COIS3040 Lecture 9 and 10

Availability

Availability

• **Availability** refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specific time interval.

• (programming) error -> software fault -> software (runtime) failure

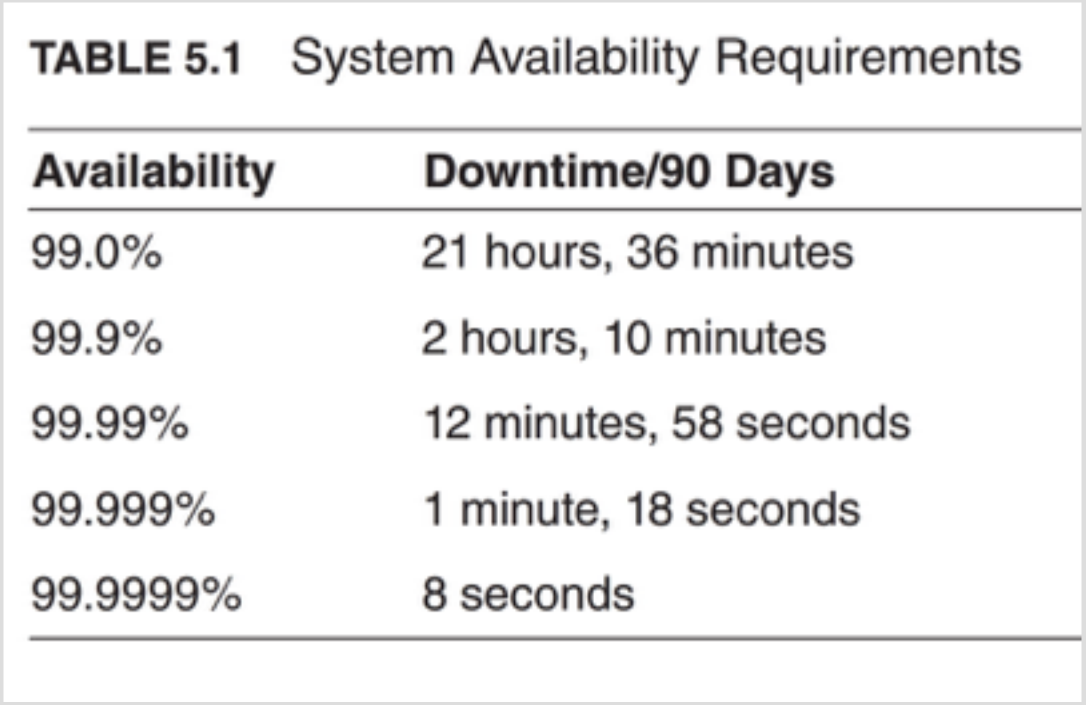
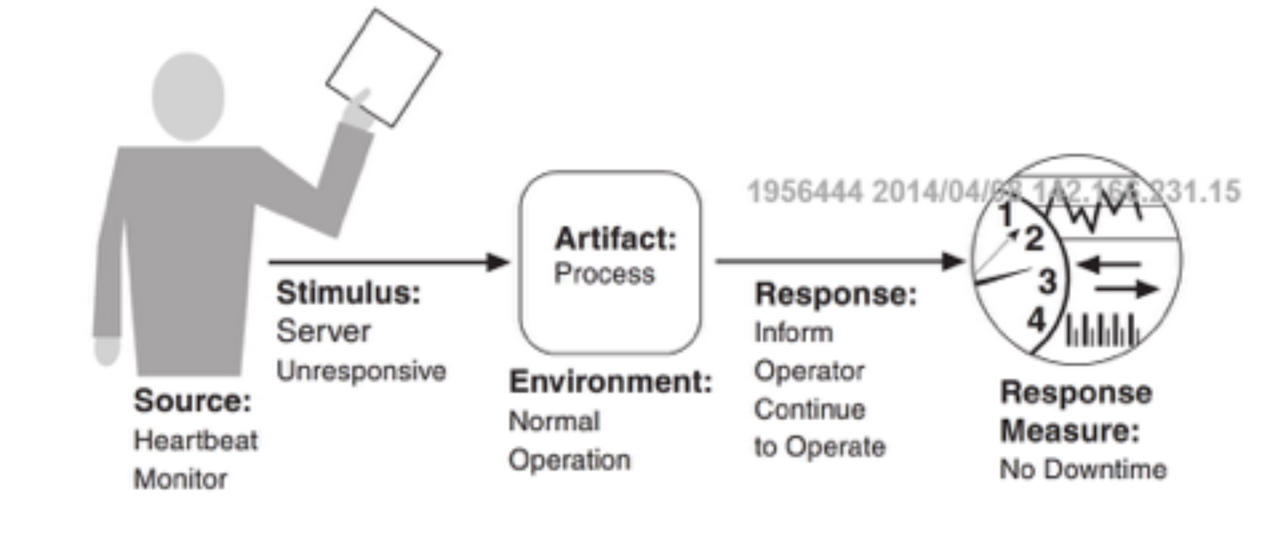


TABLE 5.1 System Availability Requirements

Availability	Downtime/90 Days
99.0%	21 hours, 36 minutes
99.9%	2 hours, 10 minutes
99.99%	12 minutes, 58 seconds
99.999%	1 minute, 18 seconds
99.9999%	8 seconds

Availability



Source: Heartbeat Monitor
Stimulus: Server Unresponsive
Artifact: Process
Environment: Normal Operation
Response: Inform Operator Continue to Operate
Response Measure: No Downtime
1956444 2014/04/08 142.166.231.15
1
2
3
4

