分布式系统

08-分布式系统的一致性与复制 Consistency and Replication in DS

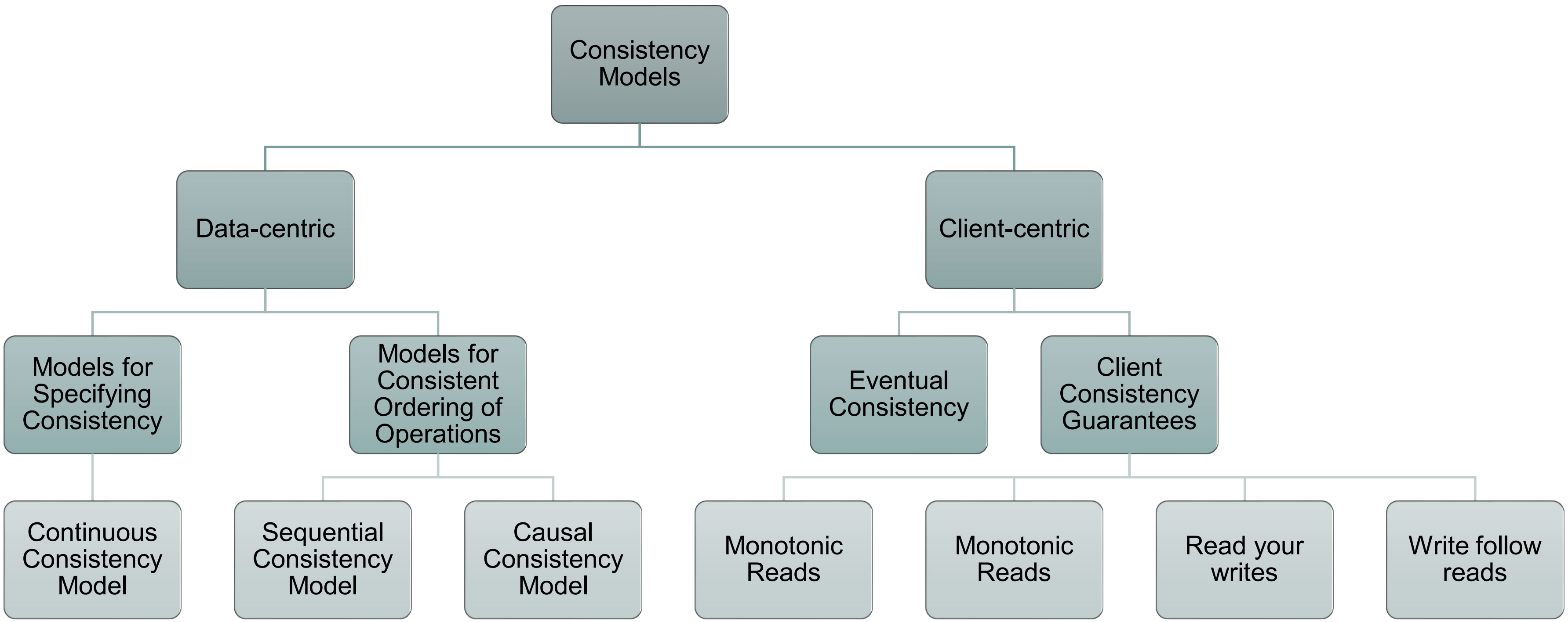
Weixiong Rao 饶卫雄
Tongji University 同济大学软件学院
2023 秋季
wxrao@tongji.edu.cn

同济大学软件学院
School of Software Engineering, Tongji University

Today…

- Last Session
  - Consistency and Replication
    - Consistency Models: Data-centric and Client-centric
- **Today's session**
  - Consistency and Replication – Part III
    - Replica Management
    - Consistency Protocols

Recap: Topics covered in Consistency Models



Overview

- Consistency Models
- Replica Management
- Consistency Protocols

Replica Management

- Replica management describes where, when and by whom replicas should be placed

- We will study two problems under replica management
  1. Replica-Server Placement
     - Decides the best locations to place the replica server that can host data-stores
  2. Content Replication and Placement
     - Finds the best server for placing the contents

Overview

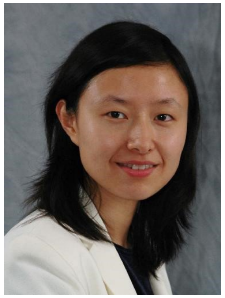
- Consistency Models
- Replica Management
  - Replica Server Placement
  - Content Replication and Placement
- Consistency Protocols

Replica Server Placement

- Factors that affect placement of replica servers:
  - What are the possible locations where servers can be placed?
    - Should we place replica servers close-by or distribute it uniformly?
  - How many replica servers can be placed?
    - What are the trade-offs between placing many replica servers vs. few?
  - How many clients are accessing the data from a location?
    - More replicas at locations where most clients access improves performance and fault-tolerance
- If **K** replicas have to be placed out of **N** possible locations, find the *best **K** out of **N*** locations **(K<N)**

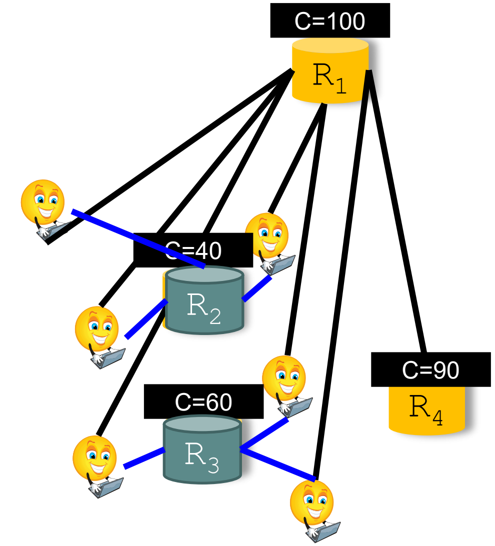
Replica Server Placement – *An Example Approach*

- Problem: **K** replica servers should be placed on some of the **N** possible replica sites such that
  - Clients have low-latency/high-bandwidth connections
- Qiu *et al.* [2] suggested a Greedy Approach
  1. Evaluate the cost of placing a replica on each of the **N** potential sites
     - Examining the *cost* of **C** clients connecting to the replica
     - Cost of a link can be **`1/bandwidth`** or **`latency`**
  2. Choose the lowest-cost site
  3. In the second iteration, search for a second replica site which, in conjunction with the already selected site, yields the lowest cost
  4. Iterate steps 2,3 and 4 until **K** replicas are chosen



邱锂力博士

- 微软亚洲研究院副院长， 负责微软亚洲研究院（上海）的研究工作，以及与产学研各界的合作
- 国际计算机学会无线及移动系统专委 ACM SIGMOBILE的主席



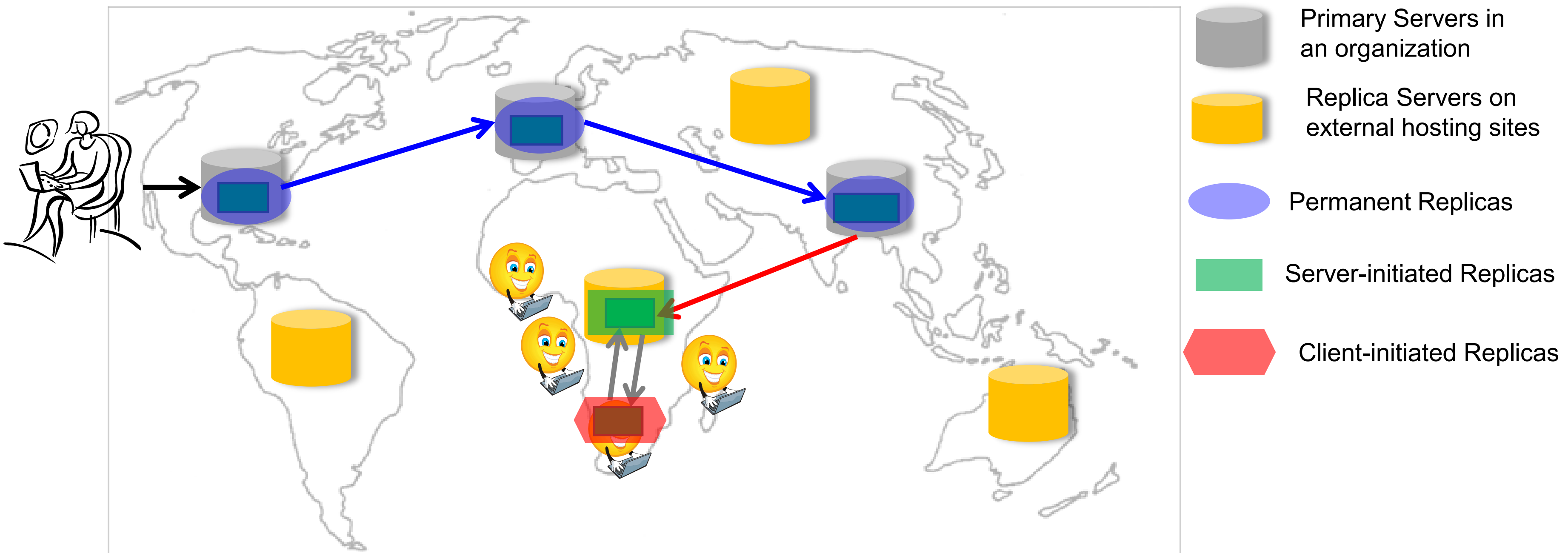
Overview

- Consistency Models
- Replica Management
  - Replica Server Placement
  - Content Replication and Placement
- Consistency Protocols

