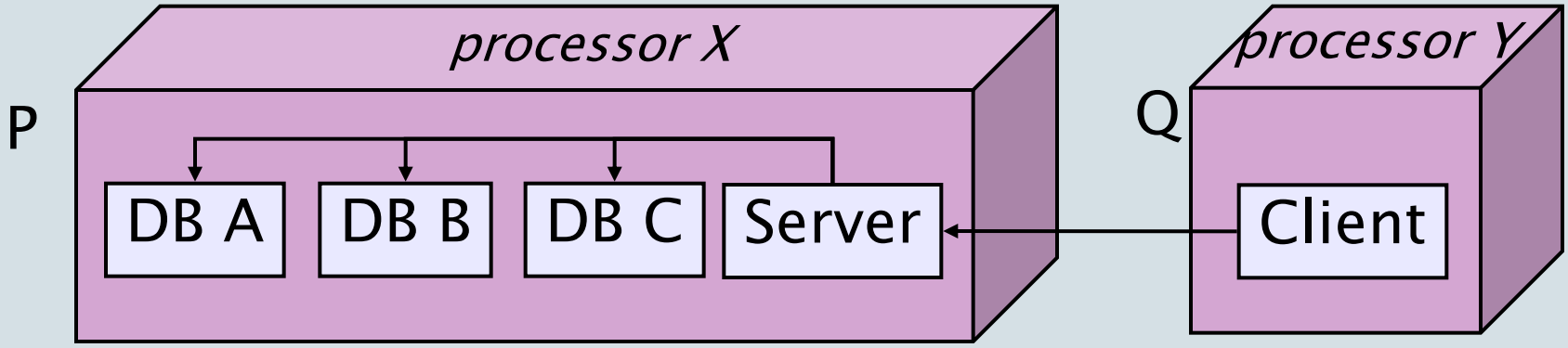
typical flow of control
for procedure-call style



deployment diagram



Example Reliability: Deployment View



Assume Server, DB's are in constant use and share CPU equally.

$$\lambda_P = (\lambda_{DB\,A} + \lambda_{DB\,B} + \lambda_{DB\,C} + \lambda_S) / 4$$

$$\lambda_Q = \lambda_{Client}$$

$$\lambda_{P+Q} = \lambda_P + \lambda_Q$$

If number of clients grows to *n*, then $\lambda_{P+Q} = \lambda_P + n \cdot \lambda_Q$

RBD Application Heuristics

- Not all systems can be reduced to series/parallel graphs. Hence, you may need to simplify the design.

  For instance by
  - consider only the critical paths through the system

- Consider fragment of time spent per component

- There are tools available that help you compute reliability of RBD's

Research Question

- How can we determine the reliability of a piece of software?
  - analytically?
  - empirically?
    - observe a system? Prototype?

- Not all defects threaten reliability

References Reliability & RBD

J.-C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, pages 2-11, 1985.

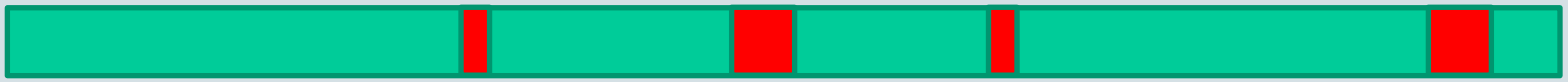
Abd-Allah, A., "Extending Reliability Block Diagrams to Software Architectures," USC Technical Report USC-CSE-97-501, Center for Software Engineering, University of Southern California, March 1997.

Availability

Availability = the probability that a system is operational at a given time

- i.e. the amount of time a device is actually operating as the percentage of total time it should be operating.
- High-availability systems may report availability in terms of minutes or hours of downtime per year.

Availability = uptime / (uptime + downtime)



