DISTRIBUTED SYSTEMS (COMP9243)

Lecture 8 (A): Fault Tolerance

Slide 1

A Failure
A Reliable Communication
A Process Resilience
B Recovery

Slide 2

DEPENDABILITY

Availability: system is ready to be used immediately
Reliability: system can run continuously without failure
Safety: when a system (temporarily) fails to operate correctly, nothing catastrophic happens
Maintainability: how easily a failed system can be repaired

Building a dependable system comes down to controlling failure and faults.

Slide 3

CASE STUDY: AWS FAILURE 2011

April 21, 2011
EBS (Elastic Block Store) in US East region unavailable for about 2 days
13% of volumes in one availability zone got stuck
led to control API errors and outage in whole region
led to problems with EC2 instances and RDS in most popular region
due to reconfig error and re-mirroring storm.
http://aws.amazon.com/message/65648/

Slide 4

AWS EBS Overview:
Region → Availability Zones
Clusters → Nodes → Volumes
Volume: replicated in cluster
Control Plane Services: API for volumes for whole region
Networks: primary, secondary

What happened?:
US east AZ
network config problem
re-mirroring storm
CP API thread starvation
node race condition
CP election overload
**Failure**

**Terminology:**

Failure: a system fails when it does not meet its promises or cannot provide its services in the specified manner.

Error: part of the system state that leads to failure (i.e., it differs from its intended value).

Fault: the cause of an error (results from design errors, manufacturing faults, deterioration, or external disturbance).

Recursive:
- Failure can be a fault
- Manufacturing fault leads to disk failure
- Disk failure is a fault that leads to database failure
- Database failure is a fault that leads to email service failure

**Total vs Partial Failure**

Total Failure:
- All components in a system fail
  - Typical in nondistributed system

Partial Failure:
- One or more (but not all) components in a distributed system fail
  - Some components affected
  - Other components completely unaffected
  - Considered as fault for the whole system

**Categorising Faults and Failures**

Types of Faults:

- Transient Fault: occurs once then disappear
- Intermittent Fault: occurs, vanishes, reoccurs, vanishes, etc.
- Permanent Fault: persists until faulty component is replaced

Types of Failures:

- Process Failure: process proceeds incorrectly or not at all
- Storage Failure: “stable” secondary storage is inaccessible
- Communication Failure: communication link or node failure

**Failure Models**

Crash Failure: a server halts, but works correctly until it halts
  - Fail-Stop: server will stop in a way that clients can tell that it has halted.
  - Fail-Resume: server will stop, then resume execution at a later time.
  - Fail-Silent: clients do not know server has halted

Omission Failure: a server fails to respond to incoming requests
  - Receive Omission: fails to receive incoming messages
  - Send Omission: fails to send messages
Response Failure: a server’s response is incorrect
Value Failure: the value of the response is wrong
State Transition Failure: the server deviates from the correct flow of control
Timing Failure: a server’s response lies outside the specified time interval
Arbitrary Failure: a server may produce arbitrary response at arbitrary times (aka Byzantine failure)

Failure Detector:
- Service that detects process failures
- Answers queries about status of a process

Reliable:
- Failed – crashed
- Unsuspected – hint

Unreliable:
- Suspected – may still be alive
- Unsuspected – hint

Fault Tolerance:
Fault Tolerance:
- System can provide its services even in the presence of faults

Goal:
- Automatically recover from partial failure
- Without seriously affecting overall performance

Techniques:
- Prevention: prevent or reduce occurrence of faults
- Prediction: predict the faults that can occur and deal with them
- Masking: hide the occurrence of the fault
- Recovery: restore an erroneous state to an error-free state

Synchronous systems:
- Timeout
- Failure detector sends probes to detect crash failures

Asynchronous systems:
- Timeout gives no guarantees
- Failure detector can track suspected failures
- Combine results from multiple detectors
- How to distinguish communication failure from process failure?
- Ignore messages from suspected processes
- Turn an asynchronous system into a synchronous one
**Failure Prevention**

Make sure faults don’t happen:
- Quality hardware
- Hardened hardware
- Quality software

**Failure Prediction**

Deal with expected faults:
- Test for error conditions
- Error handling code
- Error correcting codes
  - checksums
  - erasure codes

**Failure Masking**

Try to hide occurrence of failures from other processes

Mask:
1. Communication Failure → Reliable Communication
2. Process Failure → Process Resilience

**Redundancy:**
- Information redundancy
- Time redundancy
- Physical redundancy
**RELIABLE COMMUNICATION**

- Communication channel experiences failure
- Focus on masking crash (lost/broken connections) and omission (lost messages) failures

**RELIABLE POINT-TO-POINT COMMUNICATION**

- Reliable transport protocol (e.g., TCP)
  - Masks omission failure
  - Not crash failure

---

**Two Army Problem:**

Non-faulty processes but lossy communication.

Slide 18

- 1 → 2 attack!
- 2 → 1 ack
- 2: did 1 get my ack?
- 1 → 2 ack ack
- 1: did 2 get my ack ack?
- etc.

Consensus with lossy communication is impossible. Why does TCP work?

