CS 5523 Operating Systems: Fault Tolerance

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu and Dr. Palden Lama for providing their slides.
Outline

- Terminology: fault, error and failures
- Fault management and failure models
- Fault tolerance (agreement) with redundancy
  - Level of redundancy vs. failure models
- Fault recovery techniques
- Checkpointing and stable storage
- Recovery in distributed systems:
  - Consistent checkpointing
  - Message logging
Components provide services to clients
- For distributed systems: either processes or channels
- Component may require the services from other components and depends on them

Faults are common
- Machine crash, software bugs, disks fail, packets lost

Faults: single machines vs. distributed systems
- Distributed systems: coordination and agreement

Questions:
- Can we hide the effects of faults?
- Can we recover from failures?
Fault Tolerance Properties

- **Availability**
  - What percentage of time is a system available for use?

- **Reliability**
  - How long can a system run **continuously without failure**? Fail 1ms out of 1 hour, availability > 99.9999% but not reliable

- **Safety**
  - Small failures should not have catastrophic effects

- **Maintainability**
  - How easy is it to repair faults?

- High reliability $\neq$ high availability!

A time interval  An instant (point) in time
Terminology

- **Failure**: what you can see
  - A system cannot meet its promises, like failure to provide a service

- **Error**: cause of failure
  - The state of system that may lead to a failure

- **Fault**: cause of an error

Being fault tolerant is called as **Dependable System**.
Types of Faults

- **Transient** faults
  - occur once and then disappear
  - E.g., a bird disturb the beam of microwave transmission

- **Intermittent** faults
  - disappear and reappear: *unpredictable and hard to diagnose*
  - E.g., Loose contact on a connector

- **Permanent** faults
  - Continue to exist until faulty component are repaired/replaced
  - E.g., software bugs or burnt out chips
Outline

- Terminology: fault, error and failures
- Fault management and failure models
- Fault tolerance (agreement) with redundancy
  - Level of redundancy vs. failure models
- Fault recovery techniques
- Checkpointing and stable storage
- Recovery in distributed systems:
  - Consistent checkpointing
  - Message logging
Failure Models

- **Crash** failure
  - component simply halts, but behaves correctly before halting

- **Omission** failure
  - component fails to respond

- **Timing** failure
  - correct output, but lies outside a specified real-time interval

- **Response** failure
  - incorrect output

- **Arbitrary/Byzantine** failure:
  - Arbitrary/Malicious output

Crash failures are the least severe; arbitrary failures are the worst
# Failure Models in Distributed Systems

<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>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to send 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>A server’s response is incorrect</td>
</tr>
<tr>
<td>Value failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>
Fault Detection

- Fail-stop
  - Notifies before crashing

- Fail-silent
  - No responses after crashes

- Fail-safe
  - Junk output that can be recognized

- Byzantine failure
  - Cannot be detected easily
Fault Management

- **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
Fault Tolerance Techniques

- **Redundancy** and agreement
  - Hiding effect of faults

- **Recovery** and rollback
  - Bringing system to a consistent state
Outline

- Terminology: fault, error and failures
- Fault management and failure models
- Fault tolerance (agreement) with redundancy
  - Level of redundancy vs. failure models
- Fault recovery techniques
- Checkpointing and stable storage
- Recovery in distributed systems:
  - Consistent checkpointing
  - Message logging
Redundancy Techniques

- **Redundancy** is the key technique to tolerance faults

- **Information** redundancy
  - e.g., parity and Hamming codes

- **Time** redundancy
  - e.g., re-execution or execute secondary/backup copy

- **Physical** (software/hardware) redundancy
  - e.g., extra cpus, multi-versions softwares
Triple Modular Redundancy (TMR)

- If A2 fails $\rightarrow$ V1: majority vote $\rightarrow$ all B get good result
- What if V1 fails?!
TMR (cont.)

Correct results are obtain via **majority vote**

- Mask **ONE** fault
Level of Redundancy

- Depends on
  - How many faults can a system handle?
  - What kind of faults can happen?

- k-fault tolerant system
  - Can handle k faulty components

- Assume crash failure semantics (i.e., \textit{fail-silent})
  - \( k + 1 \) components are needed to survive \( k \) failures
Level of Redundancy (cont.)

- Assume **arbitrary** (but **non-malicious**) failure semantics and group output defined by voting
  - Independent component failures → possible same results
  - 2k+1 components are needed to survive k component failures (**majority vote**)

- Assume **Byzantine (malicious)** failure semantics and group output defined by voting
  - Faulty components cooperate to cheat!!!
  - 3k+1 components are needed to tolerate k failures → **two-thirds** are needed for the agreement from other 2k+1 non-faulty components;
Outline

- Terminology: fault, error and failures
- Fault management and failure models
- Fault tolerance (agreement) with redundancy
  - Level of redundancy vs. failure models
- Fault recovery techniques
- Checkpointing and stable storage
- Recovery in distributed systems:
  - Consistent checkpointing
  - Message logging
Fault Recovery

- **Main idea**: when a failure occurs, we need to bring the system into an *error-free* state

- **Forward** recovery
  - Find a new state from which system continue operation
  - E.g., Error-correction codes
  - Problem: how to correct errors and move to a new state

- **Backward** recovery
  - Bring the system back into a *previous* error-free state
  - E.g., packet retransmission
  - Problem: keeping error-free state (checkpoints)
Outline

- Terminology: fault, error and failures
- Fault management and failure models
- Fault tolerance (agreement) with redundancy
  - Level of redundancy vs. failure models
- Fault recovery techniques
- Checkpointing and stable storage
- Recovery in distributed systems:
  - Consistent checkpointing
  - Message logging
Checkpoints

- Periodically store system states on stable storage
- At error-recovery, go back to the last checkpointed state

Example: packet retransmission
Stable Storage

Main idea: replicate all data on at least two disks, and keep one copy “correct” at all times.

