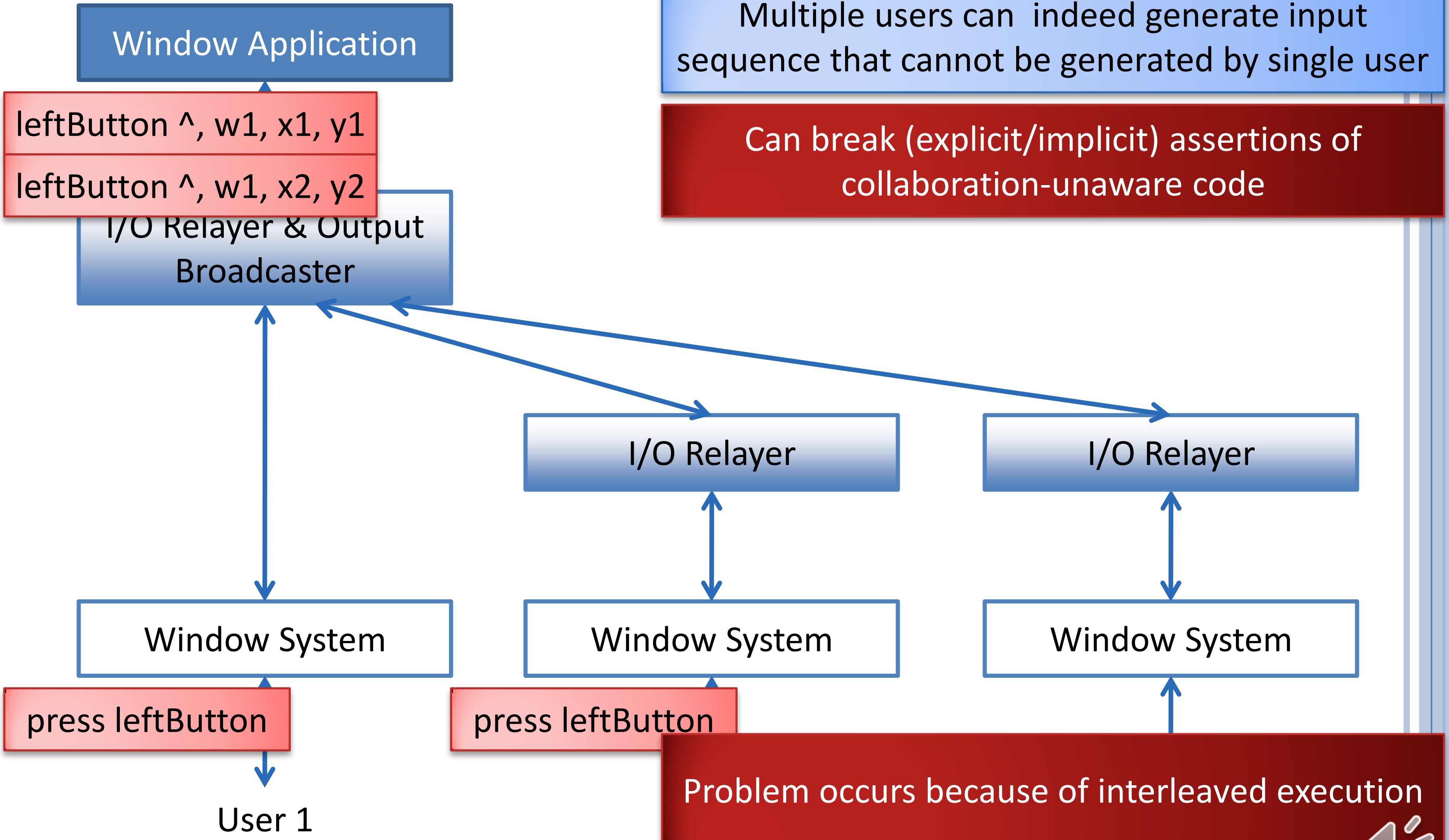
Concurrency Control

Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill
dewan@cs.unc.edu

Concurrency Control

Issue	Description
Session Management	How do distributed users create, destroy, join, and leave collaborative sessions?
Single-user Interface	What are the application semantics if there is a single user in the session?
Coupling	What is the remote feedback of a user command and when is it given?
Access Control	How do we ensure that users do not execute unauthorized commands?
Concurrency Control	How do we ensure that concurrent users do not execute inconsistent commands?

Problem in Shared Window Systems

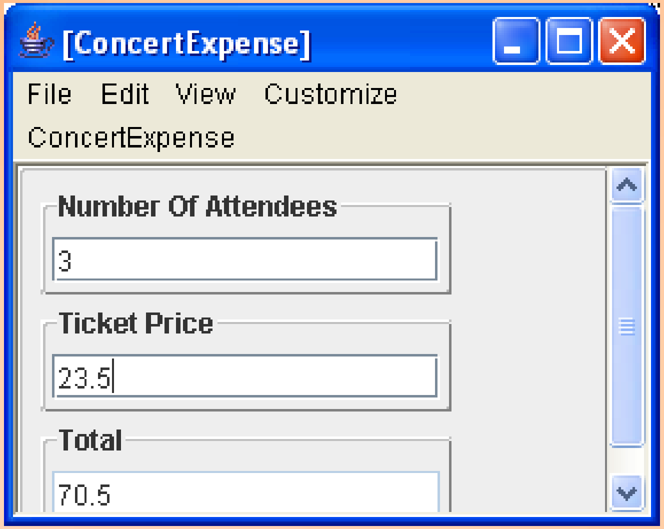


Multiple users can indeed generate input sequence that cannot be generated by single user

Can break (explicit/implicit) assertions of collaboration-unaware code

Problem occurs because of interleaved execution

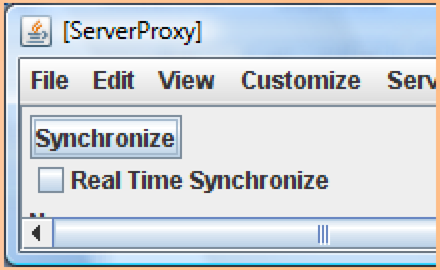
Problem in Shared Model Sytems



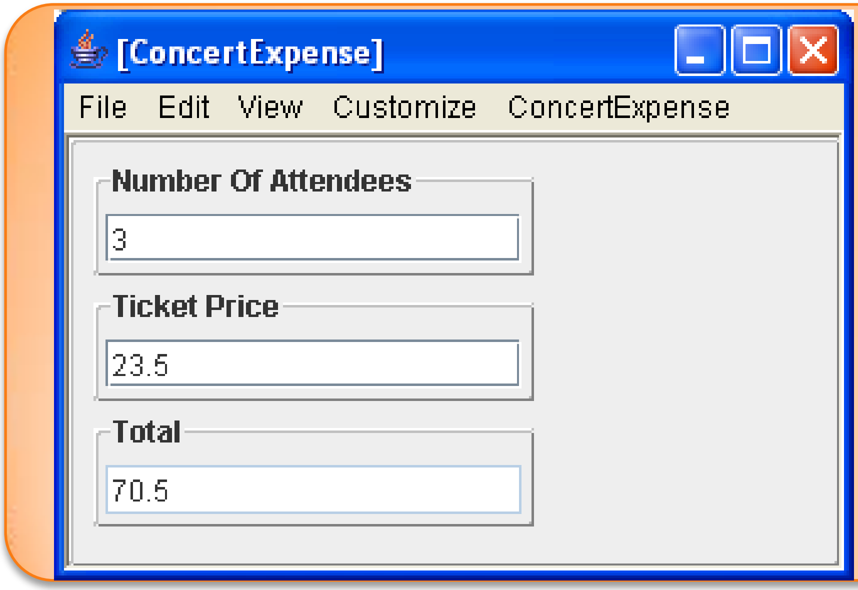
Number Of Attendees: 3

Ticket Price: 23.5

Total: 70.5



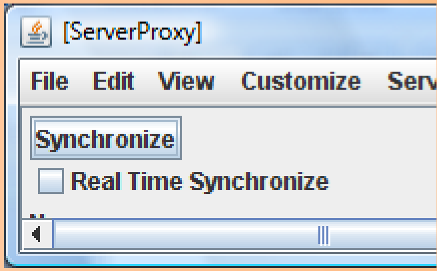
☐ Real Time Synchronize



Number Of Attendees: 3

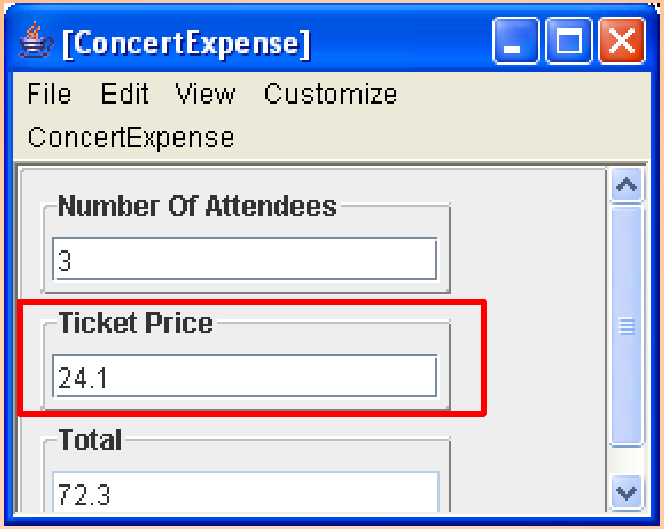
Ticket Price: 23.5

Total: 70.5



☐ Real Time Synchronize

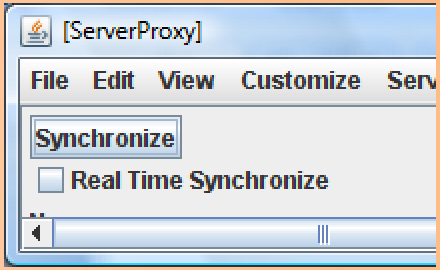
Problem in Shared Model Systems



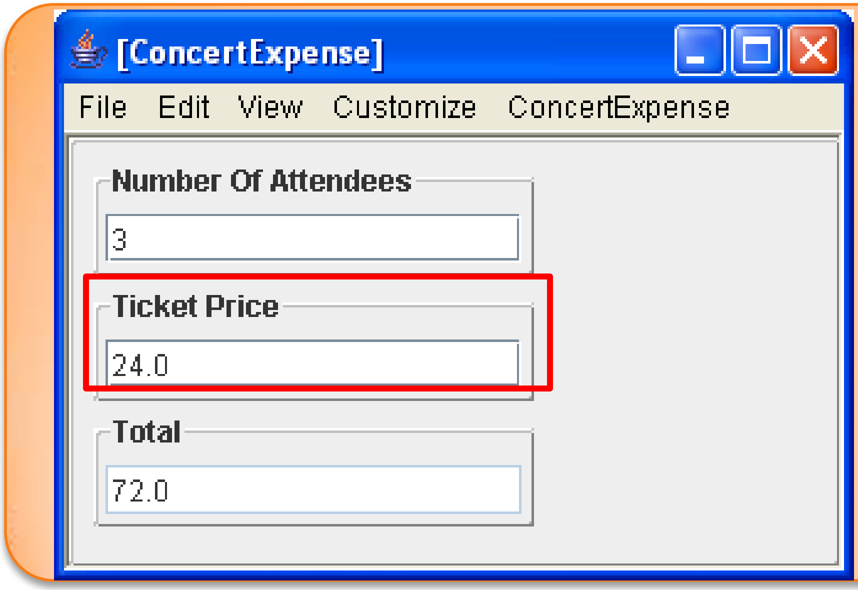
Number Of Attendees: 3

Ticket Price: 24.1

Total: 72.3



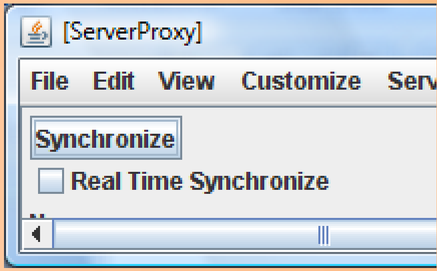
□ Real Time Synchronize



Number Of Attendees: 3

Ticket Price: 24.0

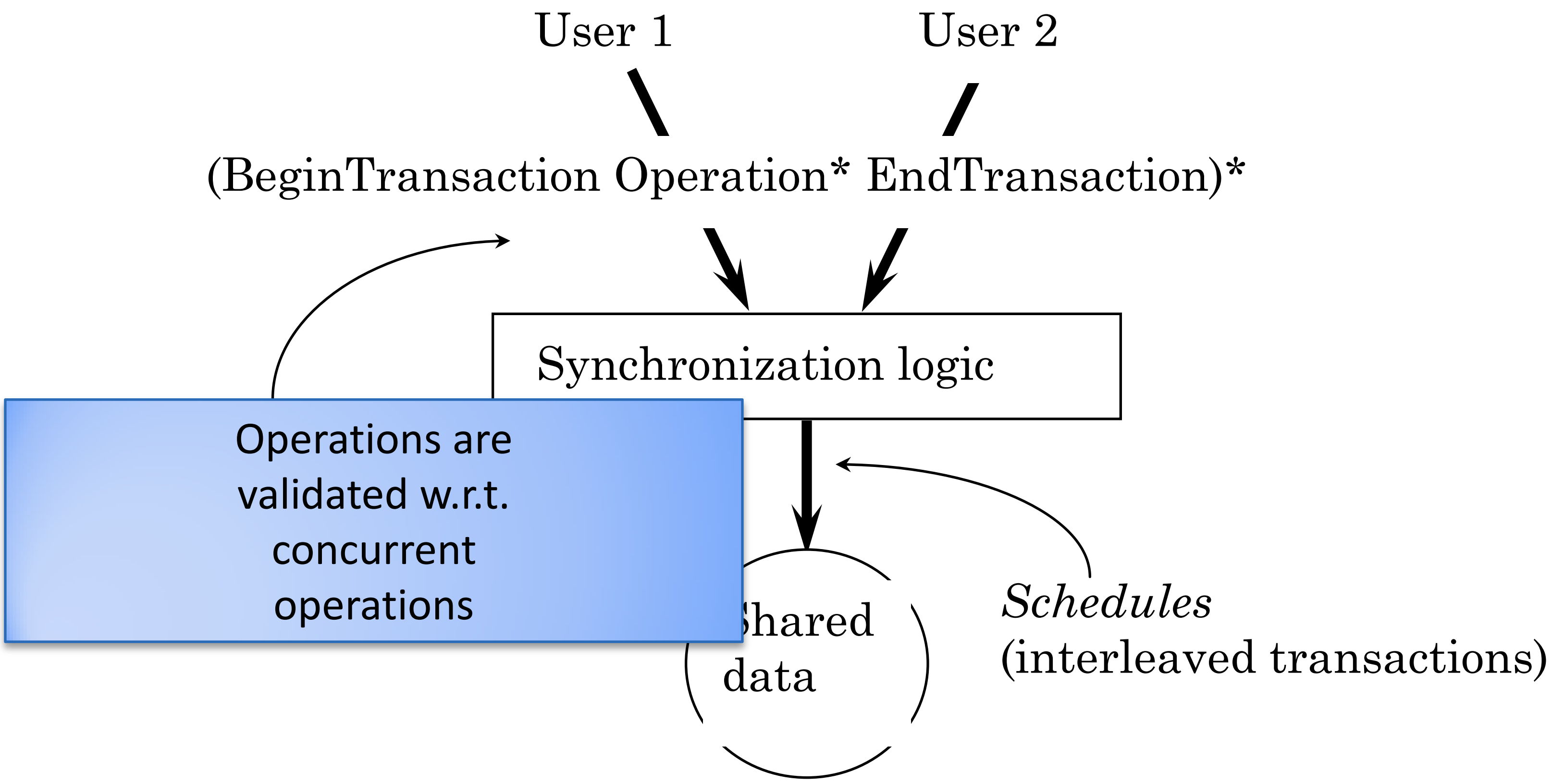
Total: 72.0



□ Real Time Synchronize

Synchronization model

- Users submit operations in *transactions*



User 1

User 2

(BeginTransaction Operation* EndTransaction)*

Synchronization logic

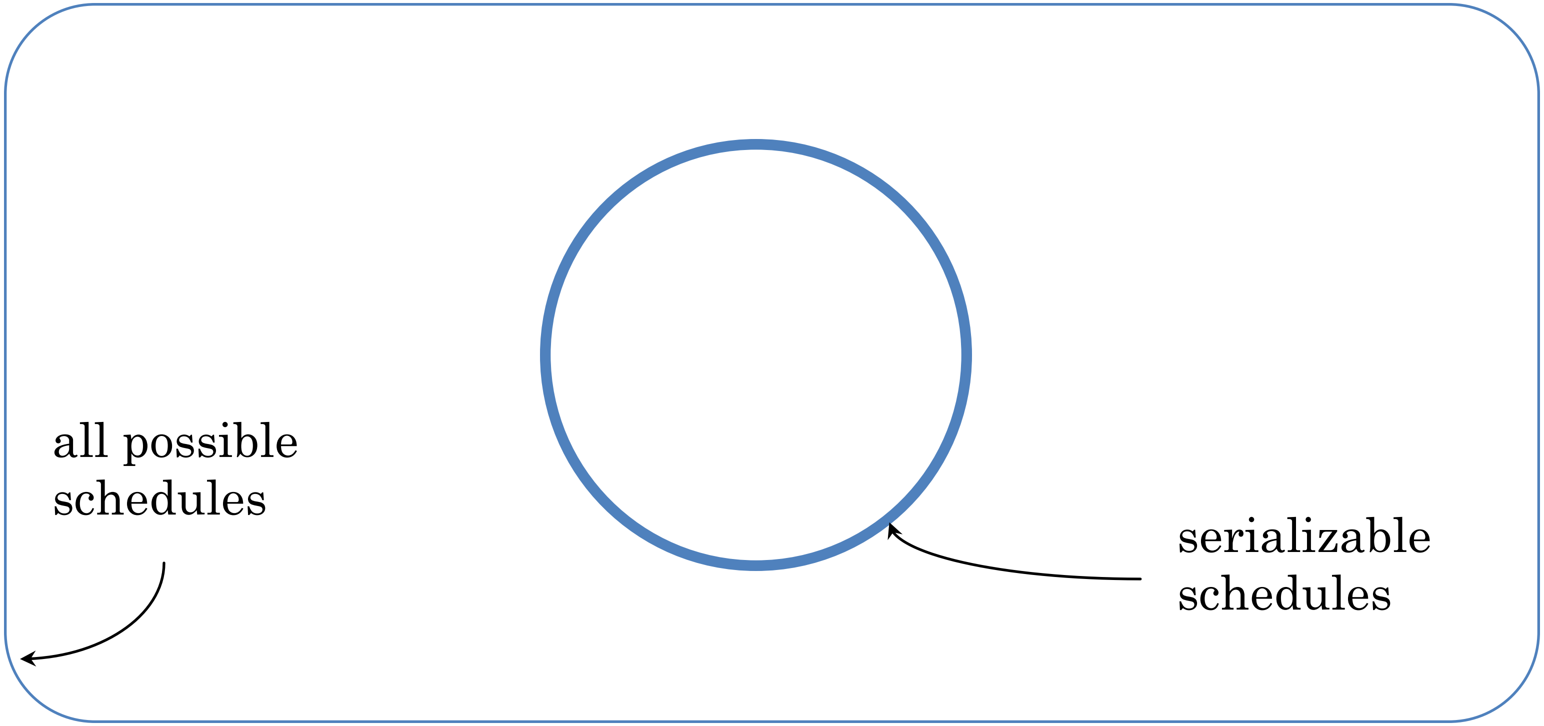
Shared data

Schedules
(interleaved transactions)

Operations are validated w.r.t. concurrent operations

Traditional Correctness criteria: Serializability

- Concurrent transactions execute as if they were submitted one after the other.



Serializability: Different Items

T^1	T^2
$R^1(d^1)$	
	$R^2(d^2)$
	$W^2(d^2)$
$W^1(d^1)$	

Serializable?