FIGURE 5.3 Sample concrete availability scenario

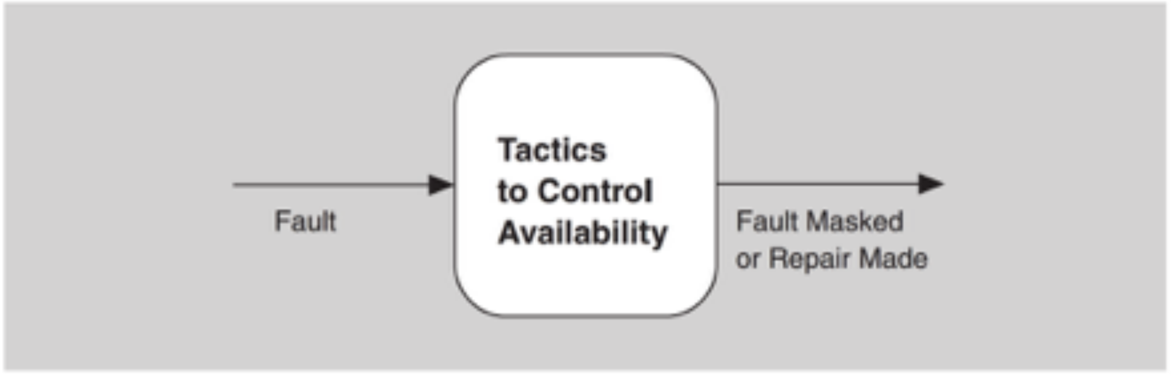


FIGURE 5.4 Goal of availability tactics

Availability

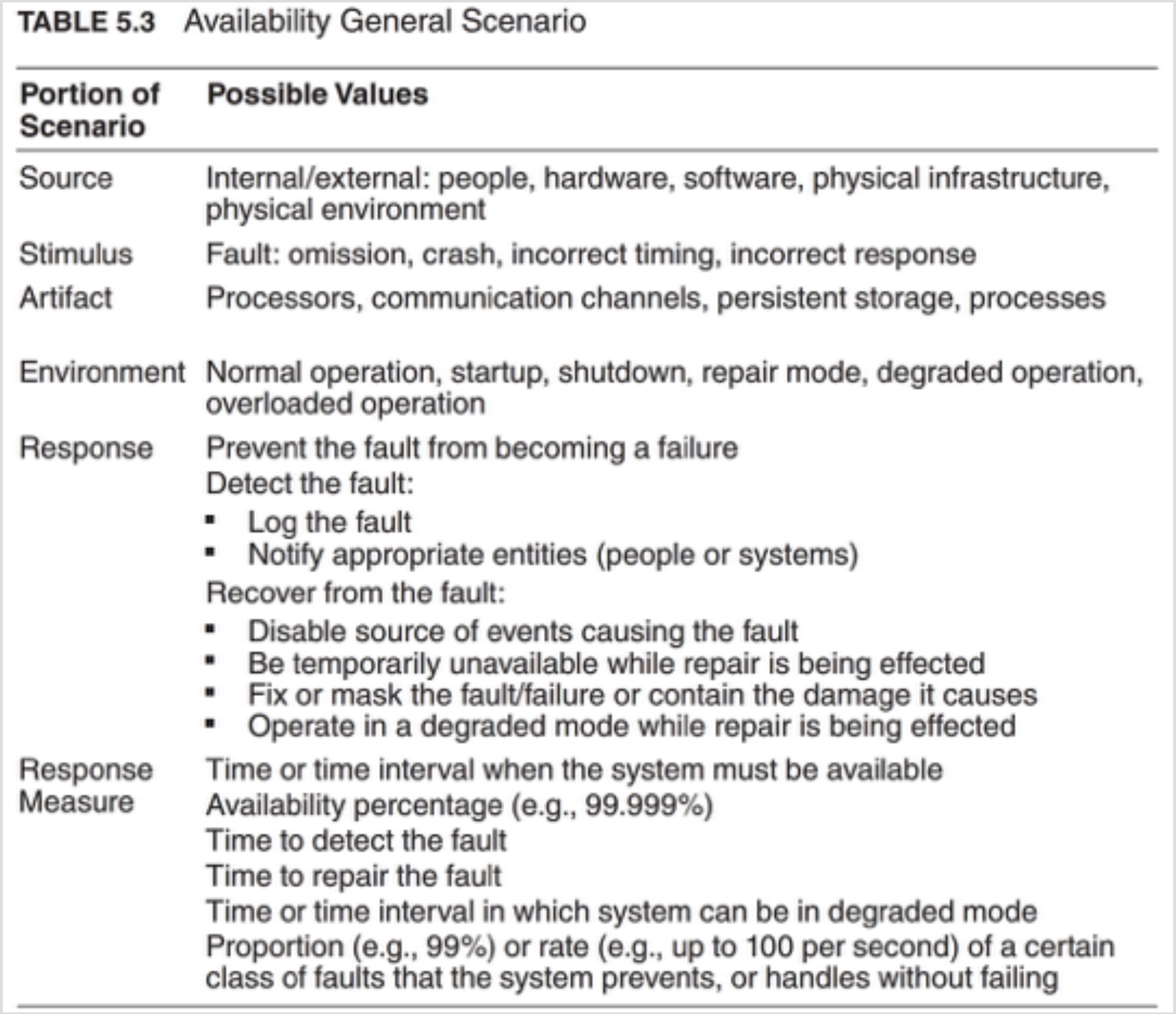


TABLE 5.3 Availability General Scenario

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

Availability

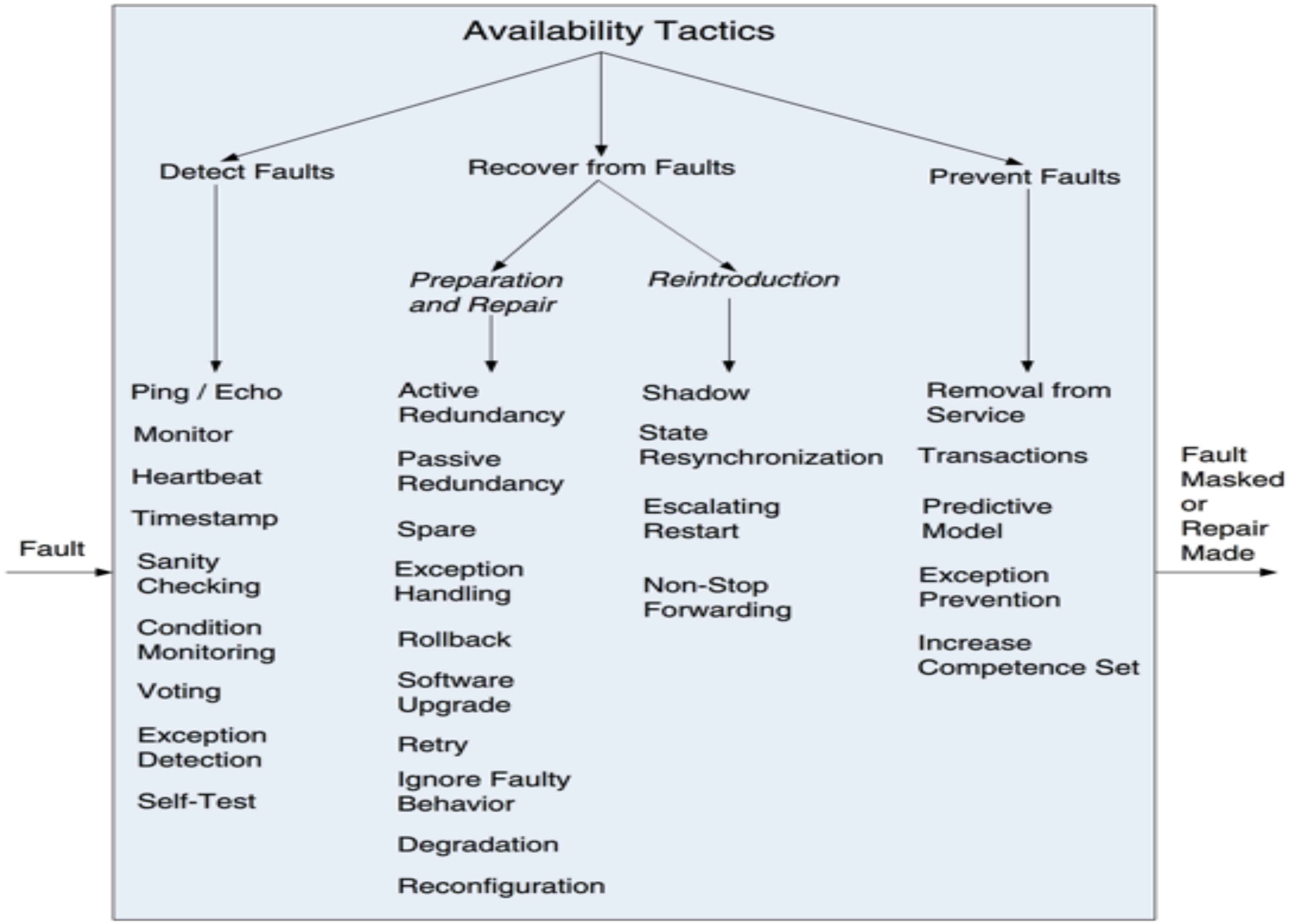


FIGURE 5.5 Availability tactics

Detect Faults

• **Ping/echo:** an asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path.

• **Monitor:** the state health of various other parts of the system: processors, processes, I/O, memory and so on.

• **Heartbeat:** periodic message exchange between a system monitor and a process being monitored.

• **Time stamp:** to detent incorrect sequences of events, primarily in distributed message passing systems.

• **Sanity checking:** checks the validity or reasonableness of specific operations or outputs of a component.

• **Condition monitoring:** checking conditions in a process or device, or validating assumptions made during the design.

• **Voting:** triple modular redundancy employs 3 components that do the same thing, each of which receives identical inputs, and forwards their output to voting logic, used to detect any inconsistency among the three output states.

Detect Faults

- **Voting:**
  - **Replication:** simplest voting, the components are exact clones of each other.
  - **Functional redundancy:** components must always give the same outputs given the same inputs, but they are diversely designed and diversely implemented.
  - **Analytic redundancy:** diversity of both components as well as inputs and outputs.
- **Exception detection:** detection of a system condition that alters the normal flow of execution.
  - **System exception:** hardware exception
  - **Parameter fence**: data patterns for runtime detection of overwriting the memory allocated for a object's variable-length parameters
  - **Parameter typing:** base/super class to define/constrain derived/subclass's variable type for detecting type exceptions.
  - **Timeout:** raise an exception when a component detects that it or another component has failed to meet its timing constraints.
- **Self-test:** run procedures to test themselves for correct operation.