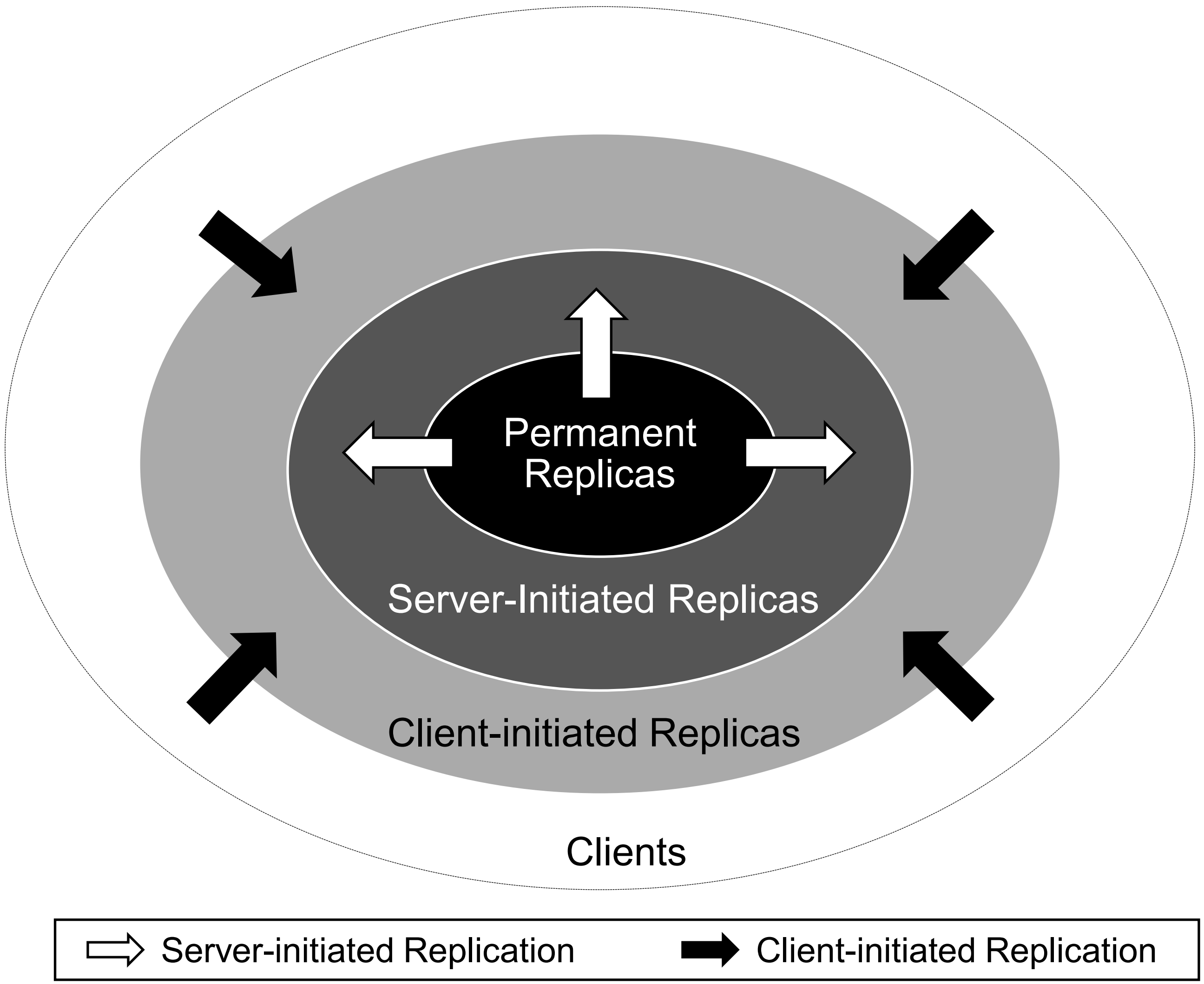
Content Replication and Placement

- In addition to the server placement, it is important:
  - how, when and by whom <u>different data items (contents)</u> are placed on possible replica servers

- Identify how webpage replicas are replicated:



Logical Organization of Replicas



同济大学软件学院
School of Software Engineering, Tongji University

1. Permanent Replicas

- Permanent replicas are the initial set of replicas that constitute a distributed data-store
- Typically, small in number
- There can be two types of permanent replicas:
  - Primary servers
    - One or more servers in an organization
    - Whenever a request arrives, it is forwarded into one of the primary servers
  - Mirror sites
    - Geographically spread, and replicas are generally statically configured
    - Clients pick one of the mirror sites to download the data

2. Server-initiated Replicas

- A third party (*provider*) owns the *secondary replica servers*, and they provide *hosting service*
  - The provider has a collection of servers across the Internet
  - The hosting service dynamically replicates files on different servers
    - Based on the popularity of the file in a region
- The permanent server chooses to host the data item on different *secondary replica servers*
- The scheme is efficient when updates are rare
- Examples of Server-initiated Replicas
  - Replicas in Content Delivery Networks (CDNs)

Dynamic Replication in Server-initiated Replicas

- Dynamic replication at secondary servers:
  - Helps to reduce the server load and improve client performance
  - But, replicas have to dynamically push the updates to other replicas

+ Rabinovich et al. [3] proposed a distributed scheme for replication:
  + Each server keeps track of:
    i. which is the closest server to the requesting client
    ii. number of requests per file per closest server
    + For example, each server **Q** keeps track of $\mathtt{cnt}_Q(P,F)$ which denotes how many requests arrived at **Q** which are closer to server **P** (for a file **F**)

  + If $\mathtt{cnt}_Q(P,F) > 0.5 * \mathtt{cnt}_Q(Q,F)$
    + Request **P** to replicate a copy of file **F**

  If some other replica is nearer to the clients, request replication over that server

  + If $\mathtt{cnt}_P(P,F) < \mathtt{LOWER\_BOUND}$
    + Delete the file at replica Q

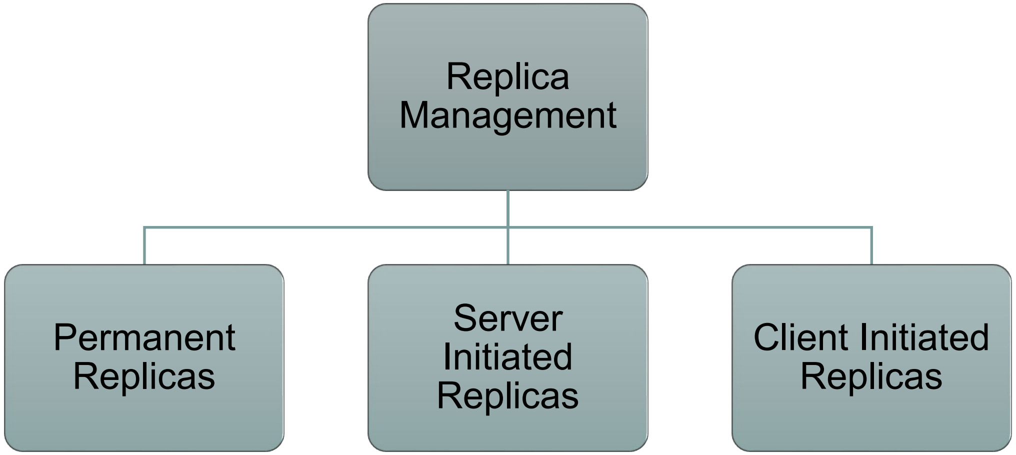
  If the replication is not popular, delete the replica

3. Client-initiated Replicas

- Client-initiated replicas are known as *client caches*
- Client caches are used only to reduce the access latency of data
  - e.g., Browser caching a web-page locally
- Typically, managing a cache is entirely the responsibility of a client
  - Occasionally, data-store may inform client when the replica has become stale

Summary of Replica Management

- Replica management deals with placement of servers and content for improving performance and fault-tolerance



Till now, we know:

- how to place replica servers and content
- the required consistency models for applications

What else do we need to provide consistency in a distributed system?

Overview

- Consistency Models
- Replica Management
- Consistency Protocols

Consistency Protocols

- A consistency protocol describes the implementation of a specific consistency model

- We are going to study three consistency protocols:
  - Primary-based protocols
    - One primary coordinator is elected to control replication across multiple replicas
  - Replicated-write protocols
    - Multiple replicas coordinate to provide consistency guarantees
  - Cache-coherence protocols
    - A special case of client-controlled replication

Overview of Consistency Protocols

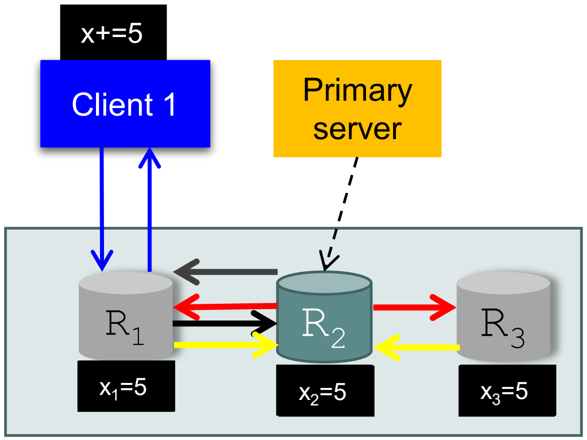
- Consistency Protocols
  - Primary-based Protocols
  - Replicated-Write Protocols
  - Cache Coherence Protocols

Primary-based protocols

- In Primary-based protocols, a simple centralized design is used to implement consistency models
  - Each data-item **x** has an associated "*Primary Replica*"
  - Primary replica is responsible for coordinating write operations
- We will study one example of Primary-based protocols that implement Sequential Consistency Model
  - Remote-Write Protocol

