Chapter 2: Why is Software Architecture Important?

Inhibiting or Enabling a System's Quality Attributes

- Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.
  - Performance
  - Modifiability
  - Security
  - Scalability
  - Reusability

Reasoning About and Managing Change

- About 80 percent of a typical software system's total cost occurs after initial deployment
  - accommodate new features
  - adapt to new environments,
  - fix bugs, and so forth.
- Every architecture partitions possible changes into three categories
  - A *local* change can be accomplished by modifying a single element.
  - A *nonlocal* change requires multiple element modifications but leaves the underlying architectural approach intact.
  - An *architectural* change affects the fundamental ways in which the elements interact with each other and will require changes all over the system.

Reasoning About and Managing Change

- Obviously, local changes are the most desirable
- A good architecture is one in which the most common changes are local, and hence easy to make.

Predicting System Qualities

- When we examine an architecture we can confidently predict that the architecture will exhibit the associated qualities.
- The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.

Enhancing Communication Among Stakeholders

- The architecture—or at least parts of it—is sufficiently abstract that most nontechnical people can understand it.
- Most of the system's stakeholders can use as a basis for creating mutual understanding, negotiating, forming consensus, and communicating with each other.
- Each stakeholder of a software system is concerned with different characteristics of the system
  - Users, client, manager, architect

Earliest Design Decisions

- Software architecture is a manifestation of the earliest design decisions about a system.
- These early decisions affect the system's remaining development, its deployment, and its maintenance life.
- What are these early design decisions?
  - Will the system run on one processor or be distributed across multiple processors?
  - Will the software be layered? If so, how many layers will there be? What will each one do?
  - Will components communicate synchronously or asynchronously?
  - What communication protocol will we choose?
  - Will the system depend on specific features of the operating system or hardware?
  - Will the information that flows through the system be encrypted or not?

Defining Constraints on an Implementation

- An implementation exhibits an architecture if it conforms to the design decisions prescribed by the architecture.
  - The implementation must be implemented as the set of prescribed elements
  - These elements must interact with each other in the prescribed fashion
- Each of these prescriptions is a constraint on the implementer.

Influencing the Organizational Structure

- Architecture prescribes the structure of the system being developed.
- The architecture is typically used as the basis for the work-breakdown structure.

Enabling Evolutionary Prototyping

- Once an architecture has been defined, it can be prototyped as a skeletal system.
  - A skeletal system is one in which at least some of the infrastructure is built before much of the system's functionality has been created.
- The fidelity of the system increases as prototype parts are replaced with complete versions of these parts.
- This approach aids the development process because the system is executable early in the product's life cycle.
- This approach allows potential performance problems to be identified early in the product's life cycle.
- These benefits reduce the potential risk in the project.

Improving Cost and Schedule Estimates

- Architecture is used to help the project manager create cost and schedule estimates early in the project life cycle.
- Top-down estimates are useful for setting goals and apportioning budgets.
- Bottom-up understanding of the system's pieces are typically more accurate than those that are based purely on top-down system knowledge.
- The best cost and schedule estimates will typically emerge from a consensus between the top-down estimates (created by the architect and project manager) and the bottom-up estimates (created by the developers).

Transferable, Reusable Model

- Reuse of architectures provides tremendous benefits for systems with similar requirements.
  - Not only can code be reused, but also can the requirements that led to the architecture in the first place
  - When architectural decisions can be reused across multiple systems, all of the early-decision consequences are also transferred

Using Independently Developed Components

- Architecture-based development often focuses on components that are likely to have been developed separately, even independently
- Commercial off-the-shelf components, open source software, publicly available apps, and networked services are example of interchangeable software components.
- The payoff can be
  - Decreased time to market
  - Increased reliability
  - Lower cost
  - Flexibility

Restricting Design Vocabulary

- As useful architectural patterns are collected, we see the benefit in restricting ourselves to a relatively small number of choices of elements and their interactions.
  - We minimize the design complexity of the system we are building.
  - Enhanced reuse
  - More regular and simpler designs that are more easily understood and communicated
  - More capable analysis
  - Shorter selection time
  - Greater interoperability.

