LTAT.06.007 Distributed Systems
Lecture 10 – Fault tolerance

Huber Flores, PhD
ASSOCIATE PROFESSOR

Tartu, Estonia 20/04/2020
Recap

• Explored the reasons that make replication necessary.
• Learned the relation between consistency and replication
  ▪ Different types of replication
  ▪ Levels of consistency
Agenda

• **Goal:** To understand how to handle failures in distributed systems

• **Content:**
  - Fault tolerance and dependability
  - Failure models
  - Process resilience
  - Consensus algorithms
  - Distributed commit protocols
  - Recovery from failures

**After this lecture, you should be able to:**

• Apply fault tolerance and recovery techniques to any system
Introduction to fault tolerance

Distributed systems are suspect to partial failures

**Design goal:** recovery from partial failures without seriously affecting overall performance

**Note**

- Being fault tolerant is strongly related to what are called **dependable systems**
- A key technique to handle failures is redundancy
Dependability

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. (Components are processes or channels.)
## Dependability

### Requirements of dependable systems

- **Availability**
  - Probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users

- **Reliability**
  - Probability that the system can run continuously without a failure for an interval of time

- **Safety**
  - When a system temporarily fails to operate correctly, nothing catastrophic should happen

- **Maintainability**
  - Refers to how easily a failed system can be repaired
Reliability versus availability

<table>
<thead>
<tr>
<th>Reliability R(t) of component C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional probability that C has been functioning correctly during ([0, t)) given C was functioning correctly at time (T = 0).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Traditional metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Mean Time To Failure</strong> (MTTF): The average time until a component fails.</td>
</tr>
<tr>
<td>• <strong>Mean Time To Repair</strong> (MTTR): The average time needed to repair a component.</td>
</tr>
<tr>
<td>• <strong>Mean Time Between Failures</strong> (MTBF): Simply MTTF + MTTR.</td>
</tr>
</tbody>
</table>
Reliability versus availability

### Availability A(t) of component C

Average fraction of time that C has been up-and-running in interval \([0, t)\).

- **Long-term availability** \(A: A(\infty)\)

- **Note:**
  \[
  A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}
  \]

### Observation

**Reliability** and **availability** make sense only if we have an accurate notion of what a **failure** actually is.
## Terminology (fault tolerance)

### Failure, error, fault

- **Failure**: A component is not living up to its specifications.
- **Error**: part of system’s state that may lead to failure
- **Fault**: The cause of an error
  - **Transient fault**: occurs once and then disappears
  - **Intermittent fault**: occurs, vanishes of its own accord, reappears, ...
  - **Permanent fault**: continues to exist until repaired

### Fault tolerance

A system can provide its services even in the presence of faults.
Failure models

Different types of failures

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td></td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server’s response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td></td>
</tr>
<tr>
<td>Value failure</td>
<td>A server’s response is incorrect</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

Observation

- **Fail-stop failures**: Server crash is announced by the server or properly detected by other processes
- **Fail-silent failures**: Server crash is only detected by other processes, possibly incorrectly
- **Fail-safe failures**: Server exhibits arbitrary failures in a benign way
Redundancy for failure masking

Redundancy is key technique for masking faults

• Information redundancy
  ▪ Extra bits are added to allow recovery from garbled bits
• Time redundancy
  ▪ (Trans)action is redone for no harm if needed
• Physical redundancy
  ▪ Extra hardware or processes added to tolerate loss
  ▪ Example: TMR (Triple Modular Redundancy)
Process resilience

Process replication

• Organizing several identical processes into a group allows masking a faulty process
  ▪ When a message is sent to the group, all processes receive it
  ▪ Process groups may be dynamic
    New groups can be created and old groups can be destroyed
    A process can join a group or leave one during system operation
    A process may be a member of several groups at the same time

• Allows treating a collection of processes as a single abstraction

• Need mechanisms for managing groups and group memberships
Process resilience

Flat groups
- All processes are equal
- No single point of failure
- Complicated decision making

Hierarchical groups
- Coordinator delegates requests to workers
- Single point of failure
- Clear decision making
Process resilience