Remote-Write Protocol

- Rules:
  - All write operations are forwarded to the primary replica
  - Read operations are carried out locally at each replica
- Approach for write ops: (Budhiraja *et al.* [4])
  - Client connects to some replica R_C
  - If the client issues write operation to R_C:
    - R_C forwards the request to the primary replica R_P
    - R_P updates its local value
    - R_P forwards the update to other replicas R_i
    - Other replicas R_i update, and send an ACK back to R_P
  - After R_P receives all ACKs, it informs the R_C that write operation is successful
  - R_C acknowledges to the client that write operation was successful

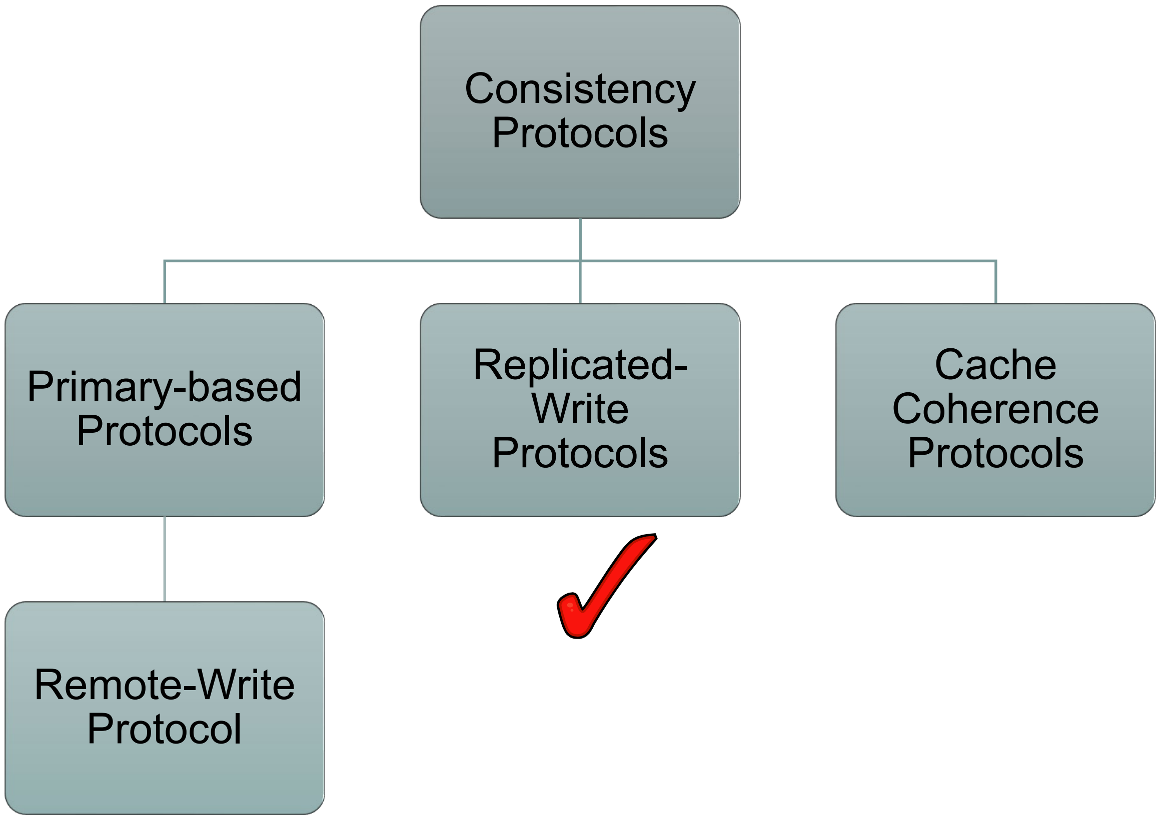


Data-store

Remote-Write Protocol – Discussion

- Remote-Write provides
  - A simple way to implement sequential consistency
  - Guarantees that client see the most recent write operations
- However, latency is high in Remote-Write Protocols
  - Client blocks until all the replicas are updated
  - In what scenarios would you use remote-write protocols?
- Remote-Write Protocols are applied to distributed databases and file systems that require fault-tolerance
  - Replicas are placed on the same LAN to reduce latency

Overview of Consistency Protocols

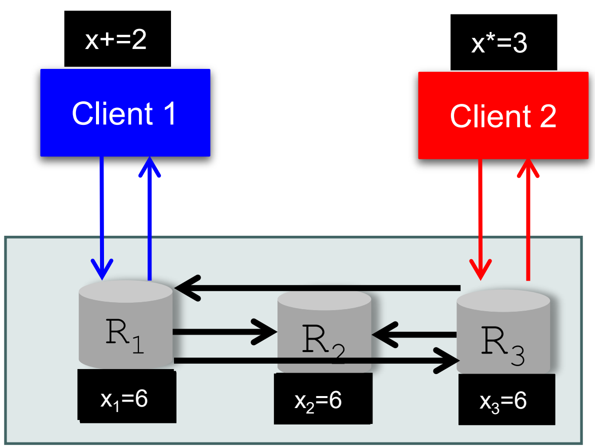


Replicated-Write Protocol

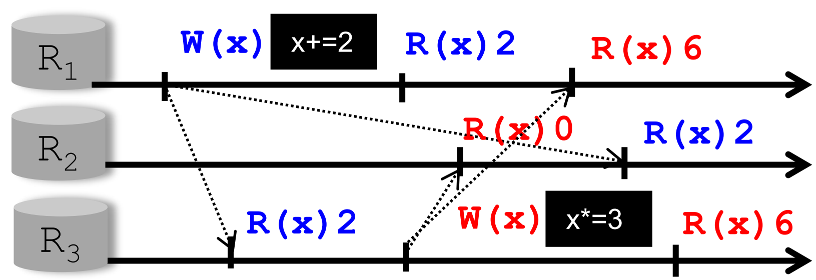
- In a replicated-write protocol, updates can be carried out at multiple replicas
- We will study one example replicated-write protocol called Active Replication Protocol
  - Here, clients write at any replica
  - The replica will propagate updates to other replicas

Active Replication Protocol

- When a client writes at a replica, the replica will send the write operation updates to all other replicas
- Challenges with Active Replication
  - Ordering of operations cannot be guaranteed across the replicas