Basis for Training

- The architecture can serve as the first introduction to the system for new project members.
- Module views show someone the structure of a project
  - Who does what, which teams are assigned to which parts of the system, and so forth.
- Component-and-connector explains how the system is expected to work and accomplish its job.

Summary

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.

Summary

8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that form the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training a new team member.

Chapter 4: Understanding Quality Attributes

Architecture and Requirements

- System requirements can be categorized as:
  - **Functional requirements** state what the system must do, how it must behave or react to run-time stimuli.
  - **Quality attribute requirements** qualify functional requirements, e.g., how fast the function must be performed, how resilient it must be to erroneous input, etc.
  - **Constraints.** A constraint is a design decision with zero degrees of freedom.

Functionality

- **Functionality** is the ability of the system to do the work for which it was intended.
- Functionality has a strange relationship to architecture:
  - functionality does not determine architecture;

Quality Attribute Considerations

- If a functional requirement is "when the user presses the green button the Options dialog appears”:
  - a **performance qualification** might describe how quickly the dialog will appear;
  - an **availability qualification** might describe how often this function will fail, and how quickly it will be repaired;
  - a **usability qualification** might describe how easy it is to learn this function.

Two Categories of Quality Attributes

- The ones that describe some properties of the system at runtime
  - Availability, performance, usability, security
- The ones that describe some properties of the development of system
  - Modifiability
  - Testability

Quality Attribute Considerations

- There are problems with previous discussions of quality attributes:

1. **Untestable definitions.** The definitions provided for an attribute are not testable. It is meaningless to say that a system will be “modifiable”
2. **Overlapping concerns.** Is a system failure due to a denial of service attack an aspect of availability, performance, security, or usability?

Quality Attribute Considerations

- A solution to the problems (untestable definitions and overlapping concerns) is to use *quality attribute scenarios* as a means of characterizing quality attributes.

Specifying Quality Attribute Requirements

- We use a common form to specify all quality attribute requirements as scenarios.
- Our representation of quality attribute scenarios has these parts:
  1. **Stimulus**
  2. **Stimulus source**
  3. **Response**
  4. **Response measure**
  5. **Environment**
  6. **Artifact**

Specifying Quality Attribute Requirements

1. **Stimulus**. The stimulus is a condition that requires a response when it arrives at a system.
2. **Source of stimulus**. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
3. **Response**. The response is the activity undertaken as the result of the arrival of the stimulus.
4. **Response measure**. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.
5. **Environment**. The stimulus occurs under certain conditions. The system may be in an overload condition or in normal operation, or some other relevant state.
6. **Artifact**. This may be a collection of systems, the whole system, or some piece or pieces of it. Some artifact is stimulated.

Specifying Quality Attribute Requirements

- *General* quality attribute scenarios are system independent and can, potentially, pertain to any system
- *Concrete* quality attribute scenarios are specific to the particular system under consideration.

Specifying Quality Attribute Requirements

- Example general scenario for availability:

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

Response:
Prevent fault from becoming failure
Detect fault: log, notify
Recover from fault: disable event source, be unavailable, fix/mask, degraded mode