**Group membership**
- Method needed for creating and deleting groups
- Method needed for allowing processes to join and leave groups

**Group server**
- Handles all requests, maintains database of groups
- Pros: Straightforward, efficient, easy to implement
- Cons: Single point of failure

**Distributed group membership management with multicasting (Challenges)**
- Discovery of crashed group members
- Synchronizing leaving/joining with message delivery
- Rebuilding group in case of many crashes
Groups and failure masking

Important assumptions

• All members are identical
• All members process commands in the same order

Result: We can now be sure that all processes do exactly the same thing
Groups and failure masking

Process replication via primary-backup or replicated-write protocol

<table>
<thead>
<tr>
<th>Type</th>
<th>Failure Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary-backup (hierarchical group)</td>
<td>single point of failure</td>
</tr>
<tr>
<td>Replicated-write (flat group)</td>
<td>no single point of failure, cost of distributed coordination</td>
</tr>
</tbody>
</table>

k-fault tolerant group

When a group can mask any $k$ concurrent member failures ($k$ is called degree of fault tolerance).

How large does a k-fault tolerant group need to be?

- With halting failures (crash/omission/timing failures): we need a total of $k+1$ members as no member will produce an incorrect result, so the result of one member is good enough.
- With arbitrary failures: we need $2k+1$ members so that the correct result can be obtained through a majority vote.
Example on sequential consistency

In a fault-tolerant process group, each nonfaulty process executes the same commands, and in the same order, as every other nonfaulty process.

Reformulation

Nonfaulty group members need to reach consensus on which command to execute next.
# Flooding-based consensus

## System model

- A process group $P = \{P_1, \ldots, P_n\}$
- Fail-stop failure semantics, i.e., with reliable failure detection
- A client contacts a $P_i$ requesting it to execute a command
- Every $P_i$ maintains a list of proposed commands

## Basic algorithm (based on rounds)

- In **round** $r$, $P_i$ multicasts its known set of commands $C_i^r$ to all others
- At the end of $r$, each $P_i$ merges all received commands into a new $C_i^{r+1}$
- Next command $cmd_i$ selected through a **globally shared, deterministic function**: $cmd_i \text{ select}(C_i^{r+1})$. 
Flooding-based consensus
Flooding-based consensus
Flooding-based consensus
Summary

• $P_2$ received all proposed commands from all other processes → makes decision.
• $P_3$ may have detected that $P_1$ crashed, but does not know if $P_2$ received anything, i.e., $P_3$ cannot know if it has the same information as $P_2$) cannot make decision (same for $P_4$).
<table>
<thead>
<tr>
<th>Assumptions (rather weak ones, and realistic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• A <strong>partially synchronous</strong> system (in fact, it may even be asynchronous).</td>
</tr>
<tr>
<td>• <strong>Communication</strong> between processes may be <strong>unreliable</strong>: messages may be lost, duplicated, or reordered.</td>
</tr>
<tr>
<td>• <strong>Corrupted message can be detected</strong> (and thus subsequently ignored).</td>
</tr>
<tr>
<td>• All <strong>operations are deterministic</strong>: once an execution is started, it is known exactly what it will do.</td>
</tr>
<tr>
<td>• Processes may exhibit <strong>crash failures</strong>, but <strong>not arbitrary failures</strong>.</td>
</tr>
<tr>
<td>• Processes <strong>do not collude</strong>.</td>
</tr>
</tbody>
</table>

**Note**

Paxos depicts the implementation of flood-based consensus.
Consensus in faulty systems with arbitrary failures

Essence

We consider process groups in which communication between processes is **inconsistent**: (a) improper forwarding of messages, or (b) telling different things to different processes.
# Consensus in faulty systems with arbitrary failures

## System model

- We consider a primary $P$ and $n-1$ backups $B_1, \ldots, B_{n-1}$.
- A client sends $v \in \{T,F\}$ to $P$.
- Messages may be lost, but this can be detected.
- Messages cannot be corrupted beyond detection.
- A receiver of a message can reliably detect its sender.

