

# Distributed Systems (ICE 601)

## *Fault Tolerance*

Dongman Lee  
ICU

## Class Overview

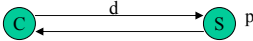
- Introduction
- Failure Model
- Fault Tolerance Models
  - state machine
  - primary-backup

# Introduction

- Dependability
  - availability
  - reliability
  - safety
  - maintainability
- Fault
  - failure, error, & fault
  - system is considered *faulty* once its behavior is no longer consistent with its specification [Schneider]
- Separation property of distribution systems lead to *partial failure property*
  - components that one component depends on may fail to respond due to various reasons
    - ♦ system or network failure
    - ♦ system or network overload

Distributed Systems - Fault Tolerance

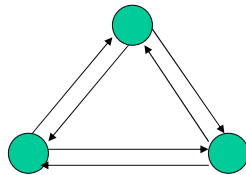
# Failure Model

- Failure semantics
    - description of the ways in which a service may fail
- 
- The diagram illustrates a client-server interaction. On the left is a green circle labeled 'C' representing the client. On the right is a green circle labeled 'S' representing the server. A double-headed arrow connects the two circles, with the letter 'd' positioned above the arrow, signifying a dependency or a communication channel between the client and the server.
- recovery actions depends on the likely failure behavior of the server when its failure is detected
  - designers should ensure that the behavior of the server conforms to a specified failure semantics
    - ♦ e.g. network with omission/time failure semantics
      - need to guarantee detection of message corruption such as checksum
    - ♦ stronger failure semantics costs more in general
  - adequacy of failure semantics would require preliminary stochastic analyses

Distributed Systems - Fault Tolerance

## Failure Model (cont.)

- Representative faulty behavior
  - Byzantine failures
    - ♦ system exhibits arbitrary and malicious behavior which may collude with other systems
  - fail-stop failures
    - ♦ when system fails, it changes to a state that allows others to detect its failure and then stops



Distributed Systems - Fault Tolerance

## Failure Model (cont.)

- Failure classification [Cristian]
  - omission failure
  - timing failure (performance failure)
  - response failure
  - crash failure

Distributed Systems - Fault Tolerance

## Failure Model (cont.)

- Failure classification : omission failure
  - a server omits to respond to an input
    - ♦ fail to perform actions a process or communication channel is supposed to do
  - communication omission failures
    - ♦ fail to transport a message from a sender's outgoing buffer to a receiver's incoming buffer
    - ♦ possible causes
      - buffer overflow and/or transmission error
    - ♦ derived failures
      - send-omission failure
      - channel failures
      - receive-omission failures

## Failure Model (cont.)

- Failure classification: timing failure (performance failure)
  - a server responds correctly but not in time (early or late)
  - applicable only in synchronous systems
    - ♦ time limits are set on process execution, message delivery, and clock drift rate
  - clock failures
    - ♦ exceeding the bounds on clock drift rate
  - performance failures
    - ♦ exceeding the bounds on the interval between two processing steps or message transmission

## Failure Model (cont.)

- Failure classification: response failure (arbitrary failure)
  - the term *arbitrary* or Byzantine to describe the worst possible failure semantics (cf. omission and timing failures are called *benign*) -> a server responds incorrectly
  - a process arbitrarily omits intended steps or takes unintended steps
    - > set or return wrong values
      - ◆ value failure: incorrect output
      - ◆ state transition failure: incorrect state transition
      - ◆ can't detect by timeout
  - communication arbitrary failures
    - ◆ message contents corruption or delivery of non-existent messages and duplicate messages
    - ◆ detect by checksums or sequence numbers

## Failure Model (cont.)

- Failure classification: crash failure
  - a server repeatedly fails to respond to inputs: process omission failures
    - ◆ *crash*: halt and remain halted
    - ◆ a process crash is *fail-stop* if other processes can detect certainly that the process has crashed
      - detection by timeout in synchronous systems
      - cf. asynchronous systems
  - failure management depends on server state at restart
    - ◆ amnesia crash: no record of state at crash; reset to initial state
    - ◆ partial amnesia crash: partially recorded
    - ◆ pause crash: restart in state before crash
    - ◆ halting crash: no restart

