Crash Recovery in a Distributed Data Storage System

Lampson and Sturgis 1979

Presented by Raymond Xu

Motivation

- Crashes may cause the state of a system to become inconsistent

```
transfer(src, dst, amount) {
    src_bal = read(src)
    write(src, src_bal - amount)
    dst_bal = read(dst)
    write(dst, dst_bal + amount)
}
```

Motivation

- Crashes may cause the state of a system to become inconsistent

```
transfer(src, dst, amount) {
    src_bal = read(src)
    write(src, src_bal - amount)
    dst_bal = read(dst)
    write(dst, dst_bal + amount)
}
```

Goal: Atomic transactions

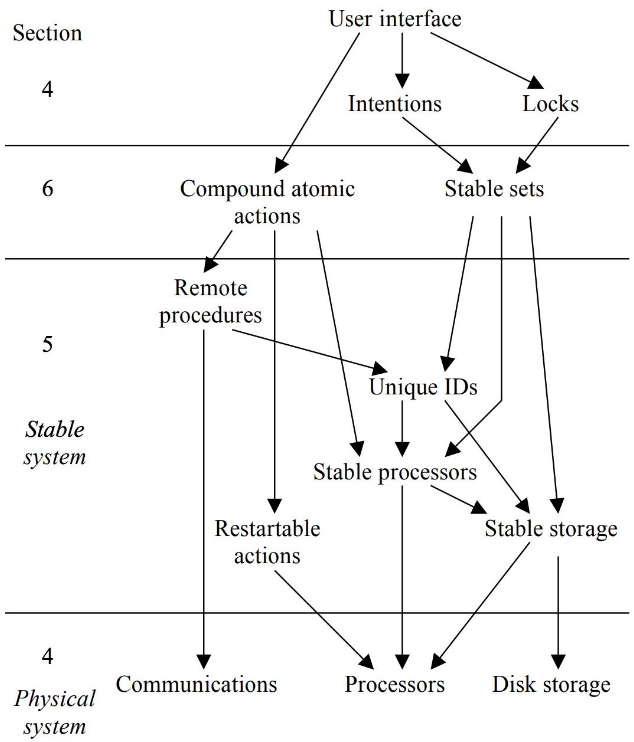
Foundation

- Distributed storage system
  - Clients and servers
  - Clients talk directly to servers containing data
  - Servers can send messages to each other
- Consistency
  - A state is consistent if it satisfies the invariant of the system
- Transaction
  - A sequence of read/write commands that, after recovery from a crash, will have all been executed or not executed at all

Failure model

- Events can be **desired** or **undesired**
- Undesired events can be **expected (errors)** or **unexpected (disasters)**
- The paper's algorithm assumes no disasters but any number of errors

Abstraction Lattice



Physical System

Disk storage

```
Get(at: Address) returns (status: (good, looksBad), data: Dblock)

Put(at: Address, data: Dblock)
```

Disk storage

```
Get(at: Address) returns (status: (good, looksBad), data: Dblock)
```

- (error) Soft read error
  - page is good, but status is bad
- (disaster) Persistent read error
  - soft read error but successively
- (disaster) Undetected error
  - page is bad but status is good, or data returned is wrong block

Disk storage

`Put(at: Address, data: Dblock)`

- (error) Null write
    - page is unchanged
- (error) Bad write
    - data is written to page, but page status becomes bad

Disk storage

- **Decay set:** pages that may decay together
- **Decay:** spontaneous event that changes a subset of a decay set from good to bad
- *Assumption:* it is possible to partition disk storage into pairs of units that are not decay-related

- (error) Infrequent decay
  - Interval T_D separates all decays in a unit
- (error) Revival
  - A page goes from bad to good
- (disaster) Frequent decay
  - No interval T_D in play
- (disaster) Undetected error
  - A page's data changes

Processor

- A crash is an error that resets the state of the processor (volatile state) to a standard value
- *Assumption:* No other processor errors are allowed — any malfunction can be detected and converted into a crash

Communication

```
Send(to: Processor, data: Mblock)

Receive returns (status: (good, bad), data: Mblock)
```

Communication

```
Send(to: Processor, data: Mblock)

Receive returns (status: (good, bad), data: Mblock)
```

- (error) Loss
  - some message is destroyed
- (error) Duplication
  - some new message identical to an existing one is created
- (error) Decay
  - some message status changes from good to bad
- (disaster) Undetected error
  - some message changes from bad to good, or the data or recipient process change for a good message

Atomicity

An action is atomic if it is:

1. **Unitary:** If the action returns, it was carried out completely; if the system crashes before the action returns, then after the crash the action has been carried out completely or not started

AND

2. **Serializable:** When actions are done by concurrent processes, the result is as if the individual actions were carried out one at a time in some order. Action order must also be preserved within a process.

Atomicity

An action is atomic if it is:

1. **Unitary:** If the … y; if the system crashes befor… action has been carried out co…

“Unfortunately, we are unable to make all the actions in our various abstractions atomic” (9)

2. **Serializable:** … ses, the result is as if the individual actions were carried out one at a time in some order. Action order must also be preserved within a process.