## Byzantine agreement: requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA1</td>
<td>Every nonfaulty backup process stores the same value.</td>
</tr>
<tr>
<td>BA2</td>
<td>If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary had sent.</td>
</tr>
</tbody>
</table>

## Observation

1) Primary faulty, 2) Primary not faulty
Is \textbf{3k} processes not enough?
$3k+1$ processes is enough
Realizing fault tolerance

Observation

Considering that the members in a fault-tolerant process group are so tightly coupled, we may bump into considerable performance problems, but perhaps even situations in which realizing fault tolerance is impossible.

Question

- Are there limitations to what can be readily achieved?
- What is needed to enable reaching consensus?
- What happens when groups are partitioned?
Distributed consensus: when can it be reached

Formal requirements for consensus

- Processes produce the same output value
- Every output value must be valid
- Every process must eventually provide output
Consistency, availability, and partitioning

CAP theorem

Any networked system providing shared data can provide only two of the following three properties:

C: **consistency**, by which a shared and replicated data item appears as a single, up-to-date copy

A: **availability**, by which updates will always be eventually executed

P: Tolerant to the **partitioning** of process group.

Observation

In a network subject to communication failures, it is impossible to realize an atomic read/write **shared memory** that guarantees a response to every request.
## Failure detection

### Essence

> How can we reliably detect that a process has actually crashed?

### General model

- Each process is equipped with a failure detection module
- A process P **probes** another process Q for a reaction
- If Q reacts: Q is considered to be alive (by P)
- If Q does not react with t time units: Q is **suspected** to have crashed

### Observation for a **synchronous** system

> A suspected crash = a known crash
Practical failure detection

Implementation

• If P did not receive heartbeat from Q within time t: P suspects Q.
• If Q later sends a message (which is received by P):
  ▪ P stops suspecting Q
  ▪ P increases the timeout value t
• Note: if Q did crash, P will keep suspecting Q.
Reliable client-server communication

<table>
<thead>
<tr>
<th>Communication channel may exhibit</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Crash failures (broken connection)</td>
</tr>
<tr>
<td>• Omission failures (lost messages)</td>
</tr>
<tr>
<td>• Timing failures</td>
</tr>
<tr>
<td>• Arbitrary failures</td>
</tr>
</tbody>
</table>

Reliable point-to-point communication with reliable transport protocol (e.g. TCP)

| • TCP masks omission failures (lost messages) by using acknowledgements and retransmissions |
| • Crash failures are not masked → exception is raised to inform the client |
Reliable client-server communication

What can go wrong?

- The client is unable to locate the server.
- The request message from the client to the server is lost.
- The server crashes after receiving a request.
- The reply message from the server to the client is lost.
- The client crashes after sending a request

Two “easy” solutions

- (cannot locate server): just report back to client
- (request was lost): just resend message
## RPC semantics in presence of failures

### Different classes of failures in RPC systems

1. The client is unable to locate the server
   - Server may be down or binding fails due to interface mismatch
   - Error raises an **exception** (error signal)
     Not all languages support exceptions and error signals
     Compromises transparency
2. The request message from the client to the server is lost
   Start a timer when message is sent, resend if timer expires
3. The server crashes after receiving the request
4. The reply message from the server to the client is lost
   Client assigns each request a sequence number, server filters requests that have already been carried out
5. The client crashes after sending a request
RPC semantics in presence of failures

Problem
Where (a) is the normal case, situations (b) and (c) require different solutions. However, we don’t know what happened. Two approaches:

• **At-least-once-semantics:** The server guarantees it will carry out an operation at least once, no matter what.

• **At-most-once-semantics:** The server guarantees it will carry out an operation at most once.
# Why fully transparent server recovery is impossible

## Three type of events at the server

(Assume the server is requested to update a document.)

