LTAT.06.007 Distributed Systems
Lecture 12 – Fault tolerance II

Huber Flores, PhD
ASSOCIATE PROFESSOR

Tartu, Estonia 03/05/2021
Recap

• Explored failure models and requirements of dependable systems

• Process resilience and consensus algorithms
  ▪ Arbitrary failures
  ▪ Crash failures and others
Roadmap (so far)

Logical order of events

- \( a \)
- \( b \)
- \( c \)

Global clock

Internal clock in components

- Amazon.com
- Ut.ee

Printer1
Agenda

Logical order of events

Global clock

Internal clock in components

Printer1
Agenda

• **Goal:** To understand failure detection and recovery in distributed systems

• **Content:**
  - Consensus algorithms
    - Other types of failures
  - Distributed commit protocols
  - Recovery from failures

After this lecture, you should be able to:

• Apply fault tolerance and recovery techniques to any system
More about consensus
Realistic consensus

Implementations

• Paxos
  • it depicts the implementation of flood-based consensus.
• Raft
We consider process groups in which communication between process 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 B₁,…, Bₙ₋₁.
• A client sends ν ∈ {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

BA1: Every nonfaulty backup process stores the same value.
BA2: If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary had sent

Observation

1) Primary faulty, 2) Primary not faulty
Consensus in faulty systems with arbitrary failures

P – Primary (Coodinator)
B – Backup (Participant)
Is \textit{3k} processes not enough?

(a) 

(b)
$3k+1$ 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
CAP
# 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.
When the network fails?
## Reliable client-server communication

**Communication channel may exhibit**

- Crash failures (broken connection)
- Omission failures (lost messages)
- Timing failures
- Arbitrary failures

**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
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
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

<table>
<thead>
<tr>
<th>Three type of events at the server</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Assume the server is requested to update a document.)</td>
</tr>
<tr>
<td>M: send the completion message</td>
</tr>
<tr>
<td>P: complete the processing of the document</td>
</tr>
<tr>
<td>C: crash</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Six possible orderings</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Actions between brackets never take place)</td>
</tr>
<tr>
<td>1. M → P → C: Crash after reporting completion.</td>
</tr>
<tr>
<td>2. M → C → P: Crash after reporting completion, but before the update.</td>
</tr>
<tr>
<td>3. P → M → C: Crash after reporting completion, and after the update.</td>
</tr>
<tr>
<td>4. P → C (→ M): Update took place, and then a crash.</td>
</tr>
<tr>
<td>5. C (→ P → M): Crash before doing anything</td>
</tr>
<tr>
<td>6. C (→ M → P): Crash before doing anything</td>
</tr>
</tbody>
</table>
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
Distributed commit protocols
Distributed commit protocols

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

Coordinator

(b)

Participant

(a)
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

#### Essence

The real problem lies in the fact that the coordinator’s final decision may not be available for some time (or actually lost).

#### Alternative

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).

#### Observation

Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes.
Coordinator in Python

```python
class Coordinator:
    def run(self):
        yetToReceive = list(participants)
        self.log.info('WAIT')
        self.chan.sendStatus(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.sendStatus(participants, GLOBAL_ABORT)
                return
            else:  # msg[1] == VOTE_COMMIT
                yetToReceive.remove(msg[0])
                self.log.info('COMMIT')
        self.chan.sendStatus(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
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 $\text{CP}_i(m)$ denote $m$th checkpoint of process $P_i$ and $\text{INT}_i(m)$ the interval between $\text{CP}_i(m-1)$ and $\text{CP}_i(m)$.

• When process $P_i$ sends a message in interval $\text{INT}_i(m)$, it piggybacks $(i,m)$.

• When process $P_j$ receives a message in interval $\text{INT}_j(n)$, it records the dependency $\text{INT}_i(m) \rightarrow \text{INT}_j(n)$.

• The dependency $\text{INT}_i(m) \rightarrow \text{INT}_j(n)$ is saved to storage when taking checkpoint $\text{CP}_j(n)$.

Observation

If process $P_i$ rolls back to $\text{CP}_i(m-1)$, $P_j$ must roll back to $\text{CP}_j(n-1)$. 
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.
### 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.
Message logging and consistency

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?
• Consensus and arbitrary failures
• Failure detection
• Recovery techniques
Next lecture(s)

Parallel programming (Prof. Eero Vainikko)

Transportation systems (Dr. Amnir Hadachi)
Questions?

E-mail: huber.flores@ut.ee