Motivating Questions

1. What are faults?
2. What is fault-tolerance?
3. What is the difference between software fault-tolerance and hardware fault-tolerance?
4. Why do we need to give special consideration to software fault-tolerance?
5. Who should care about it? (Analyst/Designer/Programmer?)
6. How do we ensure that a system tolerates faults?

After this lecture, you should have a clear idea of how to address above questions.

Outline

- Basic concepts
  - Faults, errors, failures
  - Types and nature of fault
- Challenges in software fault-tolerance
- Fault-tolerance mechanisms
  - Recovery blocks
  - Checkpointing & recovery
    - Non-Transparent & Transparent Approaches
    - State machine approach
- A fundamental theory of fault-tolerance
- Component-based design of fault-tolerance
- Verification and synthesis of fault-tolerance
Fault

- An event in the physical domain of a system
  - Component failure in hardware systems
  - Divide by zero
  - A wire is stuck at a fixed voltage
  - A process restarts
  - A message is lost in the communication channel
  - A process occasionally misses a message in communicating with others
  - A process behaves arbitrarily
  - An input sensor is corrupted
  - Load surges in the network

How about design inadequacies? (s/w, h/w)

Relation Between Faults, Errors, and Failures

- **Fault** causes an internal error state in the information domain
  - E.g., a process restarts and resets the value of all variables to zero

- **Error** states cause the observable system behaviors to go stray (failed behaviors)

- **Failure** is a deviation from specified/desired behavior
  - Depends on the specification

Fault Types

- **Crash**: a component crashes with an undetectable
  - E.g., a node crashes in a network without being detected by other nodes

- **Fail-stop**: a component fails in a detectable fashion

- **Omission**: a component does not perform a particular action
  - E.g., the receiver of a message does not reply by an ACK

- **Timing**: a component does not perform a particular action at the right time
  - E.g., the receiver of a message does not reply in a specific amount of time
Fault Types - Continued

- **Performance**: a component does not provide the required performance
  - E.g., congestion in communication channels

- **Assertive**: the communicated data is wrong (syntactically/semantically)

- **Byzantine**: a component behaves arbitrarily
  - E.g., a sensor arbitrarily changes its sampled data

Nature of Faults

- **Permanent**: faults corrupt a component permanently
  - E.g., crash

- **Transient**: faults corrupt a component momentarily; i.e., appears once and then disappears
  - E.g., Electrical surge, spurious interrupt, illegal opcode

- **Intermittent**: faults corrupt a component sporadically; i.e., appear in a short time and disappear spontaneously
  - E.g., loose contact on a connector

Program Observation of Faults

- The ability of a program to observe faults
  - Detectable
    - E.g., fail-stop
  - Undetectable
    - E.g., transient faults

- Undetectable faults are hard to mask; mostly handled by self-stabilization
Fault-Tolerance

- Providing a desired level of functionality in the presence of faults
  - E.g., MC6800 provides recovery mechanism when executing an illegal opcode
  - A distributed files system works despite the failure of a node
  - A nuclear reactor shuts down safely when something bad happens
- How do we define the “desired level of functionality”?
- Can programs tolerate all faults?

We have to define our expectation of a system in the presence of faults

Fault-Tolerance - Continued

- Fault-tolerance is defined w.r.t system specification

- Example:
  - In the case of power outage in a hospital, the emergency power will be activated to power on safety-critical medical devices, however no TV will be powered on

- Often a weaker form of specification is satisfied in the presence of faults

Software Fault-Tolerance

- What is the difference between s/w and h/w fault-tolerance?
- Hardware faults often occur due to component failure
- Fault-tolerance can be achieved by replacing a component or having a stand-by spare
- Correct design is achievable for hardware systems
- Modular reasoning in hardware design
Software Fault-Tolerance Complexity

- Why is software fault-tolerance more complicated?
- The complexity of h/w systems is far less than s/w systems
  - The total number of states
  - Combination of components
- Software systems could easily have hundreds of millions of interacting computational components
- Combinatorial nature of software systems
  - Achieving correct design is difficult in software systems
  - Fault detection is much more difficult
  - Design inadequacy; i.e., design correctness is hard to achieve

Fault-Tolerance –
A Cross-Cutting Concern

• Fault-tolerance should be provided in all levels
• Fault-tolerance should be added to the components in such a way that the entire program is fault-tolerant

Software Fault-Tolerance Mechanisms
Design Approaches

- Recovery blocks [Randall 75]
  - Wrap program with blocks of code for recovery
- Checkpointing and recovery [Strom/Yemini 85]
  - In the absence of faults, save the state of the computations
  - In the presence of faults, restore the state of the system to a legitimate saved state
- State machine approach (Replication) [Schneider 90]
  - Server-client model
  - Servers as state machines
  - Replicate servers

Recovery Blocks

- **Recovery block**: Unit of error detection and recovery
- A mechanism for
  - Switching to a spare software component
  - Detection and recovery while keeping the complexity manageable
- Goal: provide progress for computing processes in the presence of faults
- Add recovery blocks to functional code
- Can have recovery block nesting

