NTNU – Trondheim
Norwegian University of
Science and Technology

TTK4145 – Real-time Programming

Lecture 9 – Fault Model & Software Fault Masking

Example exam questions

Gray had the thought that we could build fault tolerant computer systems if we only had reliable data storage, reliable communication and reliable calculations...

- From 2011:
  1. Refer shortly how we can achieve reliable storage (that is, built on unreliable storage)
  2. Refer shortly how we correspondingly can achieve reliable communication.

- From 2014:
  1. Reliable calculations are the most difficult in this triplet since the failure modes is difficult to describe in the general case. The curriculum sets up “Checkpoint/Restart” as the first alternative: Describe how this works.
  2. A simple and straightforward alternative to checkpoints like this is to write to the “log”. How does this work, and what do we achieve compared to the checkpoints?
  3. What is, in this context, the error mode(s) of a “calculation”, and how are they detected?
  4. Another way to achieve reliable calculations is process pairs. Give a short explanation on how this works. What do we gain here compared to checkpoint/restart?

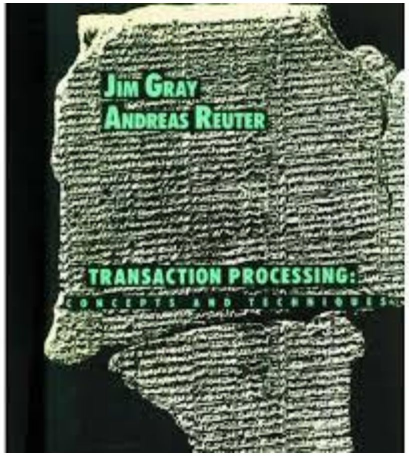
Learning goals:
Fault Model & Software Fault Masking

- Understanding of the three cases in low level design for fault tolerance by redundancy: Storage, Computation and Communication.

- Understanding of the work method:
  1. Find error model
  2. Detect errors and merge failure modes (+error injection for testing)
  3. Handling/masking with redundancy ... aiming for progression of failfast, reliable and available systems

- Ability to Implement (simple) Process Pairs-like systems

Fault Model & Software Fault Masking



- Lecture covers topics from Chapter 3.7 in **Transaction Processing: Concepts and Techniques** by Gray & Reuter

- To avoid confusion with other parts of the curriculum the
  - **storage**, **messages** and **processes**

    are in this lectured referred to as
  - **storage**, **communication** and **calculation**

Context of chapter

- Fault Model
  - Expanding the definition from last lecture
  - Failure modes, in which way can our software fail?
  - Probabilities of faults, how reliable are our software?
    - What is the *MMTF* (mean-time-to-fail) of the component?
  - Specification states reliability demand, which frequency of failure is tolerated?
    - How many hard drives do we need in parallel to get below required failure rate?

- Software Fault Masking
  - Masking by static redundancy
  - Faults become invisible through techniques such as N-version programming

Failfast

The crash and burn way of handling things

Failure modes revisited

- In the previous lecture, failure mode classifications were discussed, classifying systems in terms of how it responds to failures
- Additionally, the concept of merging failure modes to simplify error handling was introduced
- Did not specify an approach to handling the outcome of merged failure modes for a general system
- It is desirable that modules can self-check by examining internal data structures, states, and input parameters and determine if the component is in an error state
  - We already have this, acceptance tests!
  - *Defensive programming*, analogous to the concept of *defensive driving*

Definition

- Fault latency
  - The time between the occurrence of a fault (the error) and its detection (the failure)
- Failfast (failstop)
  - A module is *failfast* if it stops execution when it detects a fault, and additionally if it has a small fault latency
- Failfast behavior is important for containment
  - Latent errors can turn a single fault into a cascade of faults
  - Failfast minimizes fault latency and so minimizes latent faults

Properties

- Failfast creates a need for instant crash and restart
- No wrong outputs from the module
  - Better to do nothing than continue the computation in some unspecified way
  - The consequences of the error doesn’t spread to other modules
- Failfast modules make suitable building blocks
  - Pair & Spare
- Easy to compute reliability figures
  - Check how often component restarts
- Benefit of knowing exactly when a component fails
  - On the other hand, the component fails more often

The three cases

Storage, Communication, Calculation

A model for fault-tolerant programs

- Aim to construct highly available systems from unreliable components

- Focus on the general method, not so much the implementation details
  - The chapter follows a relatively systematic approach