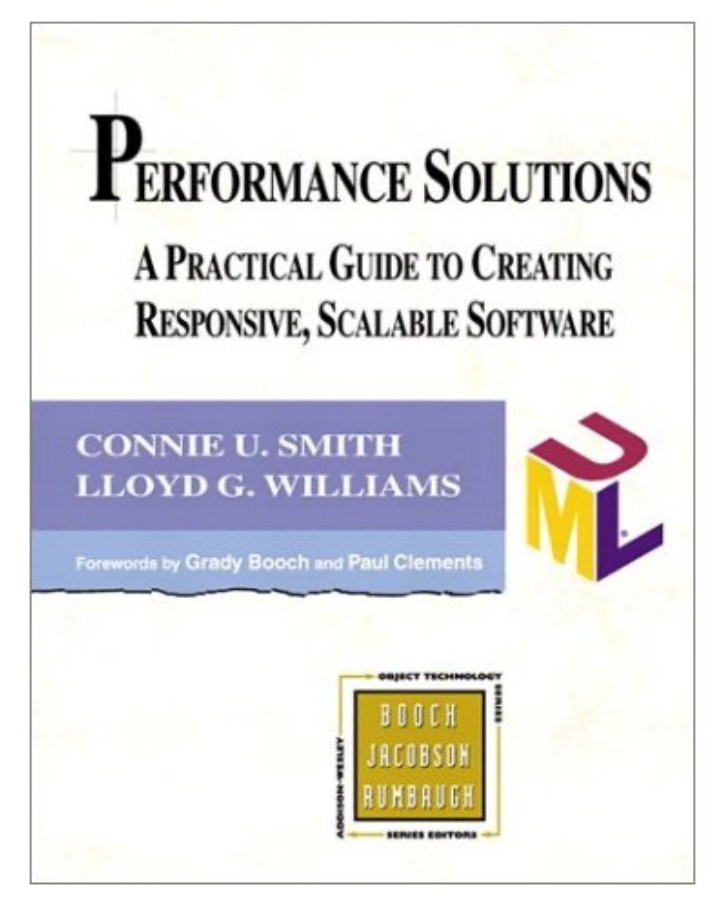
Availability = 

$$\text{Availability} = \text{MTBF} / (\text{MTBF} + \text{MTTR})$$

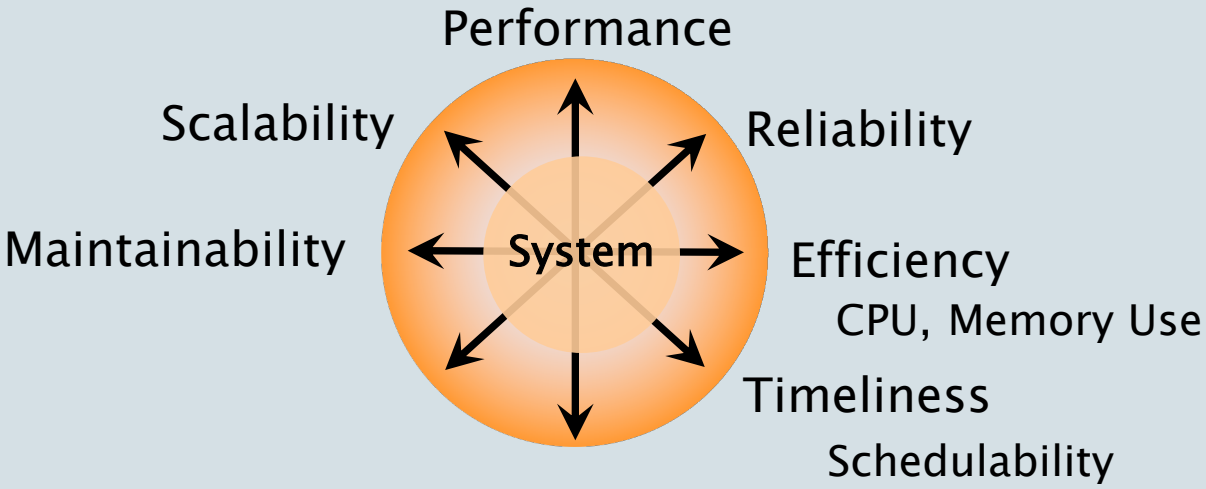
Software Performance Engineering

Connie U. Smith &
Lloyd G. Williams

Performance Solutions:
A Practical Guide to creating response, scalable software, Addison-Wesley, 2002



Extra Functional Properties



Essential system engineering problem:

- **a plurality of contradictory goals**
- **a plurality of means (technology, process)**
  each of which provides a varying degree of help or hindrance in achieving a given goal

Some more examples of *ilities

Accessibility, Understandability, Usability, Generality, Operability, Simplicity, Mobility, Nomadicity, Portability, Accuracy, Efficiency, Footprint, Responsiveness, Scalability, Schedulability, Timeliness, CPU utilization, Latency, Throughput, Concurrency, Flexibility, Changeability, Evolvability, Extensibility, Modifiability, Tailorability, Upgradeability, Expandability, Consistency, Adaptability, Composability, Interoperability, Openness, Integrability, Accountability, Completeness, Conciseness, Correctness, Testability, Traceability, Coherence, Analyzability, Modularity, Reusability, Configurability, Distributeability, Availability, Confidentiality, Integrity, Maintainability, Reliability, Safety, Security, Affordability, Serviceablility, …

Performance throughout lifecycle

	Core	Performance
Planning		
Requirements and Analysis	Functional Requirements Non Functional Requirements Technology Requirements	**Performance Objectives**
Architecture and Design	Design Guidelines Architecture and Design Review	**Performance Design Guidelines** **Performance Modeling** **Performance Architecture and Design Review**
Development	Unit Tests Code Review Daily Builds	**Performance Code Review** **Measuring**
Testing	Integration Testing System Testing	**Performance Testing** **Performance Tuning**
Deployment	Deployment Review	**Performance Deployment Review**
Maintenance		

Performance in Requirements Engineering

It is difficult to understand requirements in terms of application domain;

E.g. In digital video: good picture quality?

In airbag: should be safe?

we don't know how to quantify?

Resource needs are often underestimated when requirements are vague

Requirements are volatile:

the architecture should cater for changes

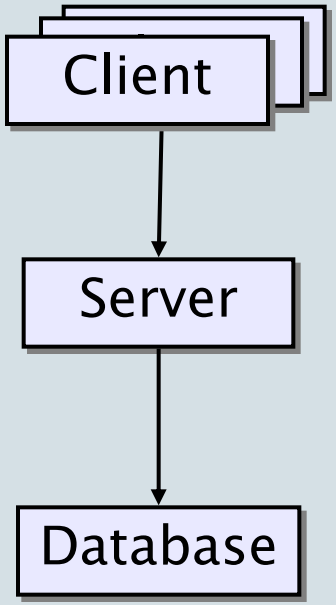
Performance Myths

- It is not possible to do anything about performance until you have something to measure
- Performance models are complex and expensive to construct.
- Architecture determines performance properties
- Simple models ('back of the envelope') can identify performance problems.

Performance Analysis

Performance is the ability of a system to meet its objectives for timeliness

- How well can this system handle the anticipated demand?
- What will the average response time be?
- Given a max. accept. response time, what is the highest load the system can handle?
- Which component is the bottleneck?



Performance Engineering

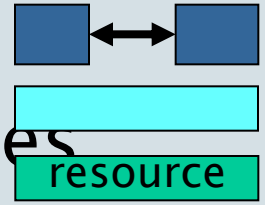
Important performance characteristics:

- Response Time
  - The time it takes the system to react to an impulse/trigger
- Throughput
  - The rate at which jobs are completed
- Capacity
  - Typically: storage, processing, bandwidth

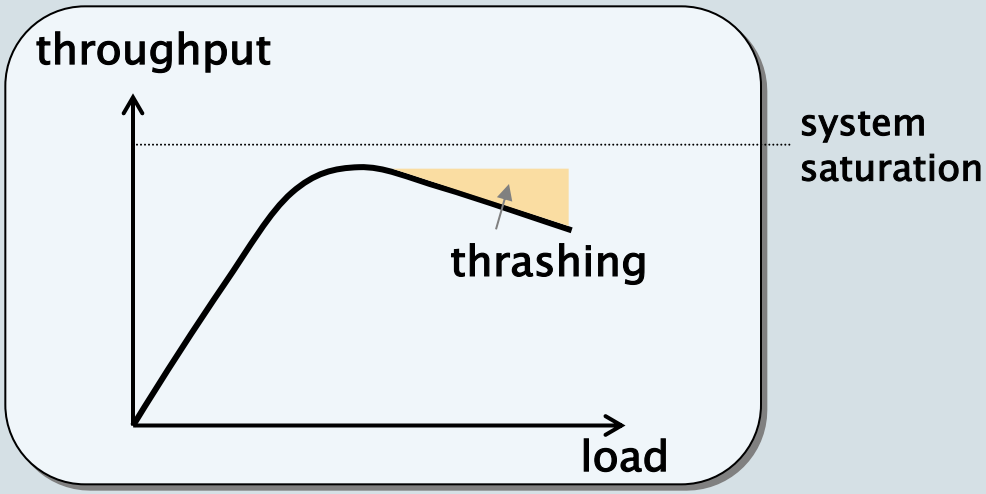
- Performance Engineering is also known as: Capacity Planning