Recover from Faults

Preparation-and-repair:

- **Active redundancy (hot spare):** all nodes (active or redundant spare) in a protection group and process identical input in parallel, allowing the redundant spare(s) to maintain synchronized state with the active node(s).

- **Passive redundancy (warm spare):** only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodical state updates.

- **Spare (cold spare):** the redundant spares of a protection group remain out of service until a fail-over occurs, at which point a power-on-rest procedure is initiated on the redundant spare prior to its being placed in service.

- **Exception handling:** once an exception has been detected, the system must handle it in some fashion.

- **Rollback:** permits the system to revert to a previous known good state, called roll-back line, upon detection of a failure.

- **Software upgrade:** in-service upgrades to executable code images in a non-service-affecting manner.

Recover from Faults

Preparation-and-repair

• **Retry:** assume that the fault that caused a failure is transient and retrying the operation may lead to success.

• **Ignore fault behaviour:** ignoring messages from a particular source when we determine that those messages are spurious.

• **Degradation:** maintains the most critical system functions in the presence of component failures.

• **Reconfiguration:** recover from component failures by reassigning responsibilities to the (Potentially restricted) resources left functioning, which maintaining as much functionality as possible.

Recover from Faults

Reintroduction: *where a failed component is reintroduced after it has been corrected*

• **Shadow:** operating previously failed or in-service upgraded component in a "shadow mode" for a predefined duration of time period to reverting the component back to an active role.

• **State resynchronization:** a reintroduction partner to the active redundancy and passive redundancy

• **Escalating restart:** allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the levels of service affected.

• **Non-stop forwarding:** split to 2 parts: supervisory/control plane and worker/data plane. When the control plane is restarted, it implements graceful restart to incrementally rebuild its functionality and data, while the data plane continues to operate.

Prevent Faults

• **Removal from service:** temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures.

• **Transaction:** leverage transactional semantics to ensure that asynchronous messages exchanged between distributed components are atomic, consistent, isolated, and durable (ACID), by two-phase commit, to prevent race conditions caused by two processes attempting to update the same data item.

• **Predictive model:** monitor the state of health of a system process to ensure that the system is operating within its normal operating parameters, and to take corrective action when conditions are detected that are predictive of likely future faults.

• **Exception prevention:** techniques employed for the purpose of preventing a system exceptions from occurring, to allow a system to transparently recover from system exceptions.

• **Increase competence set:** the set of states in which a component is competent to operate. To design a component to increase its competence set to handle more cases.

Modifiability

What is Modifiability?

- Modifiability is about change and our interest in it is in the cost and risk of making changes.
- To plan for modifiability, an architect has to consider three questions:
  - What can change?
  - What is the likelihood of the change?
  - When is the change made and who makes it?

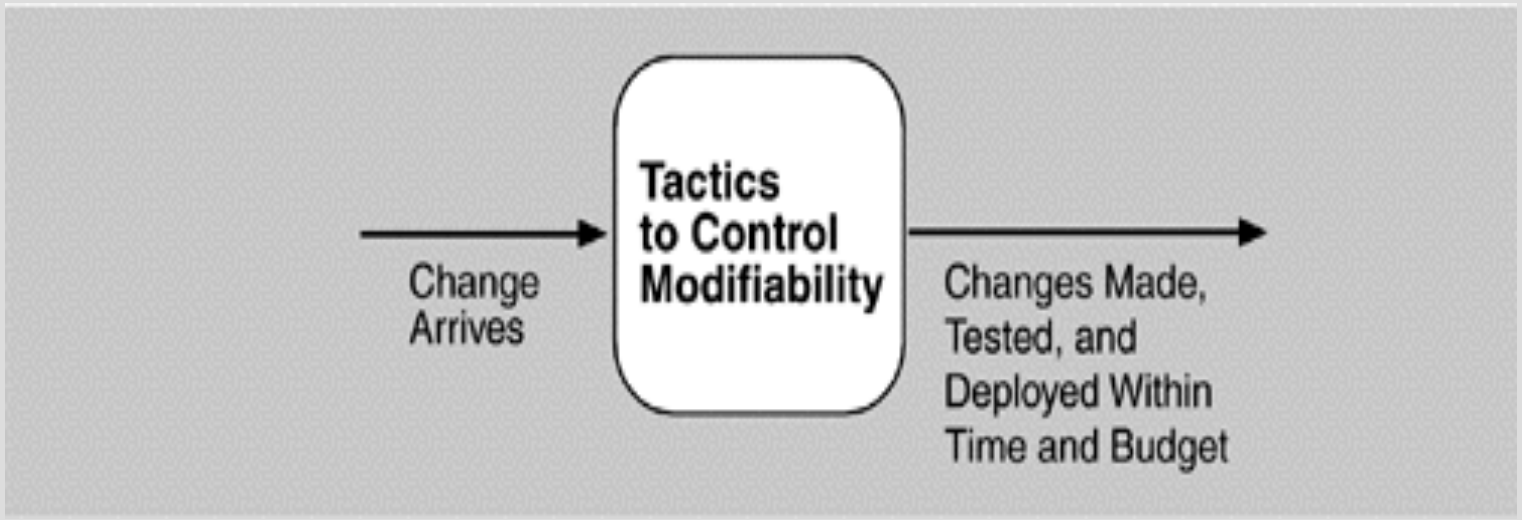
Sample Concrete Modifiability Scenario

- The developer wishes to change the user interface by modifying the code at design time. The modifications are made with no side effects within three hours.