- The model involves three entity types (cases):
  - **Storage**, underlying RAID (redundant array of independent disks)
  - **Communication**, underlying TCP
  - **Calculation**, ensuring that calculations are done reliably

- The three cases together constitute an infrastructure
  - Not that relevant for embedded systems
  - Drove development of transactional system

Systematic approach

- Progression towards fault tolerant systems
  - Failfast modules
    - Detect errors instantly
  - Reliable components
    - Behaves according to specification (functional demand)
  - Available system
    - Behaves according to specification (timing demand)

- The general approach
  - Identify failure modes
  - Detect errors and simplify (merging of error modes)
    - Add error injection for testing expected errors
    - Detect unexpected errors with acceptance tests
      - Test handling by injecting failed acceptance tests
  - Ensure error handling and software fault masking by static redundancy

Case 1 - Storage

You have to store data to make data

Failure Modes

- Contains an array of pages, each page is an array of bytes

- How can storing and retrieving data fail?
  - `status read(address, &data)`
    - `status` returns with error value, `read` simplify failed
    - The `data` returned is wrong due to bad `write()`
    - The `data` returned is out of date due to failed `write()`
    - The `data` is consistent, but comes from the wrong `address`
  - `status write(address, data)`
    - `status` returns with error value, `write` simplify failed
    - Writes the wrong or bad `data` to the page
    - Writes the `data` to the wrong `address`
    - Does not write anything at all

Error detection

- Store the page address as part of the page value
  - Detects `read()` of wrong page `address`
  - Detects `write()` to wrong page `address`

- Add a checksum to the saved data
  - Ensures upon `read()` that the data was correctly written

- Add a version id to the saved data
  - Detects if `read()` data is out of date
  - Detects if a `write()` failed to write
  - Needs redundancy to function (what is the versioned compared to?)

- Add a status bit to each page
  - Status bit indicates if there is a fault
  - Serves no real purpose, only used for injecting errors
  - Have separate thread that sporadically flips status bits to test fault tolerance

- Failfast simplification, all errors leads to fail of the module

- Create new functions for read and write, these will fail more often due to failfast
- Create the decay thread that flips status bits

Handling with redundancy

- Probability failure rates specifies the required redundancy to ensure availability according to specification
- Static redundancy: Use *n* copies of storage modules (hard drives)
  - Read and write to all redundant modules
  - Probability for *n* bad reads so small that it can be ignored
  - Keep increasing *n* if reliability is not acceptable
  - Compare version id, use data with newest id
- Dynamic redundancy: All reads leads to write-back on error
- Repair thread reads all pages regularly
  - Performs write-backs as required
  - Data used seldom might deteriorate due to age, repair thread refreshes the pages

store_read()

```
/* Reads a block from storage, performs acceptance
test and returns status */

bool store_read(group, address,&value){
   int result = read(group,address,value);
   if(result != 0 ||
      checksum fails ||
      stored address does not correspond to addr ||
      statusBit is set){
      return False;
   }else{
      return True;
   }
}
```

`store_decay()` *error injection*

```
/* There is one store_decay process for each store in the system */

#define mttvf 7E5 /* mean time (sec) to a page fail, a few days */
#define mttsf 1E8 /* mean time(sec) to disc fail is a few years */

void store_decay(astore store){
      Ulong addr;
      Ulong page_fail = time() + mttvf*randf();
      Ulong store_fail = time() + mttsf*randf();
      while (TRUE){
            wait(min(page_fail,store_fail) - time());
            if(time() >= page_fail){
                  addr = randf()*MAXSTORE;
                  store.page[addr].status = FALSE;
                  page_fail = time() - log(randf())*mttvf;
            }
            if (time() >= store_fail){
                  store.status = FALSE;
                  for (addr = 0; addr < MAXSTORE; addr++) store.page[addr].status = FALSE;
                  store_fail = time() + log(randf())*mttsf;
            }
      }
}
```

reliable_write()

```
#define nplex 2 /* code works for n>2, but do duplex */

Boolean reliable_write(Ulong group, address addr, avalue value){
    Boolean status = FALSE;

    for(int i = 0; i < nplex; i++ ){
        status = status ||
        store_write(stores[group*nplex+i],addr,value);
    }
    return status;
}
```

