Principles of Computer Systems Design

Assignment 2

Tudor Dragan
Gabriel Carp

December 2, 2014

Question 1: Serializability & Locking

Conflict-serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

A schedule is conflict-serializable if and only if it's precedence graph has no cycles. This is a graph of nodes and vertices, where the nodes are the transaction names and the vertices are attribute collisions.

Schedule 1

Schedule 1 *is not* conflict serializable because the graph is *cyclic*:

- T_1 - T_2: read-write conflict on X.
- T_2 - T_3: write-read conflict on Z.
- T_3 - T_1: read-write conflict on Y.

Because T_1 has a shared lock on X when T_2 writes to it, the schedule cannot be generated by S2PL.

Schedule 2

Schedule 2 *is* conflict serializable because the precedence graph is *acyclic*:

- T_1 - T_2: X is accessed by T_2 after T_1 has committed.
- T_2: Y is only accessed by T_2.
- T_3 - T_2: Z exclusively locked by T_3 is released prior to T_2 acquiring a shared lock.

The second schedule can be generated by S2PL because we can insert locks prior to the respective reads and writes to the objects in question.

Question 2: Optimistic Concurrency Control

Scenario 1

Because T_1 finishes before T_3 starts, the 1^{st} condition holds. We have to check that the 2^{nd} condition holds for T_2 and T_3, but because T_2 writes the object that T_3 reads from, this condition does not hold. Therefore T_3 has to *rollback*.

Scenario 2

The 2^{nd} validation condition does not hold for T_1 and T_3 because T_1 writes to object 3 and T_3 reads from it. Therefore T_3 has to rollback. We can observe that the 3^{rd} holds for T_2, because T_3 does not access object 8.

Scenario 3

The only object that T_1 writes to is object 4 and because T_3 does not read or write to that object, the 2^{nd} validation condition holds. Then we check is the 2^{nd} condition holds for T_2 and T_3. T_2 only writes to object 6 and T_3 does not access that object in any way, therefore T_3 can commit.

Programming Task

For the implementation of the locking protocol used in the ConcurrentCertainBookStore we used the *ReentrantReadWriteLock* class available in the *java.util.concurrent.locks* package.

A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock.

The ReentrantReadWriteLock allows both readers and writers to reacquire read or write locks in the style of a ReentrantLock. Non-reentrant readers are not allowed until all write locks held by the writing thread have been released. Additionally, a writer can acquire the read lock, but not vice-versa. Among other applications, reentrancy can be useful when write locks are held during calls or callbacks to methods that perform reads under read locks. If a reader tries to acquire the write lock it will never succeed.

We implemented the strict two-phase locking (S2PL) by requiring a write lock when modifying the data at the start of the method. Some methods only require read locks such as getBooks and getBooksInDemand.

Lock type	read-lock	write-lock
read-lock		**X**
write-lock	**X**	**X**

Table 1: Lock compatibility table

Testing

The testing has been performed locally. Every client has been emulated as a *Thread* that uses the helper classes that have implemented the *Runnable* interface. By creating different threads we manage to satisfy the requests concurrently.

We have implemented the 2 required tests that were presented in the assignment text and added 2 of our own that handled the posting of *ratings* and getting the *top rated books*.

Our two tests test the behavior of adding two or more ratings concurrently and checking a hardcoded value. The assertion returns true in both tests.

Discussion on the Concurrent Implementation of Bookstore

Is your locking protocol correct? Why? Argue for the correctness of your protocol by equivalence to a variant of 2PL.

Our protocol is implemented by using the conservative S2PL by acquiring a lock at the beginning of each method and releasing it before the method returns.

Can your locking protocol lead to deadlocks? Explain why or why not.

The deadlock cannot occur because we acquire the locks right at the start of the methods, transforming every call into an *atomic* one. This approach impacts the performance by lowering concurrency, but ultimately assures proper synchronization.

Is/are there any scalability bottleneck/s with respect to the number of clients in the bookstore after your implementation? If so, where is/are the bottleneck/s and why? If not, why can we infinitely scale the number of clients accessing this service?

