Consistency in Distributed Systems

Achour Mostefaoui

ANR Descartes Fontainbleau 8-10 novembre 2021
Plan

- Distributed shared data structures
- Safety properties and progress conditions
- Strong consistency (linearizability and sequential consistency)
- Weak consistency
- Progress conditions
- Conclusion
Distributed Computation

- Processes and concurrency
  - Processes may interact directly through share data structures if there is a shared memory
  - Or through messages exchanged by the different processes
Distributed shared data structures

- The basic operations are read/write on registers and possibly special instructions (C&S, T&S, ...)
- or, send/receive of messages through a communication network
- However shared data structures used by distributed applications may be more sophisticated: stacks, queues, sets, logs, graphs, etc.
- These data structures are not offered natively by processors
- Each operation on a shared data structure corresponds to a code (a function) that can be complex
Distributed shared data structures

- The push operation in Trieber/IBM’s Stack (from D. Hendler)

```
Push(int v, Stack S)
1. n := new NODE ; create node for new stack item
2. n.val := v ; write item value
3. do forever ; repeat until success
4. node top := S.top
5. n.next := top ; next points to current top (LIFO order)
6. if compare&swap(S, top, n) ; try to add new item
7. return ; return if succeeded
8. end do
```
Distributed shared data structures

- The enqueue operation in Mickael & Scott’s Queue (from D. Hendler)

```java
public boolean enq(T value) {
    Node node = new Node(value);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(null, node)) {
                    tail.compareAndSet(last, node);
                    return;
                }
            } else {
                tail.compareAndSet(last, next);
            }
        }
    }
}
```
The ABD simulation (Attiya, Bar-Noy and Dolev 1995)

- It has been proved in 1995 that a shared memory (shared registers) can be emulated over a distributed system provided that there is a majority of processes that do not crash.

However special instructions cannot be implemented on a message-passing system prone to process crashes.
A data structure is defined by two properties:
- A safety property
- A progress condition

Safety: It questions the meaningfulness of the results returned by operations on shared objects

Progress: will there be a returned value and when?
Consistency and Progress Conditions

Safety properties

Ideally: The best consistency for an implemented shared object is the one that makes it indistinguishable from a physical object accessed concurrently.

- One simple way to guarantee this property is to use locks: atomicity
  - Locks do not tolerate process crashes
  - 52% of bugs in Java concern the misuse of "synchronized"
  - False conflicts

- Otherwise
  - Complex implementation of data structures
  - Memory consuming
Progress conditions

Ideally: Each operation terminates whatever is the behavior of the other processes (contention, order, etc.)

● If one uses locks there are three progress conditions
  ○ deadlock-free (global progress)
  ○ starvation-free (local progress)
  ○ fifo

● Otherwise
  ○ wait-free (local progress)
  ○ lock-free (global progress)
  ○ obstruction-free (conditional progress - no contention)
Strong Consistency (linearizability and sequential consistency)

- Linearizability and sequential consistency are usually called strong consistency.

- A process cannot distinguish a strongly consistent implementation of a data structure with a physical data structure accesses concurrently:
  - Wait-free linearizable shared data structures are desirable but sometimes complex or inefficient.
  - Lock-free implementations may enjoy enough strong progress and acceptable complexity.
Strong Consistency (linearizability and sequential consistency)

- Linearizability (from M. Raynal)