$R^1(d^1)\ R^2(d^2)\ W^2(d^2)\ W^1(d^1)$

Commuting operations can be reordered

$R^2(d^2)\ R^1(d^1)\ W^2(d^2)\ W^1(d^1)$

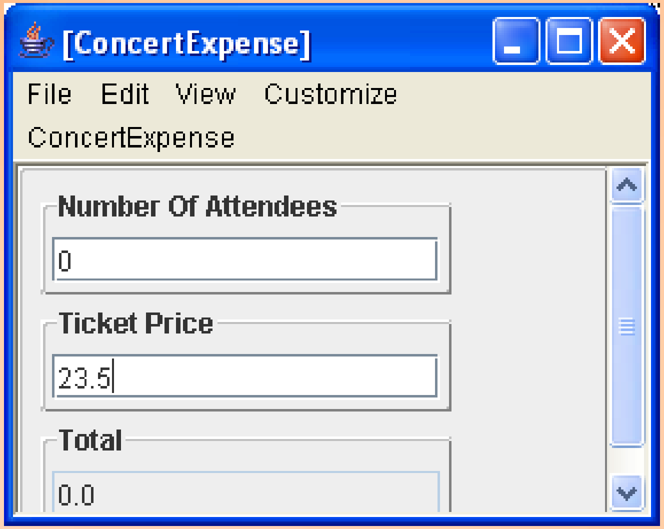
$R^2(d^2)\ W^2(d^2)\ R^1(d^1)\ W^1(d^1)$

T2 T1

T1T2

Serializable!

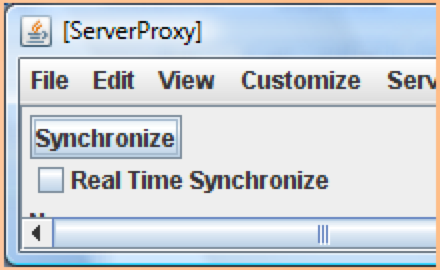
Different Items

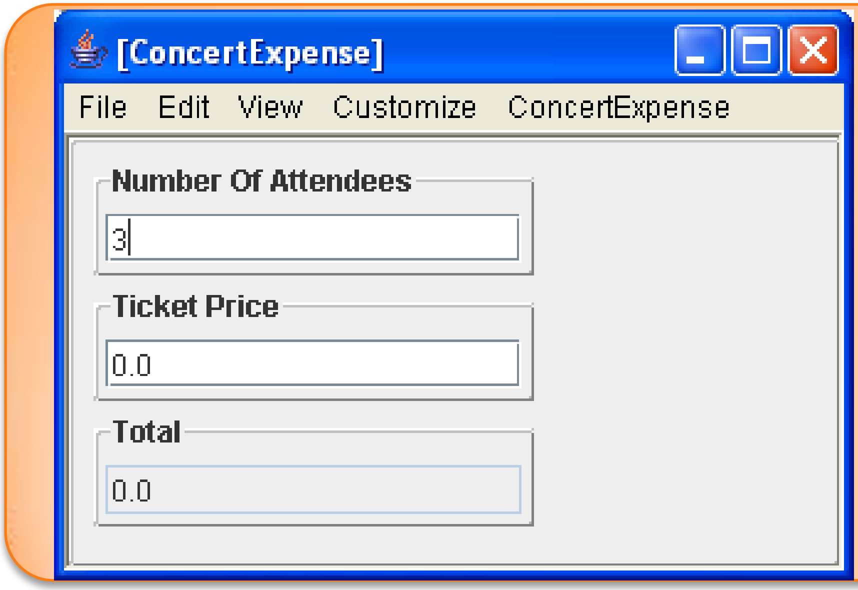


Number Of Attendees: 0

Ticket Price: 23.5

Total: 0.0

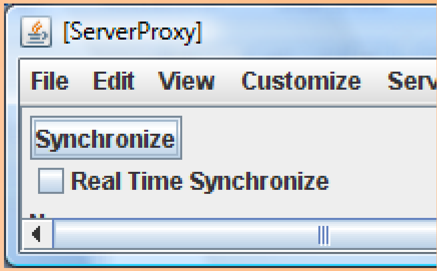




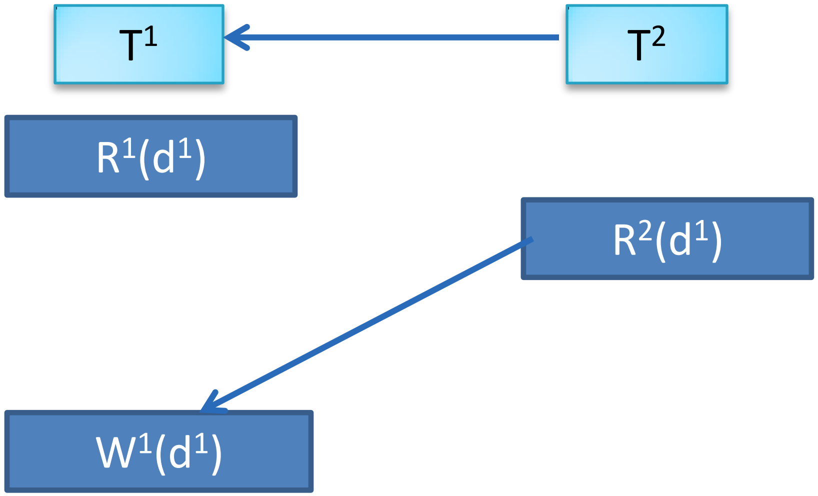
Number Of Attendees: 3

Ticket Price: 0.0

Total: 0.0



Serializability: Same Items



Serializable?

T^2 should precede T^1

$R^1(d^1)$ $R^2(d^1)$ $W^1(d^1)$

No dependencies between commuting operations

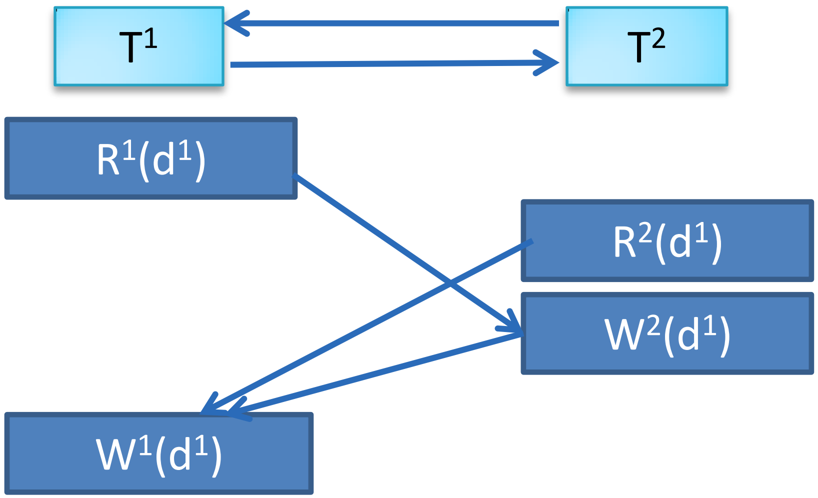
$R^2(d^1)$ $R^1(d^1)$ $W^1(d^2)$

T2 T1

Serializable!

T1T2 ✗

Serializability: Same Items



Serializable?

T^2 should precede T^1

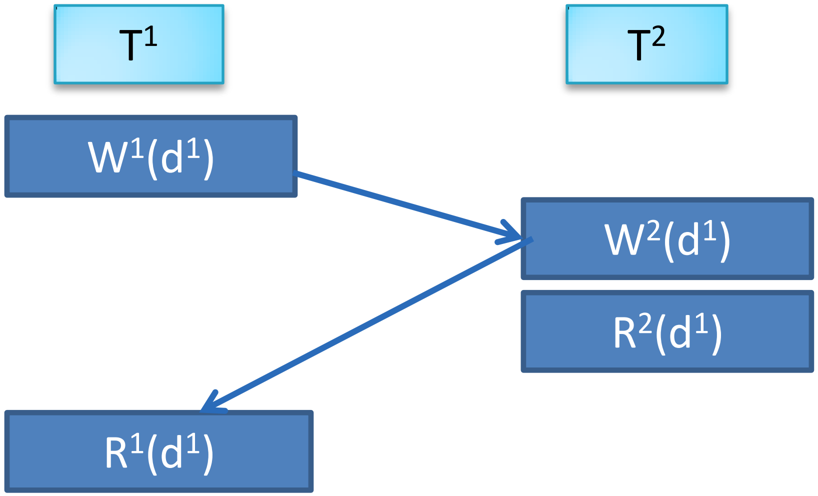
T^1 should precede T^2

Cycle in the transaction graph!

Not serializable!

Reverse reads and writes?

Serializability: Same Items



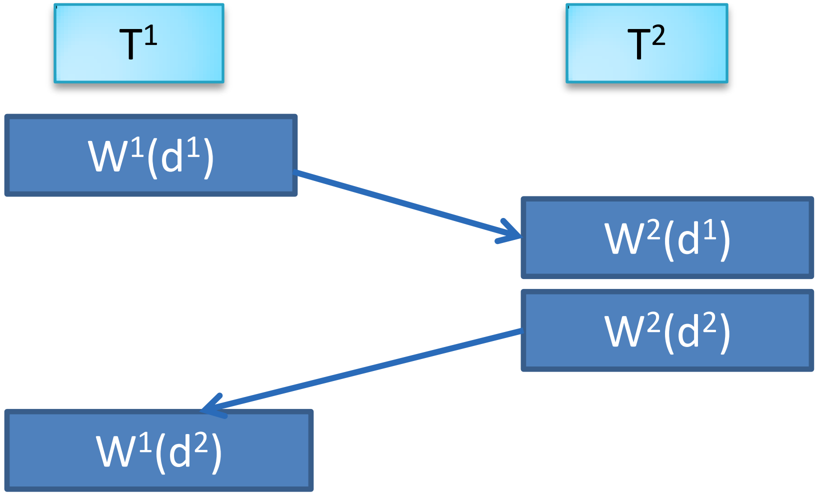
Serializable?

T^2 should precede T^1

T^1 should precede T^2

Not serializable!

Serializability: Multiple Items



T^1

T^2

$W^1(d^1)$

$W^2(d^1)$

$W^2(d^2)$

$W^1(d^2)$

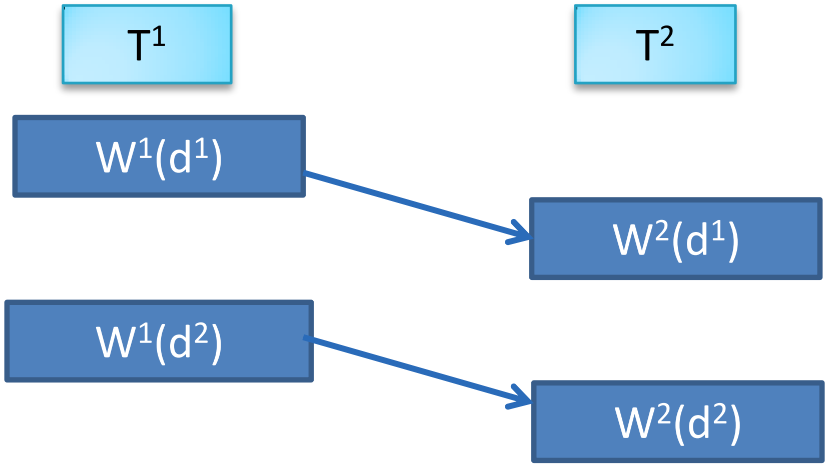
Serializable?

T^2 should follow T^1

T^1 should follow T^2

Not serializable!

Serializability: Multiple Items



Serializable?

T^2 should follow T^1

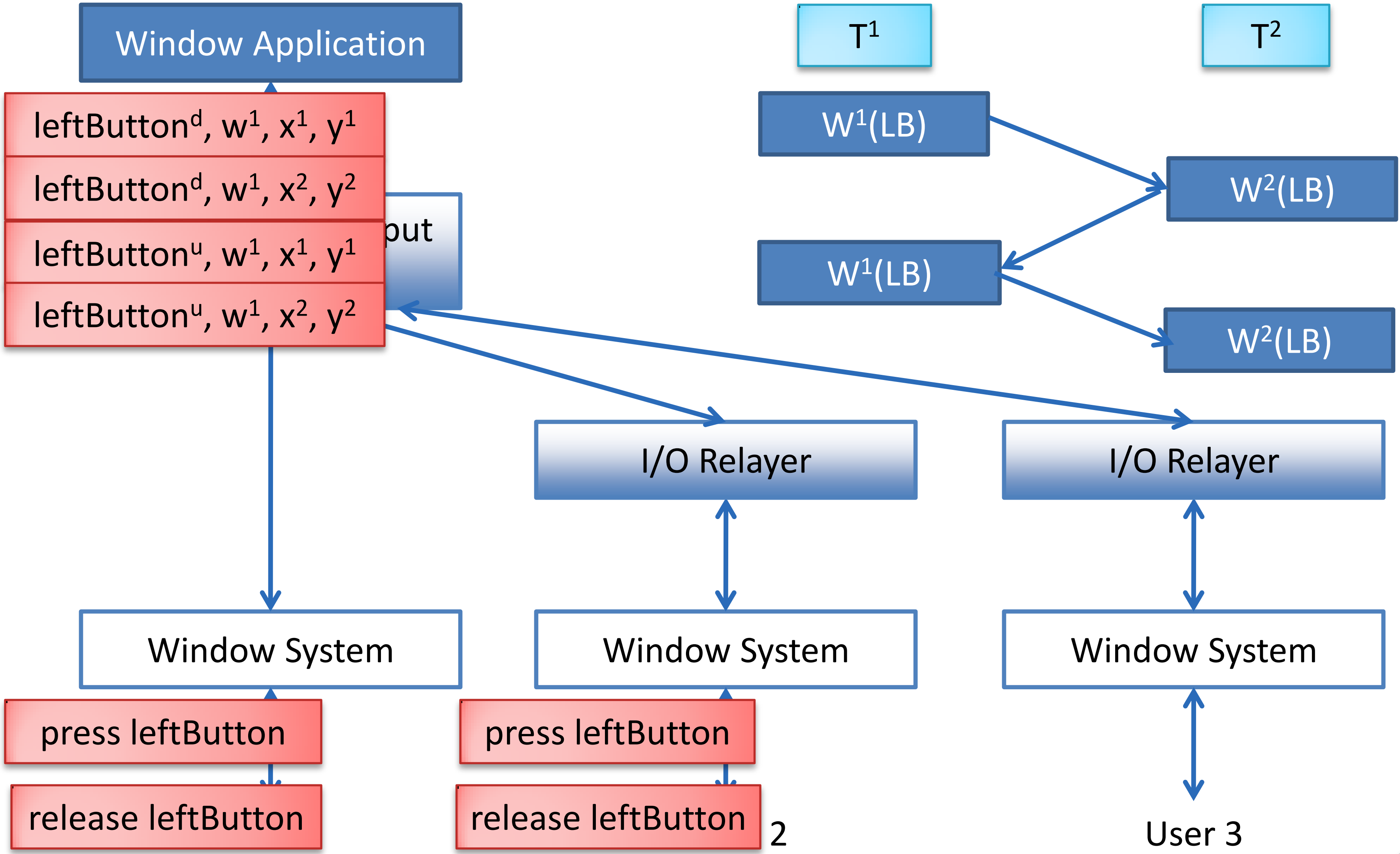
T^2 should follow T^1

Serializable!

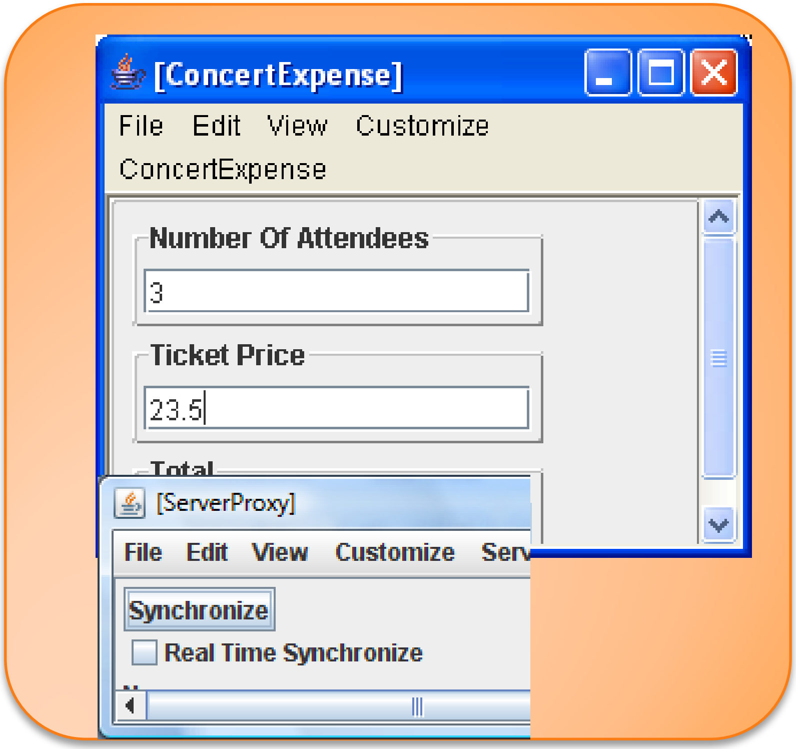
SERIALIZABILITY

- R-W Serializability
  - R-R operations (on same item) commute and hence can be reordered.
  - R-W and W-W do not commute and hence cannot be reordered. Cause R-W and W-W conflicts in non-serializable transactions