Recovery Blocks Syntax

```
<recovery block> ::= ensure <acceptance test> by
<primary alternate>
<other alternates> else error

<primary alternate> ::= <alternate>
<other alternates> ::= <empty> | <other alternates>
else by <alternate>

<alternate> ::= <statement listing>
<acceptance list> ::= <logical expression>
```

Recovery Blocks - Example

```plaintext
ensure consistent sequence (S)
by extend S with (i)
else by concatenate to S
else by warning “lost item”
else by S := construct sequence (i); warning
    “correction: lost sequence”
else by S := empty sequence; warning “lost sequence
    and item”
else error
```

Recovery Blocks - Alternates

- Primary alternate: perform the desired operation
  if the acceptance test fails
- Other alternates: perform desired operation in a
different fashion
- Example:
  `- S is a sequence of elements in an array
    ensure sorted(S)
    by quicksort(S)
    else by quicksort(S)
    else by bubblesort(S)
    else error

Providing Reset in Recovery Blocks

- For recovery
  `- Value of non-local variables must be available
    in original and modified form
- How to maintain restart information?
- How to realize which variable has been
  modified at run time?
- Recursive Cache
  `- Detect which non-local variable is modified and
    cache it
Recovery Blocks and Interacting Processes

- **Domino effect**
  - All processes in their 4th recovery block
  - Dashed lines show inter-process communication
  - What if process 1 fails?
  - What if process 3 fails?

![Diagram of processes]

Recovery Blocks and Interacting Processes - Continued

- **Causes of Domino effect**
  1. Uncoordinated recovery blocks
  2. Symmetric processes
     - In any pair of processes, the failure of one can cause the failure of the other
  - Inter-process dependencies must be taken into account
  - The global state of the system must be saved for restoration

Checkpointing and Rollback-Recovery
### Checkpointing and Recovery

- **Checkpoint**: the state of a process (program)
- **Two broad categories**
  - Checkpointing protocols
  - Log-based recovery protocols
- **Checkpointing**

Uncoordinated | Communication-Induced | Coordinated
---|---|---

---

### Uncoordinated Checkpointing

- Processes do the checkpointing without any coordination
  - Domino effect; rollback propagation
  - Complicates recovery
  - Still needs coordination for garbage collection & output commit; i.e., generating a consistent output

---

### Coordinated Checkpointing

- Processes coordinate to save the global consistent state of the system
  - Simplifies recovery and garbage collection
  - Acceptable practical performance
  - Requires global coordination

---

Communication-Induced Checkpointing

- Checkpointing is activated depending on the communication pattern of processes
- A global consistent state is saved based on the piggyback information
  - No Domino effect
  - Non-deterministic nature
    - Degrades performance
    - Complicates garbage collection

Implementation of Checkpointing-Recovery

- Non-Transparent
  - Provide language structure for programmers (e.g., recovery blocks) [Randall 75]
- Transparent
  - Middleware platforms for providing a fault-tolerant run-time system [ElnozahyAlvisiWangJohnson 2002]

Log-Based Recovery

- Combine checkpointing with the logging of non-deterministic events
  - Fault-tolerant systems that react with a non-deterministic environment
- Assumptions:
  - All non-deterministic events can be identified
  - The information necessary for replaying events can be logged
- A process can recreate its pre-failure state by replaying logged events
Schneider’s State Machine Approach

State Machine Approach

- To implement a fault-tolerant client-server system
  - Replicate the server
  - Develop a replica management protocol to coordinate the interactions between clients and the replicated server
- Model clients and servers as state machines
  - State variables
  - Atomic commands
- Failure model
  - Byzantine component behaves arbitrarily
  - Fail-stop components crash in a detectable way

Replicated Server

- Client 1
- Request
- Service
- ...
- Client m
- Request
- Service
- ...
- Replica 1
- ...
- Replica n
- Server
Replica Management Protocol

- Specification of a replicated server
  - **Agreement**: every non-faulty replica receives every request
  - **Order**: each non-faulty replica processes the requests in the same relative order

- Any correct implementation of the replicated server should satisfy the above properties

Summary

- Limitations of recovery blocks, checkpointing-recovery and state machine approaches
  - Type of faults that can be handled
  - Type of system where they can be deployed

- Limitations of the replication-based approach
  - Creates copies of the fault-intolerant program
  - Can only deal with Byzantine and fail-stop faults (transient faults?)
  - Can only be used for deterministic systems; i.e., for any input, only one correct output

Summary - Continued

- Checkpointing-recovery limitations
  - Only applicable for detectable faults (e.g., fail-stop)
  - Problematic if faults occur during recovery

- Today’s software systems are deployed in very dynamic environments
  - Change of configuration
  - Network faults
  - Adapt to sudden change of environmental conditions (network load variations, network intrusion, etc.)

- More importantly
  
  Can we anticipate all classes of faults at the design time?