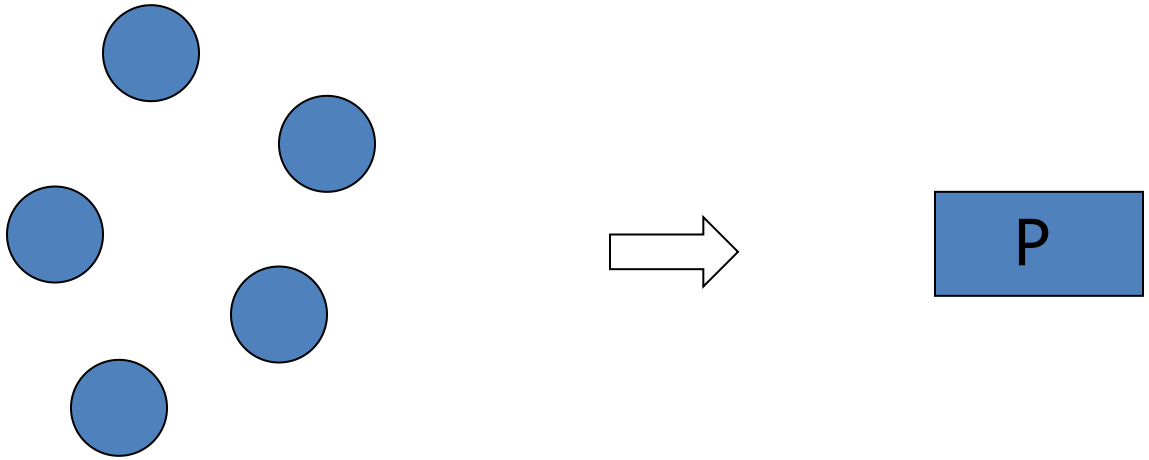
Response Measure:
Time system must be available
Availability percentage
Time in degraded mode
Repair time
Proportion of faults system handles

Achieving Quality Attributes Through Tactics

- There are a collection of primitive design techniques that an architect can use to achieve a quality attribute response.
- We call these architectural design primitives *tactics*.
- Tactics, like design patterns, are techniques that architects have been using for years.
- We do not *invent* tactics, we simply capture what architects do in practice.

Tactics: examples

- Resource scheduling is a tactic for performance



Given: *release time, workload of each task*

To determine **when** each task is executed

Objectives: to minimize *average completion time of the tasks*

Tactic v.s. Architectural pattern

- A tactic is a design decision for a single quality attribute
- A tactic does not consider tradeoffs among quality attributes
- Architectural patterns can be seen as “packages” of tactics, in which tradeoffs are considered

Achieving Quality Attributes Through Tactics

- We need to isolate, catalog, and describe the tactics. Why do we do this?
  1. Design patterns are complex, and often difficult to apply as is; architects need to modify and adapt them.
  2. If no pattern exists to realize the architect's design goal, tactics allow the architect to construct a design fragment from "first principles".
  3. By cataloguing tactics, we will have a choice of multiple tactics to improve a particular quality attribute. The choice of which tactics to use depends on factors such as the cost to implement.

Guiding Quality Design Decisions

- Architecture design is a systematic approach to making design decisions.
- We categorize the design decisions as follows:
  1. Allocation of responsibilities
  2. Coordination model
  3. Data model
  4. Management of resources
  5. Mapping among architectural elements
  6. Binding time decisions
  7. Choice of technology

Allocation of Responsibilities

- Decisions involving allocation of responsibilities include:
  - identifying the responsibilities including *basic system functions*, *architectural infrastructure*, and *satisfaction of quality attributes*.
  - determining how these responsibilities are allocated to non-runtime and runtime elements (namely, modules, components, and connectors).

Coordination Model

- Decisions about the coordination model include:
  - identify the elements of the system that must coordinate, or are prohibited from coordinating
  - determining the properties of the coordination, such as timeliness, currency, correctness, and consistency
  - choosing the communication mechanisms that realize those properties.
    - stateful vs. stateless,
    - synchronous vs. asynchronous,
    - guaranteed vs. non-guaranteed delivery, and

Data Model

- Decisions about the data model include:
  - choosing the major data abstractions, their operations, and their properties.
  - metadata needed for consistent interpretation of the data
  - organization of the data, i.e., to decide whether the data is going to be kept in a relational data base, a collection of objects or both

Management of Resources

- Decisions for management of resources include:
  - identifying the resources that must be managed and determining the limits for each
  - determining which system element(s) manage each resource
  - determining how resources are shared and the strategies employed when there is contention
  - determining the impact of saturation on different resources.