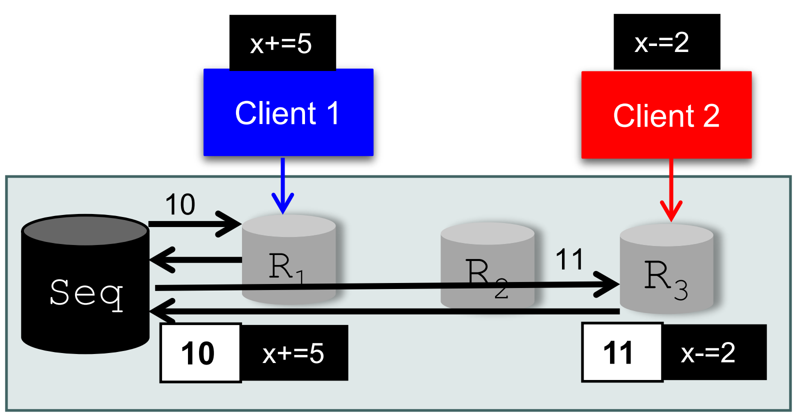


Data-store



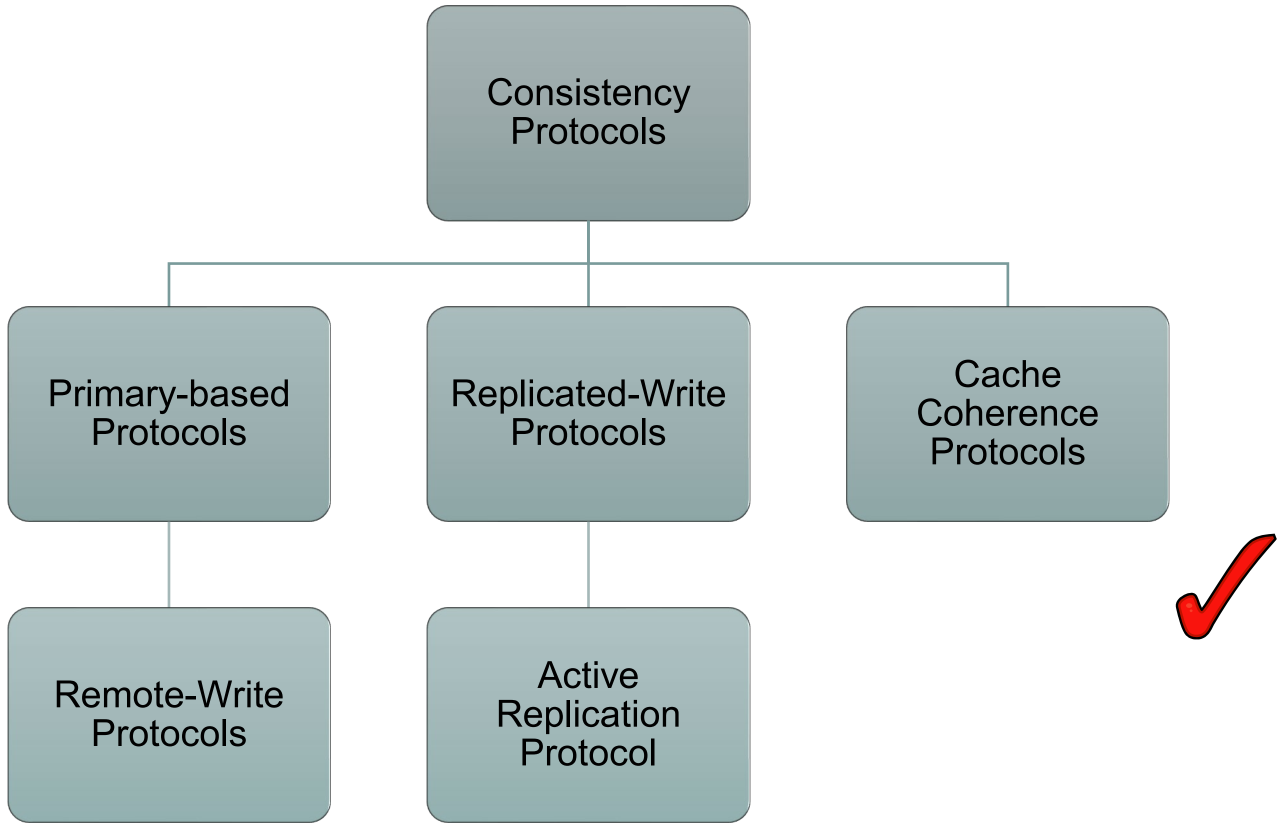
Centralized Active Replication Protocol

- Approach
  - There is a centralized coordinator called *sequencer* (`Seq`)
  - When a client connects to a replica R_C and issues a write operation
    - R_C forwards the update to the `Seq`
    - `Seq` assigns a sequence number to the update operation
    - R_C propagates the sequence number and the operation to other replicas
  - Operations are carried out at all the replicas in the order of the sequence number



Data-store

Overview of Consistency Protocols



Cache Coherence Protocols

- Caches are special types of replicas
  - Typically, caches are client-controlled replicas
- Cache coherence refers to the consistency of data stored in caches
- How are the cache coherence protocols in shared-memory multiprocessor (SMP) systems different from those in Distributed Systems?
  - Coherence protocols in SMP assume cache states can be broadcasted efficiently
  - In DS, this is not possible because caches may reside on different machines

Cache Coherence Protocols (cont'd)

- Cache Coherence protocols determine how caches are kept consistent

- Caches may become inconsistent when data item is modified:
  1. at the server replicas, or
  2. at the cache

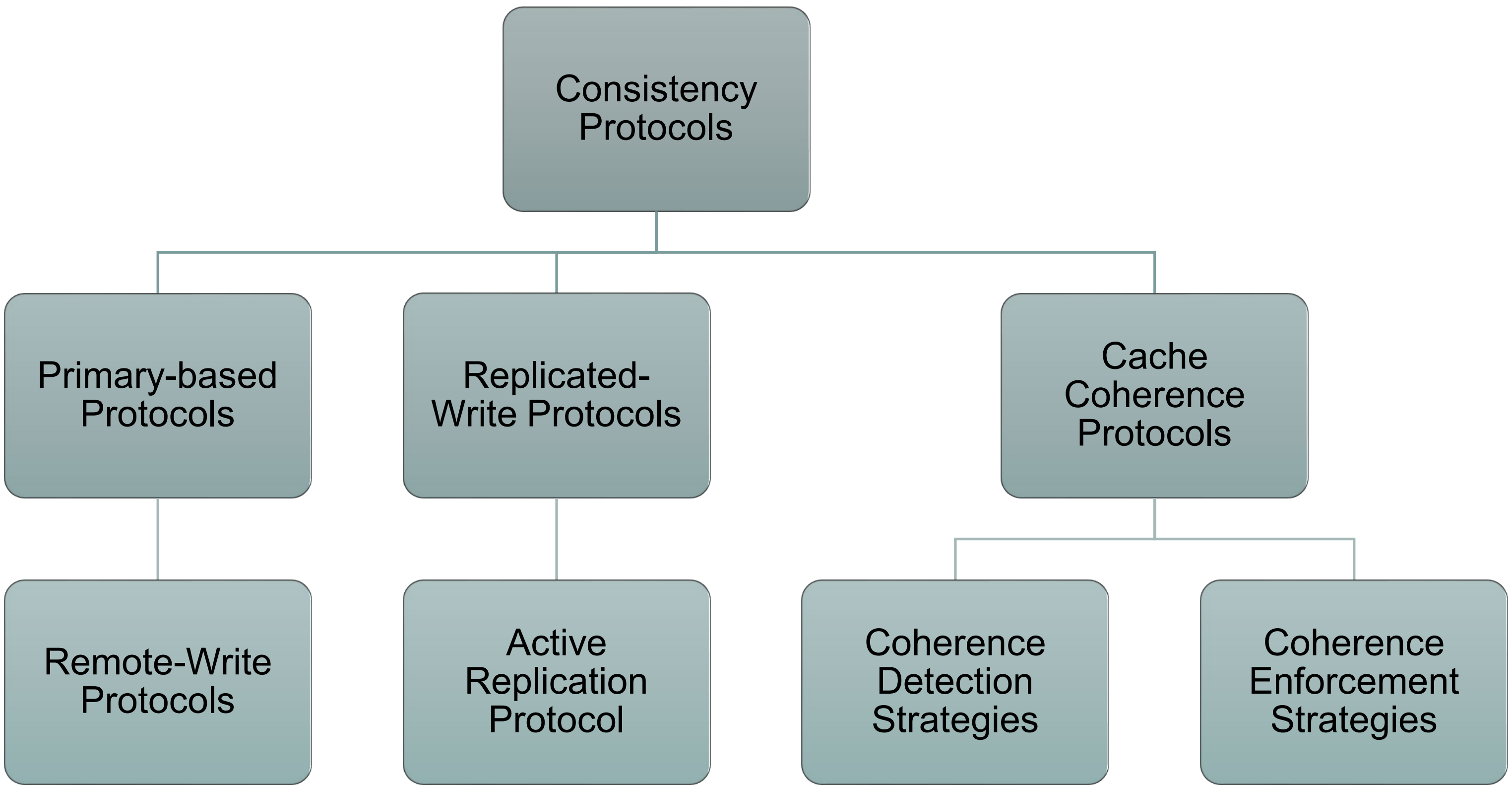
1. When Data is Modified at the Server

- Two approaches for enforcing coherence:
  1. Server-initiated invalidation
     - Here, server sends all caches an invalidation message when data item is modified
  2. Server updates the cache
     - Server will propagate the update to the cache

2. When Data is Modified at the Cache

- The enforcement protocol may use one of three techniques:
  1. Read-only cache
     - The cache does not modify the data in the cache
     - The update is propagated to the server replica
  2. Write-through cache
     - Directly modify the cache, and forward the update to the server
  3. Write-back cache
     - The client allows multiple writes to take place at the cache
     - The client batches a set of writes, and will send the batched write updates to the server replica

Summary of Consistency Protocols



Consistency and Replication – Brief Summary

- Replication improves performance and fault-tolerance
- However, replicas have to be kept reasonably consistent

Consistency Models

- A contract between the data-store and process
- Types: Data-centric and Client-centric

Replication Management

- Describes where, when and by whom replicas should be placed
- Types: Replica Server Placement, Content Replication and Placement

Consistency Protocols

- Implement Consistency Models
- Types: Primary-based, Replicated-Write, Cache Coherence

Next Classes

- Fault-tolerance
  - How to detect and deal with failures in Distributed Systems?

References

- [1] Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B.B., "Session guarantees for weakly consistent replicated data", Proceedings of the Third International Conference on Parallel and Distributed Information Systems, 1994
- [2] Lili Qiu, Padmanabhan, V.N., Voelker, G.M., "On the placement of Web server replicas", Proceedings of IEEE INFOCOM 2001.
- [3] Rabinovich, M., Rabinovich, I., Rajaraman, R., Aggarwal, A., "A dynamic object replication and migration protocol for an Internet hosting service", Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS), 1999
- [4] Navin Budhiraja, Keith Marzullo. Fred B. Schneider. Sam Toueg, "The primary-backup approach", Distributed systems (2nd Ed.), ACM Press/Addison-Wesley Publishing Co., 1993
- [5] http://www.cdk5.net
- [6] http://en.wikipedia.org/wiki/Cache_coherence

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Today…

- Last Sessions
  - Consistency and Replication
    - Consistency Models: Data-centric and Client-centric
    - Replica Management
    - Consistency Protocols

- Today's session
  - Fault Tolerance
    - General background
    - Process resilience and failure detection

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Dependability 可信任性

Basics: A component provides services to clients. To provide services, the component may require the services from other components ⇒ a component may depend on some other component.

Specifically: A component C depends on C∗ if the correctness of C's behavior depends on the correctness of C∗'s behavior.

Some properties of dependability:

- Availability: Readiness for usage
- Reliability: Continuity of service delivery
- Safety: Very low probability of catastrophes
- Maintainability: How easy can a failed system be repaired

Note: For distributed systems, components can be either processes or channels

Terminology

Failure: When a component is not living up to its specifications, a failure occurs
失效

Error: That part of a component's state that can lead to a failure
错误

Fault: The cause of an error
故障

Fault prevention: prevent the occurrence of a fault

Fault tolerance: build a component in such a way that it can meet its specifications in the presence of faults (i.e., mask the presence of faults)

Fault removal: reduce the presence, number, seriousness of faults

Fault forecasting: estimate the present number, future incidence, and the consequences of faults

Failures, Due to What?