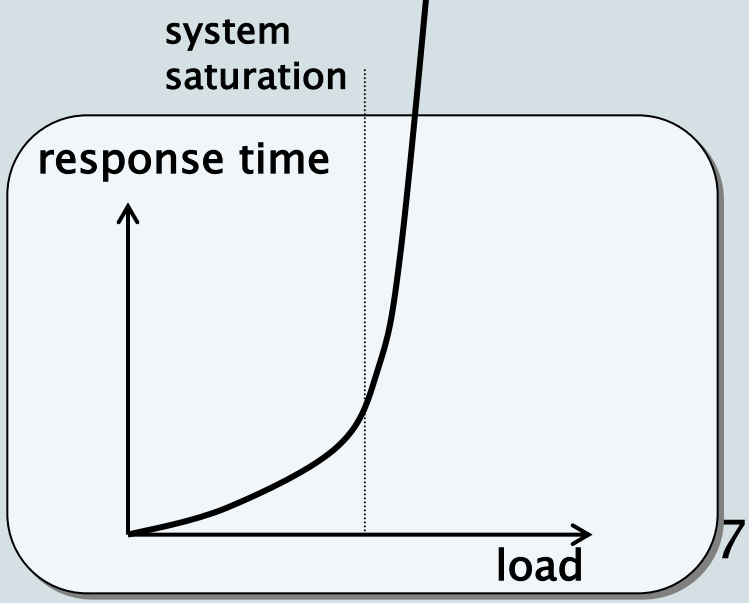
Scalability (definition)

- Scalability = the ability of a system to continue to meet its performance objectives under increasing demand



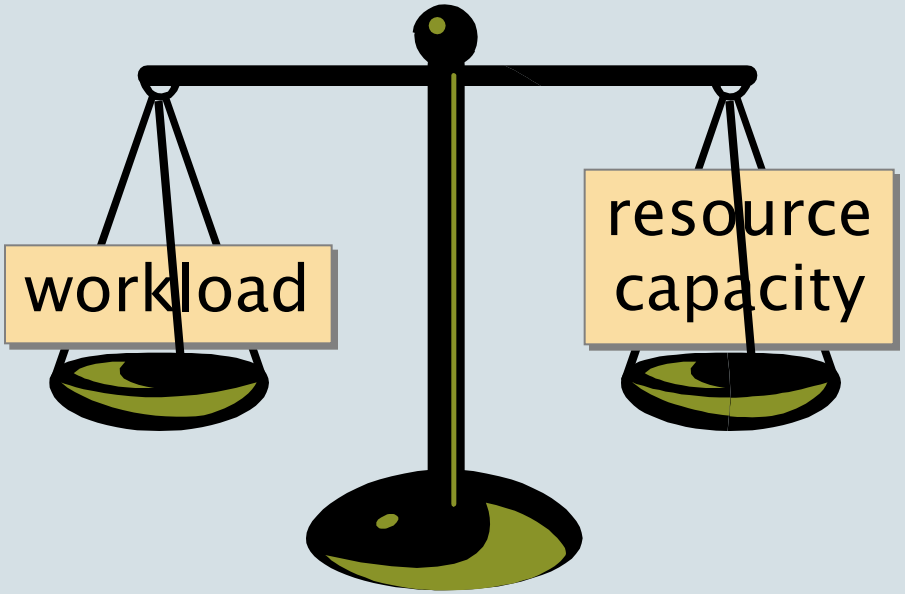
- Scalability is limited by available resources
  - Thrashing is caused by contention for CPU, disk, netw.