Stable System

Careful Storage

- `CarefulGet`
  - `Get` until good, or after `n` tries
    - Eliminates soft read errors
- `CarefulPut`
  - `Put` then `Get`, repeat until success
    - Eliminates null writes and bad writes

Stable Storage

Stable page comprised of an ordered-pair of careful disk pages (that are not decay-related) and a monitor

- `StableGet`
  - `CarefulGet` from first page, if bad then `CarefulGet` from second page
- `StablePut`
  - `CarefulPut` to first page then to second page
    - Atomic
- `Cleanup`
  - `CarefulGet` both pages:
    - If both good and same data, then do nothing
    - If one bad, then `CarefulPut` the good data to the bad address
    - If both good but different data, then `CarefulPut` one data to the other

Stable Storage

- Run `Cleanup` at initialization, after each crash, and at least every T_D
- *Invariant*: Both pages cannot be bad, and if both are good then they both have the most recent data, except during a `StablePut`
  - If first page decayed, then the other page cannot decay in T_D and cannot experience a bad write, so `Cleanup` will fix the bad page
  - If first page suffered a bad write, it will be fixed by `CarefulPut` or `Cleanup`

Stable Processors

1. Can use disk storage to store state
2. Can use stable storage to store state for crash recovery
3. Can use stable storage to construct stable monitors
    a. Monitor: data wrapper that uses locking to ensure mutual exclusion
    b. Stable monitor: all data besides the lock is in stable storage

Remote Procedures

- Use globally unique ids for messages, attainable via a stable monitor
- Periodic retries, deduplicate by id
- Receivers track largest seen id from each processor, ignore ids <= largest
- If receiving a message from a processor that you have no record of, request that processor’s current id

Compatible Actions

Weaker substitute for serializability

A history of actions is compatible if there is a serial history, in which there is no concurrency and each action occurs in isolation

The serial history must agree with the actual history:

- Mapping from serial history actions to actual history actions
- Actions that complete before a strong action starts will appear to occur before the strong action
- Actions that begin after a strong action ends will appear to occur after the strong action

Stable Sets and Compound Atomic Actions

Stable Sets

`StablePut` is good and atomic, but we want to support arbitrarily large data

A Stable Set has a unique id and a set of records

Pool: maintain a set of pages in stable storage for use by stable sets

Stable Sets

Atomic operations

- `Create(i: ID)`
- `Insert(s, t: StableSet, new: Record)`
- `Replace(s, t: StableSet, old, new: Record)`
- `IsEmpty(s, StableSet)`
- `IsMember(s: StableSet, r: Record)`

Non-atomic operations

- `Enumerate(s: StableSet, p: procedure)`
- `Erase(s: StableSet)`

Stable Sets

- All actions on a stable set are compatible
- But using atomic operations while non-atomic operations are in progress has arbitrary resolution
- The pool can be ordered in a ring to reuse space
- For a wide stable set (a stable set that spans more than one processor) store metadata in each processor (1 root, potentially many leaves)

Compound Atomic Actions

procedure A = **begin** `Save; R; Reset` **end**

- If *R* is an action compatible with a set of actions *S*, then *A* is an atomic action compatible with *S*:
  - If the procedure crashes before the `Save`, it's a no-op
  - If it crashes after the `Reset`, `R` has been done completely and will not be done again because the saved state has been erased
  - If it crashes between the `Save` and the `Reset`, it will resume after the `Save` and restart `R`

Transactions

2 phases:

1. Record intentions, after last action here the transaction is committed
2. Do the writes

If a crash occurs after commit but before all writes are complete, restart phase 2

- Recording intentions must be atomic
- Store intentions in a stable set, store locks in another stable set, committing the transaction and doing the writes is a compound atomic action

Transactions

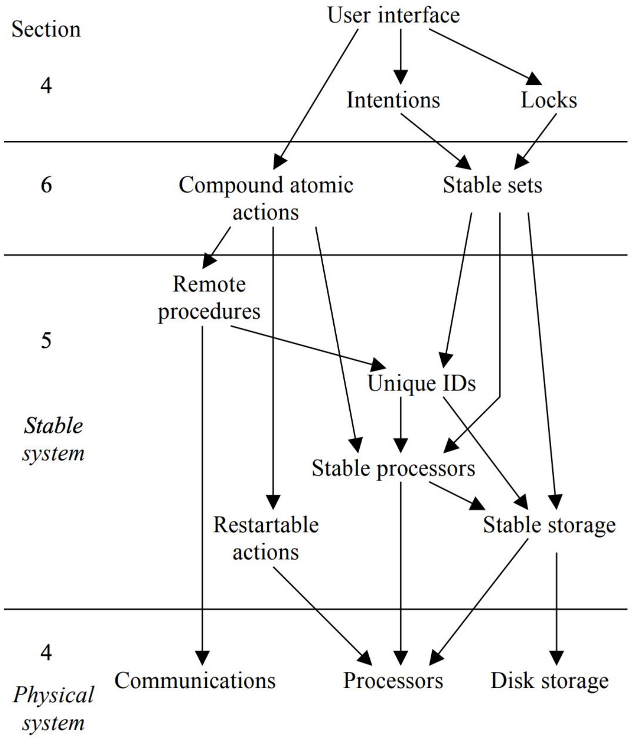
On each server store:

- Stable set containing Intentions for each page the server stores
  - Stable monitor protecting it
- Stable set containing transaction flags
  - Stable monitor protecting it
- If coordinator for a transaction, a root for a wide stable set containing transaction Intentions; a leaf if not coordinator but part of the transaction

Transactions

- Client calls `Begin` on one of the servers, which becomes the coordinator
- Client calls `Read` and `Write` on the servers that contain the corresponding pages
- After all the `Write` calls return, client calls `End` or `Abort` on the coordinator

Abstraction Lattice



Crash Recovery in a Distributed Data Storage System

Lampson and Sturgis 1979

Presented by Raymond Xu