Shared Window Systems



Shared Model Systems

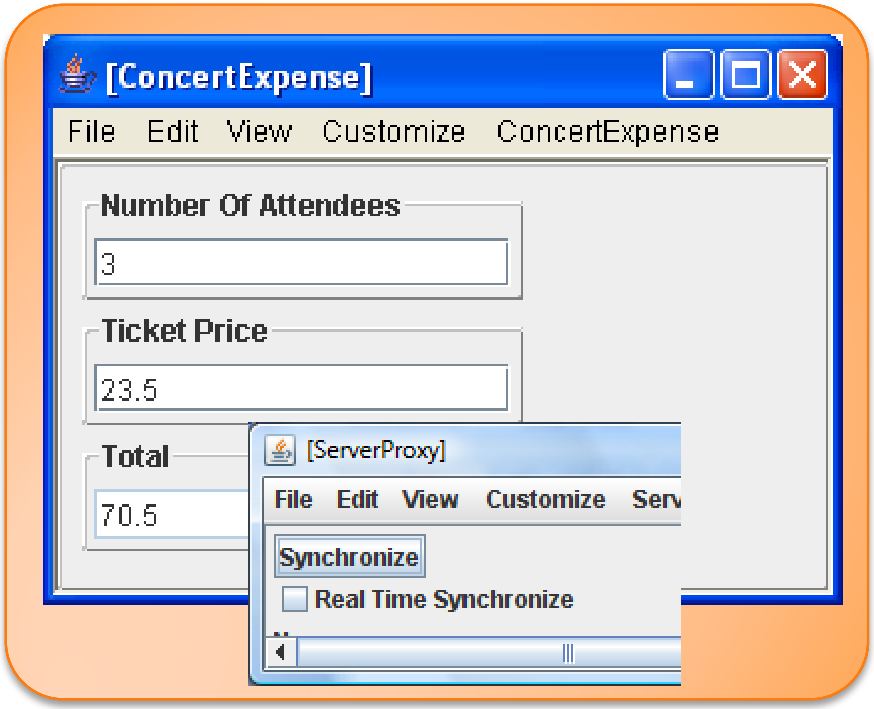

[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize

Number Of Attendees: 3

Ticket Price: 23.5

Total

☐ Real Time Synchronize


[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize

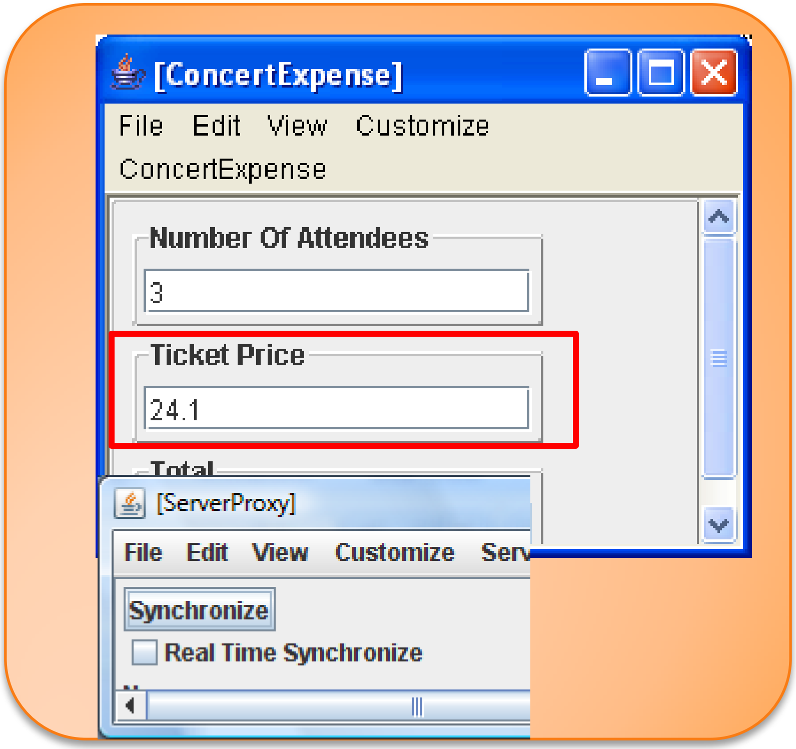
Number Of Attendees: 3

Ticket Price: 23.5

Total: 70.5

☐ Real Time Synchronize

Shared Model Systems

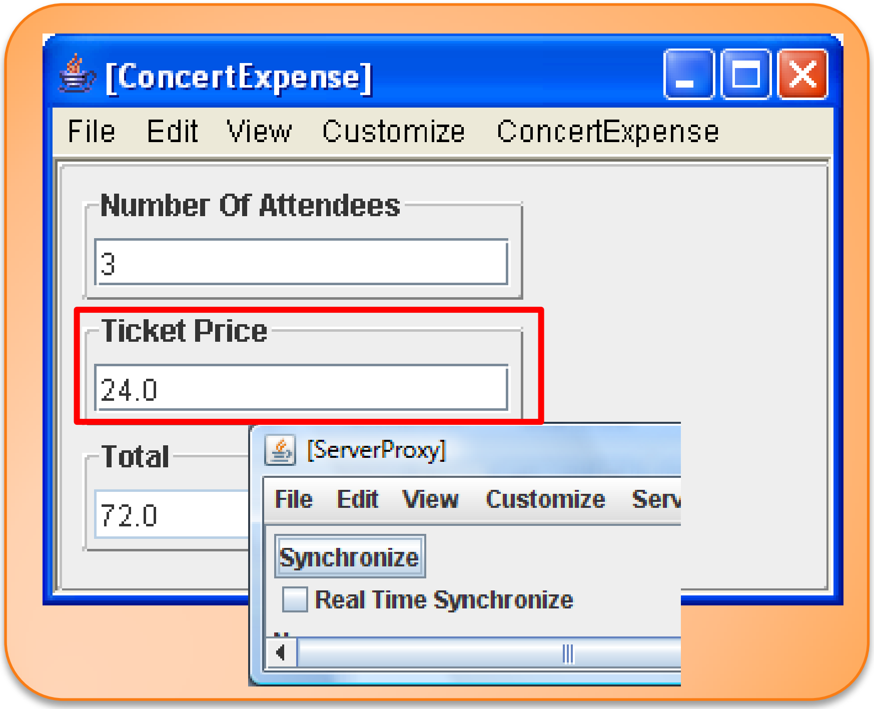

[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize

Number Of Attendees: 3

Ticket Price: 24.1

Total

□ Real Time Synchronize

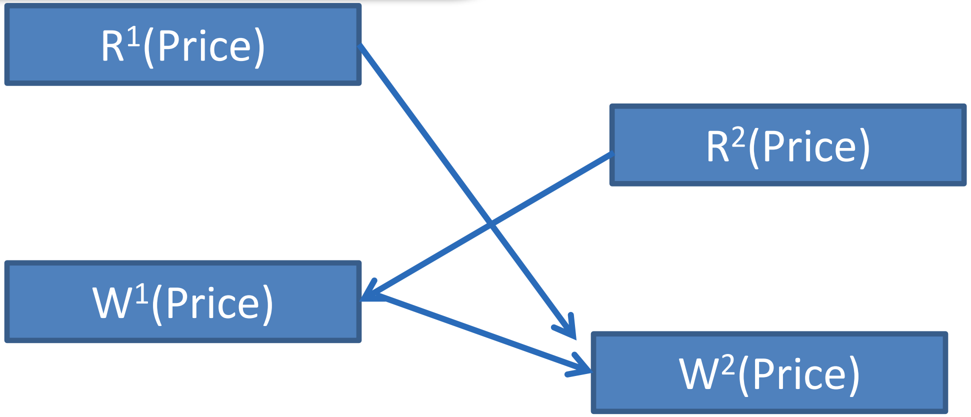

[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize

Number Of Attendees: 3

Ticket Price: 24.0

Total: 72.0

□ Real Time Synchronize



R^1(Price)

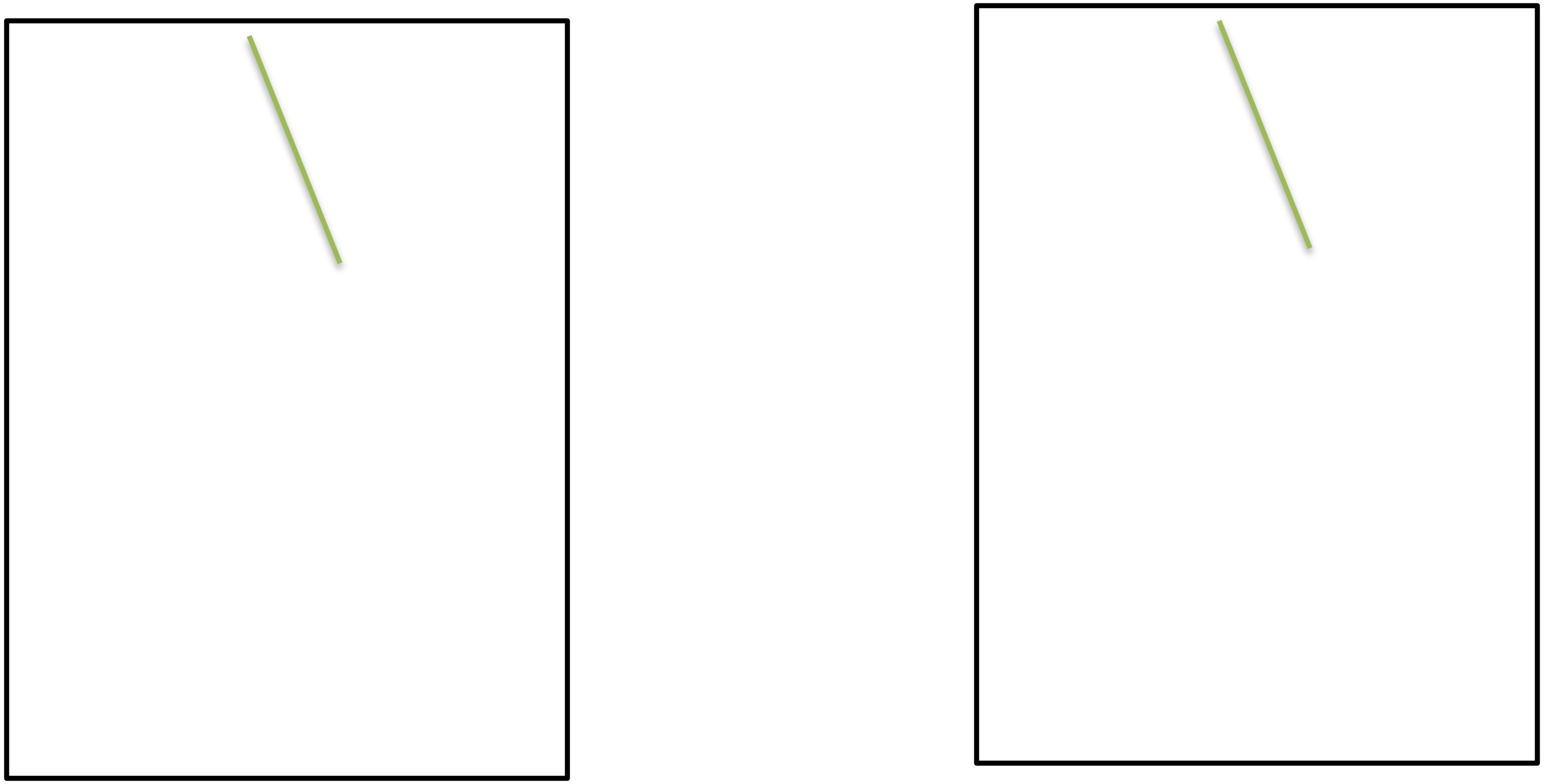
R^2(Price)

W^1(Price)

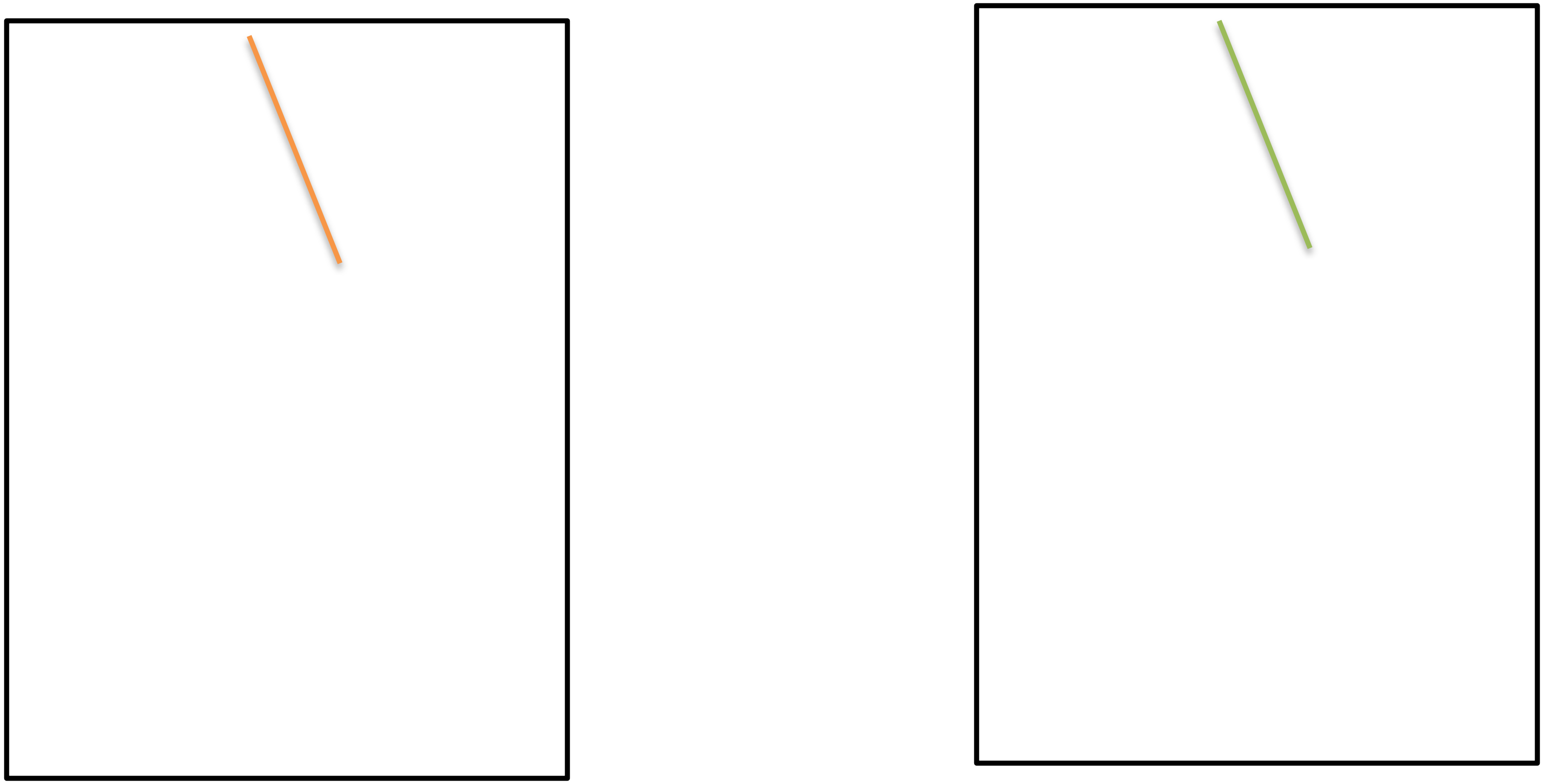
W^2(Price)

Not serializable!

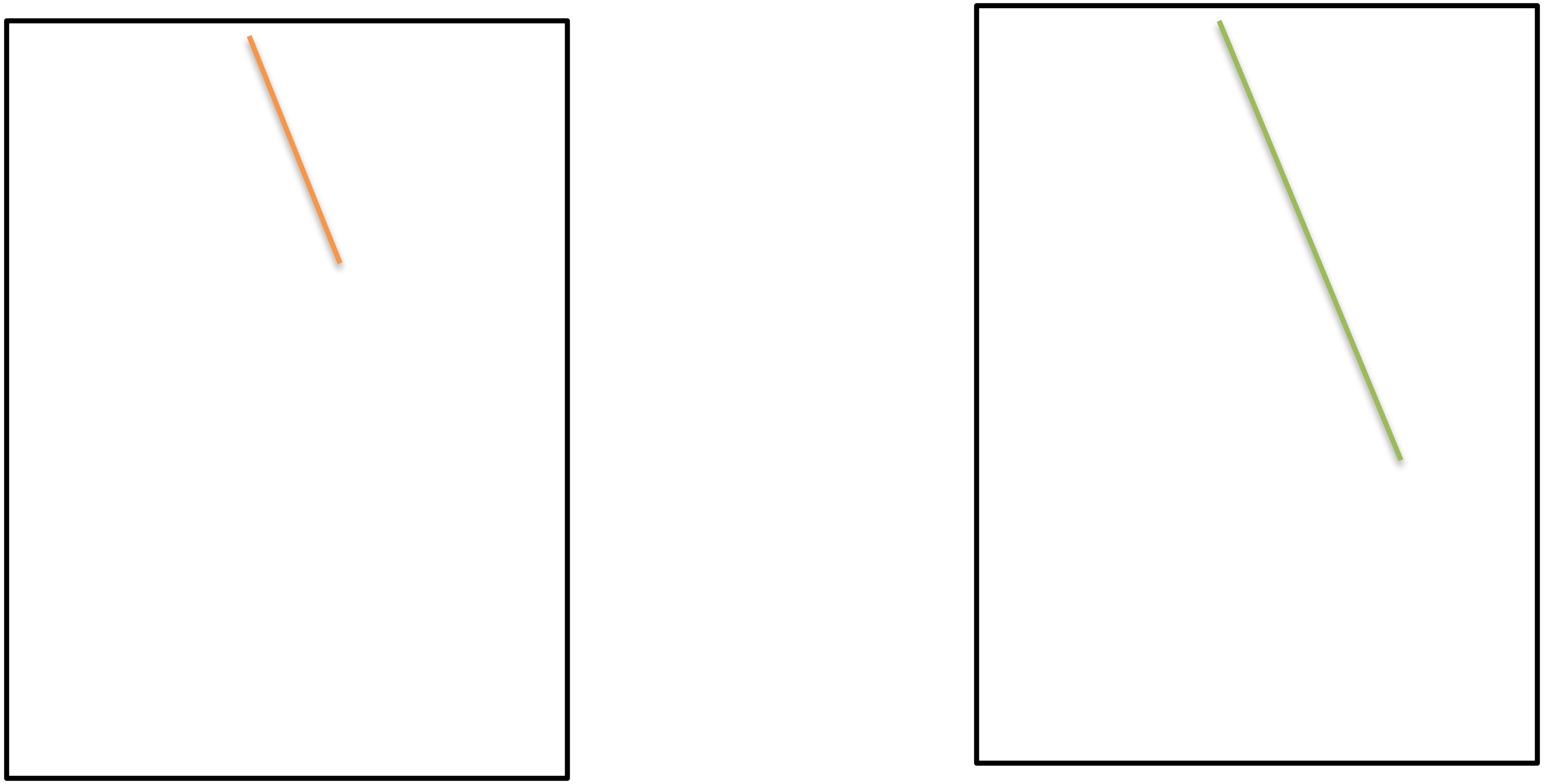
Concurrent Drawing: Initial State



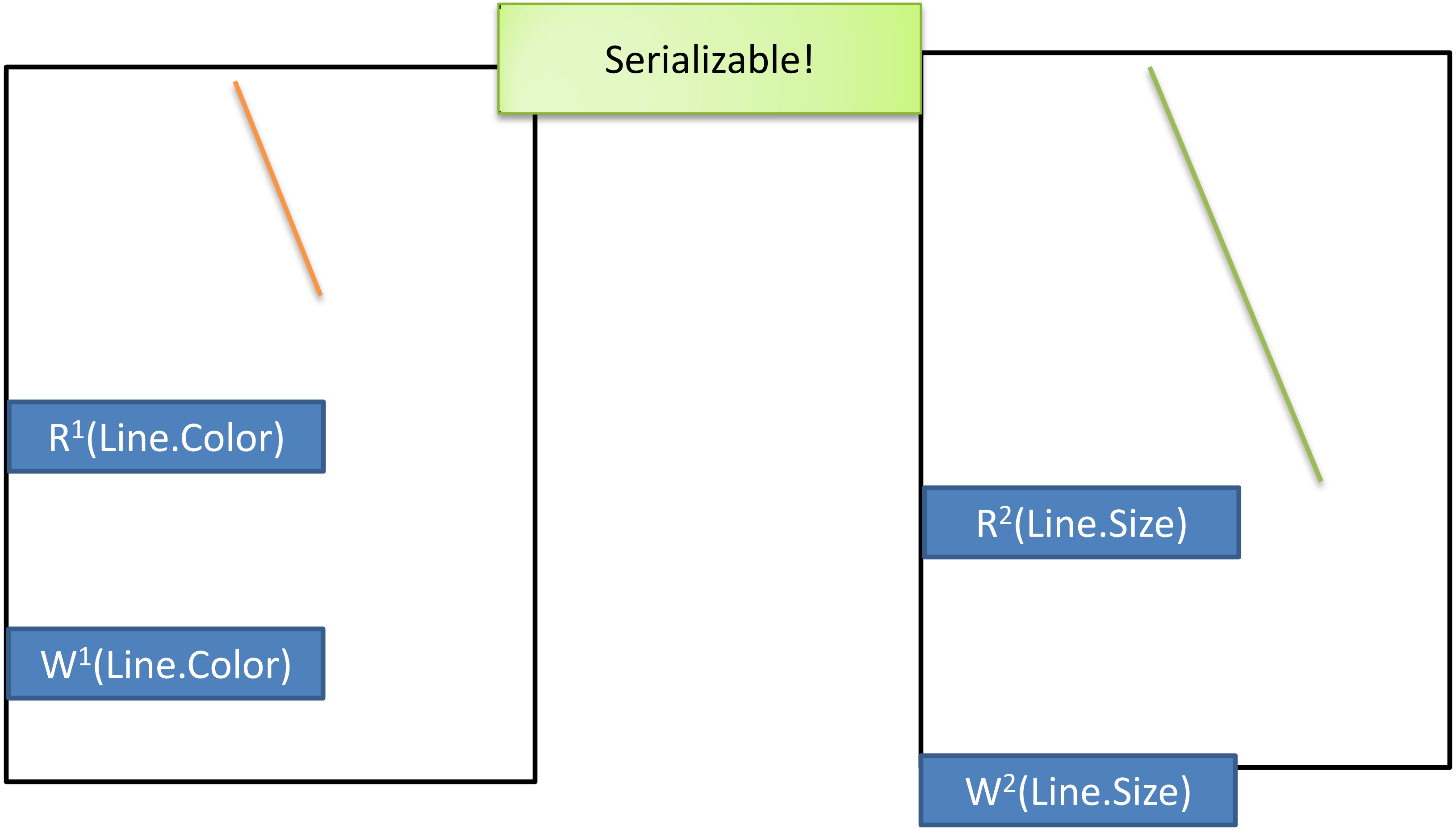
User^1 Change Not Seen By User^2



Modeling Concurrent Drawing



Fine-Grained Modeling Of Read



Serializable!