Scalability is a problem because *we lock the entire datastore* when we are modifying the database. We have taken into account the difference between readers and writers and we allow for multiple reads but not for multiple writes. A solution to this problem would be the fact that we acquire writing locks on the table entry level. For example, if a client modifies the stock of a book in the book store, we should not wait for the stock update when adding a rating.

Discuss the overhead being paid in the locking protocol in the implementation vs. the degree of concurrency you expect your protocol to achieve.

It is safe to assume that the overhead is not that big. Because we are dealing with a bookstore, the customers that *get and read* more information about the books are more frequent, compared

to the buyers that *modify and change* the bookstore's stock and ratings. Our locking protocol is suitable enough for providing adequate isolation and performance.

Appendix 1: Serializability & Locking

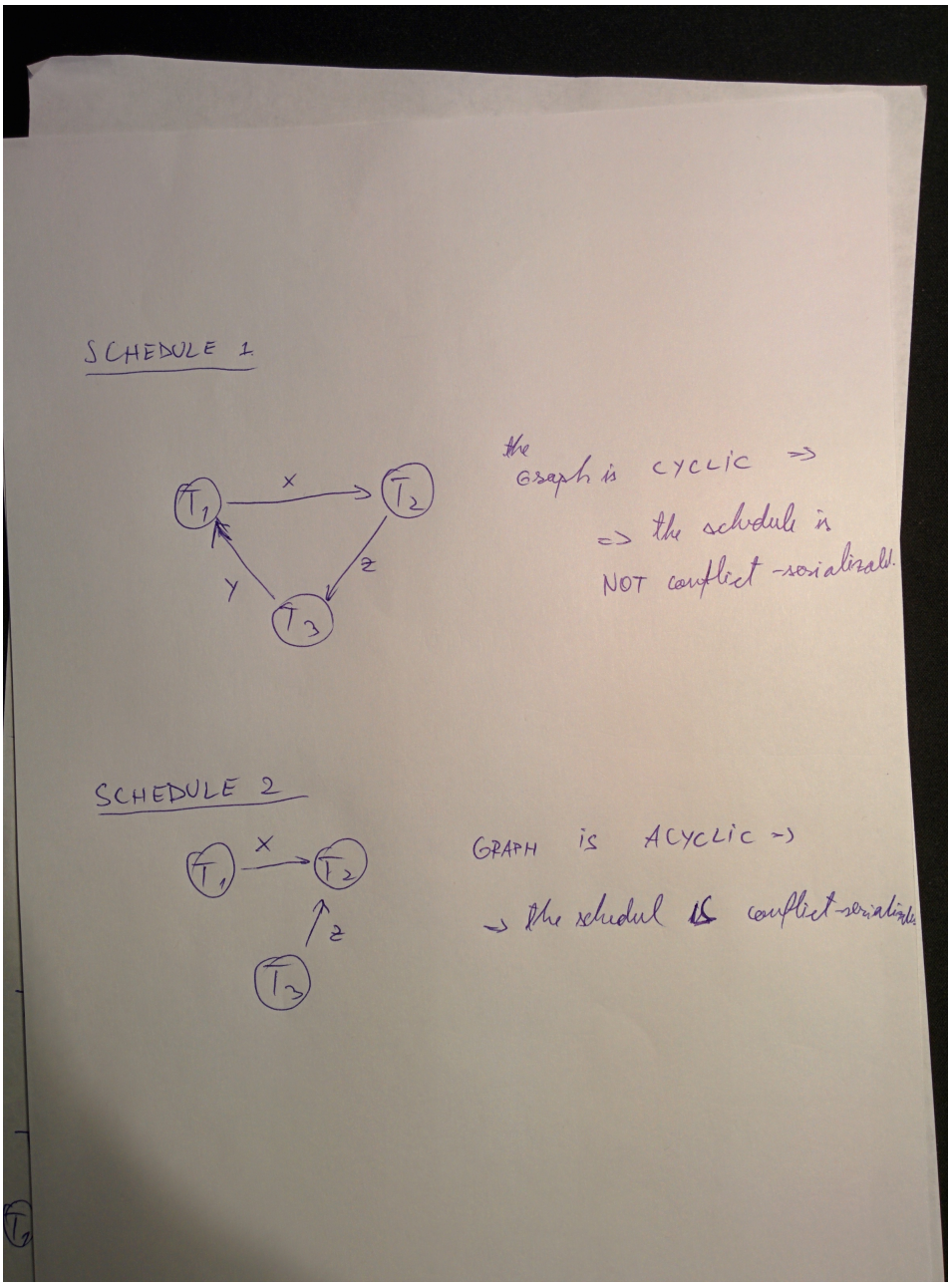

Figure 1: The precedence graphs for the two schedules

Appendix 2: Optimistic Concurrency Control

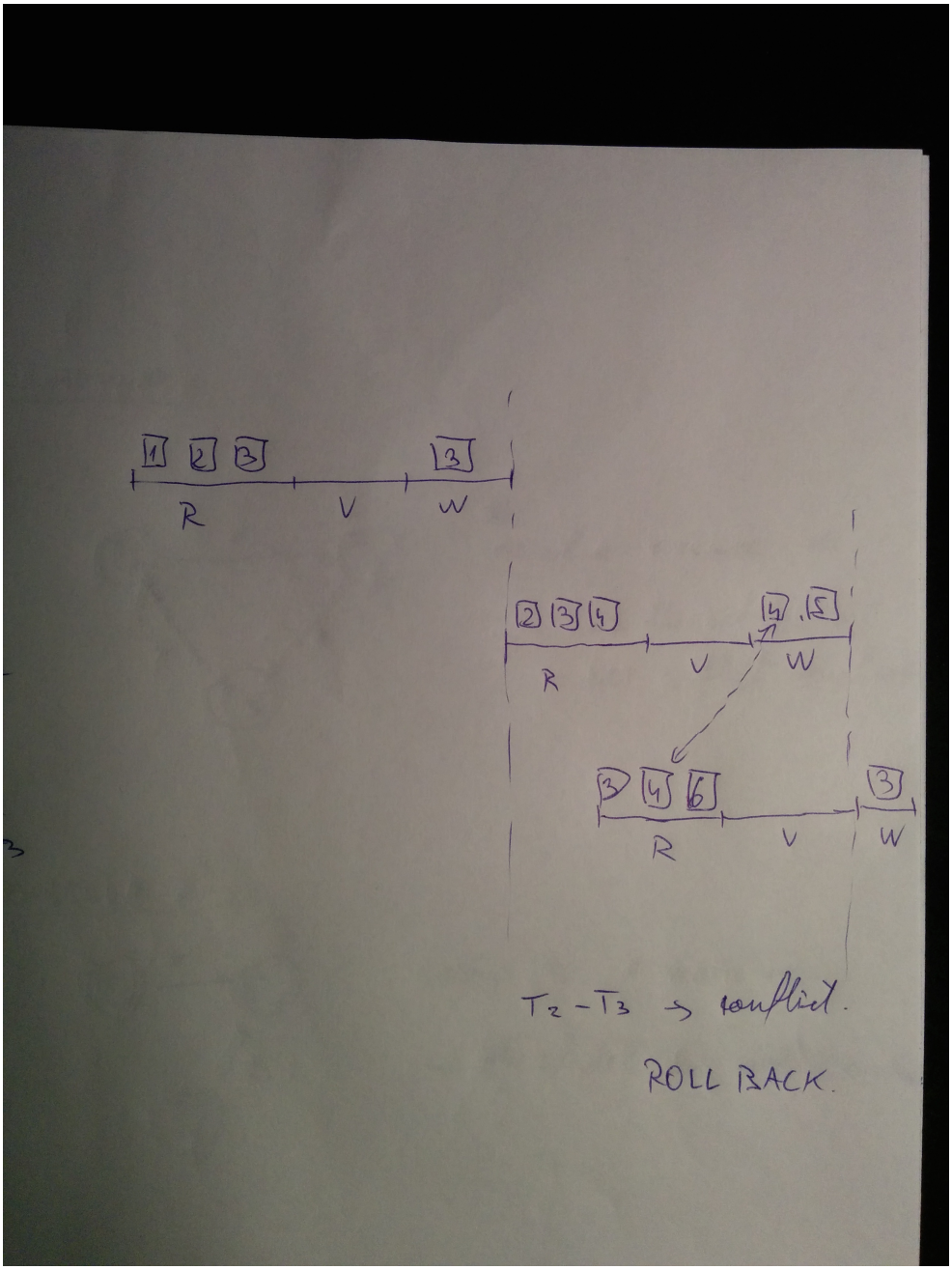

Figure 2: Scenario 1

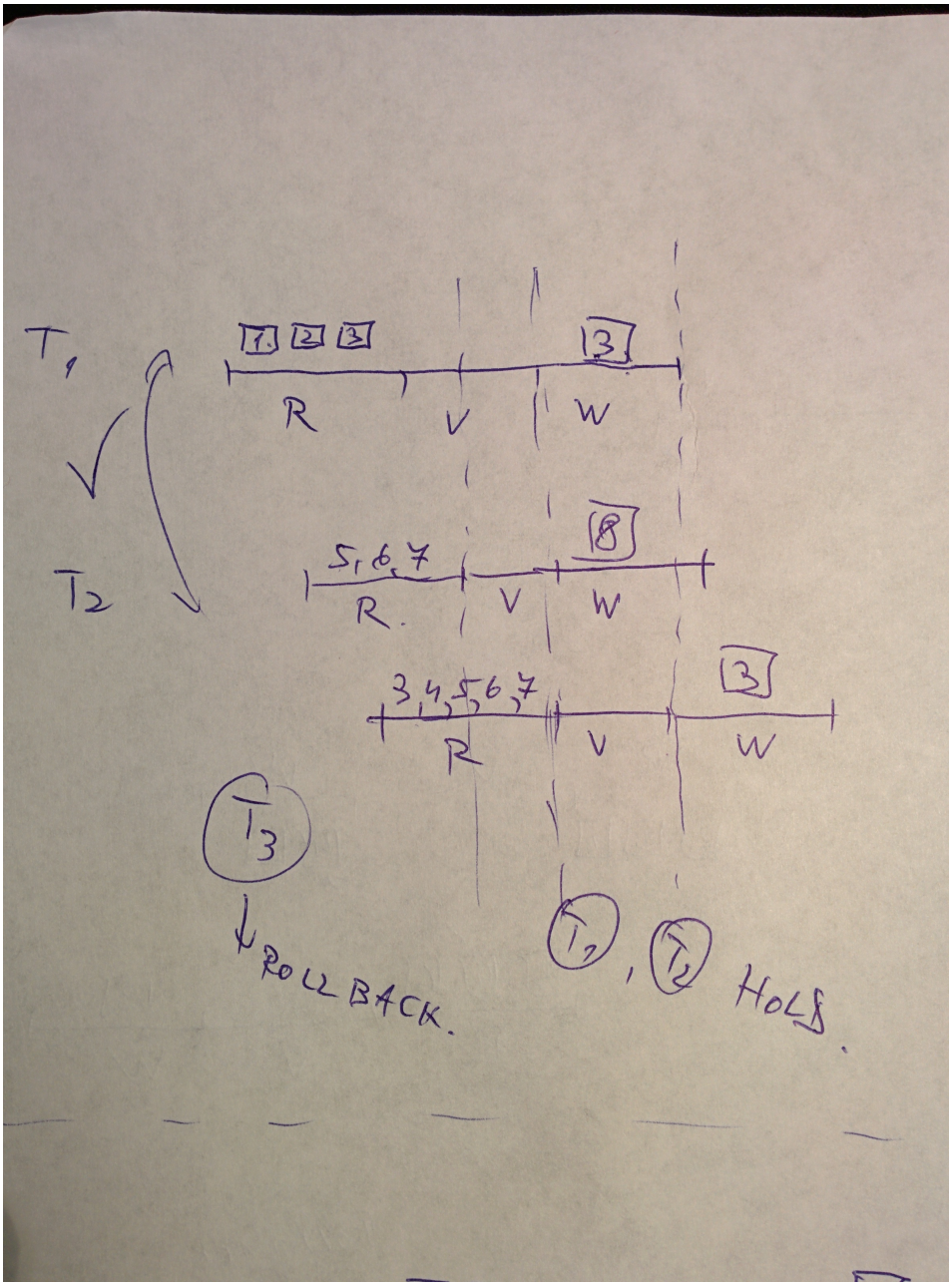

Figure 3: Scenario 2

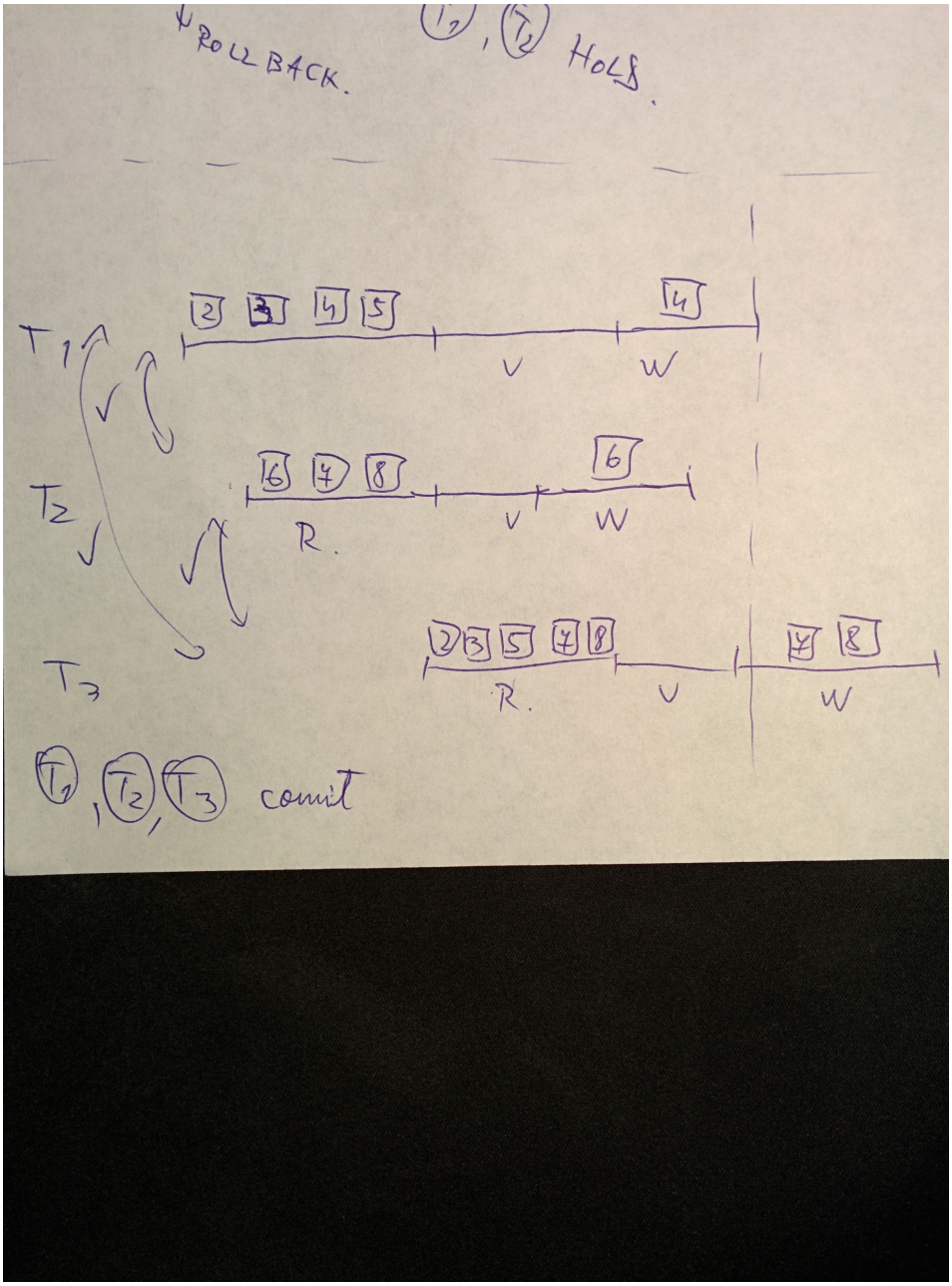

Figure 4: Scenario 3