Zen of Performance: Balance

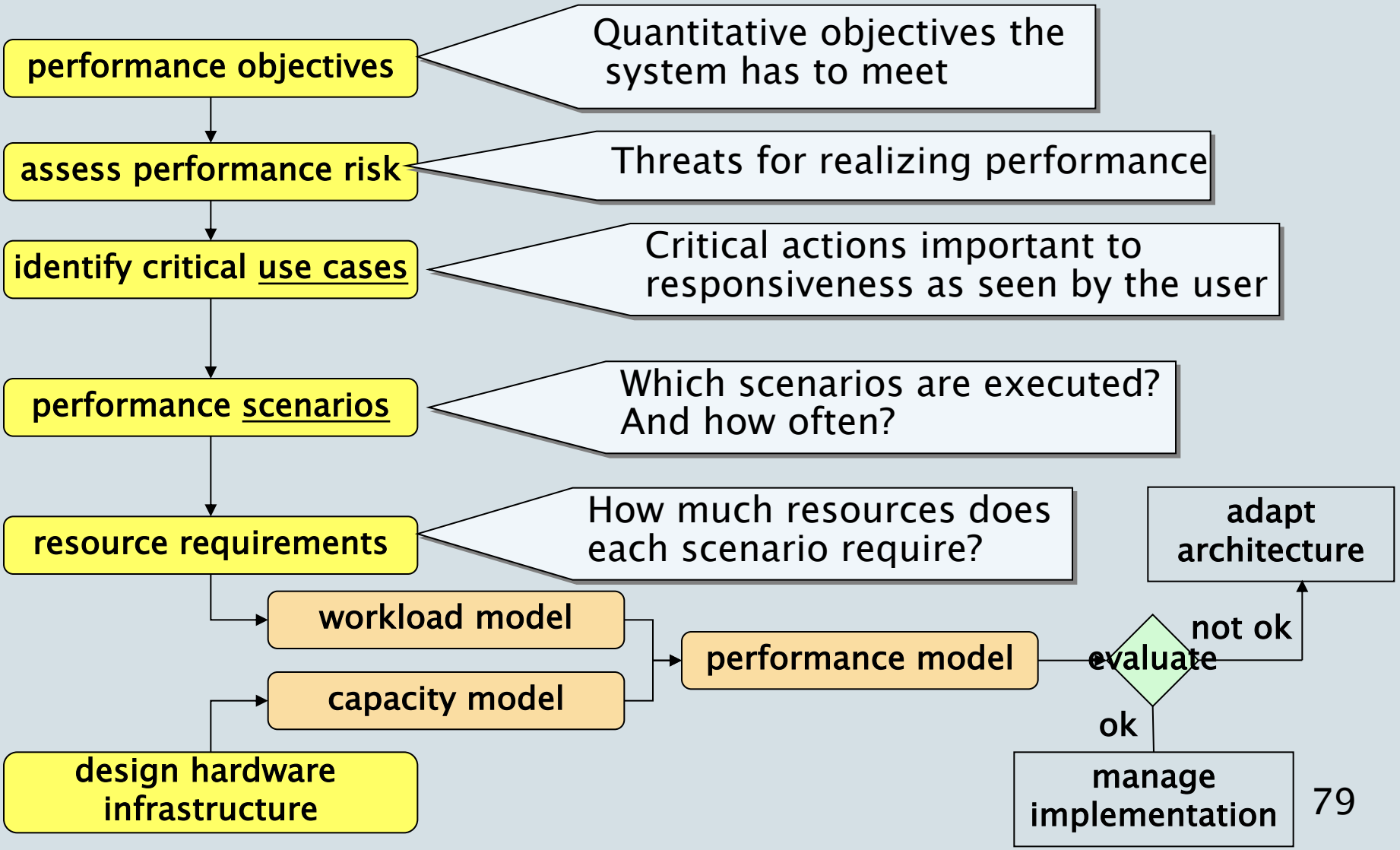
demand



Architect's goal: to establish the (minimum) amount of hardware that will allow the system to meet its performance goals.

Architect's aim: to understand the *trade-off* between resources and performance

Software Performance Engineering Process



Determining workload – Exercise

- How many requests does the university library system receive per hour (during office hours)?
- How many requests does a bank receive per hour for processing transactions / ATM withdrawal requests?
- How many requests does an airline booking system receive for checking availability of seats on flights?
- How many telephone calls go through a base station per hour?

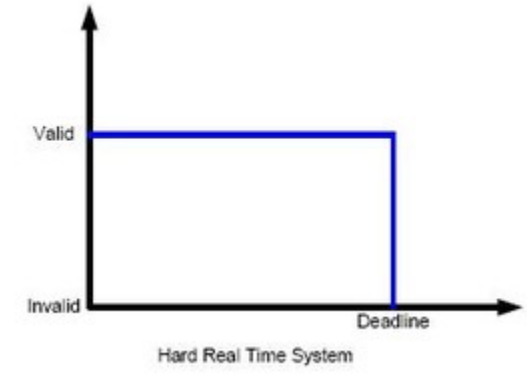
1 Determine Performance Objectives

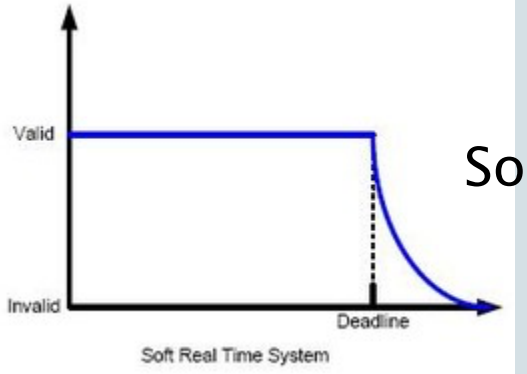
Deduce performance objectives from business objectives

- What makes the system
  - Acceptable (minimum)
  - Attractive
  - Future-proof

  - What is the cost of missing deadlines?

Hard-RT





Soft-RT

2 Guidelines performance risk scenario's

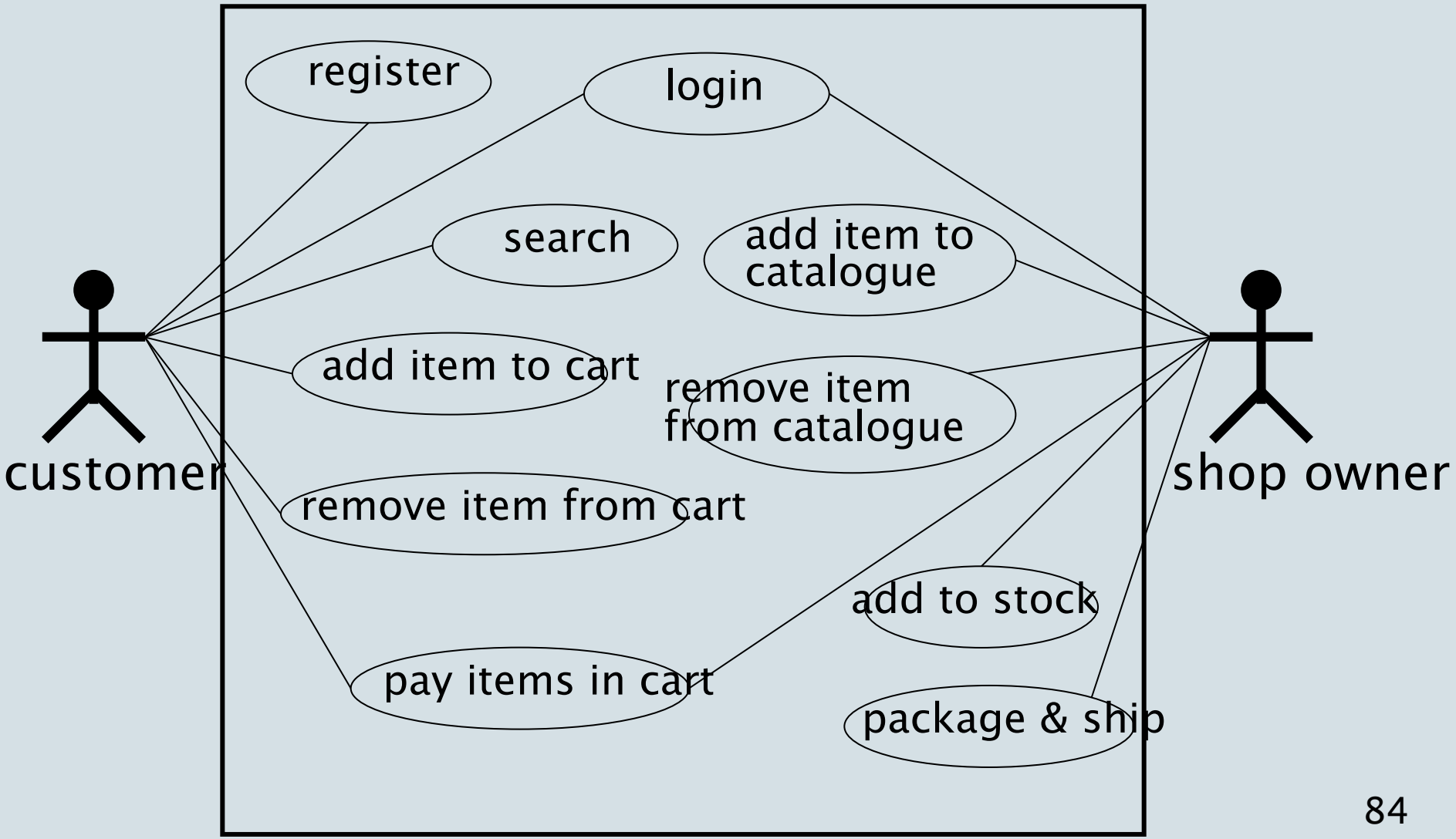
Top-down (by requirements)