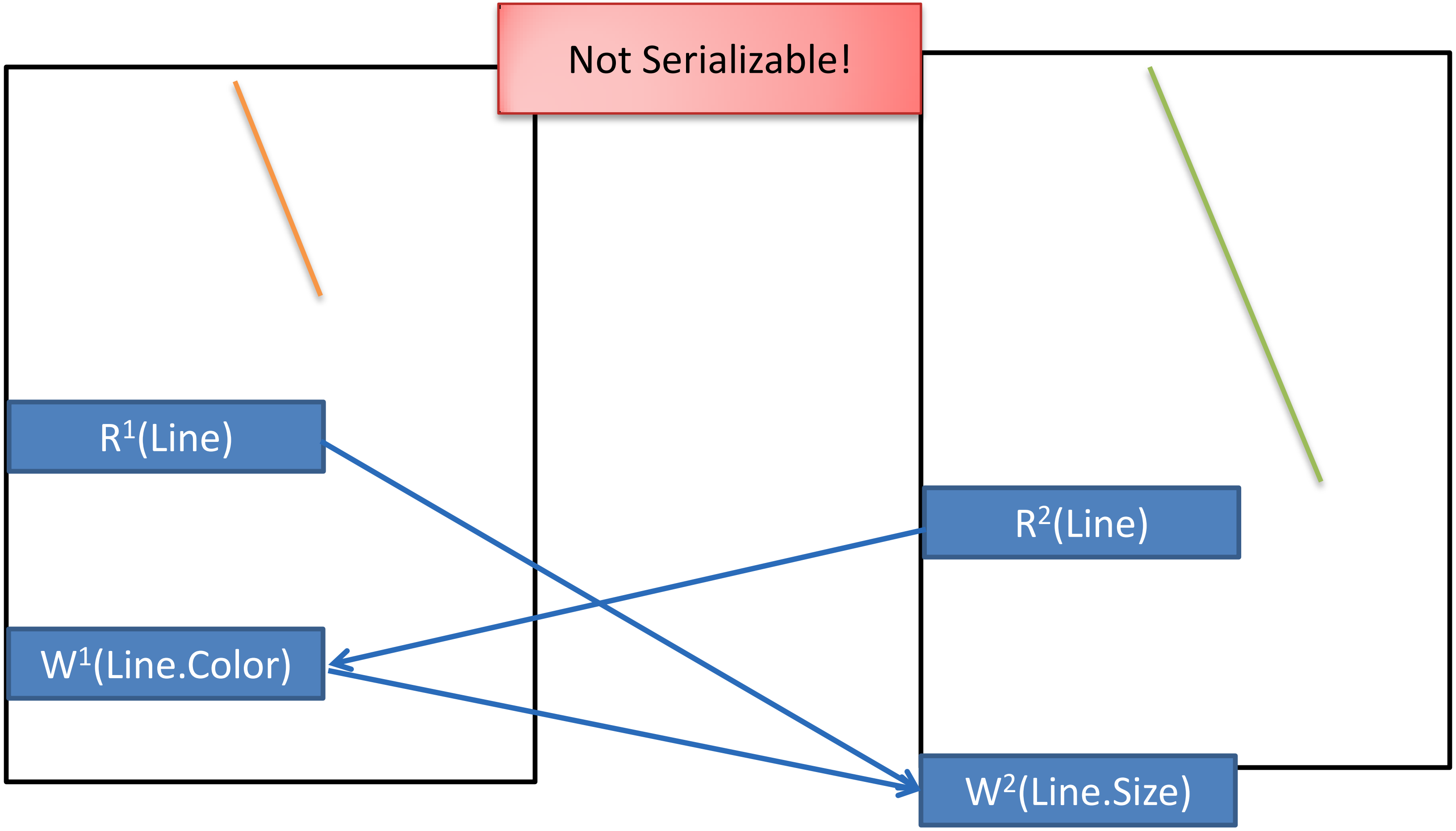
R^1(Line.Color)

W^1(Line.Color)

R^2(Line.Size)

W^2(Line.Size)

Coarse-Grained Read Modeling



Not Serializable!

R^1(Line)

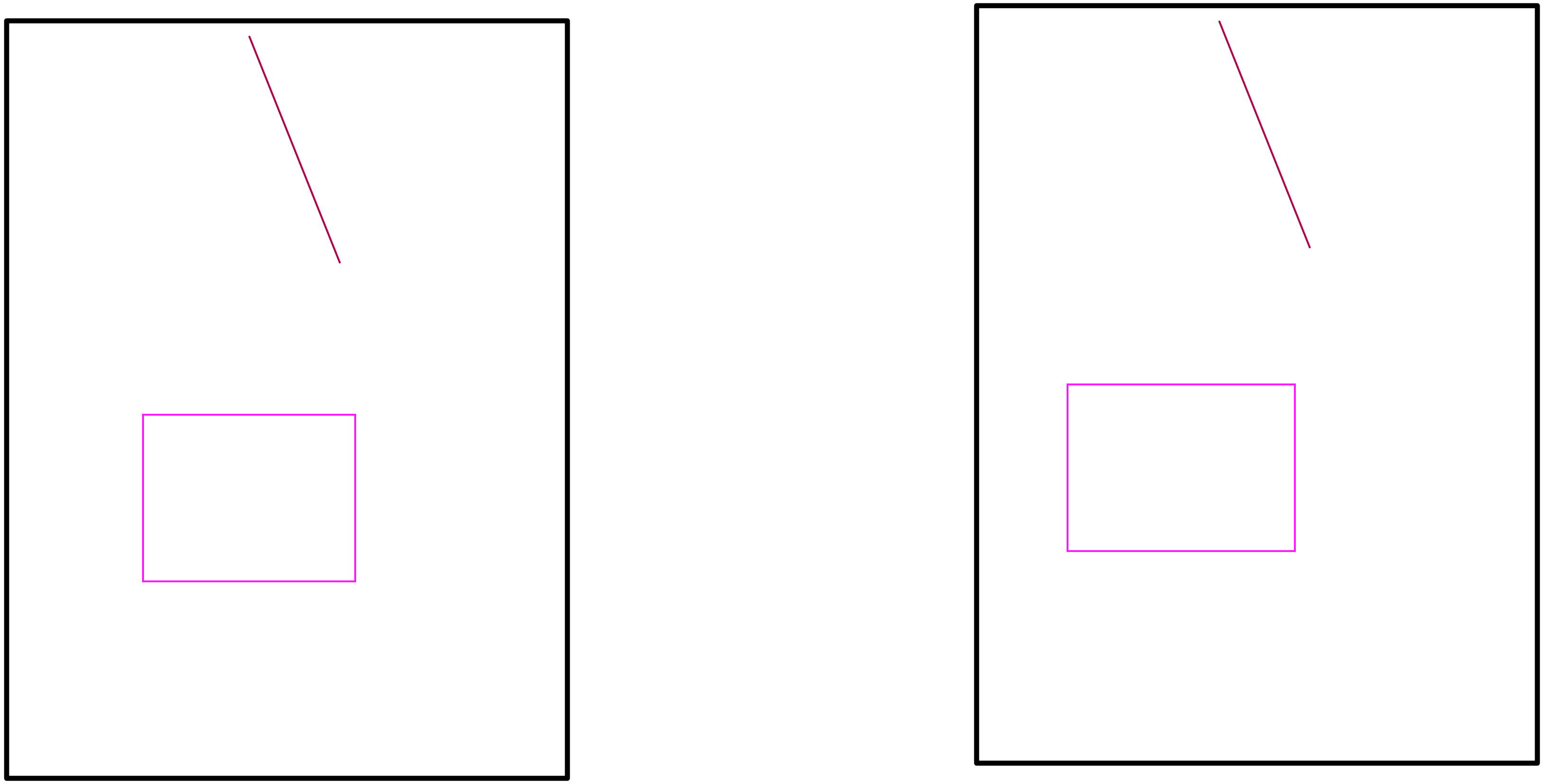
W^1(Line.Color)

R^2(Line)

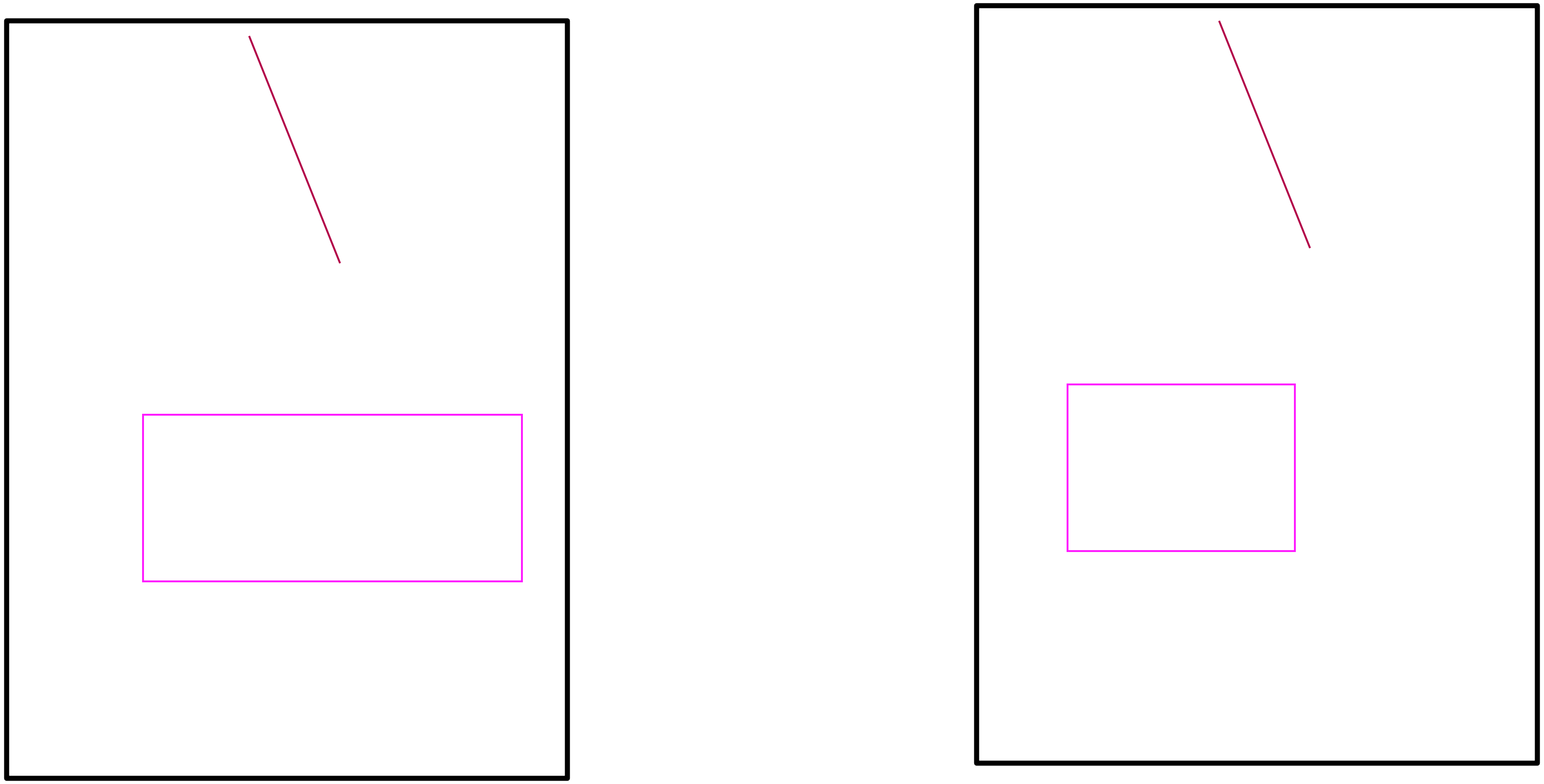
W^2(Line.Size)

Assuming whole line read

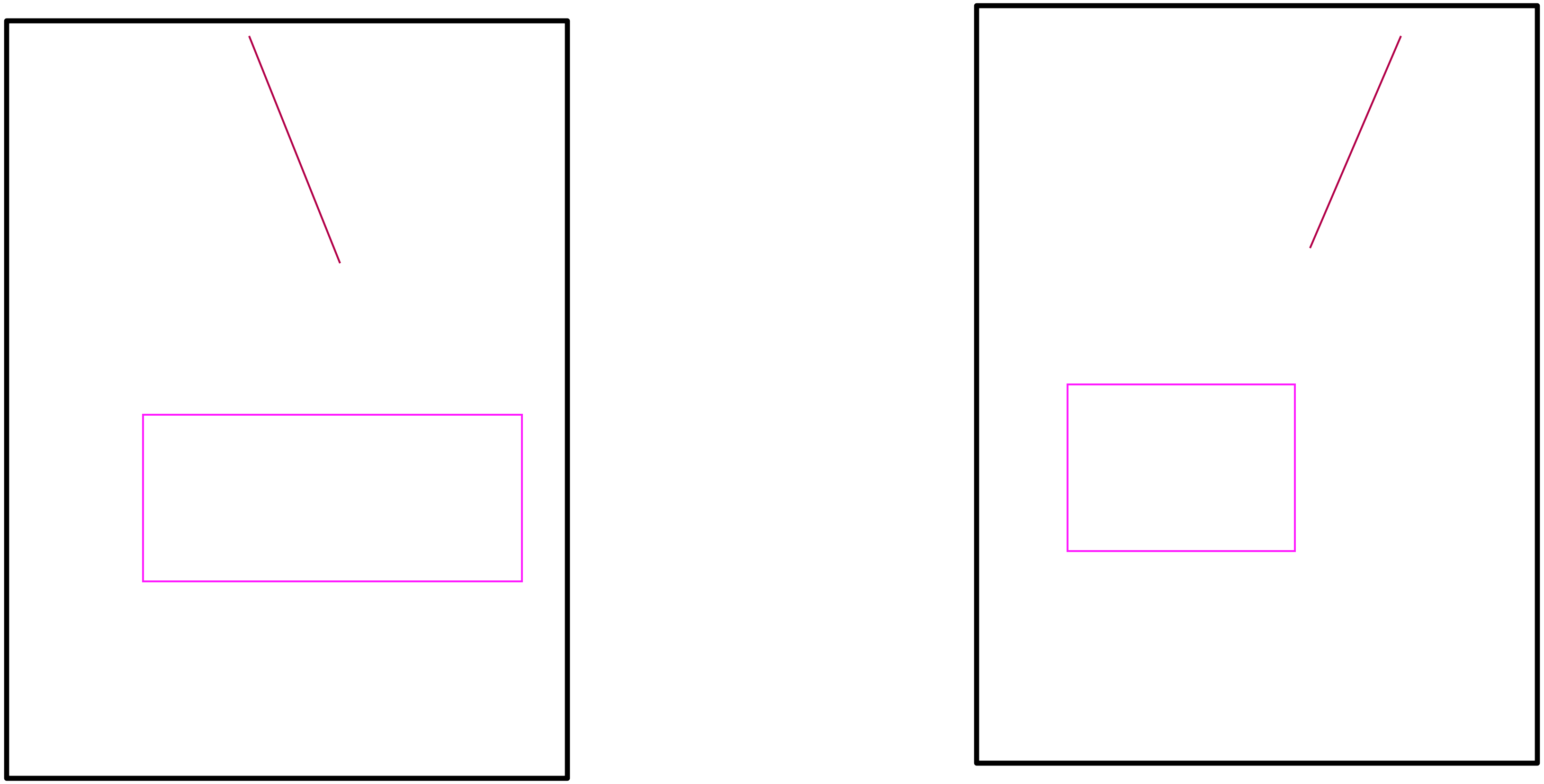
Concurrent Drawing



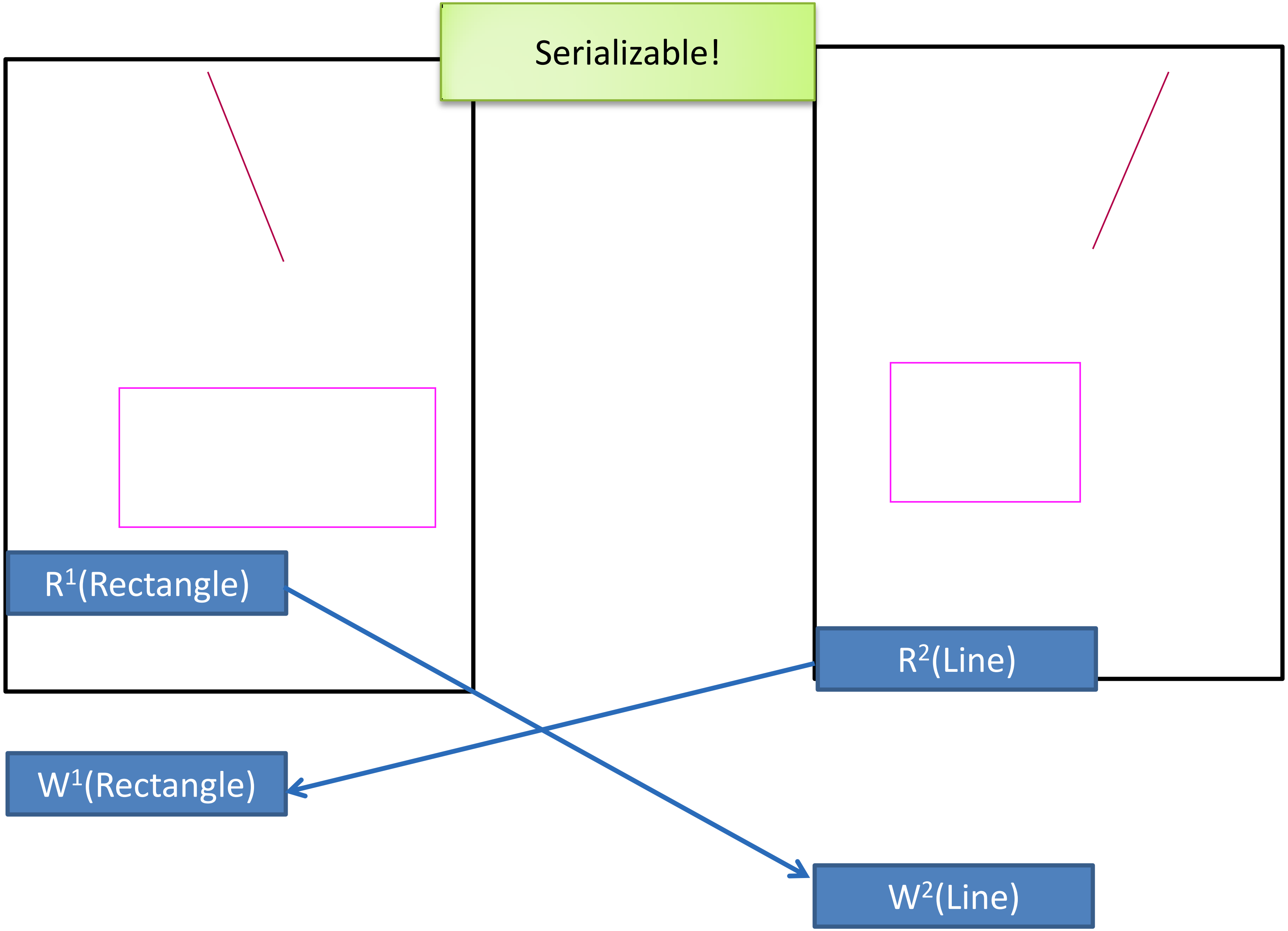
Concurrent Drawing



Concurrent Drawing



Fine-Grained Modeling



Serializable!

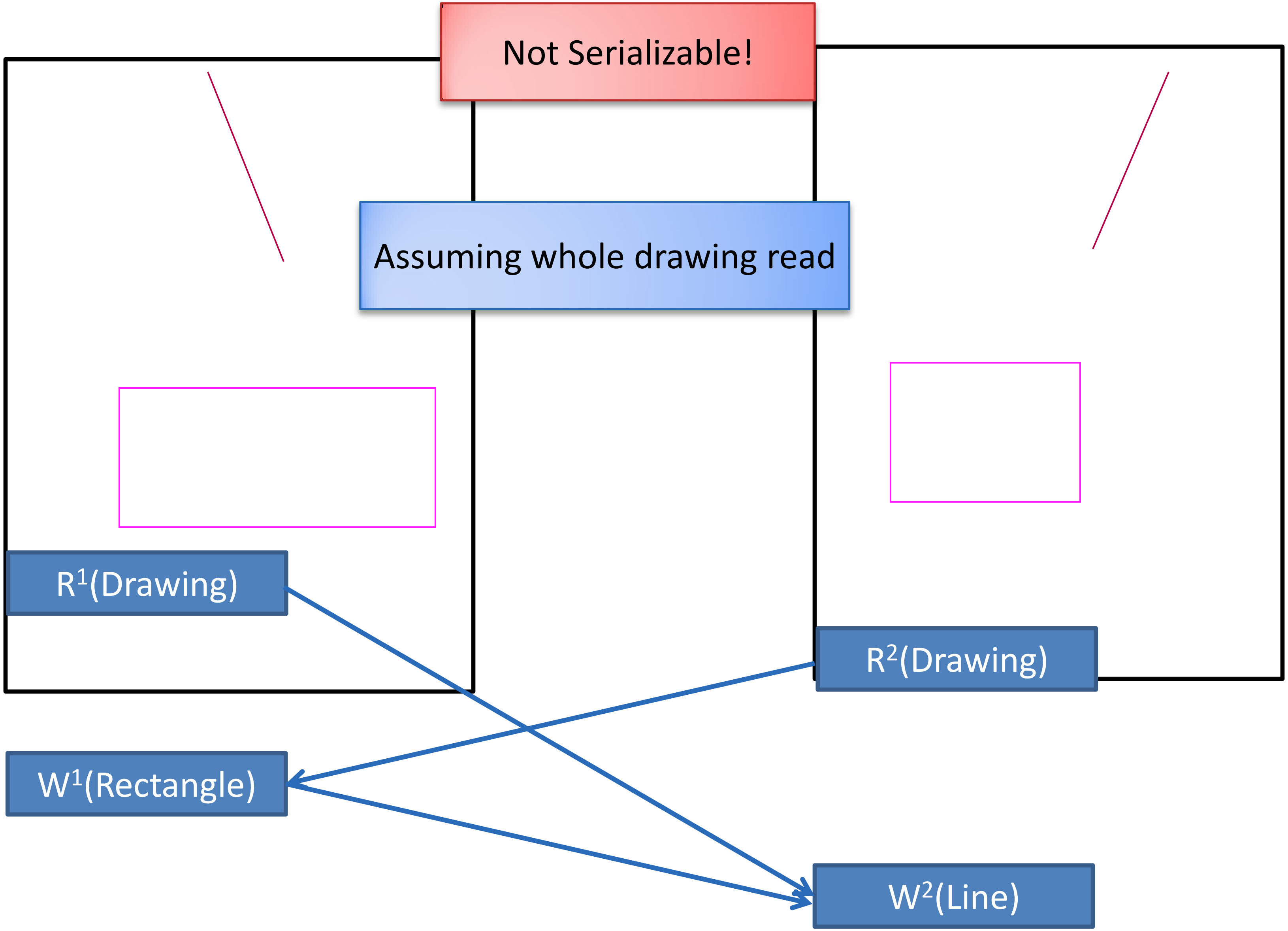
R^1(Rectangle)

R^2(Line)

W^1(Rectangle)

W^2(Line)

Coarse-Grained Modeling



Not Serializable!