失效

- Failures can happen due to a variety of reasons:
  - Hardware faults 失效
  - Software bugs
  - Operator errors 错误
  - Network errors/outages 断网

- A system is said to *fail* when it cannot meet its promises

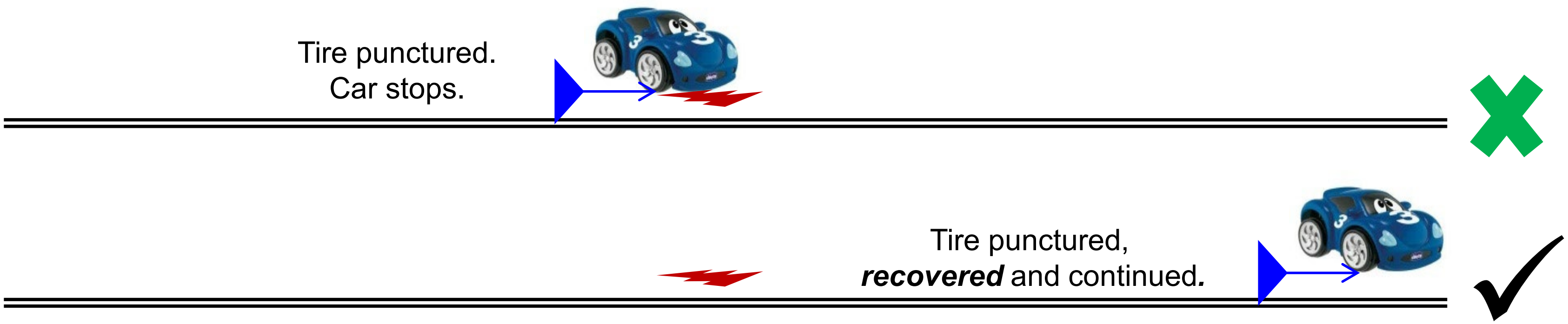
Failures in Distributed Systems

- A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of ***partial failure***

- A partial failure may happen when a component in a distributed system fails
  - This failure may affect the proper operation of other components, while at the same time leaving yet other components unaffected

Goal and Fault-Tolerance

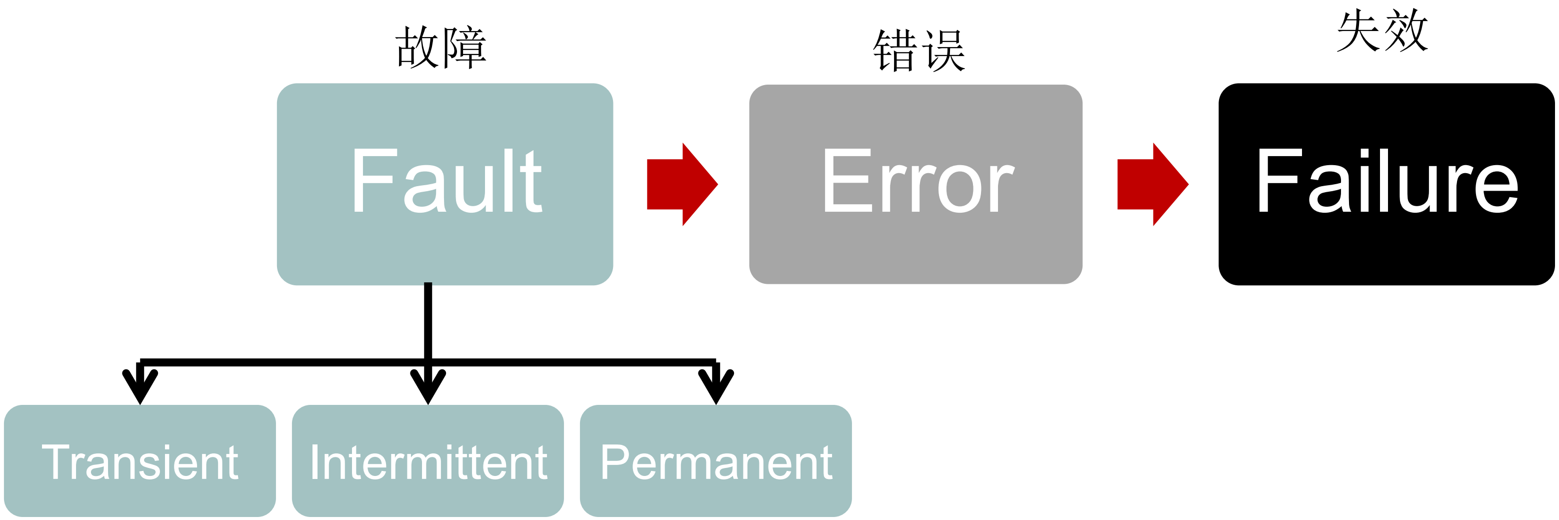
- *An overall goal in distributed systems is to construct the system in such a way that it can automatically recover from partial failures*



- **Fault-tolerance** is the property that enables a system to continue operating properly in the event of failures

- For example, TCP is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communication links which are imperfect or overloaded

Faults, Errors and Failures



A system is said to be *fault tolerant* if it can provide its services even in the presence of *faults*

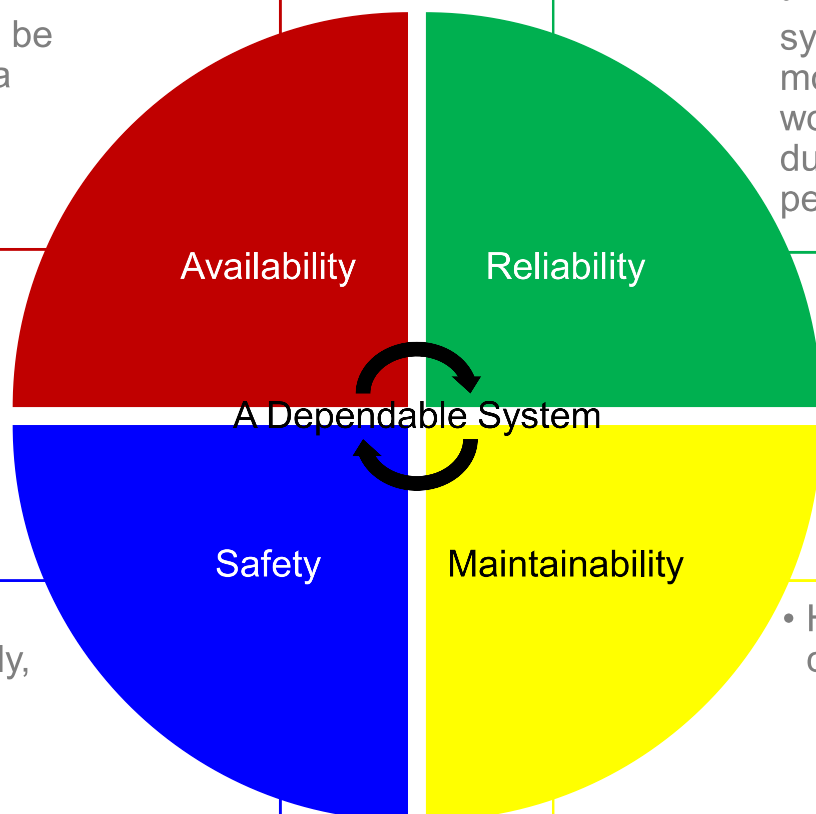
Fault Tolerance Requirements

- A robust fault tolerant system requires:

    1. No single point of failure
    2. Fault isolation/containment to the failing component
    3. Availability of reversion modes

Dependable Systems 可信系统

- Being fault tolerant is strongly related to what is called a *dependable system*



- A system is said to be highly available if it will be most likely working at a given instant in time

- A highly-reliable system is one that will most likely continue to work without interruption during a relatively long period of time

- A system temporarily fails to operate correctly, nothing catastrophic happens

- How easy a failed system can be repaired

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Failure Models 失效模式

Type of Failure	Description
• Crash Failure	• A server halts, but was working correctly until it stopped

• Omission Failure
  • Receive Omission
  • Send Omission
	• A server fails to respond to incoming requests
	  • A server fails to receive incoming messages
	  • A server fails to send messages

• Timing Failure	• A server's response lies outside the specified time interval

• Response Failure
  • Value Failure
  • State Transition Failure
	• A server's response is incorrect
	  • The value of the response is wrong
	  • The server deviates from the correct flow of control

