Concurrent and Distributed Systems - 2021–2022)

CS3: Transactions (Rev A)

* Star denotes optional/advanced exercise.

Transaction-processing systems provide concurrency control for composite operations that act on multiple distinct shared objects/data, behind the abstraction of a 'transaction'. The guarantees of transactional systems are described by the ACID properties: atomicity, consistency, isolation, and durability. Transaction systems shift the burden of managing concurrency away from the end programmer into databases, language runtimes, and operating systems – and by masking its effects, allow flexibility in how the ACID properties are implemented.

Q1 ACID properties

(a) In the context of multi-threading, 'atomicity' meant something different from its meaning in transaction systems. If we simply associate locks with objects in a transaction system to implement 2-phase locking (2PL), which ACID properties fall out naturally, and which require additional work? Why?

(b) Our definition of 'isolation' is grounded in 'serialisability': the appearance of concurrently executing transactions within a system to conform to serial execution. In the lecture, we discussed 'conflict serialisability', in which a schedule is considered serialisable if two transactions operating on overlapping sets of objects order all conflicting (non-commutative) operations the same way.

For the following two transactions, enumerate all interleavings under the assumption that individual operations on objects are atomic. For each, is it a serial execution? A conflict-serialisable execution?

```
transaction T1 {                transaction T2 {
  a = A.getBalance();             A.debit(100);
  A.credit(INTEREST*a);           B.credit(100);
}                               }
```

(c) In transaction systems, 'atomicity' refers to transactions being committed fully or not at all. Why might it be easier to implement transaction atomicity with optimistic concurrency control than with 2PL?

(d) One limitation of 2PL is that it can suffer 'cascading abort', when it implements 'isolation' rather than 'strict isolation'. In the above transactions, `B.credit(100)` may abort due to the resulting balance exceeding a yearly limit; illustrate a 2PL schedule in which aborting T2 triggers a cascading abort of T1. Why would strict 2PL have prevented this problem?

(e) 'Durability' ensures that if a transaction is reported as committed to the submitter, the results will persist across a variety of failure modes including power loss. However, the obvious

solution of simply writing out object updates to disk before returning success may encounter problems with atomicity – under what circumstances might this occur?

(f) One technique for implementing durability in transaction systems is write-ahead logging. In this scheme, a balance is sought between two logical extremes:

1. a pure log-based store (in which all object accesses are only stored as an append-only log of changes, infinitely long (as has been used in financial book keeping for the past n centuries)), and
2. a simple on-disk object store in which all stores are performed 'in place' (such as a simple file system where a record is simply stored in one place with just its current value).

What effect does varying the maximum log size have on:

1. Efficient use of persistent storage space?
2. Transaction throughput?
3. Recovery time following a crash?

(g) Write-ahead logging uses atomicity and isolation properties of single-sector writes to build atomicity and durability properties for transactions.

1. What might go wrong if single-sector write were not atomic?
2. What might go wrong if disks did not properly serialise stores – e.g., did not implement a 'happens-before' relation between sequential synchronous disk writes? [Here: 'synchronous' means no-reordering in the kernel or device driver and with the system call or write primitive not returning to the issuing thread until the operation is 'complete'.]
3. Not examinable: (*) What problems arise with SSD compared with traditional spinning disk?

Q2 2-phase locking

2-phase locking is one scheme for implementing isolation in the presence of concurrent transaction processing on multiple objects. It consists of two monotonic phases: lock expansion, and lock contraction. For this question, we will work with the following transactions:

```
transaction T1 {
  x = A.read();
  B.write(x);
}
```

```
transaction T2 {
  x = B.read();
  C.write(x);
}
```

```
transaction T3 {
  x = C.read();
  A.write(x);
}
```

Imagine that the transaction scheduler, in an effort to offer fairness to transactions, interleaves scheduling with lines of code in a round-robin manner. For example, it would, without the intervention of locking, schedule two transactions as follows:

```
(First epoch)
T1.L1
T2.L1

(Second epoch)
T1.L2
T2.L2

...
```

If a transaction is blocked on a lock, then the scheduler will skip that transaction and proceed to the next transaction in the epoch.