Assuming whole drawing read

R^1(Drawing)

R^2(Drawing)

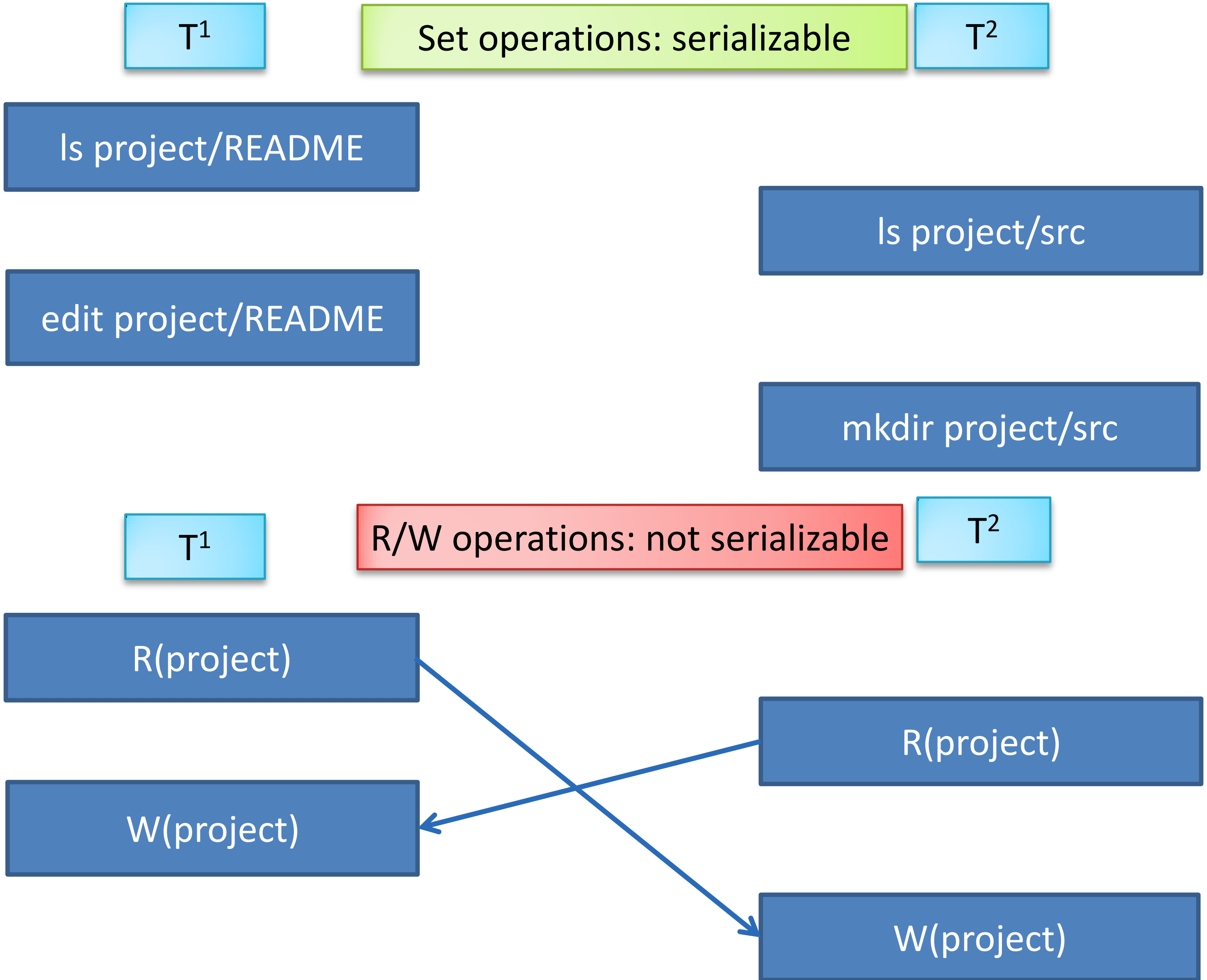
W^1(Rectangle)

W^2(Line)

THE PROBLEM OF TRACING READS

- In interactive application, not clear what user has read.
- Many collaborative systems take liberal approach, not tracking them.
- Strict serializability would require conservative approach of assuming everything on the display is read
- Eye and scroll tracking would help narrow down the read data

R/W vs. Type-Specific Serializability



T1 | Set operations: serializable | T2

ls project/README

ls project/src

edit project/README

mkdir project/src

T1 | R/W operations: not serializable | T2

R(project)

R(project)

W(project)

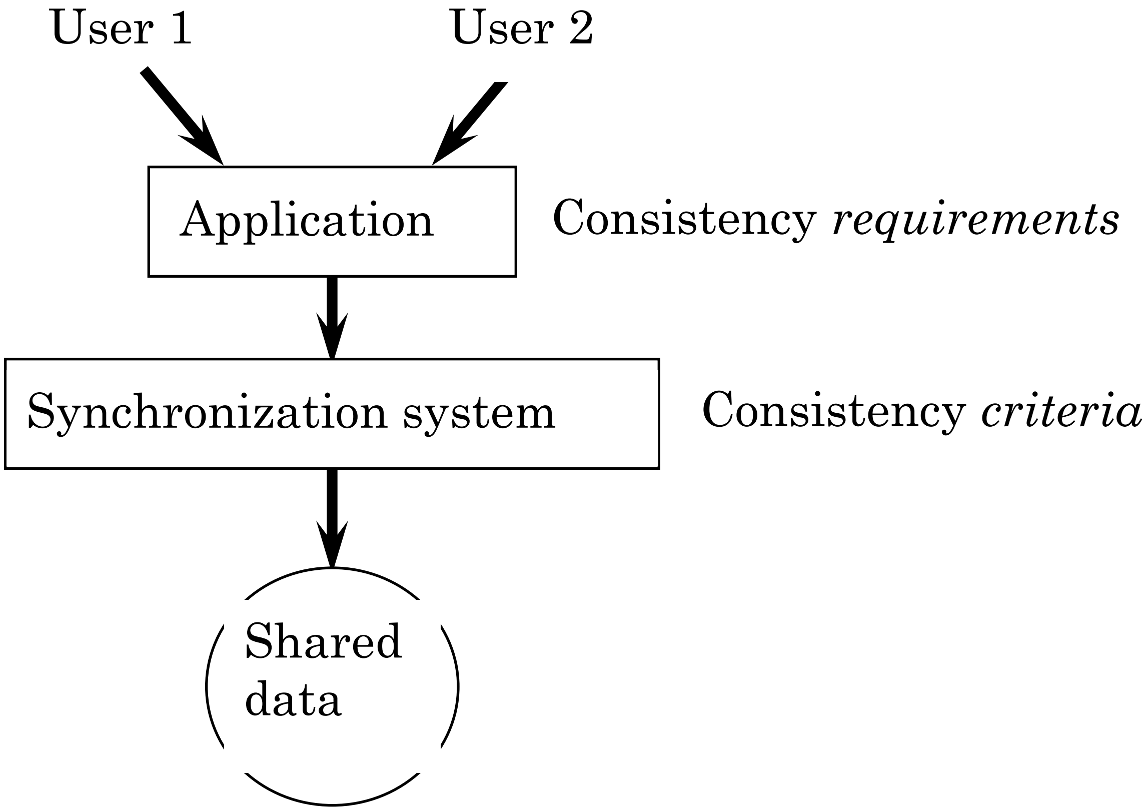
W(project)

Serializability

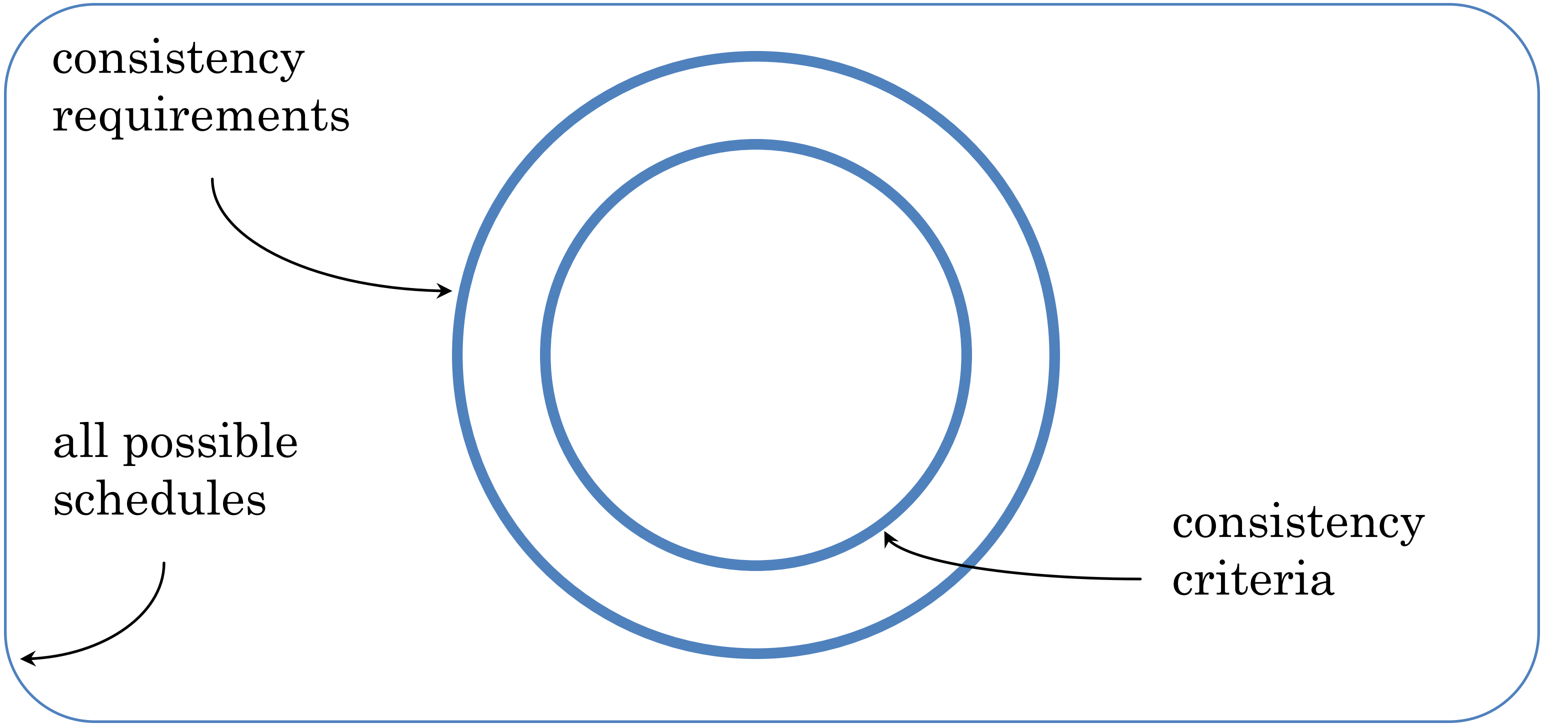
- Modeling ls as read and mkdir as write leads to directory-independent, non-serializable case
- Using type-specific semantics leads to serializable case

Synchronization systems

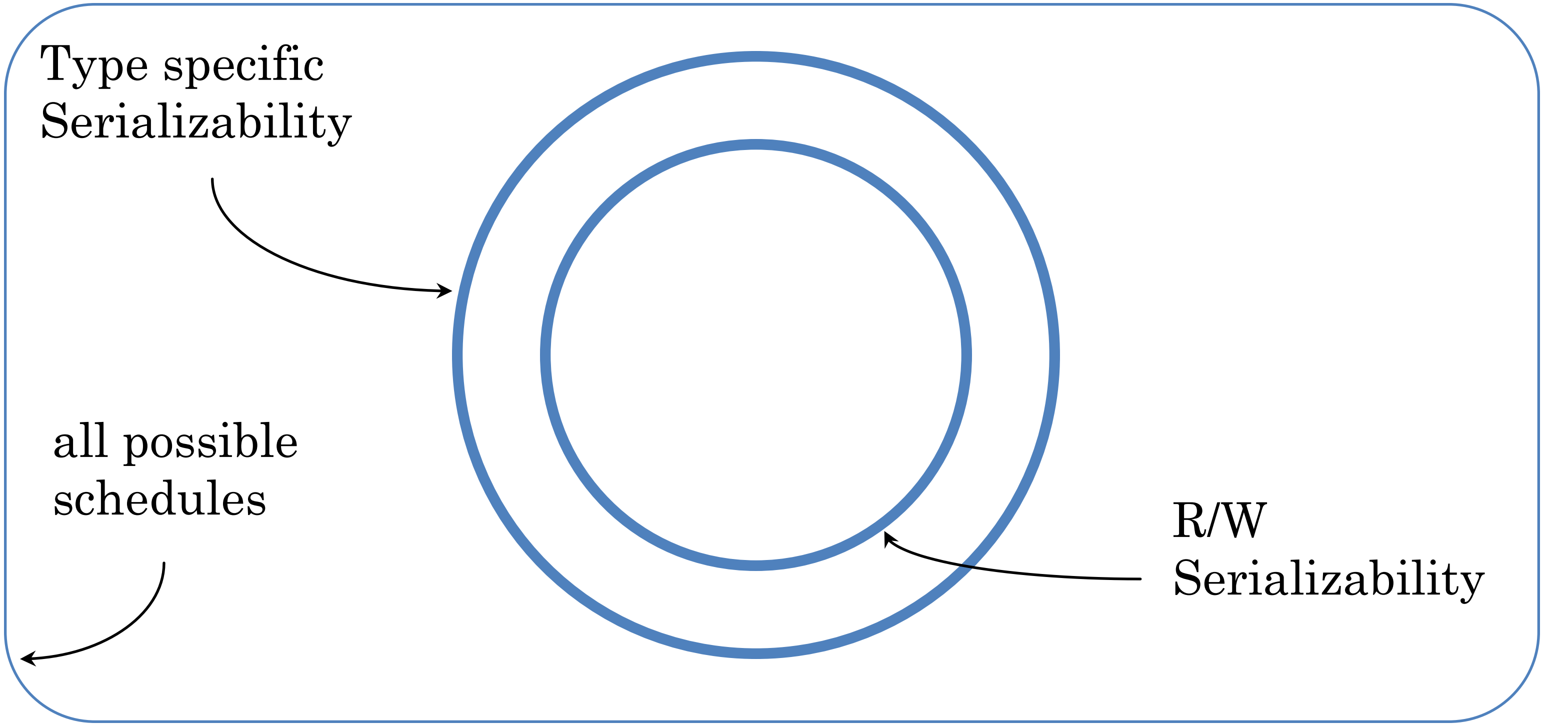
- Provide synchronization on behalf of applications



Consistency Criteria vs. Requirements



Consistency Criteria vs. Requirements



Type specific Serializability

all possible schedules

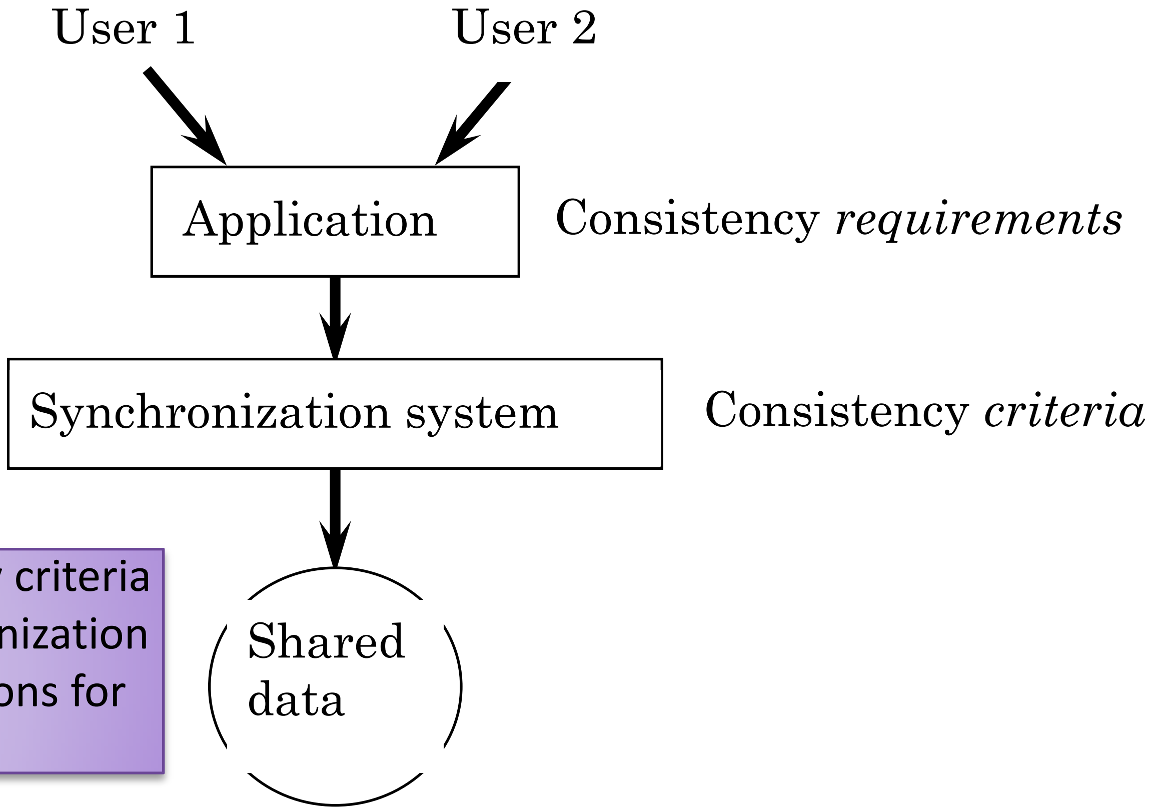
R/W Serializability

Consistency requirements & criteria

- Consistency requirements:
  - specify the set of ideally allowable schedules.
  - “Users may concurrently add room reservations (that don’t overlap), but may not concurrently change the same reservation.”
- Consistency criteria:
  - specify the set of actually allowed schedules.
  - “Users must access the set of reservations one at a time.”