- Assumptions on environment/load
- Boundary cases: Peak load
- Combinations of events
- Mode-changes

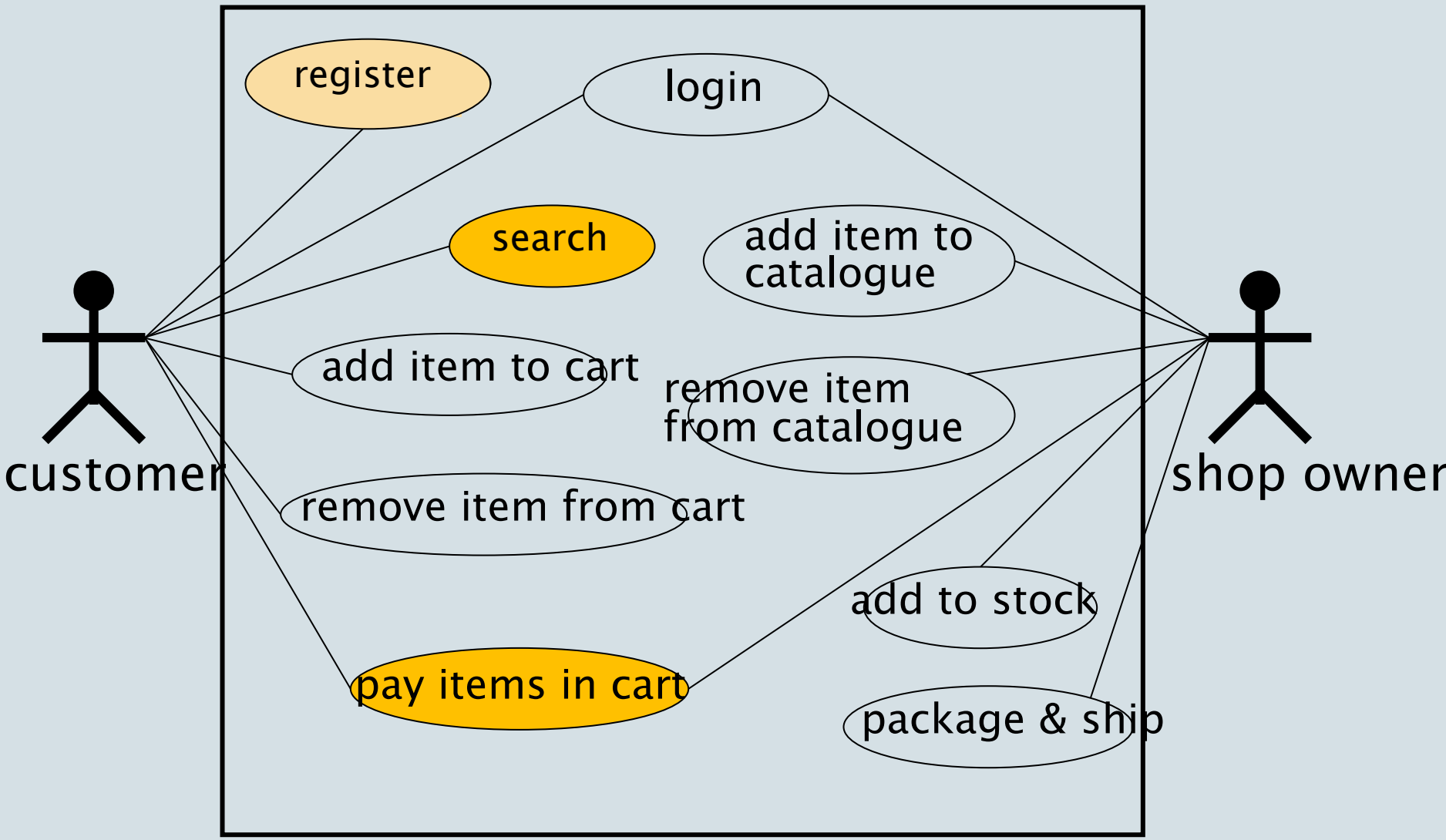
Bottom-up (from solution / technology)

- High utilization
- Interrupts, caching
- Dynamic allocation
- Complex interactions
  - (unexpected) interactions between subsystems
    e.g. between CPU and memory-management
    or between different layers (e.g. software and hardware)