reliable_read()

```
bool reliable_read(group,addr,&value){
     bool status, gotone = False, bad = False;
     Value next;

     for(int i = 0; i < nplex; i++ ){
          status = store_read(stores[group*nplex+i],addr,next);
          if (! status ){
               bad = True;
          }else{ /* we have a good read */
               if(! gotone){
                    *value = next;
                    gotone = TRUE;
               } else if (next.version != value->version){
                    bad = TRUE;
                    if (next.version > value->version)
                         *value = next;
               }
          }
     }
     if (! gotone) return FALSE; /* disaster, no good pages */
     if (bad) reliable_write(group,addr,value); /* repair any bad pages */
     return TRUE;
```

`store_repair()` *repair process*

```
void store_repair(Ulong group){
   int i;
   avalue value;
   while(TRUE){
      for (i = 0; i <MAXSTORE; i++){
         wait(1);
         reliable_read(group,i,value);
      }
   }
}
```

Case 2 - Communication

You have to talk data to make data

Failure modes

- Message lost
  - Nothing was received
- Message delayed
  - The message arrives late or out of order
- Message arrived corrupted
  - The message contents changed during sending
- Message duplicated
  - Receiver gets several copies of same message
- Wrong recipient
  - The message received was not meant for the receiver

Error detection

- Implement a session id
  - Ensures that message are only received by intended receiver
- Send a checksum with the message
  - Detects corruption of data
- Send acknowledgment (ack) back to sender
  - send reply to indicate that the message was received correctly
- Implement sequence numbers
  - Ensures that duplicates, delayed and out of order messages are detected
- Error simplification: All errors handled as message lost

Handling with redundancy

- Dynamic redundancy: Timer & resend
  - Implement timer for ack
  - Resend message on absent ack

Case 3 - Calculations

You have to calculate data to make data

Failure modes (of any calculation)

- Does not do the next correct side effect
  - Side effect is the intended result of the action

Error detection

- Implement acceptance tests
  - Must ensure acceptance test is sufficient to capture all error modes
  - Error detection list from previous lecture
  - Specification defines expected output
  - Acceptance tests are not perfect, but far better thinking in terms of acceptance test rather then handling error returns
- Perform the acceptance test
  - Ensure that the behavior is consistent and specified
  - Then perform any side effects
  - Some side effects are impossible to undo
- Error simplification: Panic/stop (Failfast)
  - Better to do nothing than continue the computation in some unspecified way

Handling with redundancy

Three different ways of handling redundancy for calculations

1) Checkpoint – restart

- Similar to backwards recovery using recovery points

- Calculation is executed in steps
  1. Calculate result and side effect
  2. Acceptance test
     - If fail, restart from previous stored checkpoint of last execution
  3. Store full state as recovery point in reliable storage
     - Write the complete state to reliable storage module
  4. Do side effect
     - In case of restart, previous side effect might be executed twice