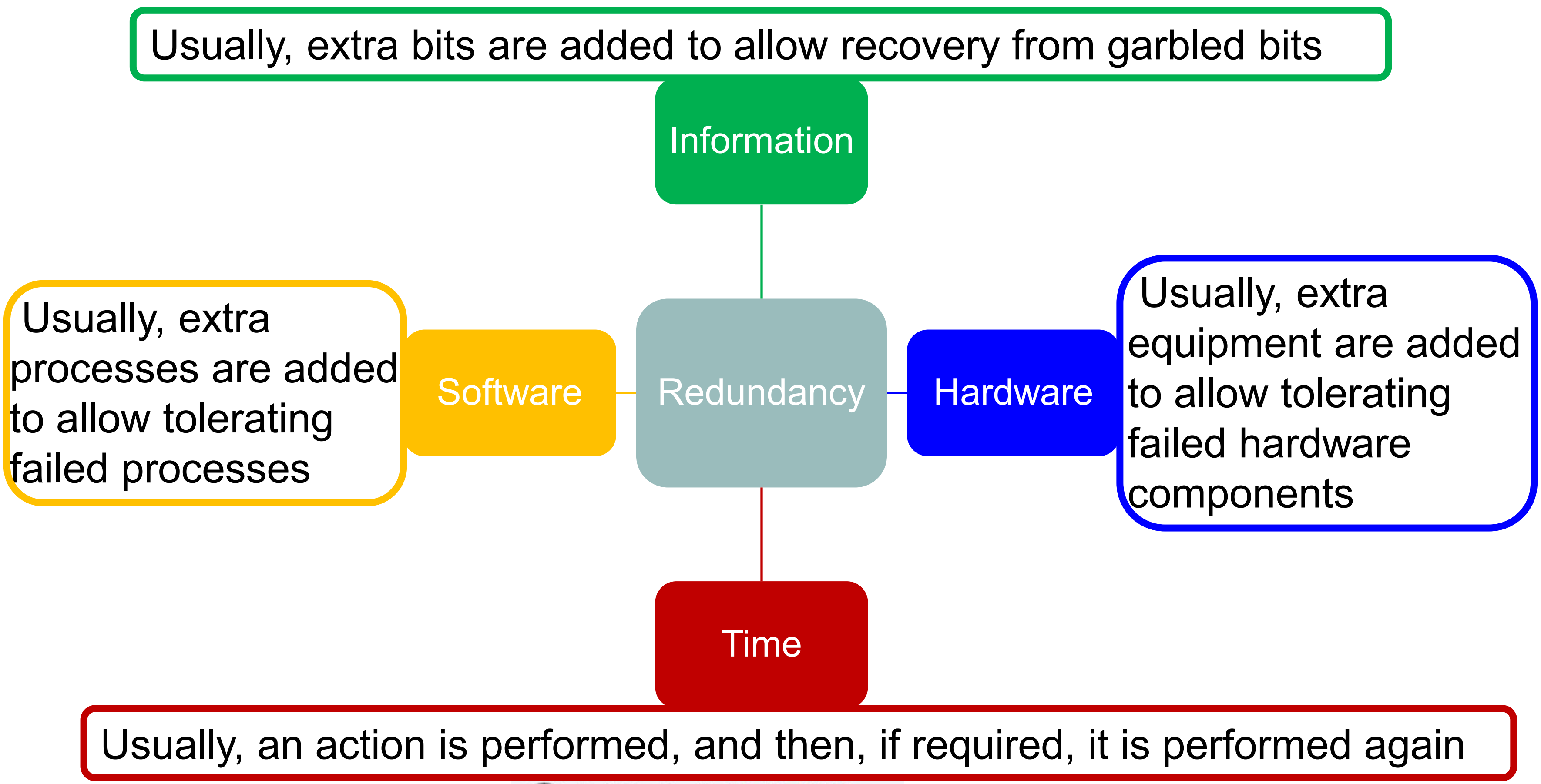
• Byzantine Failure	• A server may produce arbitrary responses at arbitrary times

A General Background

- Basic Concepts
- Failure Models
- Failure Masking by Redundancy

Faults Masking by Redundancy

- The key technique for masking faults is to use *redundancy*



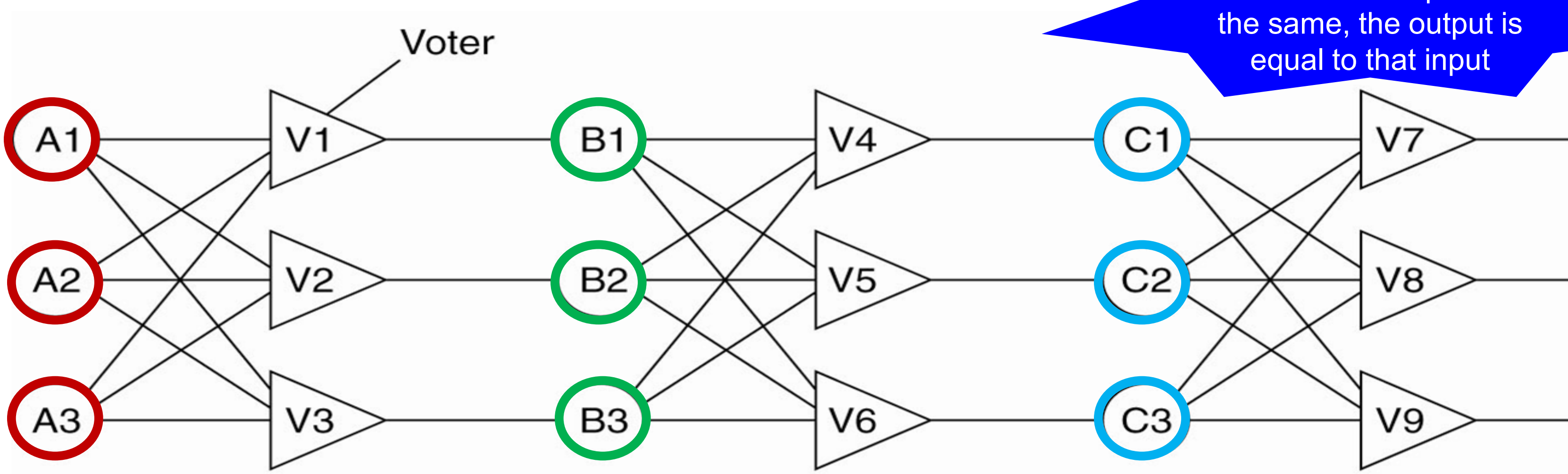
Triple Modular Redundancy

If one is faulty, the final result will be incorrect

A circuit with signals passing through devices A, B, and C, in sequence

---

If 2 or 3 of the inputs are the same, the output is equal to that input



Each device is replicated 3 times and after each stage is a triplicated voter

Process Resilience and Failure Detection

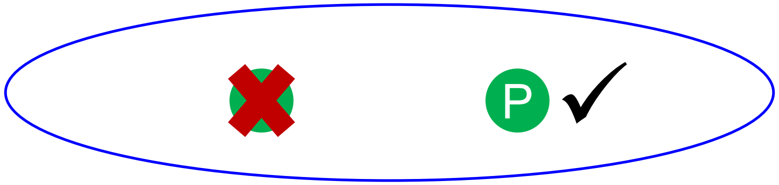
Process Resilience and Failure Detection

- Now that the basic issues of fault tolerance have been discussed, let us concentrate on how fault tolerance can actually be achieved in distributed systems

- The topics we will discuss:
  - How can we provide protection against process failures?
    - Process Groups
    - Reaching an agreement within a process group
  - How to detect failures?

Process Resilience

- The key approach to tolerating a faulty process is to organize several identical processes into a *group*



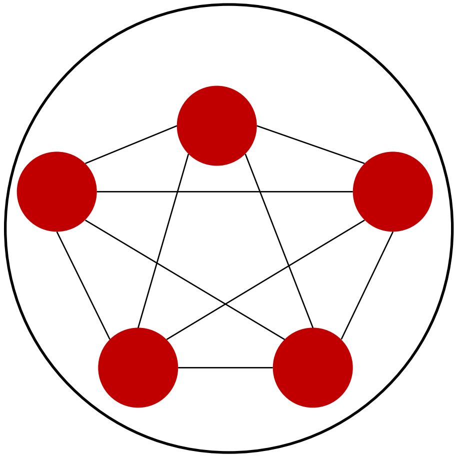
- If one process in a group fails, hopefully some other process can take over

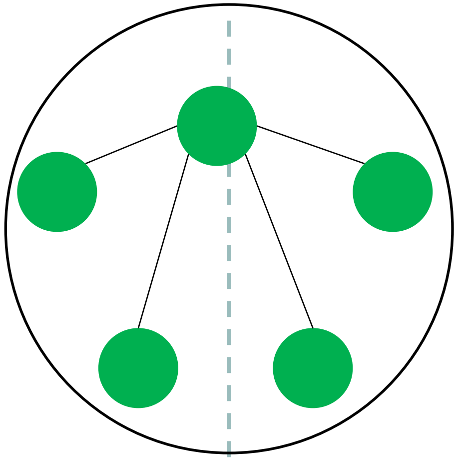
- Caveats: 注意事项

  - A process can join a group or leave one during system operation
  - A process can be a member of several groups at the same time

Flat Versus Hierarchical Groups

- An important distinction between different groups has to do with their internal structure





Flat Group:

(+) Symmetrical
(+) No single point of failure
(-) Decision making is complicated

Hierarchical Group:

(+) Decision making is simple
(-) Asymmetrical
(-) Single point of failure

K-Fault-Tolerant Systems

- A system is said to be *k-fault-tolerant* if it can survive faults in *k* components and still meet its specifications

- How can we achieve a *k-fault-tolerant* system?