Mapping Among Architectural Elements

- Useful mappings include:
  - the mapping of modules and runtime elements to each other
  - the assignment of runtime elements to processors
  - the assignment of items in the data model to data stores
  - the mapping of modules and runtime elements to units of delivery

Binding Time

- The decisions in the other categories have an associated binding time decision. Examples of such binding time decisions include:
  - Build-time v.s. run-time
  - For choice of coordination model you can design run-time negotiation of protocols.
  - For resource management you can design a system to accept new peripheral devices plugged in at run-time.

Choice of Technology

- Choice of technology decisions involve:
  - deciding which technologies are available to realize the decisions made in the other categories
  - determining whether the tools to support this technology (IDEs, simulators, testing tools, etc.) are adequate
  - determining the extent of internal familiarity and external support for the technology (such as courses, tutorials, examples, availability of contractors)
  - determining the side effects of choosing a technology such as a required coordination model or constrained resource management opportunities
  - determining whether a new technology is compatible with the existing technology stack

Summary

- Requirements for a system come in three categories.
  1. Functional. These requirements are satisfied by including an appropriate set of responsibilities within the design.
  2. Quality attribute. These requirements are satisfied by the structures and behaviors of the architecture.
  3. Constraints. A constraint is a design decision that's already been made.

Summary

- To express a quality attribute requirement we use a quality attribute scenario. The parts of the scenario are:
  1. Source of stimulus.
  2. Stimulus
  3. Environment.
  4. Artifact.
  5. Response.
  6. Response measure.

Summary

- An architectural tactic is a design decision that affects a quality attribute response. The focus of a tactic is on a single quality attribute response.
- Architectural patterns can be seen as “packages” of tactics.
- The seven categories of architectural design decisions are:
  1. Allocation of responsibilities
  2. Coordination model
  3. Data model
  4. Management of resources
  5. Mapping among architectural elements
  6. Binding time decisions
  7. Choice of technology

Class Review

1. Which one of the following statement is NOT true about software structures?

- ✓ A. Software systems are composed of many structures, and no single structure holds claim to being the architecture
- ✓ B. The module structure shows how each of the software elements behaves at run time
- ✓ C. The component & connector structure shows how the software elements are interacted at run time
- ✓ D. The deployment structure shows information about at which physical hardware the software elements are executed

2. Which one of the following structure pertains to a component & connector structure?

- ✓ A. Decomposition structure
- ✓ B. Class structure
- ✓ C. Layer structure
- ✓ D. Concurrency structure

3. Which of the following should be included in the consideration of a Module Structure?

- ✓ A. What are the major executing components and how do they interact at runtime
- ✓ B. what part of the system can run in parallel
- ✓ C. What is the primary functional responsibility assigned to each software elements
- ✓ D. How does data process through the system

4. Which one of the following structures pertains to the Allocation Structure?

- ✓ A. Service structure
- ✓ B. Concurrency structure
- ✓ C. Layer structure
- ✓ D. Deployment structure

Chapter 5: Availability

What is Availability?

- **Availability** refers to a property of software that it is there and ready to carry out its task when you need it to be.
- **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 specified time interval.
- **Availability** is about minimizing service outage time by mitigating faults

Availability

- Availability v.s. reliability or dependability
  - Availability encompasses what is normally called reliability.
  - Availability encompasses other consideration such as service outage due to period maintenance
- Availability is closely related to
  - security, e..g, denial-of-service
  - performance
  - …

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	System's 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

Sample Concrete Availability Scenario

- The heartbeat monitor detects that the server is nonresponsive during normal operations. The system informs the operator and continues to operate with no downtime.
  - **Stimulus**: non-responsiveness
  - **Response**: inform the operator
  - **Response measure**: no downtime, or 100 availability percentages
  - **Environment**: normal operation
  - **Artifact**: heartbeat monitor
  - **Stimulus source**: server