2) Process Pairs

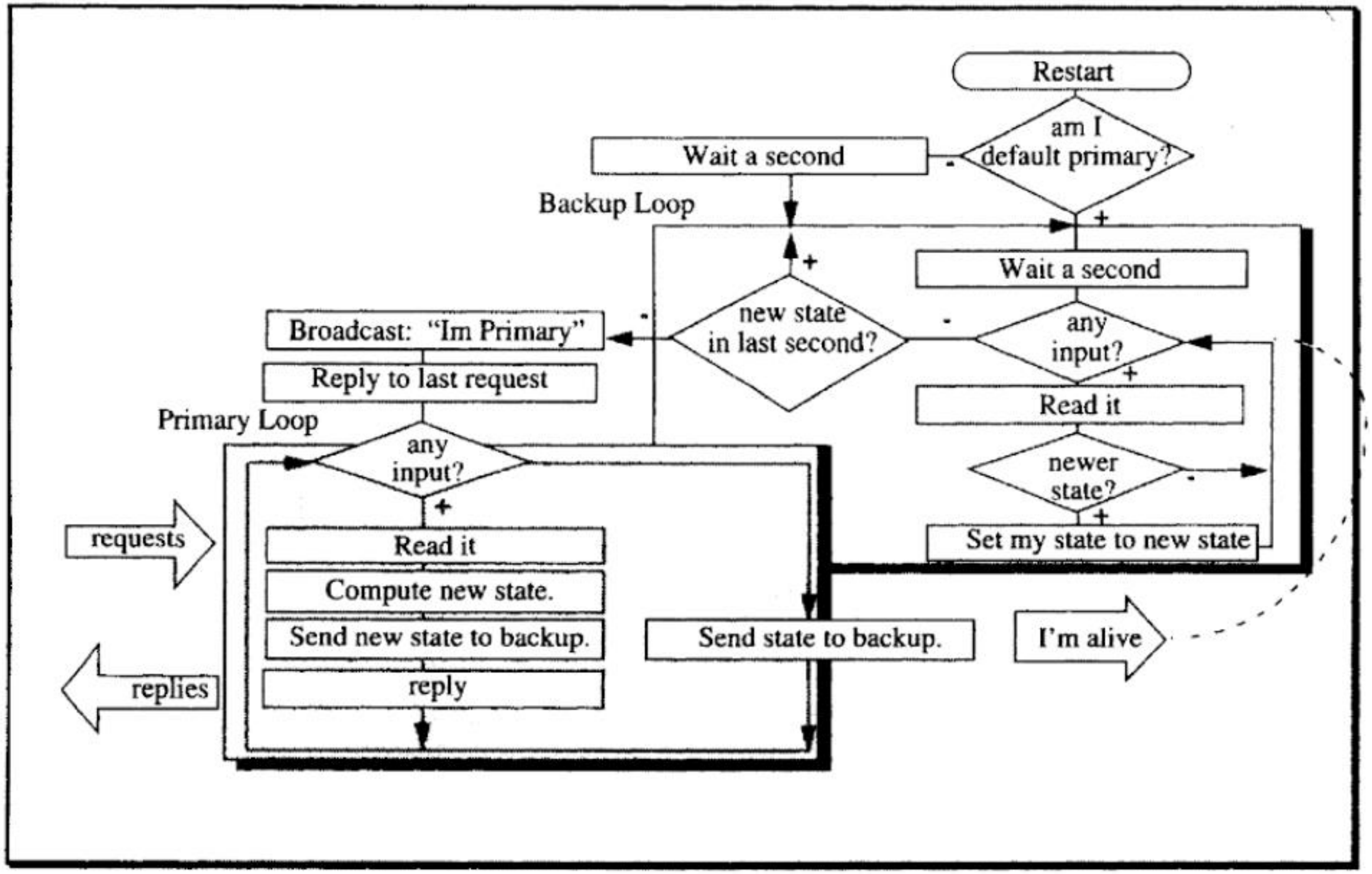
- Two processes : Primary & Backup
  - Primary performs the work and sends periodic “I’m alive” messages to Backup
  - Primary sends full state to Backup after state change
  - Backup is the checkpoint, takes over when the Primary fails
  - Failfast implementation, Primary instant crash and restart on failure

- Masks hardware failure (processor failures) as well as transient software failures (Heisenbugs)

- Redundancy by resending: Masks communication errors.

- New Primary can answer requests sent to old Primary
  - No explicit sessions between clients and Primary
  - Communication queues part of state sent to Backup

Process Pairs



Process Pairs Pseudocode

```
repeat
     // Backup mode
     Read checkpoint- & IAmAlive-messages
     Update state
Until last IAmAlive is too old

Broadcast: IAmPrimary

Finish active job // Possibly a duplicate

repeat
     // Primary mode
     if new request/task in job queue then // Part of the state
          do work
          if acceptance test fails then
               restart.
          else
               send checkpoint & IAmAlive // Unsafe communication!
          answer request/commit work.
     else
          if it is time to send then
                // Assumption that there will be more of these.
               send last checkpoint & IAmAlive.
```

3) Persistent processes

- Assumes transactional infrastructure
  - Databases to preserve data
  - Operating system that implement transactions
  - Not relevant for embedded, not even available on desktops
- All calculations are transactions
  - “atomic transformations from one consistent state to another”
  - Either done completely or not at all
- The processes are stateless
  - All data safely stored in highly available databases
  - First read data from database, then before side effects, write updated data back to database
- The vision from Gray is that all calculations will be done like this
  - Fault tolerant,
  - Failure are always handled perfectly
  - No calculations will be lost
  - The operating system takes care of fault tolerance
- Embedded systems want fault tolerance features, but they must be implemented from basic techniques