---

**Example: Failure and RPC:**

Possible failures:
- Client cannot locate server
- Request message to server is lost
- Server crashes after receiving a request
- Reply message from server is lost
- Client crashes after sending a request

How to deal with the various kinds of failure?
**RELIABLE GROUP COMMUNICATION**

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
<th>Receiver</th>
<th>Receiver</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>History buffer</td>
<td>Network</td>
<td>Last = 24</td>
<td>Last = 24</td>
<td>Last = 24</td>
</tr>
</tbody>
</table>

**SCALABILITY OF RELIABLE MULTICAST**

**Feedback Implosion:** sender is swamped with feedback messages

**Nonhierarchical Multicast:**
- Use NACKs
- Feedback suppression: NACKs multicast to everyone
- Prevents other receivers from sending NACKs if they’ve already seen one.
  - Reduces (N)ACK load on server
  - Receivers have to be coordinated so they don’t all multicast NACKs at the same time
  - Multicasting feedback also interrupts processes that successfully received message

**Hierarchical Multicast:**

**PROCESSES RESILIENCE**

**Process Resilience**
Protection against process failures
Groups:
- Organise identical processes into groups
  - Process groups are dynamic
  - Processes can be members of multiple groups
  - Mechanisms for managing groups and group membership
- Deal with all processes in a group as a single abstraction

Flat vs Hierarchical Groups:
- Flat group: all decisions made collectively
- Hierarchical group: coordinator makes decisions

**Replication**
Create groups using replication

**Primary-Based:**
- Primary-backup
- Hierarchical group
- If primary crashes others elect a new primary

**Replicated-Write:**
- Active replication or Quorum
- Flat group
- Ordering of requests (atomic multicast problem)

**Fault Tolerance:**
- Can survive faults in $k$ components and still meet its specifications
- $k + 1$ replicas enough if fail-silent (or fail-stop)
- $2k + 1$ required if if byzantine
Each replica executes as a state machine:
- $\text{state + input} \rightarrow \text{output + new state}$
- All replicas process same input in same order
- Deterministic: All correct replicas produce same output
- Output from incorrect replicas deviates

Input Messages:
- All replicas agree on content of input messages
- All replicas agree on order of input messages
- Consensus (also called Agreement)

What can cause non-determinism?

**Atomic Multicast**

A message is delivered to either all processes, or none
Requires agreement about group membership

**Process Group:**
- Group view: view of the group (list of processes) sender had when message sent
- Each message uniquely associated with a group
- All processes in group have the same view

**View Synchrony:**
A message sent by a crashing sender is either delivered to all remaining processes (crashed after sending) or to none (crashed before sending).

**Implemening View Synchrony:**
- **Stable message**: a message that has been received by all members of the group it was sent to.
- Implemented using reliable point-to-point communication (TCP)
- Failure during multicast $\rightarrow$ only some messages delivered

**Why?**
**AGREEMENT**

Examples: Election, transaction commit/abort, dividing tasks among workers, mutual exclusion

- Previous algorithms assumed no faults
- What happens when processes can fail?
- What happens when communication can fail?
- What happens when byzantine failures are possible

We want all nonfaulty processes to reach and establish agreement (within a finite number of steps)

---

**VARIANTS OF THE AGREEMENT PROBLEM**

**Consensus:**

- each process proposes a value
- communicate with each other...
- all processes decide on same value
- for example, the maximum of all the proposed values

**Interactive Consistency:**

- all processes agree on a decision vector
- for example, the value that each of the processes proposed

**Byzantine Generals:**

- commander proposes a value
- all other processes agree on the commander’s value

---

**Correctness of agreement:**

**Termination** all processes eventually decide

**Agreement** all processes decide on the same value

**Validity** C the decided value was proposed by one of the processes

IC the decided value is a vector that reflects each of the processes proposed values

BG the decided value was proposed by the commander

---

**CONSSENSUS IN A SYNCHRONOUS SYSTEM**

**Assume:**

- Execution in rounds
- Timeout to detect lost messages
Byzantine Generals Problem:
Reliable communication but faulty processes.

- $n$ generals (processes)
- $m$ are traitors (will send incorrect and contradictory info)
- Need to know everyone else’s troop strength $g_i$
- Each process has a vector: $(g_1, \ldots, g_n)$
- (Note: this is actually interactive consistency)

Byzantine Generals Impossibility:

- If $m$ faulty processes then $2m + 1$ nonfaulty processes required for correct functioning

Byzantine agreement with Signatures:
- Digitally sign messages
- Cannot lie about what someone else said
- Avoids the impossibility result
- Can have agreement with 3 processes and 1 faulty

Consensus in an Asynchronous System

Assume:
- Arbitrary execution time (no rounds)
- Arbitrary message delays (can’t rely on timeout)
IMPOSSIBILITY OF CONSENSUS WITH ONE FAILURE

Impossible to guarantee consensus with \( \geq 1 \) faulty process

Proof Outline:

- Fischer, Lynch, Patterson (FLP) 1985
- The basic idea is to show circumstances under which the protocol remains forever indecisive
- Bivalent (any result is possible) vs univalent (only single result is possible) states

1. There is always a bivalent start state
2. Always possible to reach a bivalent state by delaying messages → no termination

In practice we can get close enough