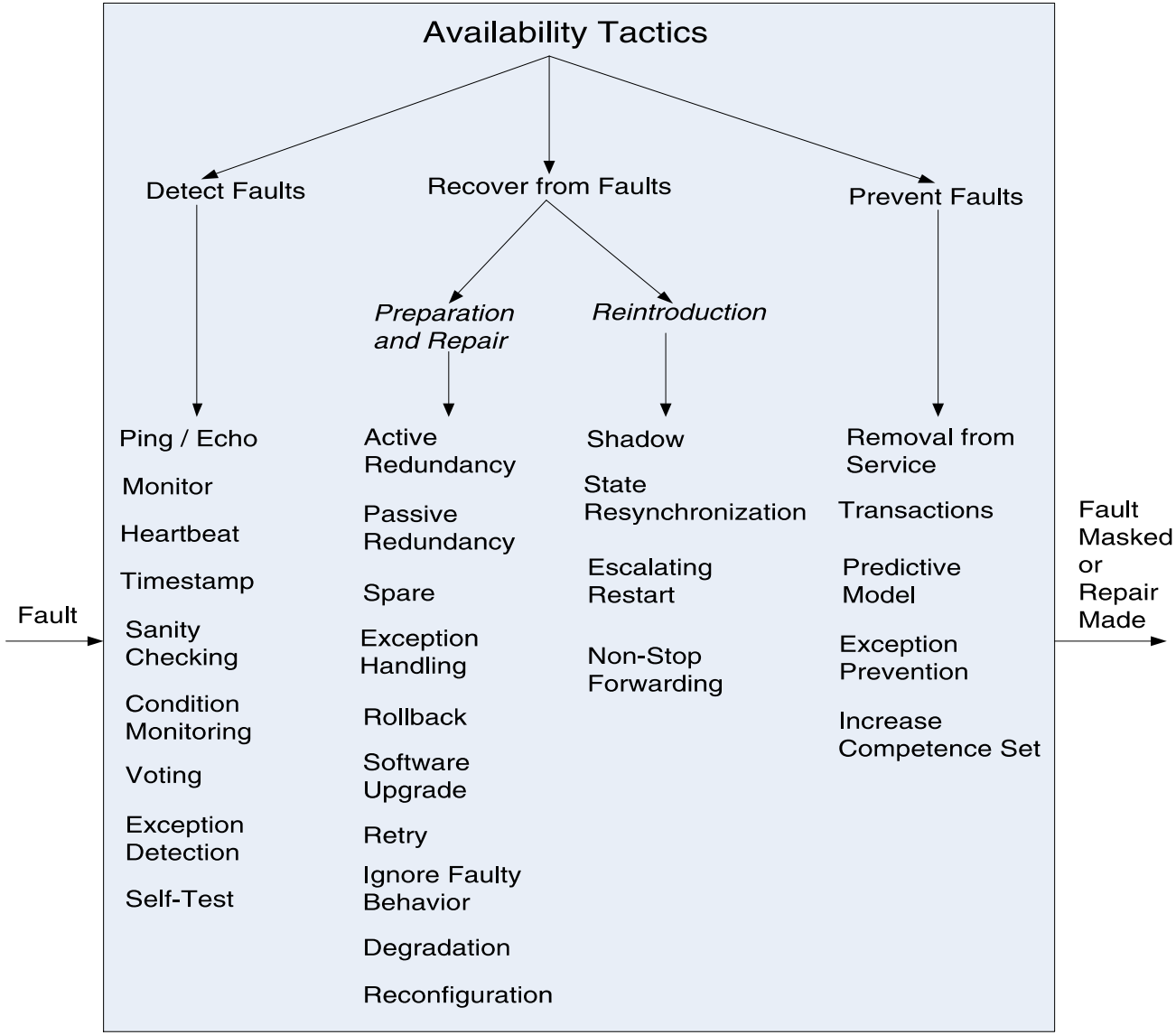
Goal of Availability Tactics

- Fault v.s. failure ?
- A **failure** occurs when the system no longer delivers a service consistent with its specification
  - this failure is observable by the system's actors.
- A **fault** (or combination of faults) has the potential to cause a failure.

Goal of Availability Tactics

- **Availability tactics** enable a system to endure faults so that services remain compliant with their specifications.
- **The tactics** keep faults from becoming failures or at least bound the effects of the fault and make repair possible.