SYNCHRONIZATION SYSTEMS

- Provide synchronization on behalf of applications



User 1

User 2

Application

Consistency *requirements*

Synchronization system

Consistency *criteria*

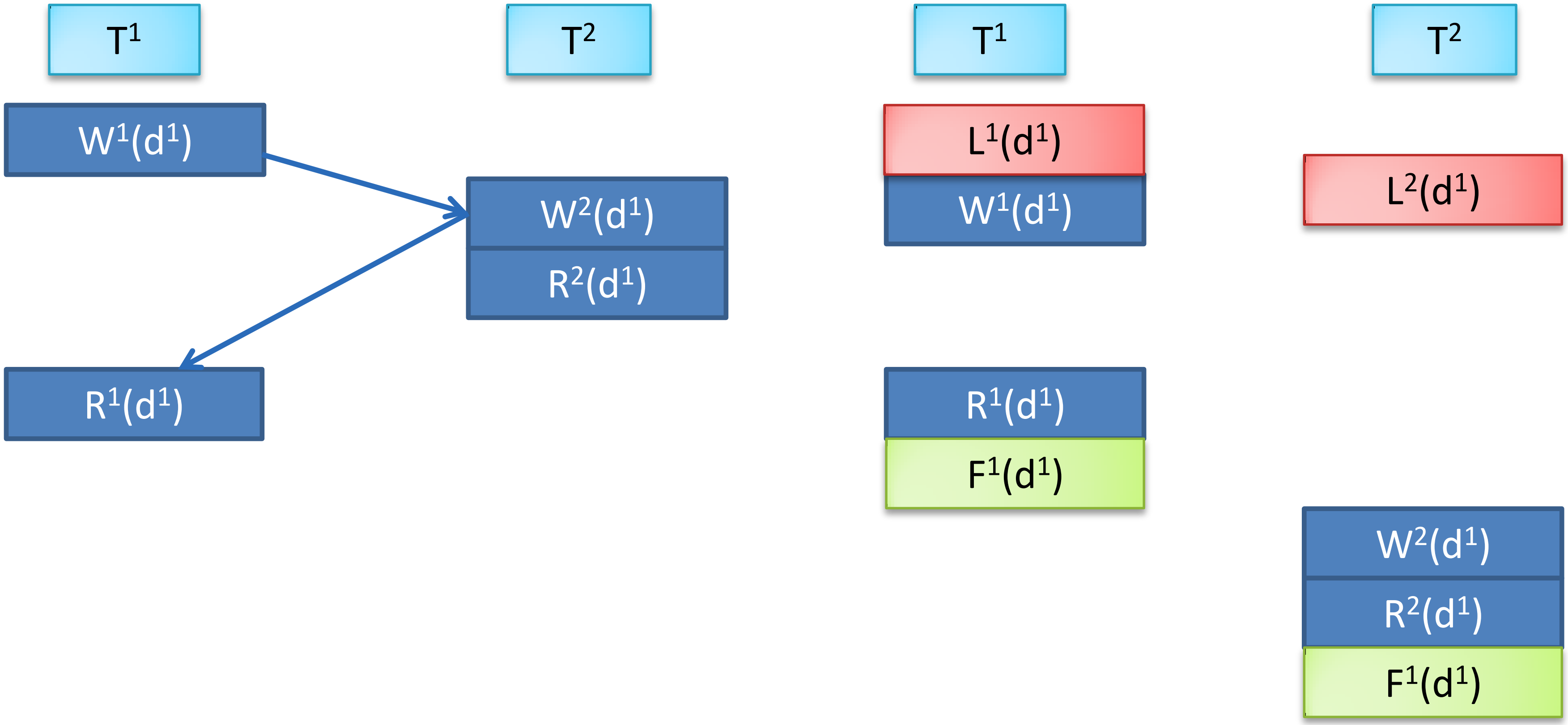
Shared data

Given some consistency criteria how should the synchronization system check transactions for serializability?

VALIDATION/CHECKING TIME

- Pessimistic
  - Early
  - Failure => block
- Optimistic
  - Late
  - Failure => abort
  - Interactive transaction?
    - Wasted human work not redoable perhaps
- Merging
  - Late, not serializable
  - Merging, new transaction to replace conflicting transactions

Locking: One Item



T^1

$W^1(d^1)$

$R^1(d^1)$

T^2

$W^2(d^1)$

$R^2(d^1)$

T^1

$L^1(d^1)$

$W^1(d^1)$

$R^1(d^1)$

$F^1(d^1)$

T^2

$L^2(d^1)$

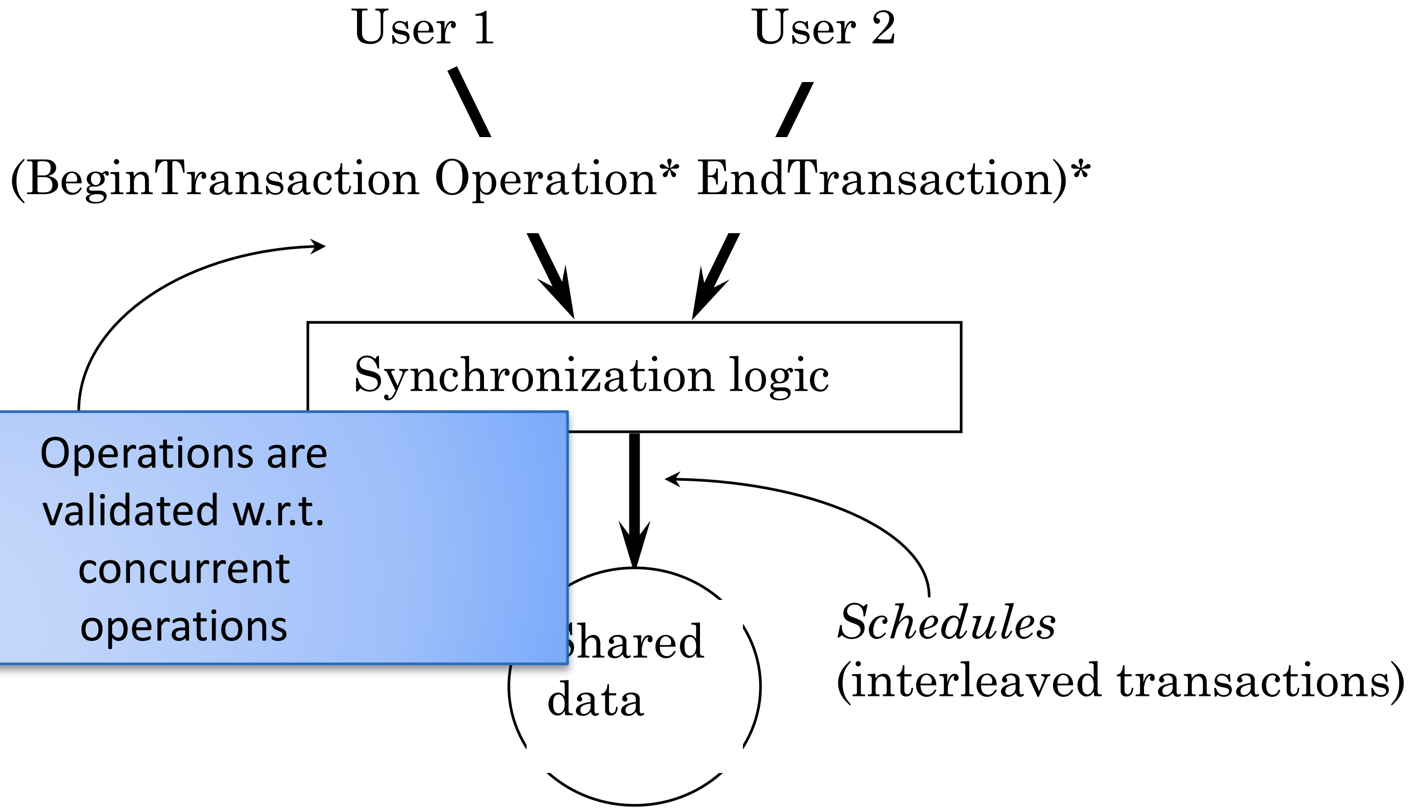
$W^2(d^1)$

$R^2(d^1)$

$F^1(d^1)$

Synchronization model (Review)

- Users submit operations in *transactions*



User 1

User 2

(BeginTransaction Operation* EndTransaction)*

Synchronization logic

Shared data

Schedules
(interleaved transactions)

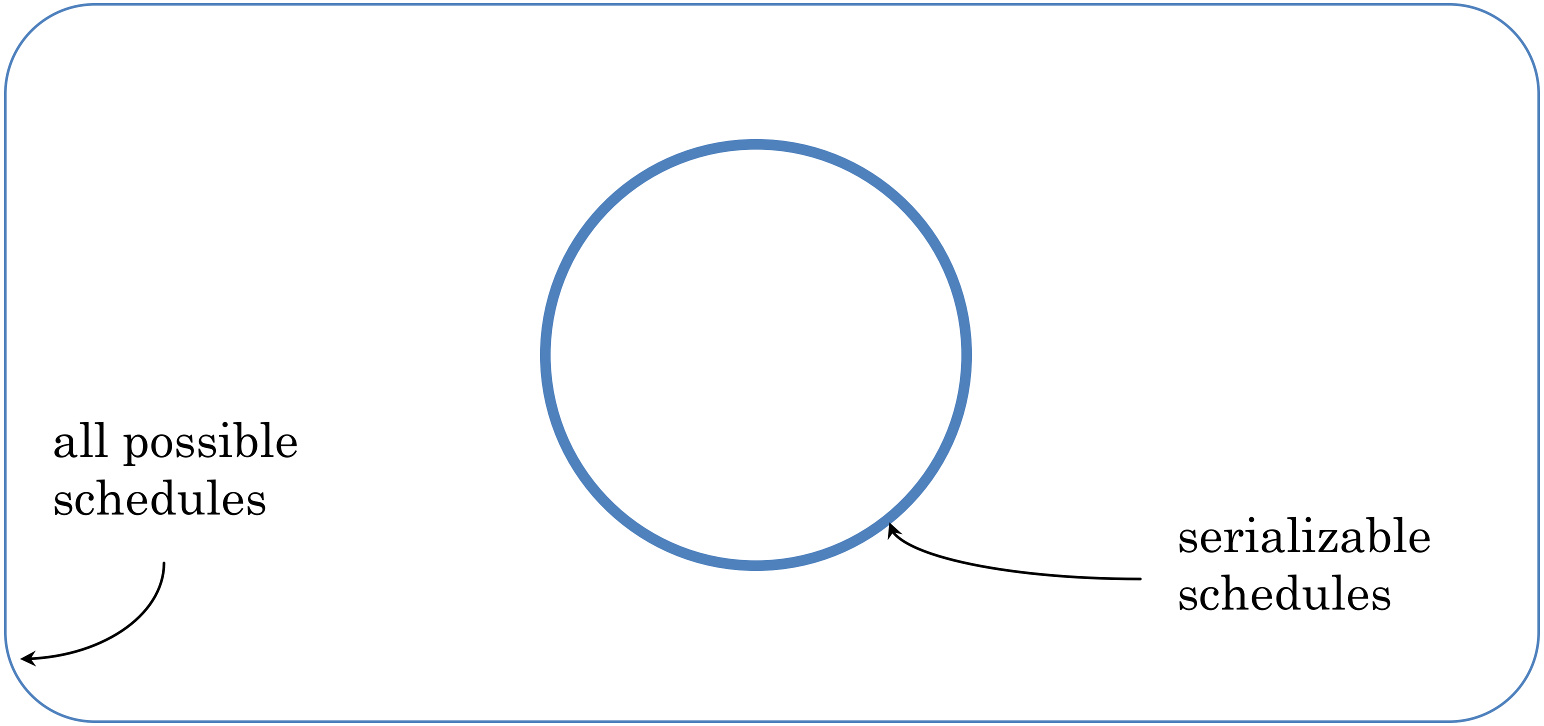
Operations are validated w.r.t. concurrent operations

Transactions (Review)

- A (tomic)
  - Either all action of a transaction occur or none
- C (onsistent)
  - Each transaction leaves shared state in a consistent state, where consistency is application-defined
- I (solation)
  - Actions of concurrent transactions are isolated so that together they leave the shared state in a consistent state
- D (urability)
  - Actions of a transaction persist – written to stable storage) vs. persistent storage
  - Stable – atomic write no errors;
  - Persistent – errors possible

Traditional Isolation criteria: Serializability (Review)

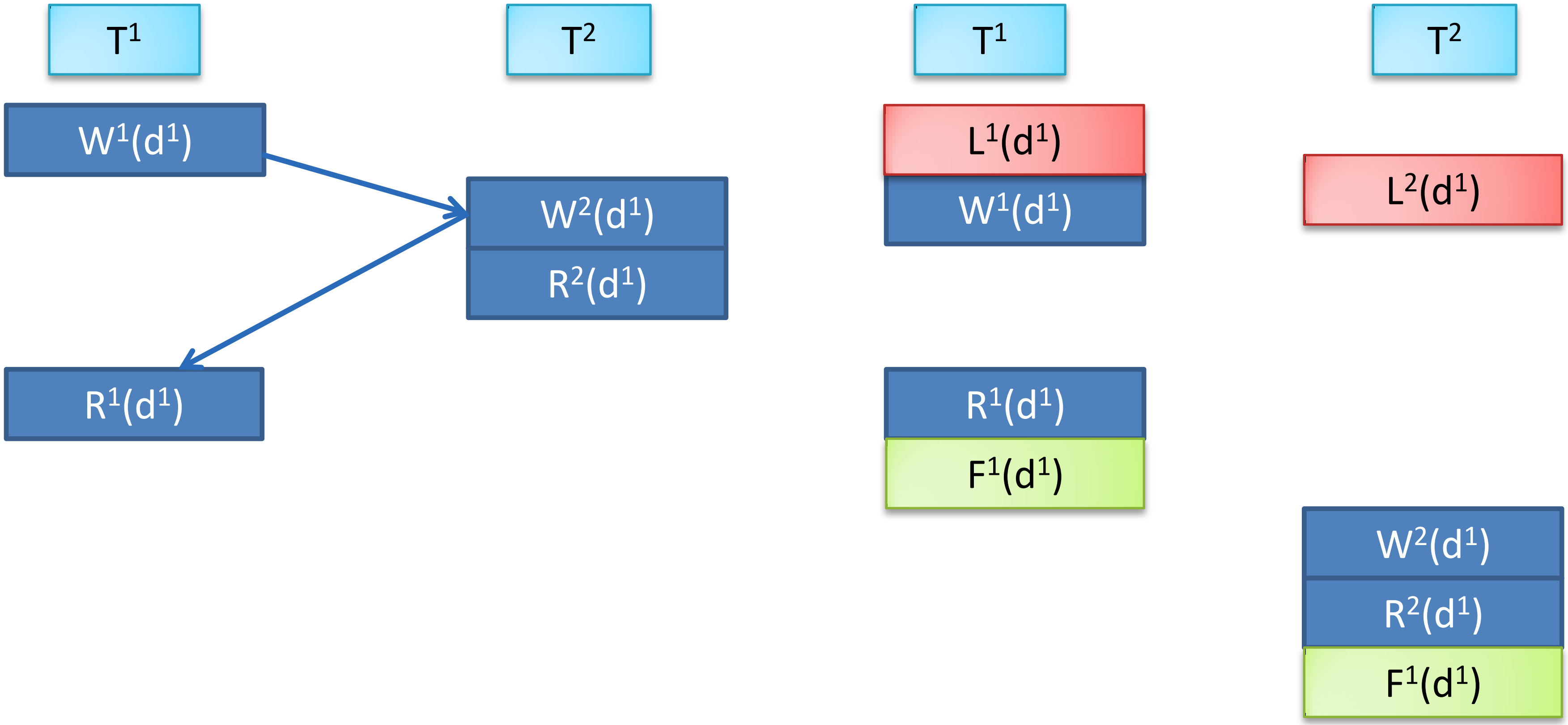
- Concurrent transactions execute as if they were submitted one after the other, leaving data in consistent state



Validation/Checking Time (Review)

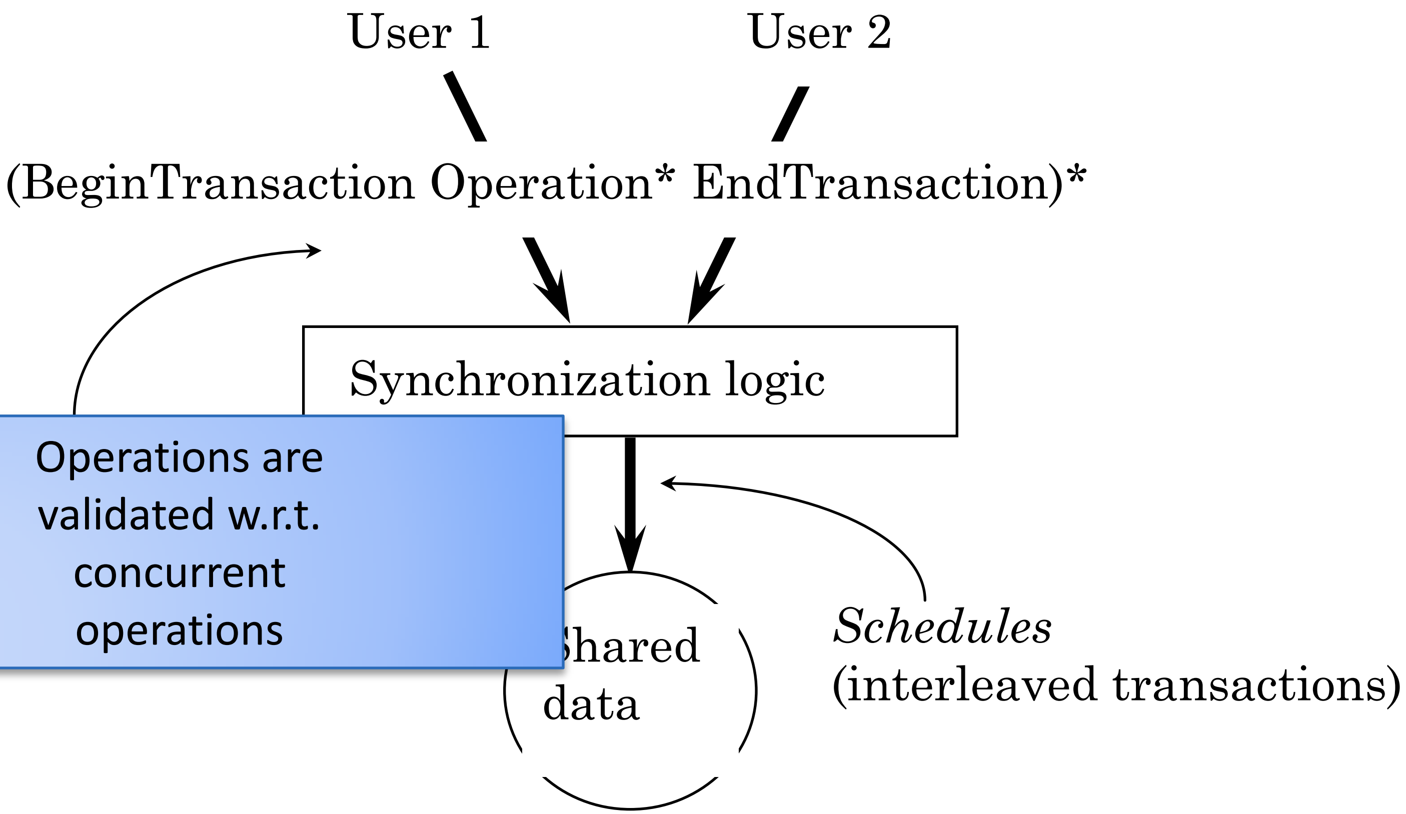
- Pessimistic
  - Early
  - Failure => block
- Optimistic
  - Late
  - Failure => abort
  - Interactive transaction?
    - Wasted human work not redoable perhaps
- Merging
  - Late, not serializable
  - Merging, new transaction to replace conflicting transactions

Locking: One Item (Review)



Synchronization model (Review)

- Users submit operations in *transactions*


User 1
User 2
(BeginTransaction Operation* EndTransaction)*
Synchronization logic
Shared
data
Schedules
(interleaved transactions)

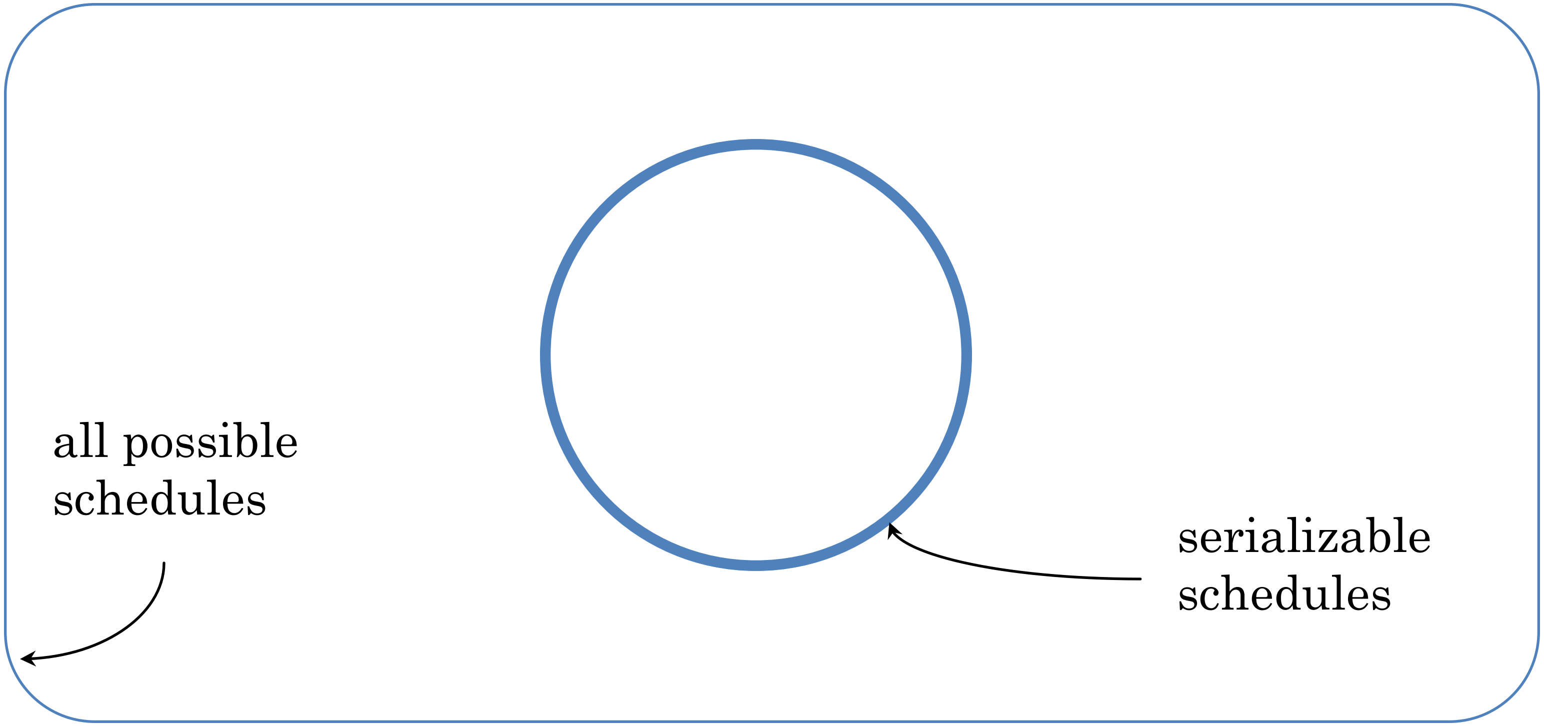
Operations are validated w.r.t. concurrent operations