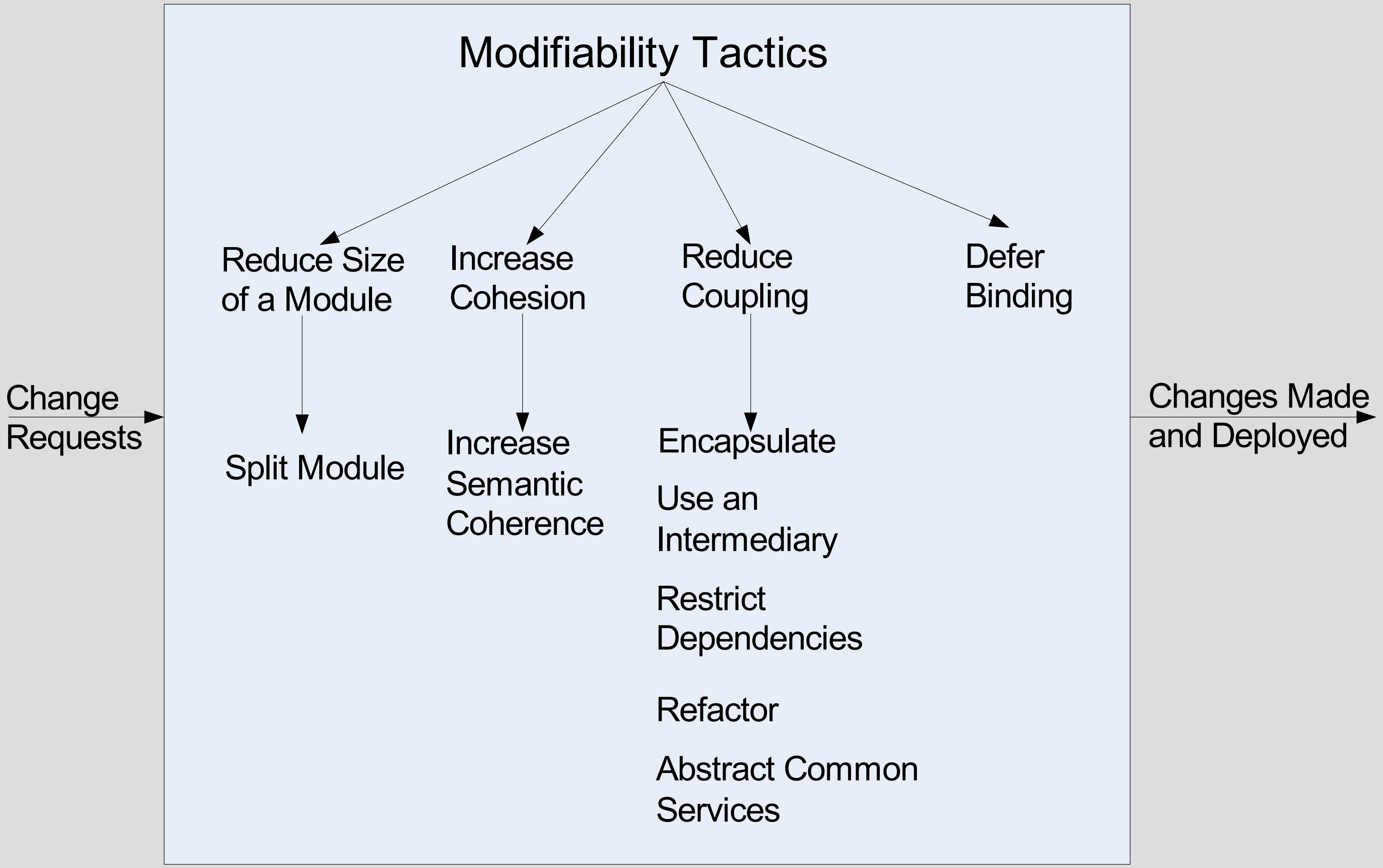
Goal of Modifiability Tactics

- Tactics to control modifiability have as their goal controlling the complexity of making changes, as well as the time and cost to make changes.

Goal of Modifiability Tactics


Change Arrives
Tactics to Control Modifiability
Changes Made, Tested, and Deployed Within Time and Budget

Modifiability Tactics



Cohesion and Coupling

- Cohesion: measures how strongly the responsibilities of a module are related. It measures that probability that a change will only affect small number of modules.
- Coupling: Measures how strongly responsibilities of different modules are related. It measures the probability that a change to a module will propagate to other modules.
- We want high cohesion and low coupling.

Reduce Size of a Module

- Split Module: If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

Increase Cohesion

- Increase Semantic Coherence: If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This may involve creating a new module or it may involve moving a responsibility to an existing module.

Reduce Coupling

- Encapsulate: Encapsulation introduces an explicit interface to a module. This interface includes an API and its associated responsibilities, such as “perform a syntactic transformation on an input parameter to an internal representation.”
- Use an Intermediary: Given a dependency between responsibility A and responsibility B (for example, carrying out A first requires carrying out B), the dependency can be broken by using an intermediary.

Reduce Coupling

- Restrict Dependencies: restricts the modules which a given module interacts with or depends on.
- Refactor: undertaken when two modules are affected by the same change because they are (at least partial) duplicates of each other.
- Abstract Common Services: where two modules provide not-quite-the-same but similar services, it may be cost-effective to implement the services just once in a more general (abstract) form.