## Failure Model (cont.)

- Masking failures
  - by hiding failures or by converting them into a more acceptable type of failures (e.g., checksums)
    - ♦ retransmission - masking communication omission failures
    - ♦ replication - masking process crashes
  - reliable communication (masking communication omission failures)
    - ♦ validity: any message is eventually delivered
    - ♦ integrity: the identical message is delivered exactly once
      - duplicate checking by sequence numbers
      - security measures against spurious message and replaying or tampering with messages

## Fault-Tolerant Approaches

- Fault tolerance
  - can detect a fault and either fail predictably or mask the fault from users
  - hiding the occurrence of errors in system components and communications
  - ⇒ incorporate *redundant* processing component to achieve fault tolerance
- *k-resilient/fault-tolerant*
  - a set of systems satisfies its specification if no more than  $k$  systems become faulty
  - $k$  is chosen based on statistical measures of system reliability
    - ♦ Byzantine failure:  $2k+1$
    - ♦ fail-stop failure:  $k+1$

## Fault-Tolerant Approaches (cont.)

- Two approaches to support fault tolerance (fault masking)
  - hierarchical masking
    - ♦ hierarchical failure and recovery management
      - error detection in layered communication protocols
      - various levels of error abstraction in OS
  - group failure masking
    - ♦ state-machine approach
    - ♦ primary-backup approach
- Fault tolerance support can be done
  - hardware
    - ♦ stable storage
  - software
    - ♦ replicated servers

## State-Machine Approach

- Requirements for  $k$  fault-tolerant state machine
  - all replicas receive and process the same sequence of requests
    - ♦ agreement: every non-faulty replica receives every request
      - specify the interaction behavior of a client with state machine replicas
      - relaxed for read-only request in fail-stop failures
    - ♦ order: every non-faulty replica processes requests it receives in the same relative order
      - specify the behavior of state machine replicas in term of how to process requests from clients
      - relaxed for commutative requests

## State-Machine Approach (cont.)

- Agreement requirement
  - to satisfy agreement requirement, state-machines should support a message broadcasting protocol which conforms to
    - ♦ IC1: all non-faulty processors agree on the same value
    - ♦ IC2: if sender of request is non-faulty, then all non-faulty processors use its value as the one on which they agree
  - message broadcasting protocol is called Byzantine agreement protocol or reliable broadcast protocol

## State-Machine Approach (cont.)

- Order requirement
  - to implement order requirement requires
    - ♦ assignment of unique identifier to each message
    - ♦ stability (a request is ready to be delivered once all the previous requests have been delivered) test
  - assumptions on order requirement
    - ♦ O1: requests issued by a single client to a given state machine  $sm$  are processed by  $sm$  in the order they were issued
    - ♦ O2: if the fact that a request  $r$  was made to a state machine  $sm$  by a client  $c$  could have caused a request  $r'$  to be made by a client  $c'$  to  $sm$ , then  $sm$  processes  $r$  before  $r'$
  - three approaches
    - ♦ logical clock-based
    - ♦ synchronized real-time clock-based
    - ♦ replica-generated identifiers-based



## State-Machine Approach (cont.)

- Order requirement: logical clock-based
  - only for fail-stop failures
  - unique id assignment: logical clock
    - ♦ LC1: timestamp is incremented after each event at p
    - ♦ LC2: upon receipt of a message with timestamp t, process p resets its timestamp  $T_p$  to  $\max(T_p, t)+1$
  - stability test
    - ♦ a request is stable at replica  $sm_i$  if a request with larger timestamp has been received by  $sm_i$  from every client running on a non-faulty processor
      - messages between a pair of processors are delivered in the order sent
      - processor p detects that a failstop process q has failed only after p has received q's last message sent to p

## State-Machine Approach (cont.)

- Order requirement: synchronized physical clock-based
  - unique id assignment
    - ♦ no client makes two or more requests between successive clock ticks => every message will have greater timestamp than its previous message (satisfies O1)
    - ♦ degree of clock synchronization is better than minimum message delivery time => timestamps of two causally related messages issued by two clients will be such that earlier one should have lower timestamp than later one (satisfies O2)
  - stability test tolerating Byzantine failures
    - ♦ request r is *stable* if local clock reads T and  $uid(r) < T-d$  (d: worst case message delivery time)
    - ♦ request r is *stable* if a request with larger uid has been received from every client