Transactions (Review)

- A (tomic)
  - Either all action of a transaction occur or none
- C (onsistent)
  - Each transaction leaves shared state in a consistent state, where consistency is application-defined
- I (solation)
  - Actions of concurrent transactions are isolated so that together they leave the shared state in a consistent state
- D (urability)
  - Actions of a transaction persist – written to stable storage) vs. persistent storage
  - Stable – atomic write no errors;
  - Persistent – errors possible

Traditional Isolation criteria: Serializability (Review)

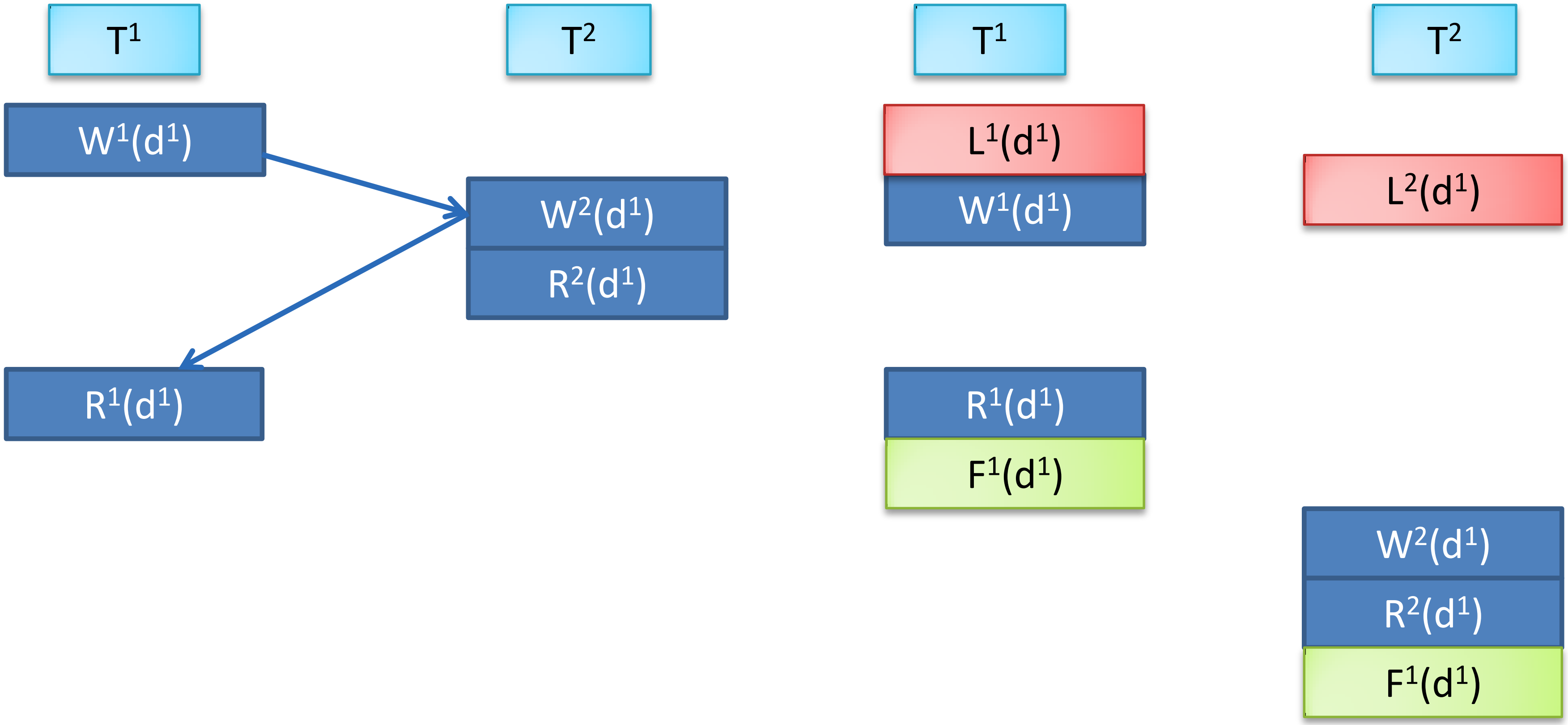
- Concurrent transactions execute as if they were submitted one after the other, leaving data in consistent state



VALIDATION/CHECKING TIME (REVIEW)

- Pessimistic
  - Early
  - Failure => block
- Optimistic
  - Late
  - Failure => abort
  - Interactive transaction?
    - Wasted human work not redoable perhaps
- Merging
  - Late, not serializable
  - Merging, new transaction to replace conflicting transactions

Locking: One Item (Review)



T^1

$W^1(d^1)$

$R^1(d^1)$

T^2

$W^2(d^1)$

$R^2(d^1)$

T^1

$L^1(d^1)$

$W^1(d^1)$

$R^1(d^1)$

$F^1(d^1)$

T^2

$L^2(d^1)$

$W^2(d^1)$

$R^2(d^1)$

$F^1(d^1)$

Lock Compatibility Matrix

Data Item D	Locked	Unlocked
Lock	No	Yes

Issues (in collaborative systems)?

Lock Compatibility Matrix (Review)

Data Item D	Locked	Unlocked
Lock	No	Yes

Issues (in collaborative systems)?

ISSUES

Lock Denial Semantics?

User Interface for Locking and Unlocking?

Implementation of locking in a distributed collaborative environment?

Lock Denial

Synchronous: Programmed blocked until lock given

UI Thread should not block

Synchronous with timeout: Like synchronous but timeout returns false

Asynchronous: Callback when lock given

Non blocking: Callback when lock available, try again

Non blocking: No callback, polling

Issues

Lock Denial Semantics? ✓

User Interface for Locking and Unlocking?

Implementation of locking in a collaborative environment?

User-Interface

Explicit/Implicit Locking

Explicit/Implicit Unlocking

UI: Explicit/Implicit Locking

Explicit	Lock O Append O, E1 Delete O, E2
Selection-implied	Select Object → Lock Object + Select Object
Key-implied	Press Key → Lock Buffer + Process Key
Dragging-implied	Start Dragging → Lock Object + Start Dragging

Explicit/Implicit Unlocking

Explicit	Append O, E1 Delete O, E2 Unlock O
Selection-implied	Unselect Object → Unselect object + Unlock object
Key-implied	Release Key → Unlock Buffer + Unlock object
Dragging-implied	Stop Dragging → Stop Dragging + Unlock Object

Analogues of explicit/implicit locking

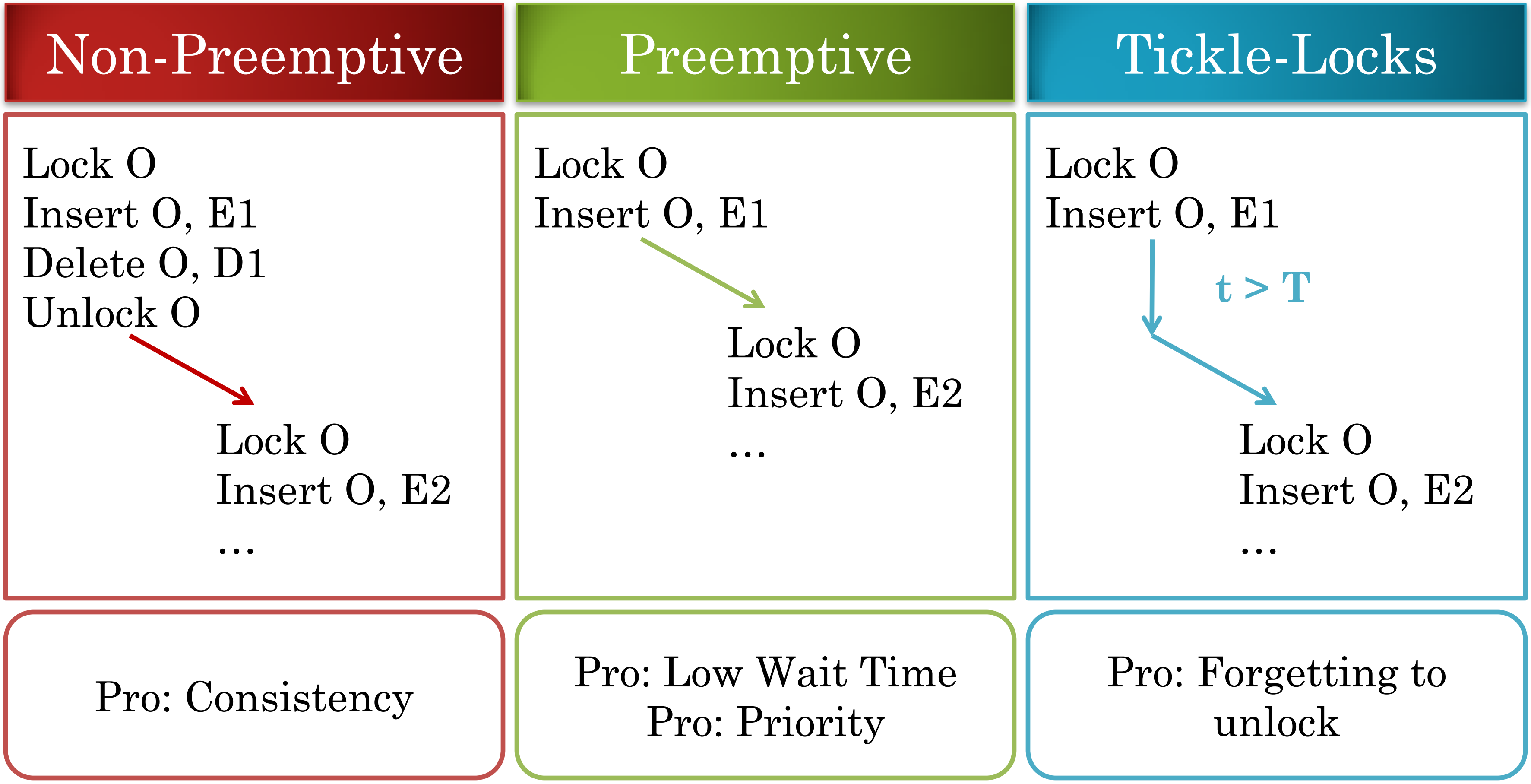
Implicit Unlocking

Tickle locks	Timeout→ Unlock Object
Preemptive locks	Lock Object→ Unlock Object + Lock Object
Tickle + Preemptive	Timeout + Lock Object→ Unlock Object + Lock Object

Unlocked object may not be consistent!

Unlocking user may be able to restore consistency of another user to essentially do a joint (nested) transaction

Consistency vs Concurrency



Non-Preemptive

Lock O
Insert O, E1
Delete O, D1
Unlock O

Lock O
Insert O, E2
…

Pro: Consistency

Preemptive

Lock O
Insert O, E1

Lock O
Insert O, E2
…

Pro: Low Wait Time
Pro: Priority

Tickle-Locks

Lock O
Insert O, E1

t > T

Lock O
Insert O, E2
…

Pro: Forgetting to unlock

Issues

Lock Denial Semantics? ✓

User Interface for Locking and Unlocking? ✓

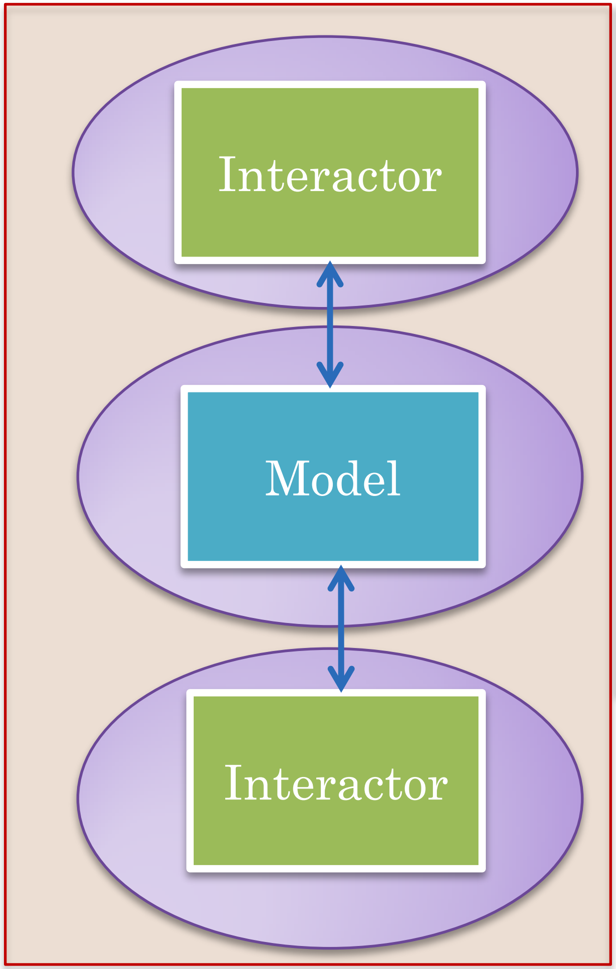
Implementation of locking in a collaborative environment?

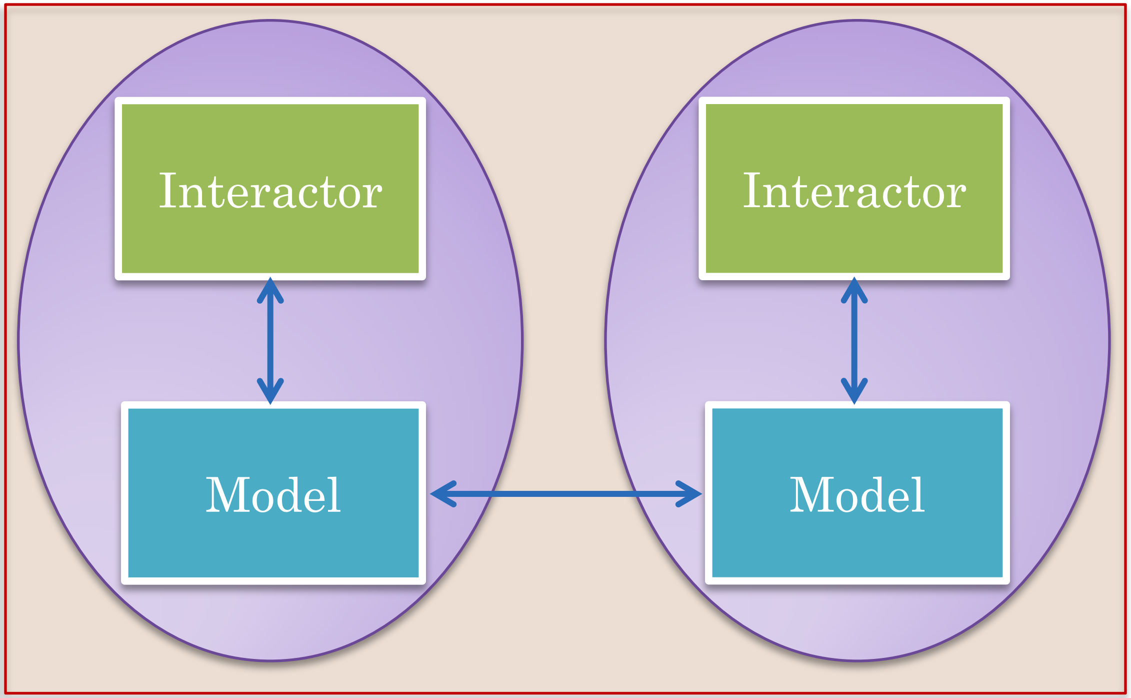
Implementation

Need to share a locking model among multiple users

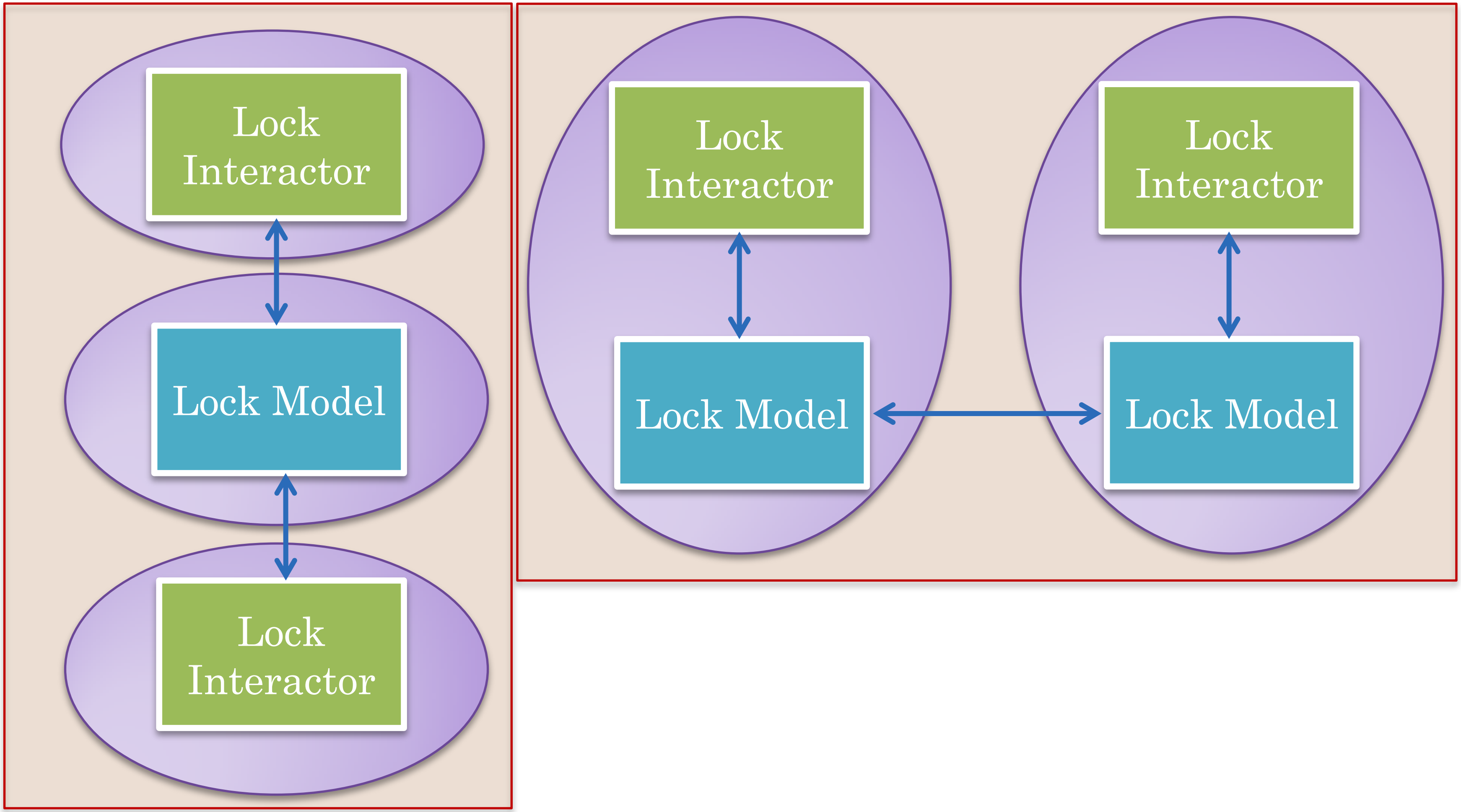
Already know how to share an object

Replicated vs Centralized

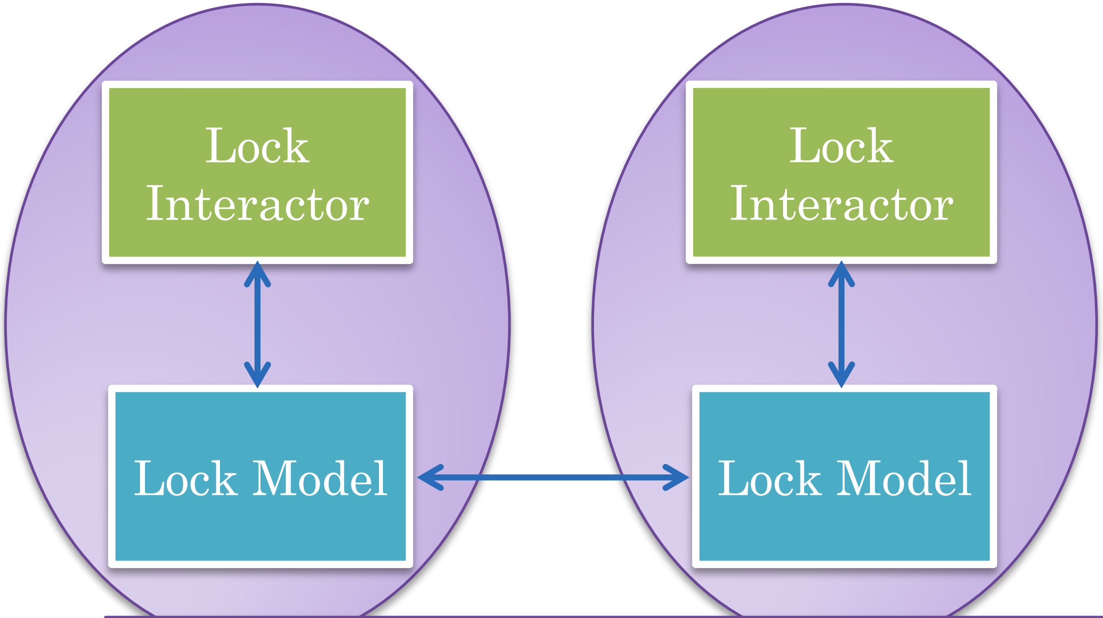




Replicated vs Centralized



Replicated Model: Issues



Who solves the consistency problems of the consistency enforcer!