Defer Binding

- In general, the later in the life cycle we can bind values, the better.
- If we design artifacts with built-in flexibility, then exercising that flexibility is usually cheaper than hand-coding a specific change.
- However, putting the mechanisms in place to facilitate that late binding tends to be more expensive.

Interoperability

What is Interoperability?

- Interoperability is about the degree to which two or more systems can usefully exchange meaningful information.
- Like all quality attributes, interoperability is not a yes-or-no proposition but has shades of meaning.

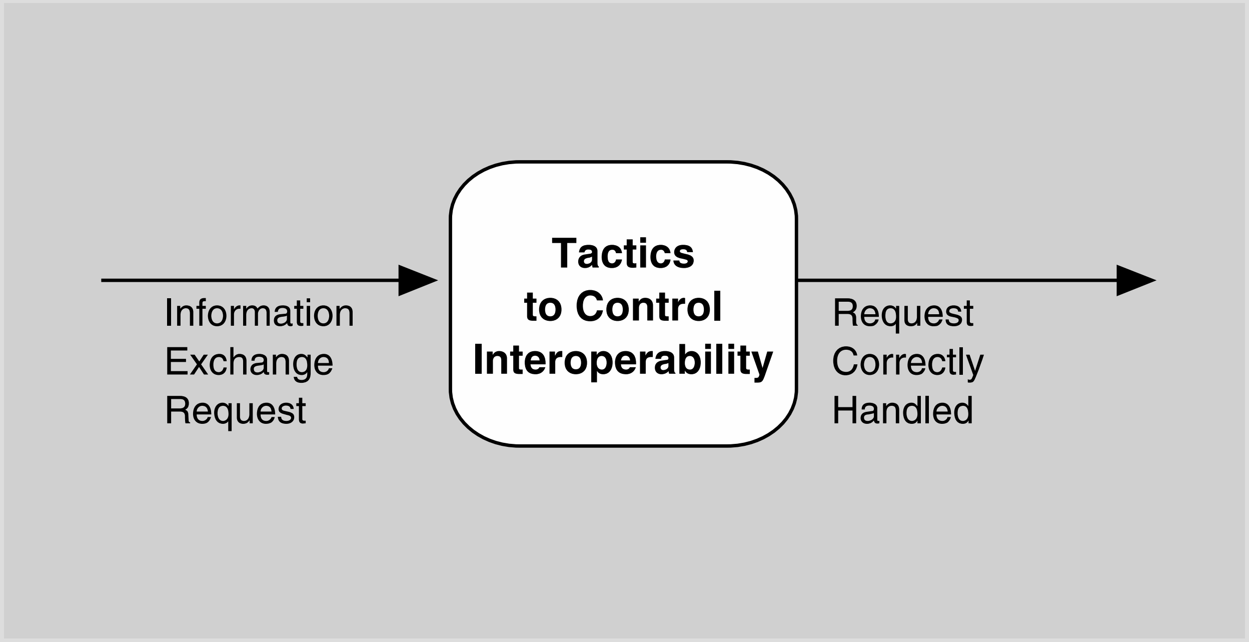
Sample Concrete Interoperability Scenario

- Our vehicle information system sends our current location to the traffic monitoring system. The traffic monitoring system combines our location with other information, overlays this information on a Google Map, and broadcasts it. Our location information is correctly included with a probability of 99.9%.

