Concurrent and Distributed Systems - 2015–2016

Supervision 2: Transactions

Transaction-processing systems provide a more programmer-interface to concurrency by masking its effects for snippets of code known as ”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 than 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 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 {
  a = A.getBalance();
  b = B.getBalance();
  return (a + b);
}
```

```
transaction T2 {
  A.debit(100);
  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”, as 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 a pure log-based store (in which all object accesses are stored as an append-only log of changes, infinitely long) and simple on-disk object stores in which all stores are performed "in place" (such as simple inode-based file systems). What effect does varying the maximum log size have on:

1. Efficient use of persistant 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 was 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?

Q2 2-phase locking

2-phase locking is one scheme to implement 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, 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 naive 2PL to implement transactions T1, T2, and T3, what schedule will be selected?

(b) Deadlock presents a serious challenge to 2-phase locking 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 naive 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 be 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, informally, might this lead us to believe that TSO could suffer from livelock under 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 transations 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; describe a scenario in which a set of transactions and a serialisation cause starvation for a transaction.