- **M**: send the completion message
- **P**: complete the processing of the document
- **C**: crash

## Six possible orderings

(Actions between brackets never take place)

1. **M → P → C**: Crash after reporting completion.
2. **M → C → P**: Crash after reporting completion, but before the update.
3. **P → M → C**: Crash after reporting completion, and after the update.
4. **P → C (→ M)**: Update took place, and then a crash.
5. **C (→ P → M)**: Crash before doing anything
6. **C (→ M → P)**: Crash before doing anything
Why fully transparent server recovery is impossible

<table>
<thead>
<tr>
<th>Reissue strategy</th>
<th>Strategy M → P</th>
<th>Strategy P → M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPC</td>
<td>MC(P)</td>
</tr>
<tr>
<td>Always</td>
<td>DUP</td>
<td>OK</td>
</tr>
<tr>
<td>Never</td>
<td>OK</td>
<td>ZERO</td>
</tr>
<tr>
<td>Only when ACKed</td>
<td>DUP</td>
<td>OK</td>
</tr>
<tr>
<td>Only when not ACKed</td>
<td>OK</td>
<td>ZERO</td>
</tr>
</tbody>
</table>

OK = Document updated once
DUP = Document updated twice
ZERO = Document not updated at all
Reliable RPC: Client crash

Client crash

1. Results in orphan(s) (computation) at servers
   • Waste CPU cycles, lock up files, tie up resources
   • Generate replies that can confuse rebooted client
2. Solution 1: extermination
   • Client logs its actions on permanent storage, kills orphans after reboot
   • Problems: expensive disk writes, grandorphans, partitioned network
3. Solution 2: reincarnation
   • Client divides time into sequentially numbered epochs, after reboot broadcasts start of new epoch, server kills remote calls of old epochs
   • In partitioned network replies of old epochs are easily detected
4. Solution 3: gentle reincarnation
   • Like reincarnation, but upon epoch broadcast each server kills calls whose owner cannot be found
5. Solution 4: expiration
   • Each RPC is given a standard amount of time, T, to finish
   • If client waits time T after rebooting, orphans are sure to be gone
Simple reliable group communication

Intuition

A message sent to a process group $G$ should be delivered to each member of $G$. Important: make distinction between receiving and delivering messages.
Simple reliable group communication

Reliable communication, but assume nonfaulty processes

Reliable group communication now boils down to **reliable multicasting**: is a message received and delivered to each recipient, **as intended by the sender**.
Reliable communication in the presence of faulty processes

Group communication is reliable when it can be guaranteed that a message is received and subsequently delivered by all nonfaulty group members.

Tricky part

Agreement is needed on what the group actually looks like before a received message can be delivered.
## Problem

Have an operation being performed by each member of a process group, or none at all.

- **Reliable multicasting:** a message is to be delivered to all recipients.
- **Distributed transaction:** each local transaction must succeed.
Two-phase commit protocol (2PC)

Essence

The client who initiated the computation acts as coordinator; processes required to commit are the participants.

• Phase 1a: Coordinator sends VOTE-REQUEST to participants (also called a pre-write)

• Phase 1b: When participant receives VOTE-REQUEST it returns either VOTE-COMMIT or VOTE-ABORT to coordinator. If it sends VOTE-ABORT, it aborts its local computation

• Phase 2a: Coordinator collects all votes; if all are VOTE-COMMIT, it sends GLOBAL-COMMIT to all participants, otherwise it sends GLOBAL-ABORT

• Phase 2b: Each participant waits for GLOBAL-COMMIT or GLOBAL-ABORT and handles accordingly.
2PC - Finite state machines

(a) Coordinator

(b) Participant
2PC – Failing participant

Analysis: participant crashes in state S, and recovers to S

- **INIT**: No problem: participant was unaware of protocol

- **READY**: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make) log the coordinator’s decision

- **ABORT**: Merely make entry into abort state idempotent, e.g., removing the workspace of results

- **COMMIT**: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.
2PC – Failing participant

Alternative

When a recovery is needed to READY state, check state of other participants → no need to log coordinator’s decision

Recovering participant P contacts another participant Q

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Result

If all participants are in the READY state, the protocol blocks. Apparently, the coordinator is failing. Note: The protocol prescribes that we need the decision from the coordinator.
### 2PC – Failing coordinator