Availability Tactics



Detect Faults

- **Ping/echo:** used to determine reachability and the round-trip delay through the associated network path.
- **Monitor**: a component used to monitor the state of health of other parts of the system

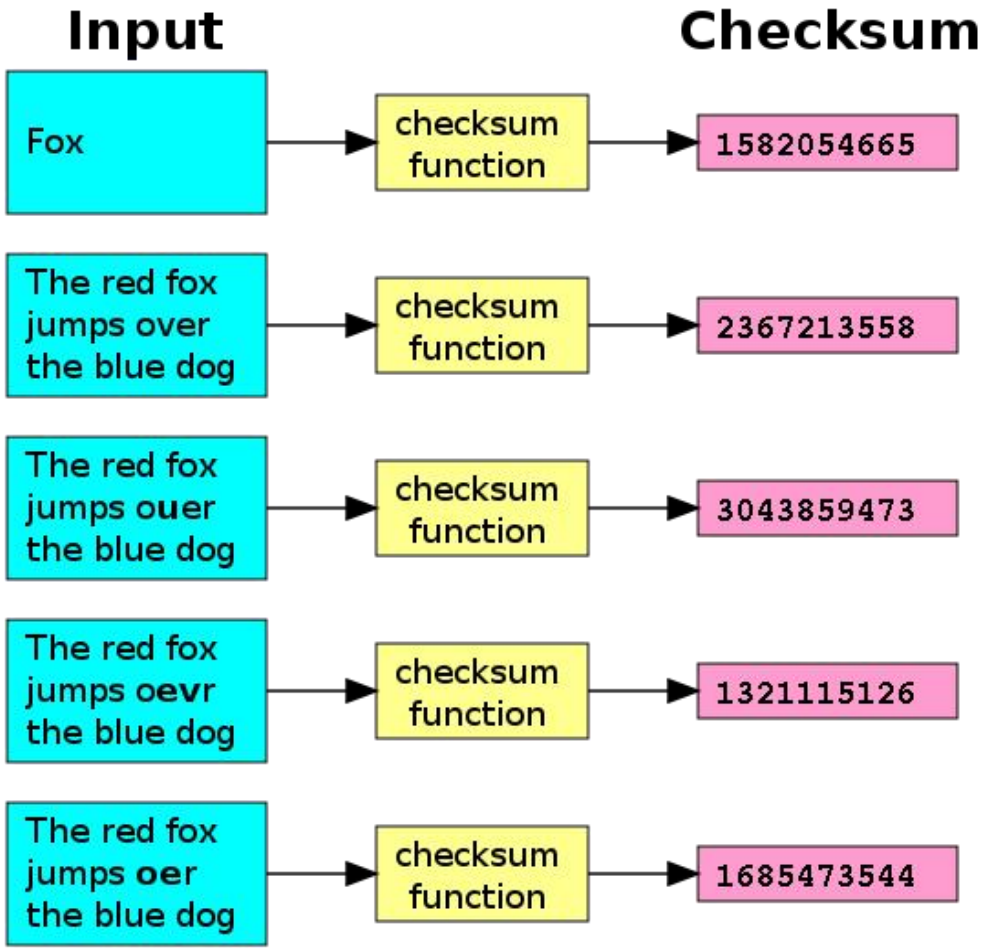
Detect Faults: Heartbeat

- **Heartbeat**: a periodic message exchange between a system monitor and a process being monitored.
  - The process periodically resets the *watchdog* timer in its monitor,
  - Piggybacking the heartbeat messages on to other control messages reduces the overhead
- Difference between ping and heartbeat?
  - Who initiates the health check?