  - This would require an *agreement protocol* applied to a process group

Agreement in Faulty Systems (1)

- A process group typically requires reaching an *agreement* in:
  - Electing a coordinator
  - Deciding whether or not to commit a transaction
  - Dividing tasks among workers
  - Synchronization

- When the communication and processes:
  - are perfect, reaching an agreement is often straightforward
  - are not perfect, there are problems in reaching an agreement

Agreement in Faulty Systems (2)

- **Goal:** have all non-faulty processes reach consensus on some issue, and establish that consensus within a finite number of steps

- Different assumptions about the underlying system require different solutions:
  - Synchronous versus asynchronous systems
  - Communication delay is bounded or not
  - Message delivery is ordered or not
  - Message transmission is done through unicasting or multicasting

Agreement in Faulty Systems (3)

- Reaching a distributed agreement is only possible in the following circumstances:

		Message Ordering				
		Unordered		Ordered		
Process Behavior		Unicast (Message Transmission)	Multicast (Message Transmission)	Unicast (Message Transmission)	Multicast (Message Transmission)	Communication Delay
Synchronous		✓	✓	✓	✓	Bounded
Synchronous				✓	✓	Unbounded
Asynchronous					✓	Bounded
Asynchronous					✓	Unbounded

Agreement in Faulty Systems (4)

- In practice most distributed systems assume that:
  - Processes behave asynchronously
  - Message transmission is unicast
  - Communication delays are unbounded

- Usage of ordered (reliable) message delivery is typically required

- The agreement problem has been originally studied by Lamport and referred to as the *Byzantine Agreement Problem* [Lamport *et al.*]

Byzantine Agreement Problem (1)

- Lamport assumes:
  - Processes are synchronous
  - Messages are unicast while preserving ordering
  - Communication delay is bounded
  - There are *N* processes, where each process *i* will provide a value v_i to the others
  - There are at most *k* faulty processes

Byzantine Agreement Problem (2)

- **Lamport's Assumptions:**
  - Processes are synchronous
  - Messages are unicast while preserving ordering
  - Communication delay is bounded
  - There are N processes, where each process i will provide a value v_i to the others
  - There are at most k faulty processes

Process Behavior	Communication Delay	Unordered – Unicast	Unordered – Multicast	Ordered – Unicast	Ordered – Multicast
Synchronous	Bounded	✓	✓	✓	✓
Synchronous	Unbounded			✓	✓
Asynchronous	Bounded				✓
Asynchronous	Unbounded				✓

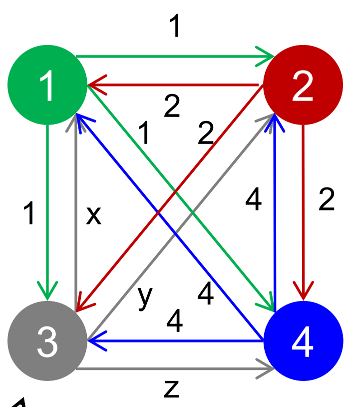
Message Ordering: Unordered / Ordered; Message Transmission: Unicast / Multicast

Lamport suggests that each process i constructs a vector V of length N, such that if process i is non-faulty, $V[i] = v_i$. Otherwise, $V[i]$ is undefined

Byzantine Agreement Problem (3)

- **Case I: N = 4 and k = 1**

Step1: Each process sends its value to the others



Faulty process

Step2: Each process collects values received in a vector

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

Step3: Every process passes its vector to every other process

1 Got

(1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

2 Got

(1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

4 Got

(1, 2, x, 4)
(1, 2, y, 4)
(i, j, k, l)

Byzantine Agreement Problem (4)

Step 4:

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a ***majority***, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

The algorithm reaches an agreement

Result Vector:
(1, 2, UNKNOWN, 4)

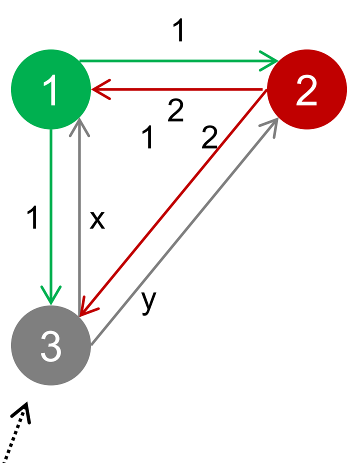
Result Vector:
(1, 2, UNKNOWN, 4)

Result Vector:
(1, 2, UNKNOWN, 4)

Byzantine Agreement Problem (5)

- **Case II: N = 3 and k = 1**

Step1: Each process sends its value to the others



Faulty process

Step2: Each process collects values received in a vector

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

Step3: Every process passes its vector to every other process

1 Got	**2** Got
(1, 2, y) (a, b, c)	(1, 2, x) (d, e, f)

Byzantine Agreement Problem (6)

Step 4:

- Each process examines the *i*th element of each of the newly received vectors
- If any value has a *majority*, that value is put into the result vector
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

1 Got

(1, 2, y)
(a, b, c)

The algorithm has failed to produce an agreement

2 Got

(1, 2, x)
(d, e, f)

Result Vector:
(UNKOWN, UNKNOWN, UNKNOWN)

Result Vector:
(UNKOWN, UNKNOWN, UNKNOWN)

Concluding Remarks on the Byzantine Agreement Problem

- In their paper, *Lamport et al.* (1982) proved that in a system with ***k*** faulty processes, an agreement can be achieved only if ***2k+1*** correctly functioning processes are present, for a total of ***3k+1***.
    - i.e., An agreement is possible only if more than two-thirds of the processes are working properly.
- *Fisher et al.* (1985) proved that in a distributed system in which ordering of messages **cannot** be guaranteed to be delivered within a known, finite time, no agreement is possible even if only one process is faulty.

Process Failure Detection

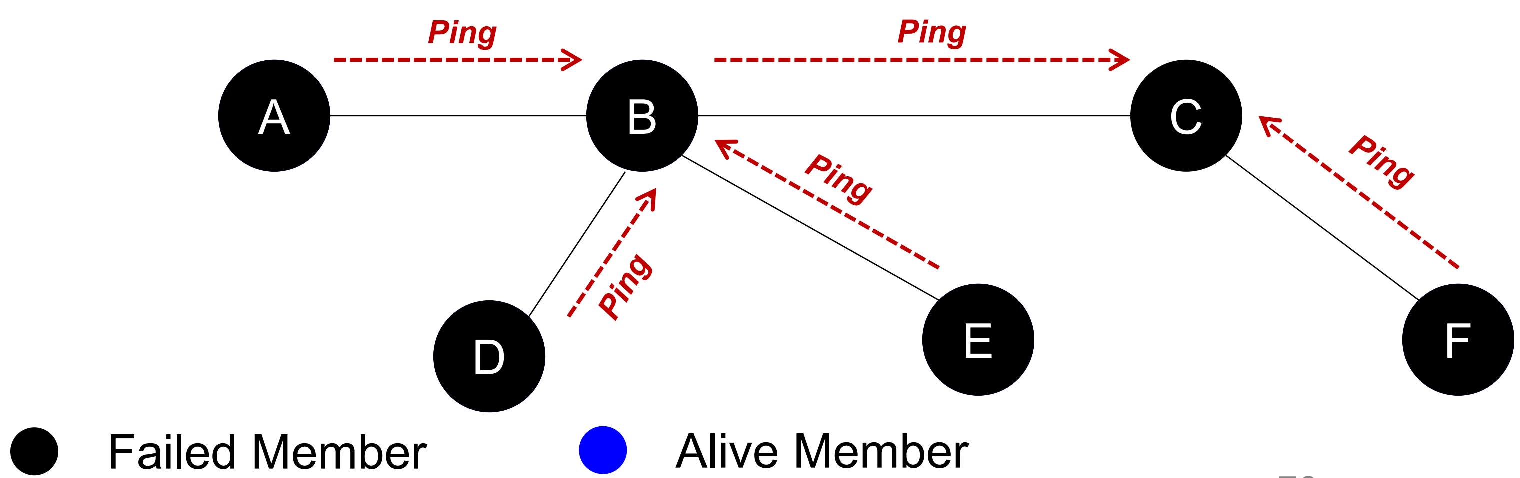
- Before we properly mask failures, we generally need to detect them
- For a group of processes, non-faulty members should be able to decide who is still a member and who is not
- Two policies:
  - Processes actively send "are you alive?" messages to each other (i.e., *pinging each other*)
  - Processes passively wait until messages come in from different processes

Timeout Mechanism

- In failure detection a ***timeout mechanism*** is usually involved
  - Specify a **timer**, after a period of time, trigger a timeout
  - However, due to unreliable networks, simply stating that a process has failed because it does not return an answer to a ping message may be wrong

Example: FUSE

- In FUSE, processes can be joined in a group that spans a WAN
- The group members create a spanning tree that is used for monitoring member failures
- An active (pinging) policy is used where a single node failure is rapidly promoted to a group failure notification



Failure Considerations

- There are various issues that need to be taken into account when designing a failure detection subsystem:

  1. Failure detection can be done as a side-effect of regularly exchanging information with neighbors (e.g., ***gossip-based information dissemination***)

  2. A failure detection subsystem should ideally be able to distinguish network failures from node failures

  3. When a member failure is detected, how should other non-faulty processes be informed

Recovery

- So far, we have mainly concentrated on algorithms that allow us to tolerate faults
- However, once a failure has occurred, it is essential that the process where the failure has happened can *recover* to a *correct state*
- In what follows we focus on:
  - What it actually means to recover to a correct state
  - When and how the state of a distributed system can be recorded and recovered, by means of *checkpointing* and *message logging*

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Error Recovery

- Once a failure has occurred, it is essential that the process where the failure has happened can recover to a correct state