<table>
<thead>
<tr>
<th>Essence</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>The real problem lies in the fact that the coordinator’s final decision may not be available for some time (or actually lost).</td>
<td>Let a participant P in the READY state timeout when it hasn’t received the coordinator’s decision; P tries to find out what other participants know (as discussed).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes</td>
</tr>
</tbody>
</table>
class Coordinator:

def run(self):
    yetToReceive = list(participants)
    self.log.info('WAIT')
    self.chan.sendTo(participants, VOTE_REQUEST)
    while len(yetToReceive) > 0:
        msg = self.chan.recvFrom(participants, TIMEOUT)
        if (not msg) or (msg[1] == VOTE_ABORT):
            self.log.info('ABORT')
            self.chan.sendTo(participants, GLOBAL_ABORT)
            return
        else: # msg[1] == VOTE_COMMIT
            yetToReceive.remove(msg[0])
            self.log.info('COMMIT')
    self.chan.sendTo(participants, GLOBAL_COMMIT)
```
class Participant:
    def run(self):
        msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if not msg:  # Crashed coordinator - give up entirely
            decision = LOCAL_ABORT
        else:  # Coordinator will have sent VOTE_REQUEST
            decision = self.do_work()
            if decision == LOCAL_ABORT:
                self.chan.sendTo(coordinator, VOTE_ABORT)
            else:  # Ready to commit, enter READY state
                self.chan.sendTo(coordinator, VOTE_COMMIT)
                msg = self.chan.recvFrom(coordinator, TIMEOUT)
                if not msg:  # Crashed coordinator - check the others
                    self.chan.sendTo(all_participants, NEED_DECISION)
                    while True:
                        msg = self.chan.recvFromAny()
                        if msg[1] in [GLOBAL_COMMIT, GLOBAL_ABORT, LOCAL_ABORT]:
                            decision = msg[1]
                            break
                    else:  # Coordinator came to a decision
                        decision = msg[1]
            while True:  # Help any other participant when coordinator crashed
                msg = self.chan.recvFrom(all_participants)
                if msg[1] == NEED_DECISION:
                    self.chan.sendTo([msg[0]], decision)
```
# Recovery: Background

## Essence

When a failure occurs, we need to bring the system into an error-free state:

- **Forward error recovery**: Find a new state from which the system can continue operation
- **Backward error recovery**: Bring the system back into a previous error-free state

## Practice

Use backward error recovery, requiring that we establish recovery points

## Observation

Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover
Consistent recovery state

Requirement
Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line
Assuming processes regularly **checkpoint** their state, the most recent **consistent global checkpoint**.
## Coordinated checkpointing

### Essence

Each process takes a checkpoint after a globally coordinated action.

### Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a checkpoint request message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a checkpoint done message to allow all processes to continue

### Observation

It is possible to consider only those processes that depend on the recovery of the coordinator, and ignore the rest
Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let CP_i (m) denote mth checkpoint of process Pi and INT_i (m) the interval between CP_i (m-1) and CP_i (m).
- When process P_i sends a message in interval INT_i (m), it piggybacks (i,m).
- When process P_j receives a message in interval INT_j (n), it records the dependency INT_i (m) → INT_j (n).
- The dependency INT_i (m) → INT_j (n) is saved to storage when taking checkpoint CP_j (n).

Observation

If process P_i rolls back to CP_i (m-1), P_j must roll back to CP_j (n-1).
Observation

If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time. We have a so-called **cascaded rollback**.

![Diagram of cascaded rollback](image)
## Message logging

### Alternative

Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint → store messages in a log.

### Assumption

We assume a piecewise deterministic execution model:
- The execution of each process can be considered as a sequence of state intervals
- Each state interval starts with a nondeterministic event (e.g., message receipt)
- Execution in a state interval is deterministic

### Conclusion

If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.
When should we actually log messages?

Avoid **orphan processes**:
• Process Q has just received and delivered messages $m_1$ and $m_2$
• Assume that $m_2$ is never logged.
• After delivering $m_1$ and $m_2$, Q sends message $m_3$ to process R
• Process R receives and subsequently delivers $m_3$: it is an orphan.
Summary

What we learned?

• The importance of fault tolerance
• Replication as fault-tolerance technique
• Trade-offs between recovery and performance
Next lecture(s)

Parallel programming (Prof. Eero Vainikko)

Transportation systems (Dr. Ammir Hadachi)
Questions?

E-mail: huber.flores@ut.ee