Detect Faults

- **Timestamp**: used to detect incorrect sequences of events, primarily in distributed message-passing systems.
- **Condition Monitoring**: checking conditions in a process or device, or validating assumptions made during the design.
  - For example, **checksum** in data storage and transmission

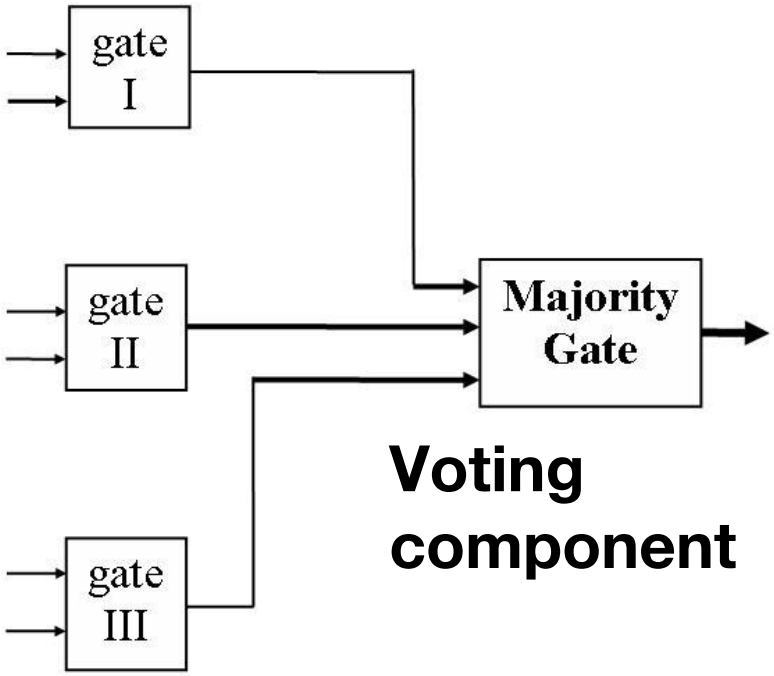
Checksum



Effect of a typical checksum function (the Unix cksum utility)

Detect Faults

- **Voting**: the common realization of this tactic is Triple Modular Redundancy (TMR)



Recover from Faults (Preparation & Repair)

- **Active Redundancy (hot spare):** all nodes in a *protection group* process identical inputs in parallel, allowing redundant spares to maintain *synchronous state* with the active nodes.

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

Recover from Faults (Preparation & Repair)

- **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 periodic state updates.

Recover from Faults (Preparation & Repair)

- **Rollback**: revert to a previous known good state, referred to as the “rollback line”.
- This tactic is combined with redundancy tactics
- After a rollback has occurred, a standby version of the failed component becomes active
- Rollback depends on a copy of a previous state (a checkpoint)
- Checkpoint can be stored in a fixed location and needs to be updated regularly

Recover from Faults (Preparation & Repair)

- **Retry**: where a failure is transient and retrying the operation may lead to success.
  - For example, network re-transmission
- **Ignore Faulty Behavior**: ignoring messages sent from a source when it is determined that those messages are spurious.
  - E.g., ignore the messages from a denial of service attacker

Recover from Faults (Preparation & Repair)

- **Degradation**: maintains the most critical system functions in the presence of component failures, dropping less critical functions.
- **Reconfiguration**: reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible.

Prevent Faults

- **Removal From Service**: temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures
- **Transactions**: bundling state updates so that asynchronous messages exchanged between distributed components are *atomic*, *consistent*, *isolated*, and *durable*.
- **Predictive Model**: take corrective action when conditions are detected that are predictive of likely future faults.

Prevent Faults

- **Increase Competence Set:** designing a component to handle more cases—faults—as part of its normal operation

Summary

- Availability refers to the ability of the system to be available for use when a fault occurs.
- The fault must be recognized (or prevented) and then the system must respond.
- The response will depend on the criticality of the application and the type of fault
  - can range from “ignore it” to “keep on going as if it didn’t occur.”

Summary

- Tactics for availability are categorized into detect faults, recover from faults and prevent faults.
- Detection tactics depend on detecting signs of life from various components.
- Recovery tactics are retrying an operation or maintaining redundant data or computations.
- Prevention tactics depend on removing elements from service or limiting the scope of faults.