- (a) Stable storage
- (b) Crash after drive 1 is updated
- (c) Bad spot
Recovery in Distributed Systems

- **Consistent Recovery State**
  - Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a **consistent state** from where to recover
  - **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, **recovery line is the most recent consistent global checkpoint**
Cascaded Rollback: Domino Effects

- If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time
Recovery with Checkpoints

- Initial state
- Recovery line
- Checkpoint
- Failure
- Inconsistent collection of checkpoints

Message sent from P2 to P1
Independent Checkpointing

- Each process independently takes checkpoints
  - Let $CP[i](m)$ denote $m^{th}$ checkpoint of process $Pi$ and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
  - When process $Pi$ sends a message in interval $INT[i](m)$, it piggybacks $(i,m)$
  - When process $Pj$ receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
  - The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$
- If process $Pi$ rolls back to $CP[i](m-1)$, $Pj$ must roll back to $CP[j](n-1)$.
Coordinated Checkpointing

- Each process takes a checkpoint after a **globally coordinated** action

- **Simple solution**: 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, queuing the incoming 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

- Incoming messages will not be considered as part of checkpoints.
Message Logging

- Checkpoints are expensive to make
  - store messages in a log \(\Rightarrow\) replay your (communication) behavior from the most recent checkpoint

- Assumption: 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

However, with the use of multithreading, this doesn’t hold for most applications
Message Logging and Consistency

**Problem:** When should we actually log messages?

Avoid **orphans:** survive the crash of another process, but whose state is inconsistent with the crashed process after recovery

- Process Q has just received messages $m_1$ and $m_2$
- Assume that $m_2$ is never logged
- After getting $m_1$ and $m_2$, Q sends message $m_3$ to process R
- Process R receives and subsequently delivers $m_3$
Summary

- Terminology: fault, error and failures
- Fault management and failure models
- Fault tolerance (agreement) with redundancy
  - Level of redundancy vs. failure models
- Fault recovery techniques
- Checkpointing and stable storage
- Recovery in distributed systems:
  - Consistent checkpointing
  - Message logging
DOUBLETAKE: Evidence-Triggered Dynamic Analysis

Tongping Liu, Charlie Curtsinger, Emery Berger
Memory Corruptions:
Old but Prominent Security Threats

- Unsafe languages (C/C++) are prone to memory errors:
  - Buffer Overflows
  - Use-after-free/Dangling pointers
  - Double frees
  - Invalid frees
  - Memory leaks

OpenSSL fails to do proper bounds checking on a buffer.
Impact: Community Health Systems lost 4.5 million patient records. Canada Revenue Agency reported a theft of Social Insurance Numbers……
Buffer Overflow

A buffer overflow is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory.

```c
char buf[5];
……
strcpy(buf, “overflow”);
```

Most overflows are triggered only at specific inputs, which makes them hard to be detected.
Checking every memory access introduces significant overhead, preventing its uses in real deployment.
Basic Idea of DOUBLETAKE

Program Progress

Snapshot

Normal execution

Irrevocable system calls

Check errors

Errors detected

Rollback

Instrumented re-execution

Report errors
Snapshot a program at the beginning of each epoch:

a. User registers

b. Stack

c. Changeable memory: heap and globals of apps and libraries
Normal Execution Phase

✧ Preparing for detection
  • Implant tripwires

✧ Preparing for deterministic replaying
  • Handle system calls correspondingly
    – Repeatable: getpid
    – Recordable: time
    – Revocable: read, write
    – Irrevocable: Socket IO, fork, exec, open

Note:
  a. Intercepting glibc APIs
  b. Revocable read/write: tracking file position pointers
Detecting Memory Errors

- **Tripwires (canaries)** for detecting buffer overflows

<table>
<thead>
<tr>
<th>Red = possible bad object</th>
<th>known random value</th>
<th>dead canary = corruption</th>
<th>Blue = not bad object</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 10 2 9 3</td>
<td>4 5 1 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Detecting Memory Errors

- **Use-after-frees**

Quarantine (FIFO)

- allocated Heap object

- Valid pointer to free

- header | canary | requested | canary

- freed object 1

- freed object 2

- header | canary | requested | canary

- allocated Heap object

- header | canary

- Invalid-frees
Deterministic Replay to Identify Errors

- **Watchpoint Mechanism**
- **Deterministic Memory Uses**

Red = possible bad object

Blue = not bad object

known random value  
dead canary = corruption

8 10 2 9 3 4 5 1 7

a. Report where problems are  
b. Repeatable replaying
Precisely Report Memory Errors

- **Buffer Overflows**
  - Object allocation site and overflowing sites

- **Memory Use-after-free**
  - Memory allocation and de-allocation site
  - Memory reference site after de-allocation

- **Double-free**
  - Memory allocation and de-allocation site
  - Double-free site

- **Memory leak**
  - Memory allocation site
Future Work

Fault Tolerance:
- Memory errors
- Deadlock
- Race conditions
- …..

Conservative Protection

Deployed Applications

Error/Anomaly Detection

Attack Identification

Nullifying Vulnerabilities

Compiler-aided Automatic Fixes

Traditional Software Testing Process