## State-Machine Approach (cont.)

- Order requirement: replica-generated identifiers-based
  - 2 phases are used
    - ♦ phase 1: replicas propose uid as part of agreement protocol (SEEN)
    - ♦ phase 2: one of candidates is selected and becomes uid (ACCEPTED)
  - stability test
    - ♦ request  $r$  that has been accepted by  $sm_i$  is *stable* if there is no request that has
      - been seen by  $sm_i$ ,
      - not been accepted by  $sm_i$ , and
      - for which  $cuid(sm_i, r) \leq uid(r)$  holdswhere  $cuid(sm_i, r) = \max(SEEN_i, ACCEPT_i) + 1 + i$   
 $SEEN_i$ : largest  $cuid(sm_i, r)$  assigned to any request  $r$  so far seen by  $sm_i$   
 $ACCEPT_i$ : largest uid( $r$ ) assigned to any request  $r$  so far accepted by  $sm_i$   
 $uid(r) = \max_{sm_j \in NF} (cuid(sm_j, r))$  where  $NF$  be the set of replicas from which candidate unique identifiers(*cuid*'s) were received

## State-Machine Approach (cont.)

- Order requirement: replica-generated identifiers-based (cont.)
  - stability test for Byzantine failure
    - ♦  $sm_i$  uses timeout for agreement protocol for disseminating *cuid*
    - ♦ if  $sm_i$  determines that  $sm_j$  timeout has happened, broadcast “ $sm_j$  timeout” to all replicas
    - ♦ decision group is a set of replicas except any replica which  $k+1$  or more replicas have determined was timeout

# Primary-Backup Approach

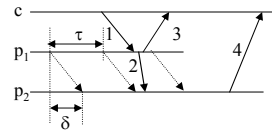
- Cost metrics of primary-backup protocols
    - degree of replication
      - ♦ # of servers for fault tolerance
    - blocking time
      - ♦ worst case period between a request and its response in any failure-free execution
    - failover-time
      - ♦ worst-case period during which requests can be lost because there is no primary
- ⇒ Smallest degree of replication, blocking time, failover-time for  $k$ -fault-tolerance?

# Primary-Backup Approach (cont.)

- Protocol properties
    - Pb1: there is at most one server whose state satisfies a condition being a primary
      - ♦ no more than one server is the primary at a time
    - Pb2: each client maintains a server identity to which the client can send a message
      - ♦ a client sends a request to the service by sending it to the server it believes to be the primary
    - Pb3: if a client request arrives at a server that is not a primary, then that request is not enqueued (thus, not processed)
      - ♦ messages to a backup are ignored
    - Pb4: there exist fixed value  $k$  and  $\Delta$  such that the service behaves like a single  $(k, \Delta)$ -bofo server\*
- \*  $(k, \Delta)$ -bofo server (bounded outage, finitely often) : all server failures can be grouped into at most  $k$  intervals of time with each interval having length at most  $\Delta$

## Primary-Backup Approach (cont.)

- Simple primary-backup protocol
  - assumption
    - ♦ one primary server  $p_1$  and one backup server  $p_2$ , connected via a communication link (message delivery time upper bound:  $\delta$ )
    - ♦ operations when  $p_1$  receives a request from a client
      - processes the request and updates its state
      - send update info to  $p_2$  (a state update message)
      - send a response to the client without waiting for ack from  $p_2$
    - ♦  $p_1$  sends a dummy message every  $\tau$  seconds; If  $p_2$  receives a dummy message for  $\tau + \delta$  seconds,  $p_2$  becomes a primary
  - spec conformance
    - ♦ Pb1:  $(p_1 \text{ has not crash}) \wedge (p_2 \text{ has not received a message from } p_1 \text{ for } \tau + \delta) = \text{false}$ 
      - Failover time:  $\tau + 2\delta$
    - ♦ Pb2: client  $c$  sends a message to  $p_1$
    - ♦ Pb3: requests are not sent to  $p_2$  until after  $p_1$  has failed
    - ♦ Pb4: a single  $(1, \tau + 4\delta)$ -bofo server



Distributed Systems - Fault Tolerance

## State-machine vs. Primary-backup

- Comparison

	State-machine	Primary-backup	Remarks
Arbitrary Failure support	Yes	No	$2k+1$ replication for $k$ -resilience
Request loss	No	Possible	Loss happens when a primary fails
Failure handling	Voting	Failover	
Request copy	as many servers as $k$ -resilience suffices	Only to primary	$2k+1$ for arbitrary $k+1$ for failstop
Overall cost	expensive	cheap	Primary-backup approach is more popular in commercial applications

Distributed Systems - Fault Tolerance