(a) If we use naïve 2PL to implement transactions T1, T2, and T3, what schedule will be selected?

(b) Deadlock presents a serious challenge to 2PL in the presence of composite operations. Imagine that the transaction system implements a simple deadlock detector in which, if there are in-flight transactions but none can be scheduled, then all in-flight transactions will be aborted (and their locks released) to restart in the next epoch. What schedule arises with the above three transactions?

(c) Livelock can also be a challenge for 2PL in the presence of a naïve deadlock resolution scheme. How might the scheme in (b) be modified to provide guarantees of progress in the event that a deadlock is detected?

Q3 Timestamp ordering (TSO) and optimistic concurrency control (OCC)

Timestamp ordering allows transactions to operate concurrently based on a serialisation selected as transactions start. Each transaction is issued a timestamp (sequence or ticket number) that is checked against object timestamps to detect conflicts, and to taint objects written or read to trigger detection of conflicts with other transactions that will touch the same object later.

(a) Timestamp ordering is conservative, in that it is committed to a particular serialisation of a transaction attempt at the moment that the timestamp is selected, causing it to potentially reject other valid serialisations. Give an example of two transactions, timestamp assignments, and an interleaved schedule in which a valid serialisation is rejected by TSO, leading to an unnecessary abort.

(b) TSO aborts transactions as conflicting operations are detected. Why could TSO could suffer from livelock under heavy contention?

Optimistic concurrency control likewise allows transactions to operate concurrently; unlike TSO, it 'searches' for a valid serialisation on transaction commit, rather than at transaction start. This is argued to give OCC greater flexibility in avoiding unnecessary aborts seen in TSO, which might exclude valid serialisations.

(c) Give an example of two transactions and a schedule that would be accepted by OCC, but

rejected by TSO.

(d) OCC executes transactions against local copies of data ('shadows') rather than globally visible original copies, which avoids the need to explicitly handle cascading aborts. However, copying all objects at the start of the transaction is problematic if the set of objects to be operated on is determined as part of the transaction itself. Why does on-demand copying of objects complicate transaction validation?

(e) OCC aborts transactions as conflicting commits are detected. Why, informally, might we argue that this conflict behaviour is more resistant to whole-system livelock than TSO?

(f) How could OCC's validation model lead to starvation? Give the general nature of a mixture of transactions that might typically lead to starvation for a subset of that mixture.

(g) Why is OCC good for online booking on the web?

Q4 Auction Controller (*)

(*) An auction controller needs to record the highest bid received before a pre-programmed auction end time and date. It is manifested by an instance of an object in an OO programming language. According to your design preference, this instance could be a simple object that uses concurrency primitives, a monitor, or an actor in a message-passing system.

A single, large computer hosts the controller instance along with individual threads and private data for each of several thousand bidders. Bidder threads may contain private/proprietary behaviour but all components are trusted to be well-behaved by all parties.

The bidding threads/actors need to be able to find out the value of the most recent successful bid and be able to make their own bids. A successful bid should cost 5 pence to the bidder, whereas an unsuccessful bid has no cost. A bid is successful if it makes the bidder the current highest bidder.

You may neglect the part of the system that notifies the winner and handles the resulting sale.

(a) Sketch a suitable API used between the bidder and the auction controller to enable the bidding process and describe the principle data structures needed.

(b) In your design, what locking mechanism is appropriate to ensure a bid fee is not paid for an unsuccessful bid. [Hint: you may need very little locking for this application, depending on how you design it.]

(c) Could 2PL or TSO be usefully used to ensure a bidder does not bid the same price as an earlier bidder?

(d) Now consider that a bidder paticipates in multiple auctions at once and has a limited budget. Bids that the bidder cannot afford are automatically ignored by the auction controllers. How would you modify your code to use optimisitic concurrency control?

Note: when there might be latency between seeing the current price and placing a bid, whether

the latency arises from thread schedulling or networking delay, we are starting to consider a distributed system...