3 Identify Critical Use Cases

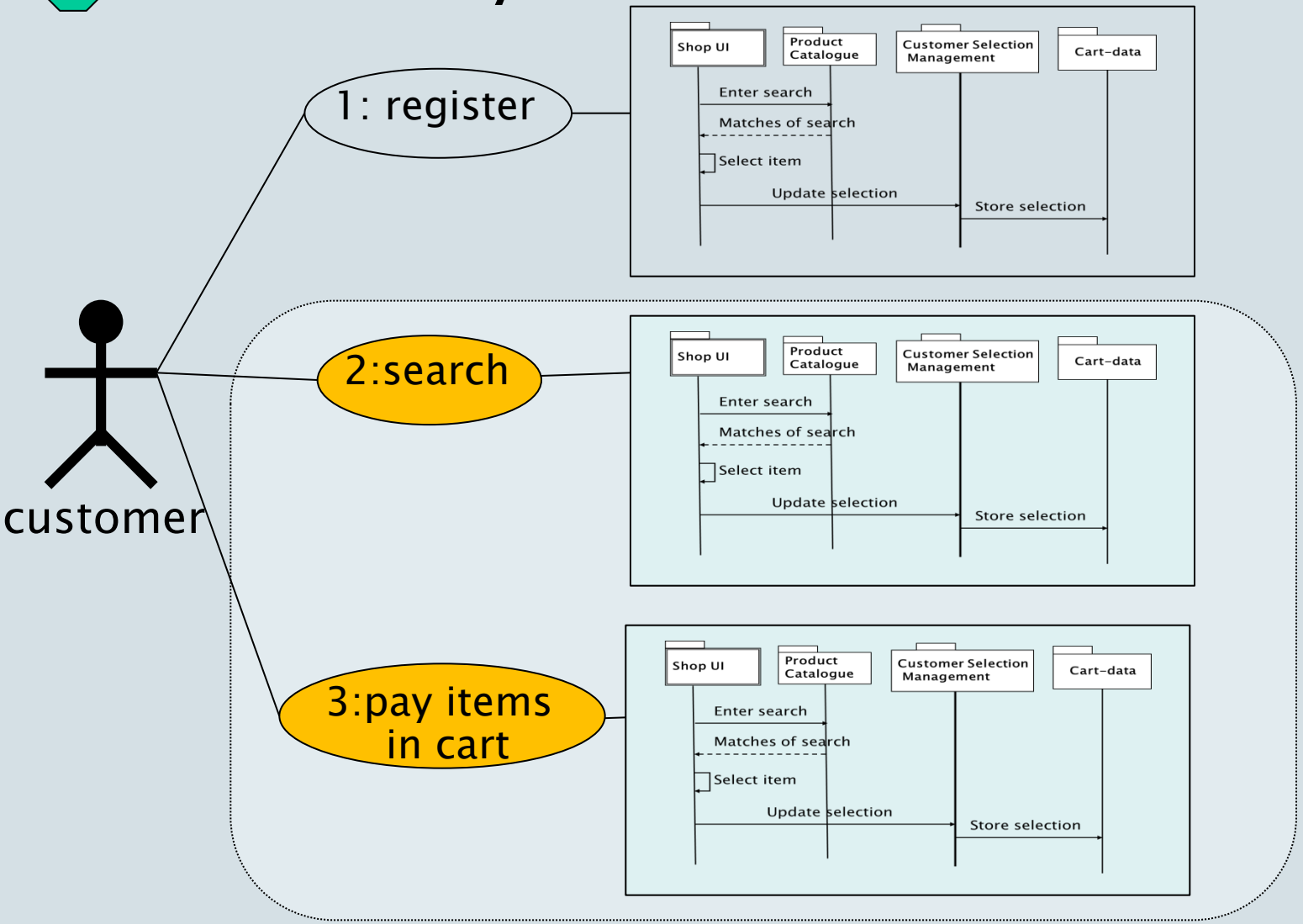


3 Identify Critical Use Cases



4

Identify Performance Scenario's



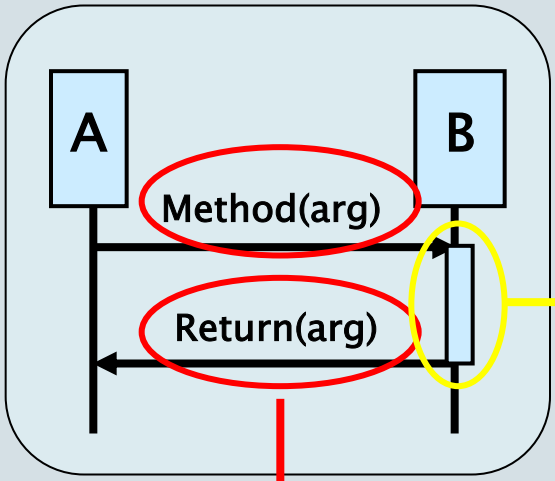
Worst case Scenario: 1000*(2:search) || 100*(3: pay) / sec

5 Determine Software Resource Requirements

- Understand the types of software resource requirements – at the level of the run-time architecture
  - Method Calls
  - Database queries
  - Data Loading:
    - How many files/tables? What is their size?
  - Caching

Discuss this with:
- the software engineer that did the implementation
- the project's performance expert

Derive Load from Sequence Diagrams



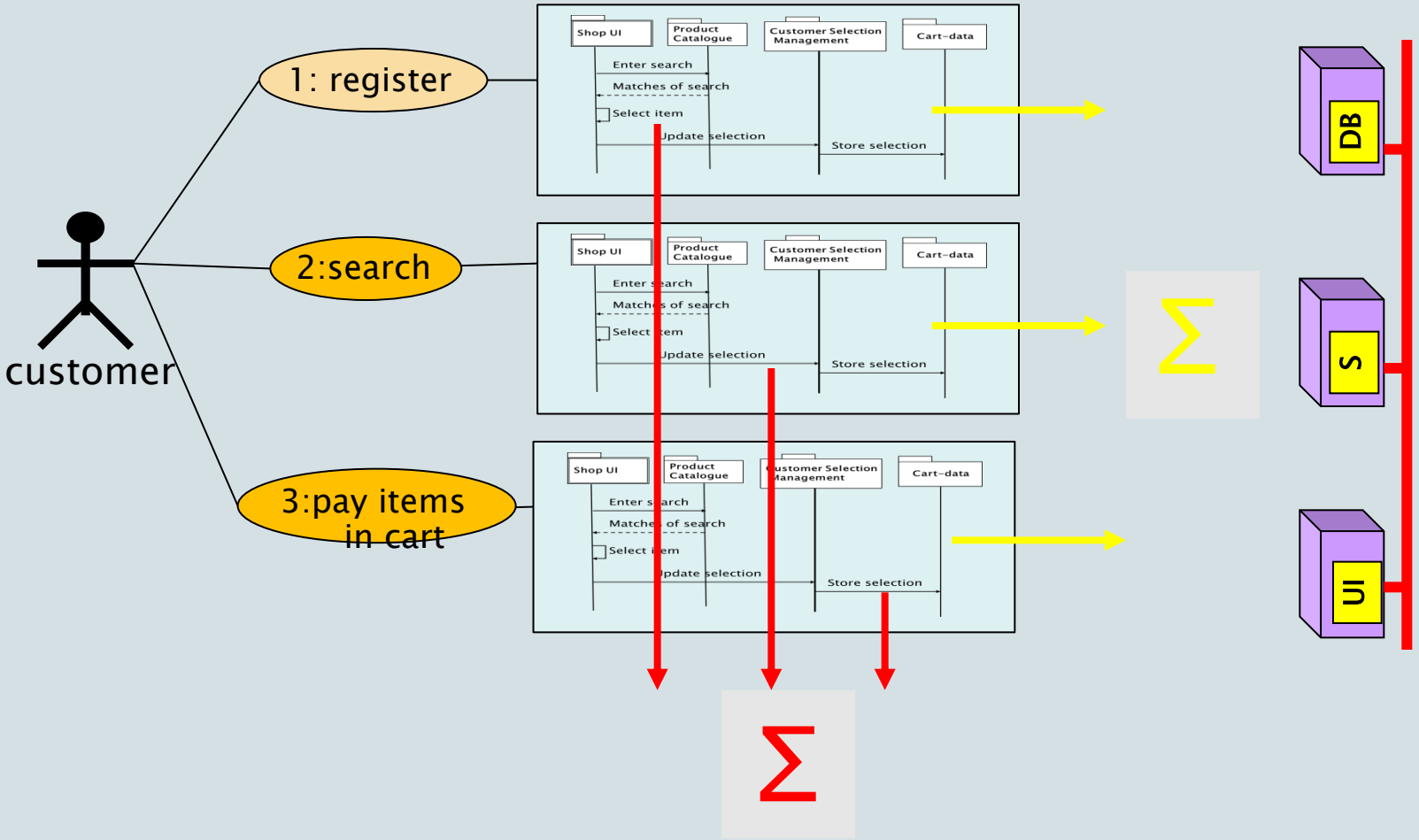
Processing load:
– DB / cache / processor