- Fundamental to fault tolerance is the recovery from an error

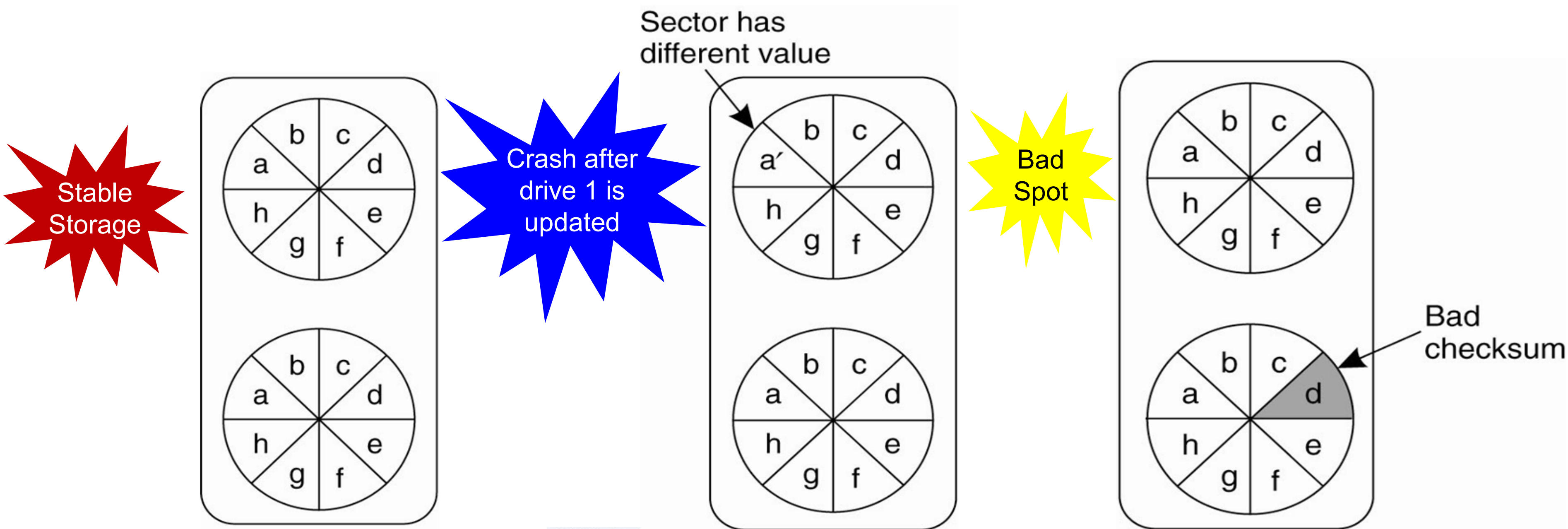
- The idea of *error recovery* is to replace an erroneous state with an error-free state

- There are essentially two forms of error recovery:

    1. *Backward recovery*
    2. *Forward recovery*

1. Backward Recovery (1)

- In backward recovery, the main issue is to bring the system from its present erroneous state back to a previously correct state
- It is necessary to record the system's state *from time to time* onto a *stable storage*, and to restore such a recorded state when things go wrong



1. Backward Recovery (2)

- Each time (part of) the system's present state is recorded, a *checkpoint* is said to be made

- Problems with backward recovery:
  - Restoring a system or a process to a previous state is generally expensive in terms of performance
  - Some states can never be rolled back (e.g., typing in UNIX rm –fr *)

2. Forward Recovery

- When the system detects that it has made an error, forward recovery reverts the system state to error time and corrects it, to be able to move forward

- Forward recovery is typically faster than backward recovery but requires that it has to be known in advance which errors may occur

- Some systems make use of both forward and backward recovery for different errors or different parts of one error

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Why Checkpointing?

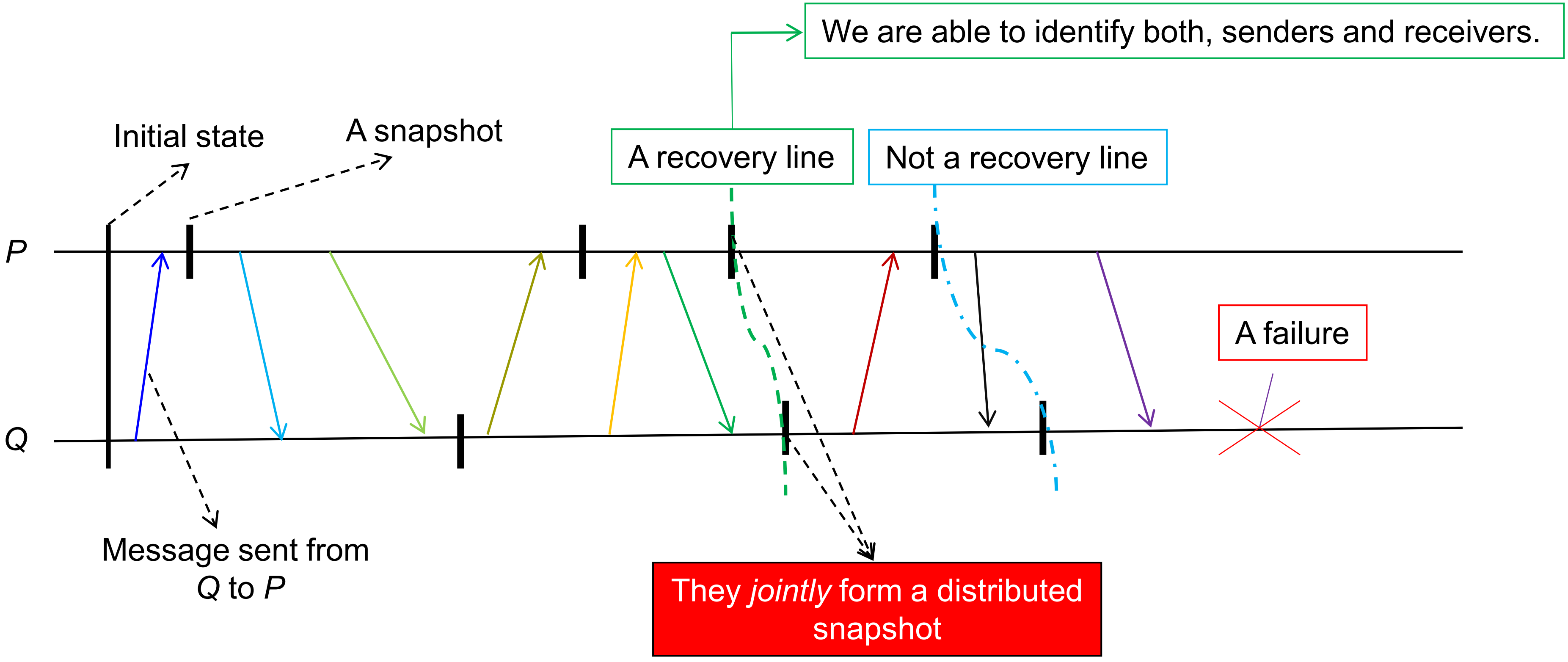
- In a fault-tolerant distributed system, backward recovery requires that the system regularly saves its state onto a stable storage

- This process is referred to as checkpointing

- In particular, checkpointing consists of storing a *distributed snapshot* of the current application state (i.e., *a consistent global state*), and later on, use it for restarting the execution in case of a failure

Recovery Line

- In a distributed snapshot, if a process *P* has recorded the receipt of a message, then there should be also a process *Q* that has recorded the sending of that message



Checkpointing

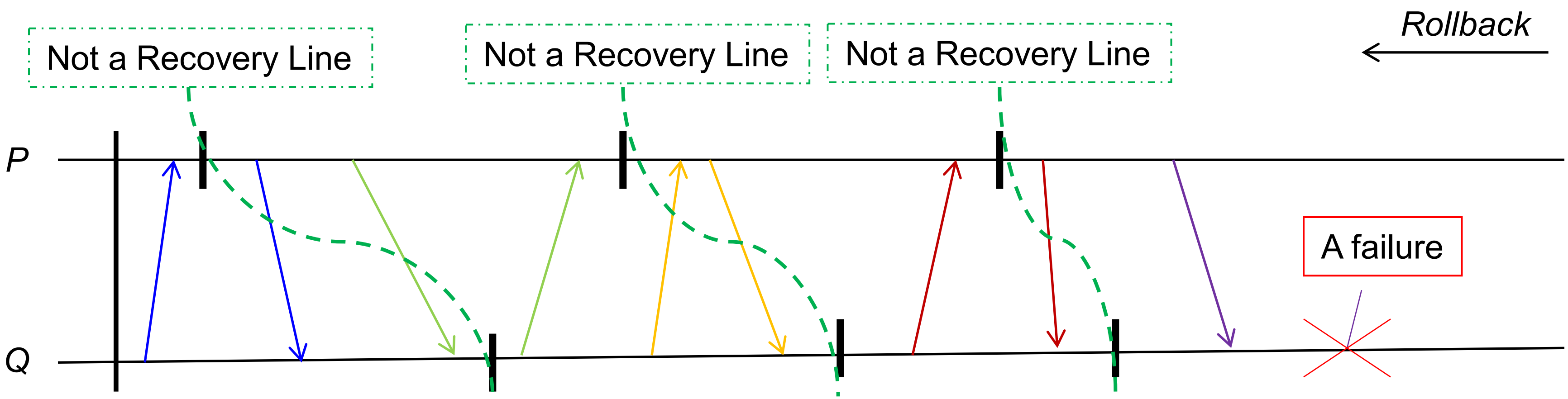
- Checkpointing can be of two types:

1. Independent Checkpointing: each process simply records its local state from time to time in an uncoordinated fashion

2. Coordinated Checkpointing: all processes synchronize to jointly write their states to local stable storages

Domino Effect

- Independent checkpointing may make it difficult to find a recovery line, leading potentially to a *domino effect* resulting from *cascaded* rollbacks



- With coordinated checkpointing, the saved state is automatically globally consistent, hence, domino effect is inherently avoided

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Why Message Logging?

- Considering that checkpointing is an expensive operation, techniques have been sought to reduce the number of checkpoints, but still enable recovery
- An important technique in distributed systems is *message logging*
- The basic idea is that if transmission of messages can be replayed, we can still reach a globally consistent state but without having to restore that state from stable storage
- In practice, the combination of having fewer checkpoints and message logging is more efficient than having to take many checkpoints

Message Logging

- Message logging can be of two types:

  1. Sender-based logging: A process can log its messages before sending them off

  2. Receiver-based logging: A receiving process can first log an incoming message before delivering it to the application

- When a sending or a receiving process crashes, it can restore the most recently checkpointed state, and from there on *replay* the logged messages (important for non-deterministic behaviors)

Replay of Messages and Orphan Processes

- Incorrect replay of messages after recovery can lead to *orphan* processes. This should be avoided