Lectures on distributed systems

Concurrency Control

Paul Krzyzanowski

Introduction

When discussing transactions, we alluded to *schedules*, or valid orders of execution. We can play it safe and use mutual exclusion on a transaction level to ensure that only one transaction is executing at any time. However, this is usually overkill and does not allow us take advantage of the concurrency that we may get in distributed systems. What we would really like is to allow multiple transactions to execute simultaneously but keep them out of each other's way and ensure serializability. This is called **concurrency control**.

Locking

One mechanism that we can use to serialize transactions is the exclusive lock on a resource. A process will lock any data that is about to be used on behalf of the transaction. Resource locking in a distributed system can be implemented with a *lock manager*. This is the same as the centralized implementation for mutual exclusion. One process serves as a lock manager. Processes request a lock for a resource (e.g., a file or a shared data object) and then they either are granted a lock or wait for it to be granted when another process releases it.

This form of locking is clearly suboptimal: there is nothing wrong with having several transactions *read* the same object concurrently as long as none of them modify it. Locking can be optimized to yield better resource usage by distinguishing *read locks* from *write locks*. If a process imposes a read lock on a resource, other processes will still be able to request read locks on that resource. However, a request for a write lock would fail (or be blocked)¹. If a process imposes a write lock on a resource, then neither read nor write locks will be granted until that process releases the resource. Another consideration in resource locking is the granularity of the lock. Quite often, there is no need to lock an entire file. More parallelism may achieved with a finer granularity of locking (a span of bytes or a record). This is particularly important with databases, where a transaction is better off obtaining locks for the records it is manipulating rather than the entire database.

Getting and releasing locks precisely can be tricky. Done improperly, it can lead to inconsistency and/or deadlocks. We need to ensure that a transaction can always commit without violating the serializability invariant. For transactions to be serial, all access to data must be serialized with respect to accesses by other transactions. To ensure that conflicting operations of multiple transactions are executed in the same order, a restriction is imposed: *a transaction is not allowed to obtain new locks once it has released a lock*. This restriction is called **two-phase locking**. The first phase is known as the <u>growing phase</u>, in which a transaction acquires all the locks it needs. The second phase is known as the <u>shrinking phase</u>, where the process releases the locks. If a process fails to acquire all the locks during the first phase, then it is obligated to release <u>all</u> of them, wait, and start over. It has been proved (Eswaren, et al., 1976) that if all transactions use two-phase locking, then all schedules formed by interleaving them are serializable.

¹ This is similar to DFS and its use of tokens in granting file access permissions.

To ensure that a transaction not access data until another transaction that was manipulating the data has either committed or aborted, locks may be held until the transaction commits or aborts. This is known as **strict two-phase locking**. This is similar to two-phase locking except that in this case, the shrinking phase takes place during the commit or abort. One effect of strict two-phase locking is that by placing the second phase at the end of a transaction, all lock acquisitions and releases can be handled by the system without the transaction's knowledge.

Optimistic concurrency control

Locking is not without problems. Locks have an overhead associated with maintaining and checking them. They may induce deadlock in a system and locking also may reduce concurrency in a system by having transactions hold on to locks longer than needed (as in strict two-phase locking). An alternative proposed by Kung and Robinson in 1981 is **optimistic concurrency control**, which tells the transaction to just go ahead and do what it has to do without worrying about what someone else is doing. In most applications, the chance of two transactions accessing the same data is low, so why induce overhead? When a conflict does arise, then the system will have to deal with it.

Optimistic concurrency control requires transactions to operate in a private workspace, so their modifications are not visible to other until they commit. When a transaction is ready to commit, a validation is performed on all the data items to see whether the data conflicts with operations of other transactions. If the validation fails, then the transaction will have to be aborted and restarted later (again, optimistically hoping for no conflicts). Optimistic control is clearly deadlock free (no locking or waiting on resources) and allows for maximum parallelism (since no process has to wait for a lock, all can execute in parallel).

Timestamps

Another approach to concurrency control is the use of **timestamp ordering**, developed by Reed in 1983. In this algorithm, we assign a timestamp to a transaction when it begins. The timestamp has to be unique with respect to the timestamps of other transactions (this is easily accomplished; for example, Lamport's algorithm can be used). Every file (object) in the system has a *read* and a *write* timestamp associated with it (two timestamps per object) telling which *committed* transaction last read it and which *committed* transaction last wrote it. Normally, when a process tries to access a file, the file's read and write timestamps will be older than the current transaction's. This implies *good ordering*. If this is not the case, and the ordering is incorrect, this means that a transaction that started later than the current one accessed the file and committed. In this case the current transaction is too late and has to abort (the rule here is that the lower numbered transaction always goes first).

For example, suppose there are three transactions: a, b, and c. a ran a long time ago and used every file needed by b and c. b and c start concurrently with b receiving a younger timestamp than c ($T_b < T_c$).

case 1 (proper ordering)

b writes a file (assume the read timestamp of the file is T_R and the write timestamp is T_W). Unless c has already committed, T_R and T_W are T_a , which is less than T_b . The write is accepted and performed tentatively, made permanent on commit. T_b is recorded as the tentative T_W .

case 2 (improper ordering)

If c has either read or written the file and committed, then the file's timestamp(s) has been modified to T_c . In this case, when b tries to access the file and compares its timestamp with that of the file, it sees that ordering is incorrect because b is an older transaction trying to modify data committed by a younger transaction (c). b must be aborted. It can apply for a new timestamp and start over.

case 3 (delaying execution)

Suppose b has written the file but not yet committed. The write timestamp of the file is a *tentative* T_b . If c now wants access to the file, we have a situation in which ordering is correct <u>but</u> the timestamp is in a tentative state. c must now <u>wait</u> for b to finish before it can access the file.

Concurrency control with timestamps is different than locking in one important way. When a transaction encounters a later timestamp, it aborts. With locking, it would either wait or proceed immediately.