transactions / sec

Network load

MB's / sec

5 Software Resource Requirements



Compute / simulate, or use a performance-analysis tool!

References on Queueing Networks

- E.D. Lazowska et.al., Quantitative System performance: computer system analysis using queueing networks, Prentice Hall, 1984
- C.H. Sauer and K.M. Chandy, Computer systems performance modeling, Prentice Hall, 1981
- Quantitative System Performance: Computer System Analysis Using Queueing Network Models, Edward D. Lazowska, John Zahorjan, G. Scott Graham, Kenneth C. Sevcik, available at: http://www.cs.washington.edu/homes/lazowska/qsp/

Classic paper

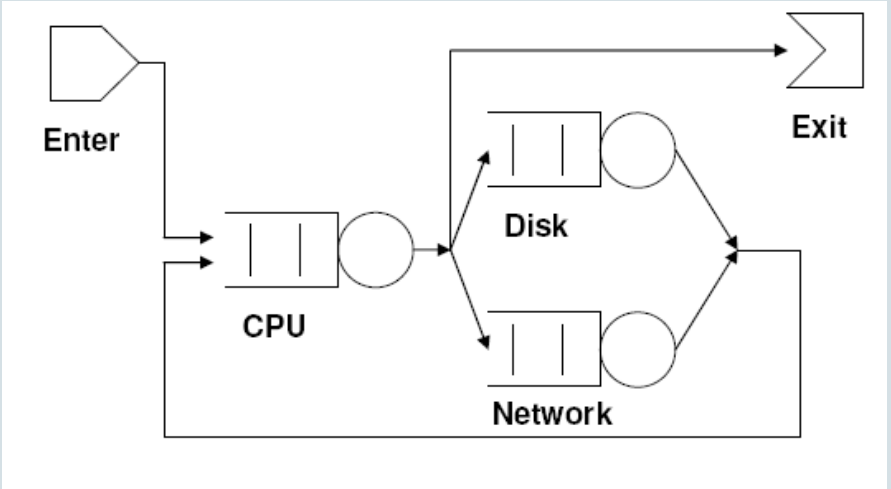
Architecture-based performance analysis, 1998

Spitznagel, Bridget, and David Garlan

http://repository.cmu.edu/cgi/viewcontent.cgi?article=1702&context=compsci

Queuing Networks

- A queuing network processes *jobs*
- The elements of the network are *service centers*, each of which has a queue
- Jobs require service from a set of service centers and wait in a queue when a desired service center is busy.
- Each job exists in only one device or queue at a time



In this Sw. Arch. class:
be able to apply the formulas.

Service Centers

- For example: bank teller, post office, database, …
- Service Center characteristics:
  - **service time**: average time needed for processing a job
  - **arrival rate**: average rate at which jobs arrive

- Usually service time and arrival rate are *assumed* to have an exponential distribution:
  - for an expected value of $1/\lambda$,

    $F(t) = \lambda \cdot e^{-\lambda \cdot t}$

example

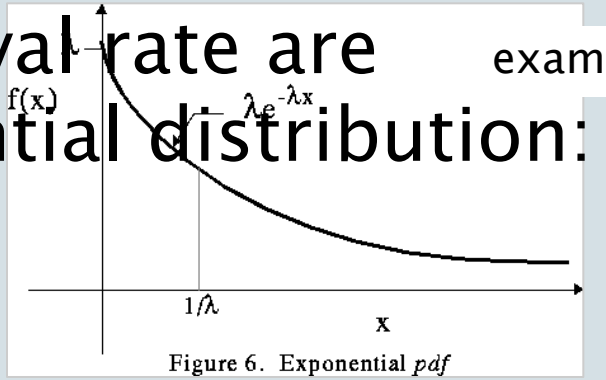


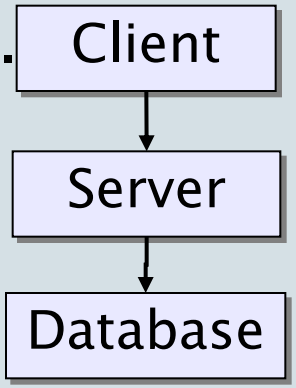
Figure 6. Exponential *pdf*

Architectural Styles and QN

- Assumptions of QN seem to fit well with the following architectural styles:
  - pipe and filter
  - client/server
- We assume messages are processed in FIFO order

QN Example

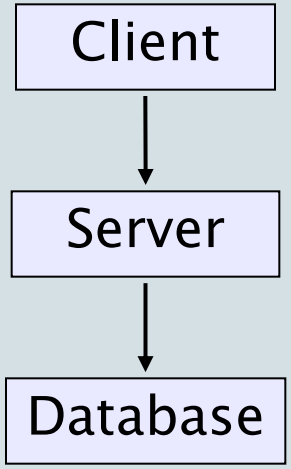
- A job enters the system at the Client and is then serviced by the Client, the Server and the Database. After that it leaves the system.
- Assume
  - An arrival rate of R jobs/second (at the Client)
  - Service times S_C , S_S and S_D respectively



- Utilization of component i: $u_i = R \cdot S_i$
- Avg. component queue length: $q_i = u_i^2 / (1 - u_i)$
- Avg. component response time: $S_i / (1 - u_i)$
- Avg. component population: $p_i = u_i / (1 - u_i)$
- System population: $P = \Sigma\, p_i$
- System response time: $S = P/R$