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>write(a)</td>
<td>write(c)</td>
</tr>
<tr>
<td>write(b)</td>
<td>read(a</td>
</tr>
<tr>
<td></td>
<td>read(a</td>
</tr>
</tbody>
</table>
```

- Sequential consistency
Strong Consistency (linearizability and sequential consistency)

- Linearizability: a possible linearization
Strong Consistency (linearizability and sequential consistency)

- Linearizability: a shared queue

**SEQUENTIAL:**

- Enq (a)
- Enq (c)
- Enq (b)
- Deq (a)
- Deq (c)

**CONCURRENT:**

- p1
  - Enq (a)
  - Enq (b)
  - Deq (a|b|c) ?

- p2
  - Enq (c)
  - Deq (a|b|c) ?
Strong Consistency (linearizability and sequential consistency)

- Linearizability

A history defines a partial order on the operations
Strong Consistency (linearizability and sequential consistency)

- Linearizability
Strong Consistency (linearizability and sequential consistency)

- Linearizability

[Diagram showing operations Enq and Deq with p1 and p2 processes]
Strong Consistency (linearizability and sequential consistency)

- Sequential consistency

A "witness" seq history:

\[ Q.\text{Enq}(b) \quad Q.\text{Enq}(a) \quad Q.\text{Deq}(b) \]
Strong Consistency (linearizability and sequential consistency)

- Sequential consistency: unfortunately, it does not compose!

\[ Q\text{-}Enq(a) \quad Q'\text{-}Enq(b') \quad Q'\text{-}Deq(b') \]

\[ Q'\text{-}Enq(a') \quad Q\text{-}Enq(b) \quad Q\text{-}Deq(b) \]

Cycle
Strong Consistency
(linearizability and sequential consistency)

- Linearizability and sequential consistency cannot be distinguished in an asynchronous system.
- Sequential consistency is “cheaper” than linearizability.
- However, **linearizability is a local property:** if all objects are linearizable, then the whole computation is linearizable!
- A distribution computation is a partial order of events.
- A good consistency criterion consists in totally ordering all events.
  - linearizability: total order on all events + causality + **real-time order**
  - sequential consistency: total order on all events + causality
Weak Consistency

- Strong consistency is usually costly in time and space.
- In message-passing systems strong consistency is usually not possible and when it is possible, operation has to last the latency of the communication network: **CAP Theorem (Consistency - Availability - Partition)** (Gilbert&Lynch 2002).
- Attiya & Welch proved in 1994 that, when possible, strong consistency needs an operation duration proportional with network latency.
- This is practically impossible for many applications such as instant messaging, collaborative editors, etc.
- In those situations, one can use weak consistency conditions:
  - Cache coherence
  - Causal consistency
  - Eventual consistency
  - PRAM consistency
  - Serializability ...
Weak Consistency

Weak consistency conditions let each process build its own total order

- Strong eventual consistency: same total order on all update operations
- Serializability (transactions in databases): not all operations terminate
- Causal consistency: all local linearization respect causal order
- PRAM consistency: local and fifo order

There is no total order on the strength of the different consistency conditions

=>$>$ which the strongest weak consistency condition?
Weak Consistency

The world of consistency conditions (from M. Perrin PhD thesis)

There are 3 basic families of consistency conditions

A consistency condition that merges all of the three families falls into strong consistency
Weak Consistency

Weak consistency for which usage

- Serialisability: substitute for strong consistency
  - maximal security
  - simple to implement
  - failures are handled by the user

- Causal/PRAM consistency: distributed algorithms
  - predictable
  - not costly
  - no convergence

- Update consistency/strong eventual consistency: collaborative applications
  - close to self-stabilization
  - quite costly
  - inconsistencies visible to the user
Weak Consistency

Small experience with instant messaging: Snapchat, Messenger, Whatsapp, Skype, Hangouts, etc.

- Hangouts: serializability
  - message sending can be aborted

- Whatsapp: PRAM consistency
  - local consistency (perhaps the least consistent instant messaging)

- Skype: strong eventual consistency
  - messages can be reordered afterwards (all users eventually see all messages in the same order)
Progress Conditions

- If one considers message-passing systems, when strong consistency is not possible, applications consider weak consistency as seen above.
  - Amazon’s Dynamo highly available key-value store

- When necessary, whatever is the cost, strong consistency is provided
  - Apache’s Zookeeper for maintaining configuration information
  - Google’s Chubbby system

- In multithreaded computing, strong consistency is not sacrificed, but instead the progress condition is weakened: lock-free instead of wait-free
Progress Conditions

- The enqueue op. in Mickael & Scott’s lock-free Queue (from D. Hendler)

```java
public boolean enq(T value) {
    Node node = new Node(value);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(null, node)) {
                    tail.compareAndSet(last, node);
                    return;
                }
            } else {
                tail.compareAndSet(last, next);
            }
        }
    }
}
```
Conclusion

- Lock-free implementation are less complex than wait-free ones but they offer weaker guarantees on progress.

- In real settings, lock-free data structures statistically guarantee termination for all operations.

- Mickael & Scott’s lock-free linearizable queue is included in the Standard Java Concurrency Package.

- There are many ongoing research to find the best special instructions and the most efficient distributed implementations of usual data structures.