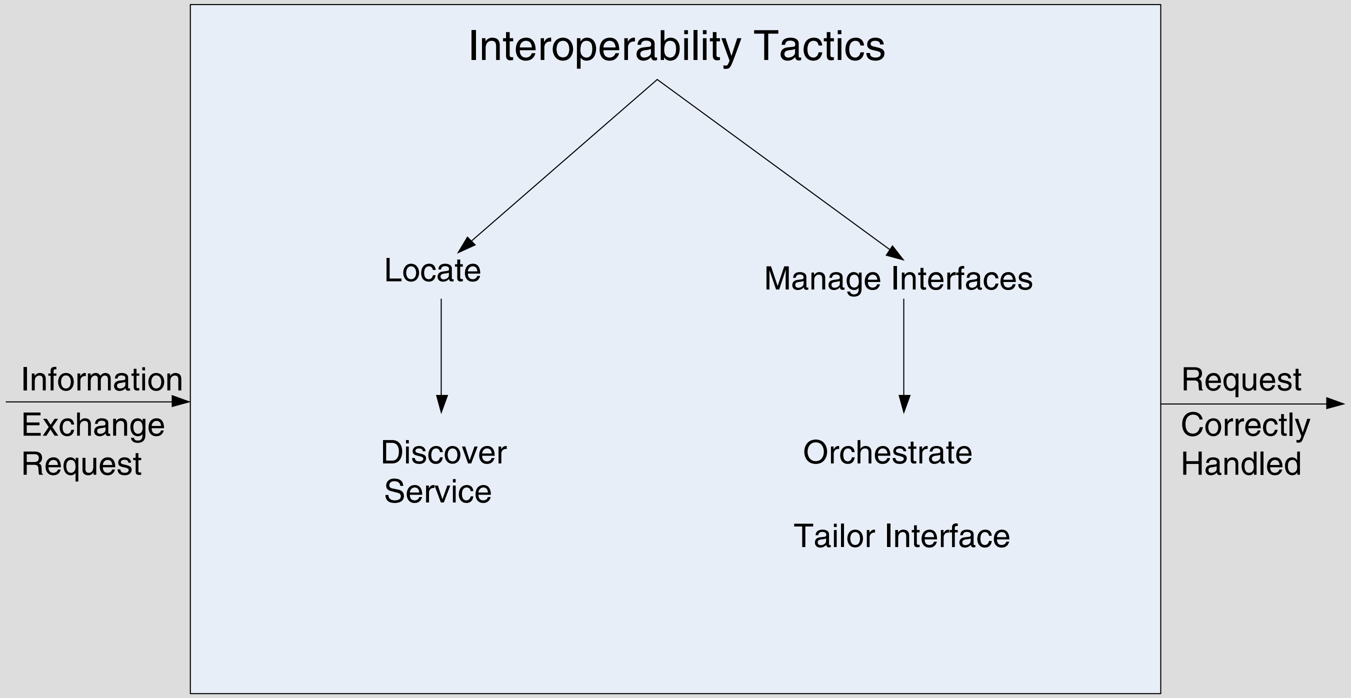
Goal of Interoperability Tactics

- For two or more systems to usefully exchange information they must
  - Know about each other. That is the purpose behind the locate tactics.
  - Exchange information in a semantically meaningful fashion. That is the purpose behind the manage interfaces tactics. Two aspects of the exchange are
    - Provide services in the correct sequence
    - Modify information produced by one actor to a form acceptable to the second actor.

Goal of Interoperability Tactics


Information Exchange Request
Tactics to Control Interoperability
Request Correctly Handled

Interoperability Tactics



Locate

- Discover service: Locate a service through searching a known directory service. There may be multiple levels of indirection in this location process – i.e. a known location points to another location that in turn can be searched for the service.

Manage Interfaces

- Orchestrate: uses a control mechanism to coordinate, manage and sequence the invocation of services. Orchestration is used when systems must interact in a complex fashion to accomplish a complex task.
- Tailor Interface: add or remove capabilities to an interface such as translation, buffering, or data-smoothing.

Manage Interfaces

- Which design pattern can be used for orchestration?

Manage Interfaces

- Which design pattern can be used for orchestration?
- The Mediator pattern.

Manage Interfaces

- Which design pattern can be used for orchestration?
- The Mediator pattern.
- Which design pattern can be used for tailoring interfaces?

Manage Interfaces

- Which design pattern can be used for orchestration?
- The Mediator pattern.
- Which design pattern can be used for tailoring interfaces?
- The decorator pattern.

Web Interoperability

- WS* and SOAP (Simple Object Access Protocol):
- SOAP is a protocol specification for XML-based information that distributed applications can use to exchange and hence interoperate.
- SOAP is often accompanied by SOA middleware interoperability standards referred to as WS*.

Web Interoperability

- REST (Representation State Transfer) is a client-server based architectural style.
- REST is about state transfer and views the web as a huge network accessible by a single URI-based addressing scheme.
- WS* has greater support for security, availability, and so on.
- RESTful implementation is simple and appropriate for read only functionality where there is minimal QoS requirements.

Performance

What is Performance?

- Performance is about time and the software system's ability to meet timing requirements.
- When events occur – interrupts, messages, requests from users or other systems, or clock events marking the passage of time – the system, or some element of the system, must respond to them in time.
- Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence is discussing performance.

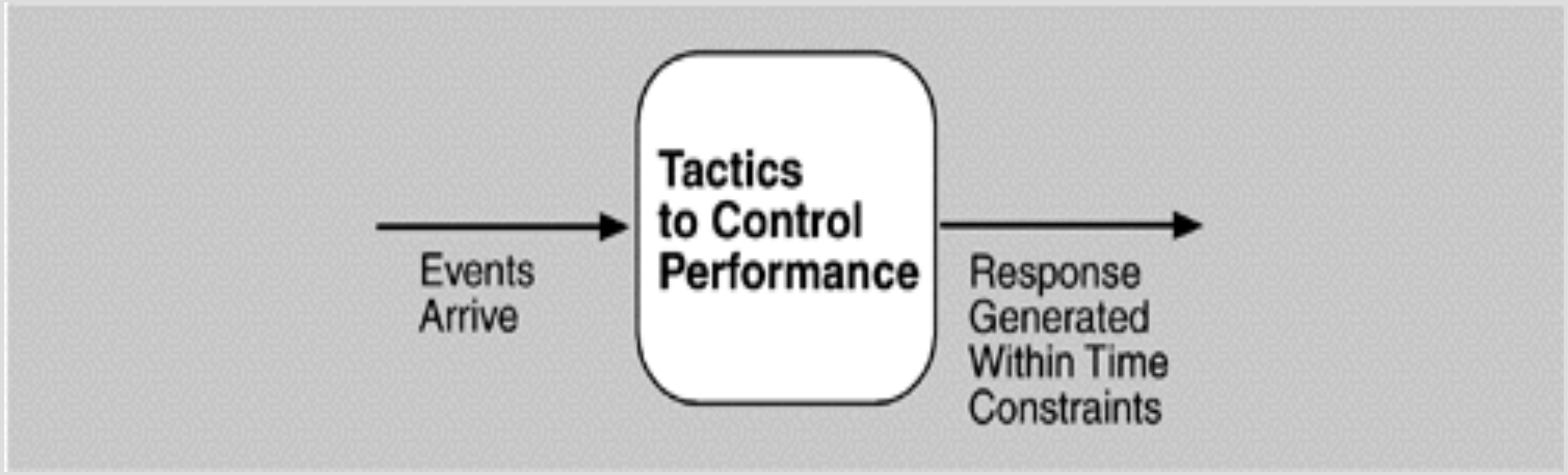
Sample Concrete Performance Scenario

- Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.