QN Example

For example if



- arrival rate

  $R = 9.5$ jobs/second

- service times

  $S_C = 65$ ms,

  $S_S = 20$ ms, and

  $S_D = 103$ ms

Then component utilization will be

DB $u_{DB} = 9.5 * 0.103 = 98\%$

Server $u_S = 9.5 * 0.020 = 19\%$

Client $u_C = 9.5 * 0.065 = 62\%$

QN Example

- Utilization ($u_i = R \cdot S_i$)

  Client $u_C = 9.5 * 0.065 = 62\%$

  Server $u_S = 9.5 * 0.020 = 19\%$

  DB $u_{DB} = 9.5 * 0.103 = 98\%$ ← close to overload

- Avg. component response time ($S_i / (1 - u_i)$):

  Client $t_C = 0.065/(1-0.62) = 0.17$ s

  Server $t_S = 0.020/(1-0.19) = 0.025$ s

  DB $t_{DB} = 0.103/(1-0.98) = 5.15$ s ← bottleneck in response time

- Avg. component population ($p_i = u_i / (1 - u_i)$):

  Client $p_C = 0.62/0.38 = 1.6$

  Server $p_S = 0.19/0.81 = 0.23$

  DB $p_{DB} = 0.98/0.02 = 49$

- System population: $P = \Sigma\, p_i = 50.8$
- System response time: $S = P/R = 50.8/9.5 = 5.4$ s

QN Example

- Avg. component queue length: $q_i = u_i 2 / (1 - u_i)$

Client $q_C = 0.38/0.38 = 1.0$
Server $q_S = 0.19/0.81 = 0.044$
DB $q_{DB} = 0.98/0.02 = 48$ ← long queue length

If arrival rate or service time estimates are a little bit off, we're in big trouble.

Options: increase DB performance
replicate DB
reduce demand (arrival rate)

Which option to take?

It depends .... on other non-functional goals, such as cost, scalability, reliability, ..

Automated Software Performance Analysis

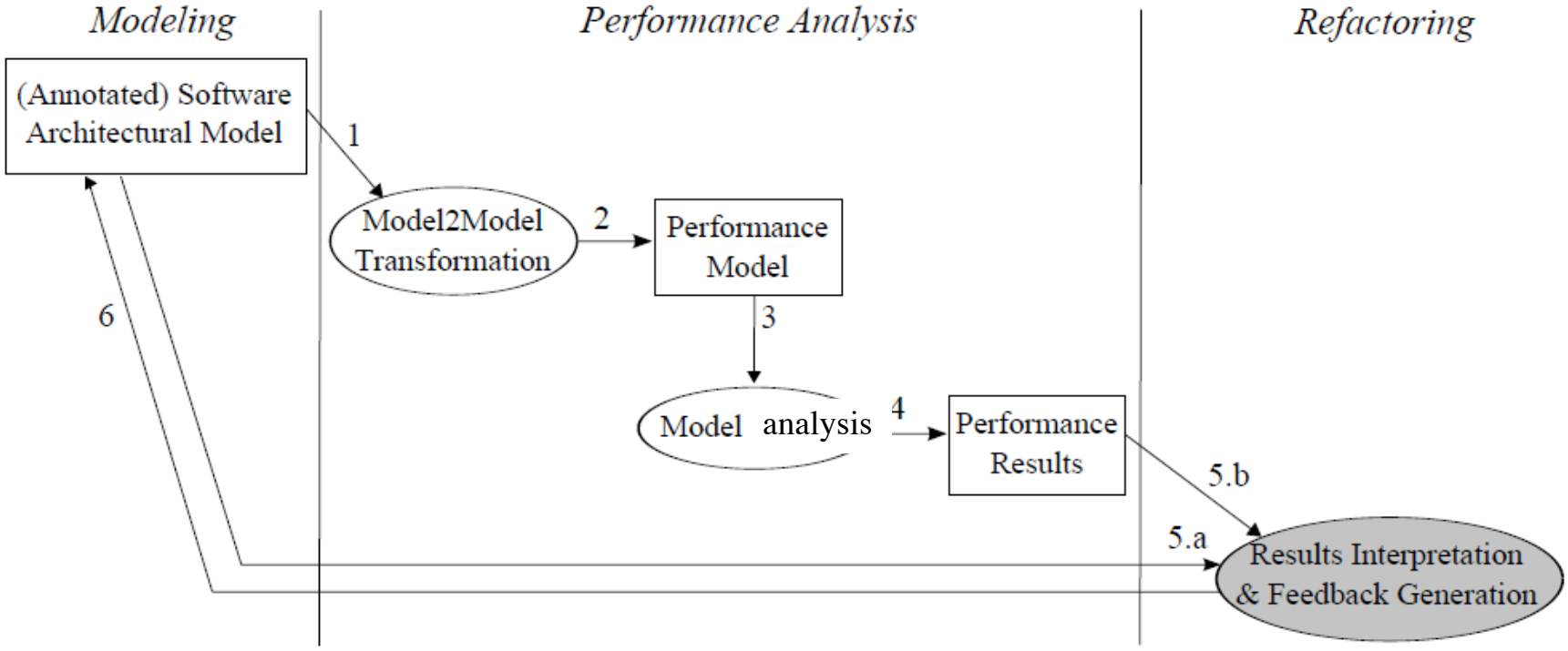


Figure 1.1: Automated software performance process.

From Catania Trubiani, Ph.D. thesis, 2011

1-slide Summary of Best Architecting Practices

- Get stakeholder involvement & feedback early and frequently
- Understand the drivers for the project (business, politics)
- Understand the requirements incl. quality properties
  - SMART & prioritized
- Develop iteratively and incrementally
- Describe architecture using multiple views
  - abstract, but precise, design decisions & rationale
- Monitor that architecture is implemented
- Design for change (modularity, low coupling, infor
- **Simplify, simplify, simplify**
- Analyze in an early stage (use maths! and scenari
- Regularly update planning and risk analysis
- Get good people, make them happy, set them loose

Summary

If you haven't analyzed it, don't build it.

- Use analytical methods to support architectural decision making
  - Performance → Queuing networks

- Many analyses are of 'back of the envelope' size.

  →little effort, lots of value

  → even if your model is not perfect (which they never are)

Conclusion

- Be able to model tactics with UML views
- Reliability Block Diagrams
  - Be able to make simple analyses for small-sized system
- Performance Analysis
  - Goals and steps to perform performance analysis & engineering
  - (Nice to have) Queuing Network