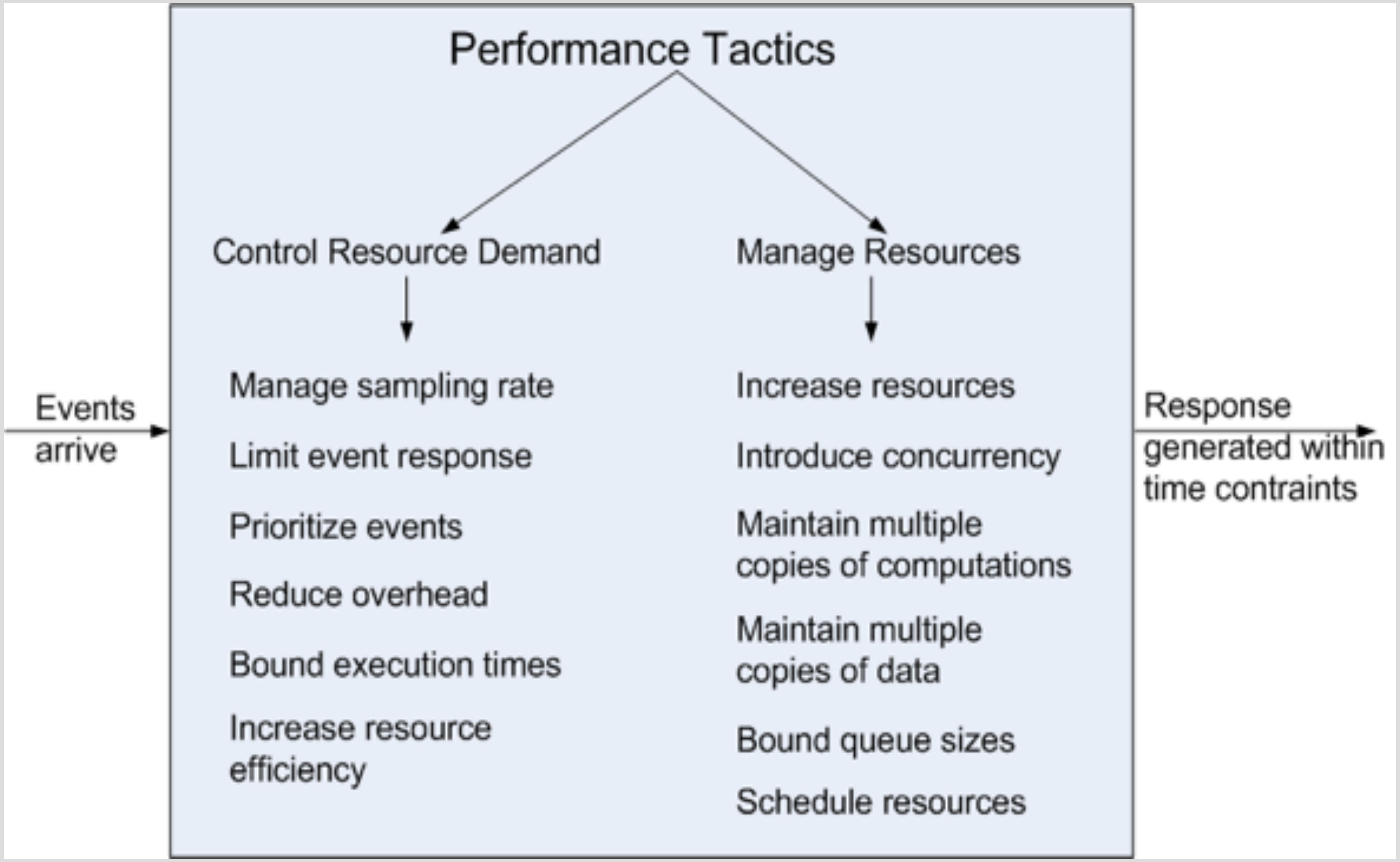
Goal of Performance Tactics

- Tactics to control Performance have as their goal to generate a response to an event arriving at the system within some time-based constraint.

Goal of Performance Tactics



Performance Tactics



Control Resource Demand

- Manage Sampling Rate: If it is possible to reduce the sampling frequency at which a stream of data is captured, then demand can be reduced, typically with some loss of fidelity.
- Limit Event Response: process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed.
- Prioritize Events: If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them.

Control Resource Demand

- Reduce Overhead: The use of intermediaries (important for modifiability) increases the resources consumed in processing an event stream; removing them improves latency.
- Bound Execution Times: Place a limit on how much execution time is used to respond to an event.
- Increase Resource Efficiency: Improving the algorithms used in critical areas will decrease latency.

Manage Resources

- Increase Resources: Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.
- Increase Concurrency: If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities.
- Maintain Multiple Copies of Computations: The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server.

Manage Resources

- Maintain Multiple Copies of Data: keeping copies of data (possibly one a subset of the other) on storage with different access speeds.
- Bound Queue Sizes: control the maximum number of queued arrivals and consequently the resources used to process the arrivals.
- Schedule Resources: When there is contention for a resource, the resource must be scheduled.