Consistency issues of causality and concurrent operations (to be addressed later)

Correctness and performance issues when model is non deterministic, accesses central resources, and has side effects

Distributed Consensus Problem

A set of processes have to agree on a common value (Byzantine generals)

There may be failures in machines and communication

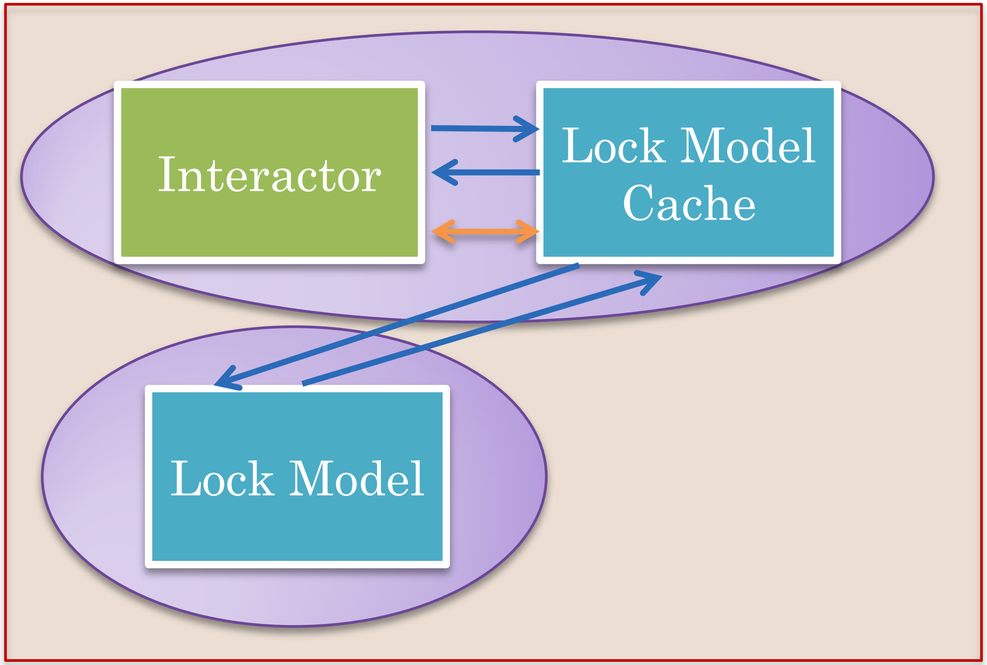
Some processes may be malicious

2 Phase Commit : Coordinator takes vote in first phase and reports majority outcome in second

Not to be confused with 2 Phase Locking (later)

Will simply use the centralized cache solutions assuming no faults

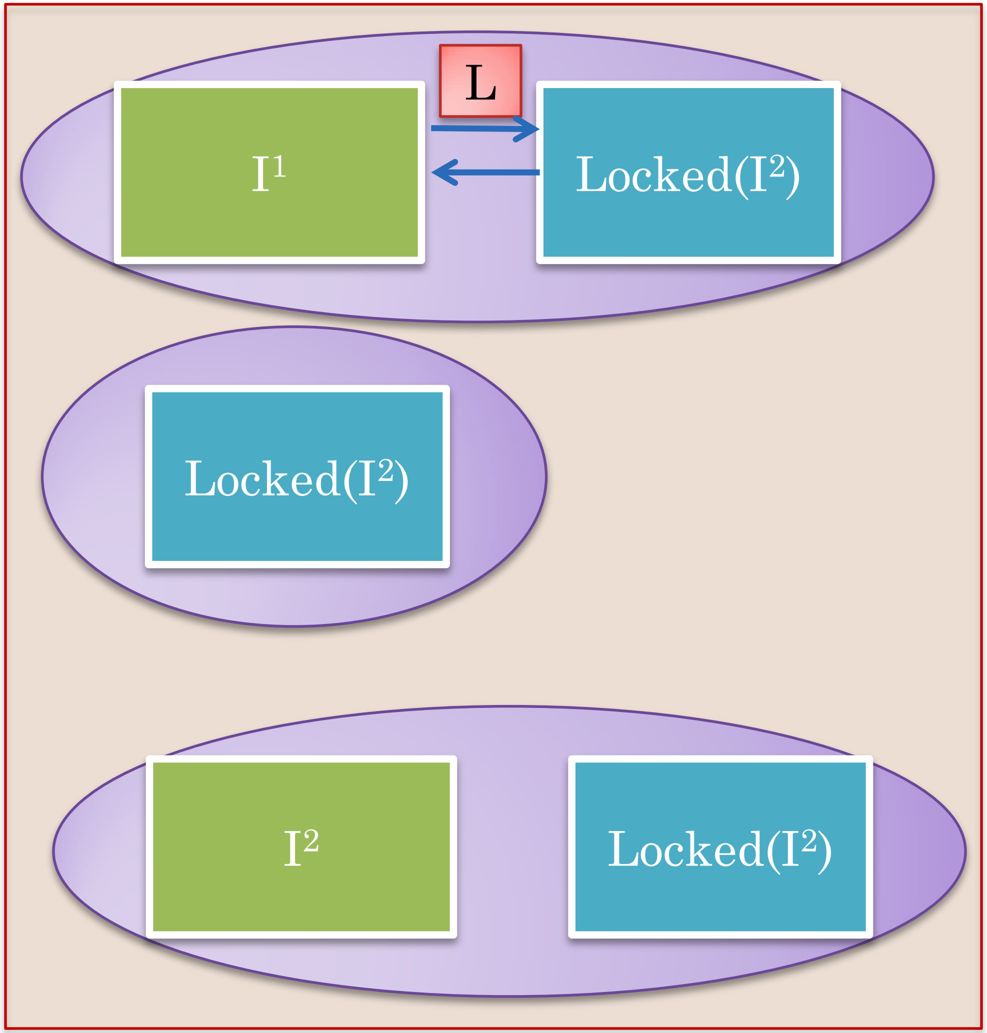
Distribution Unaware Interactor With Model Cache/Proxy



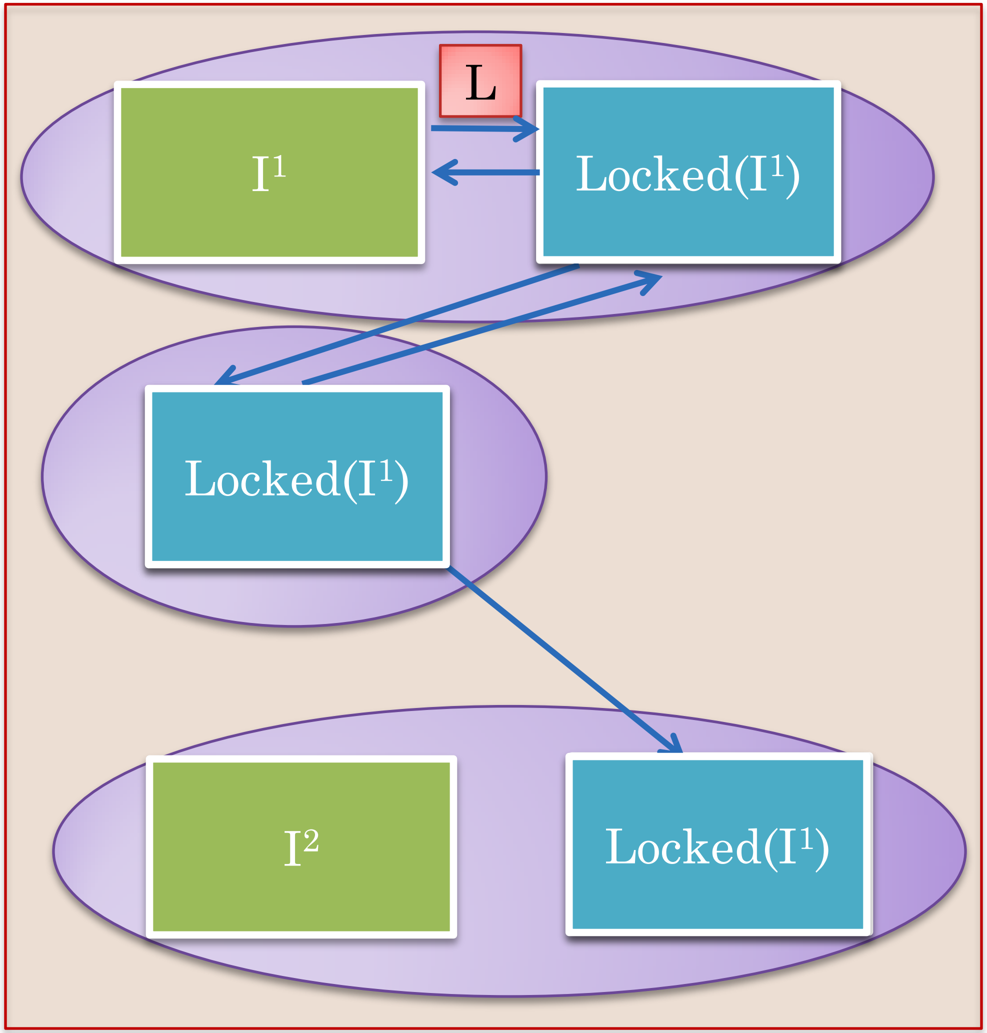
Model cache is a proxy that forwards write (lock, release) operation without changing its data

Read operations (checking lock) access cached data

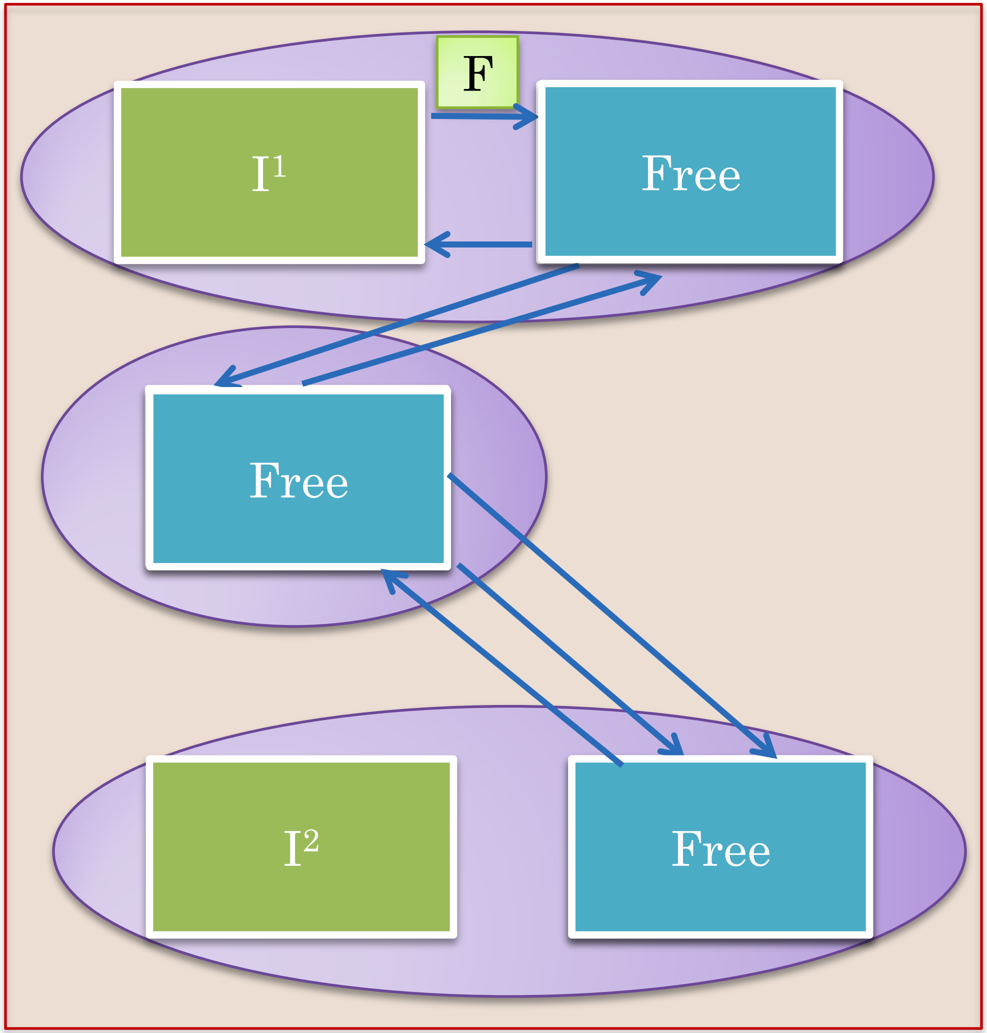
Request for Locked Resource



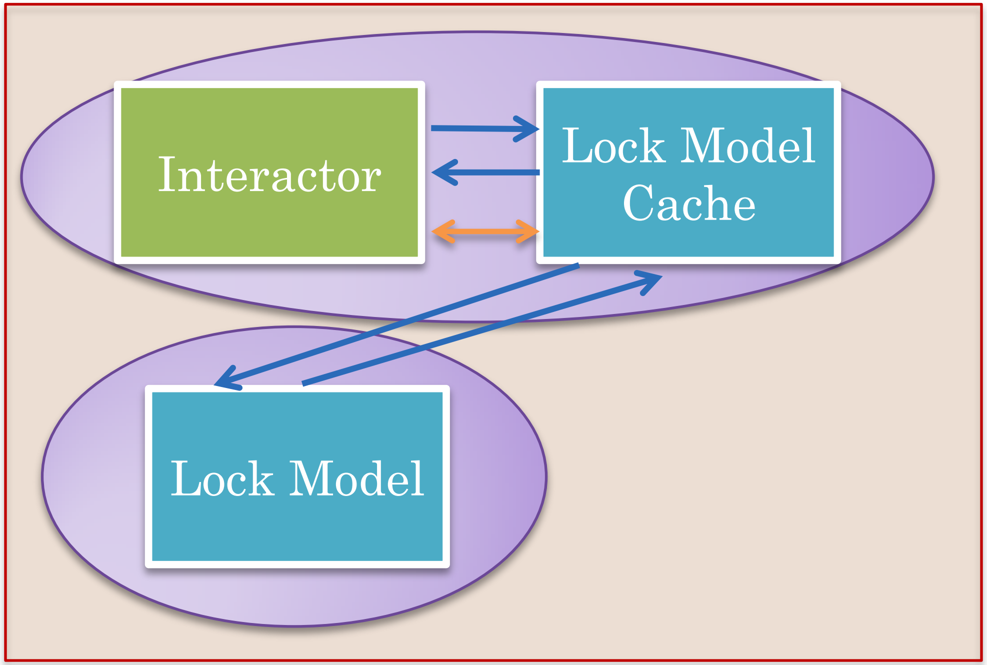
Request for Unlocked Resource



Free Request for Locked Resource



Distribution Unaware Interactor With Model Cache/Proxy



Model cache is a proxy that forwards write (lock, release) operation without changing its data

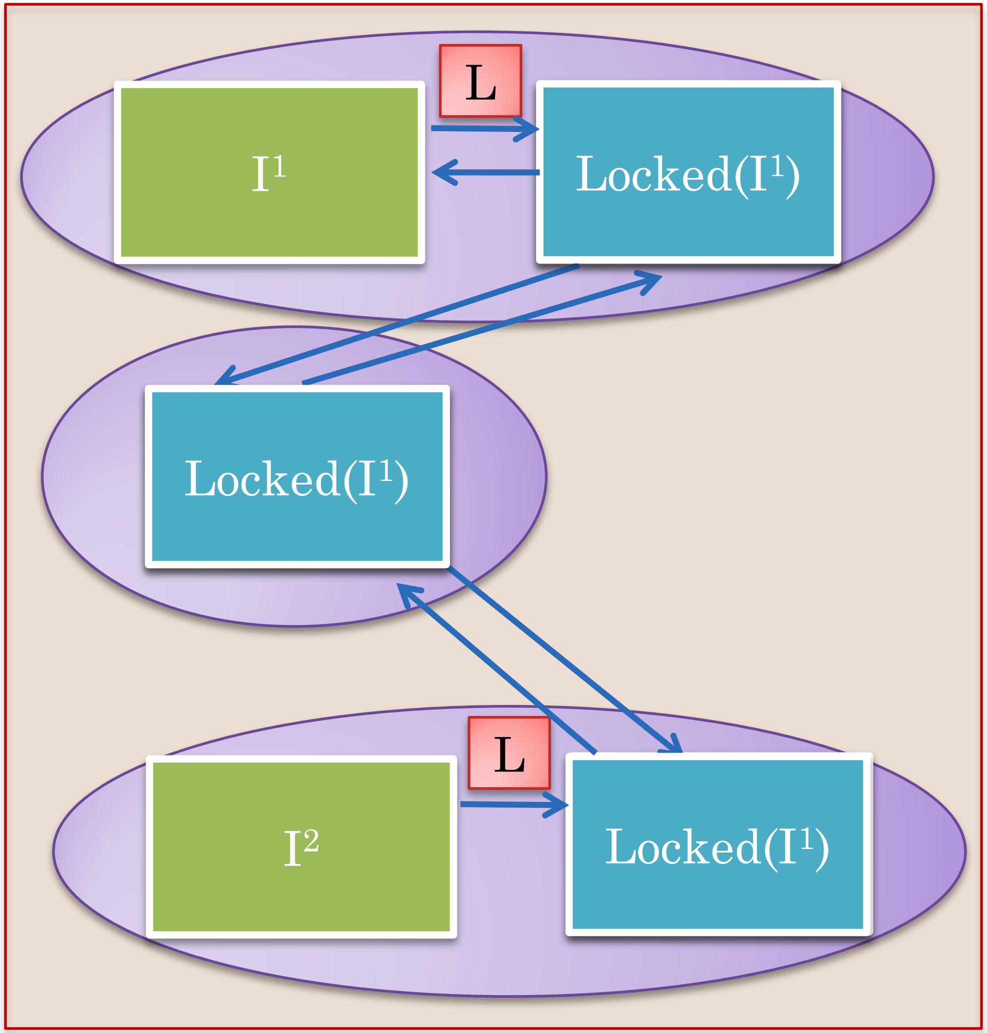
Read operations (checking lock) access cached data

Works?

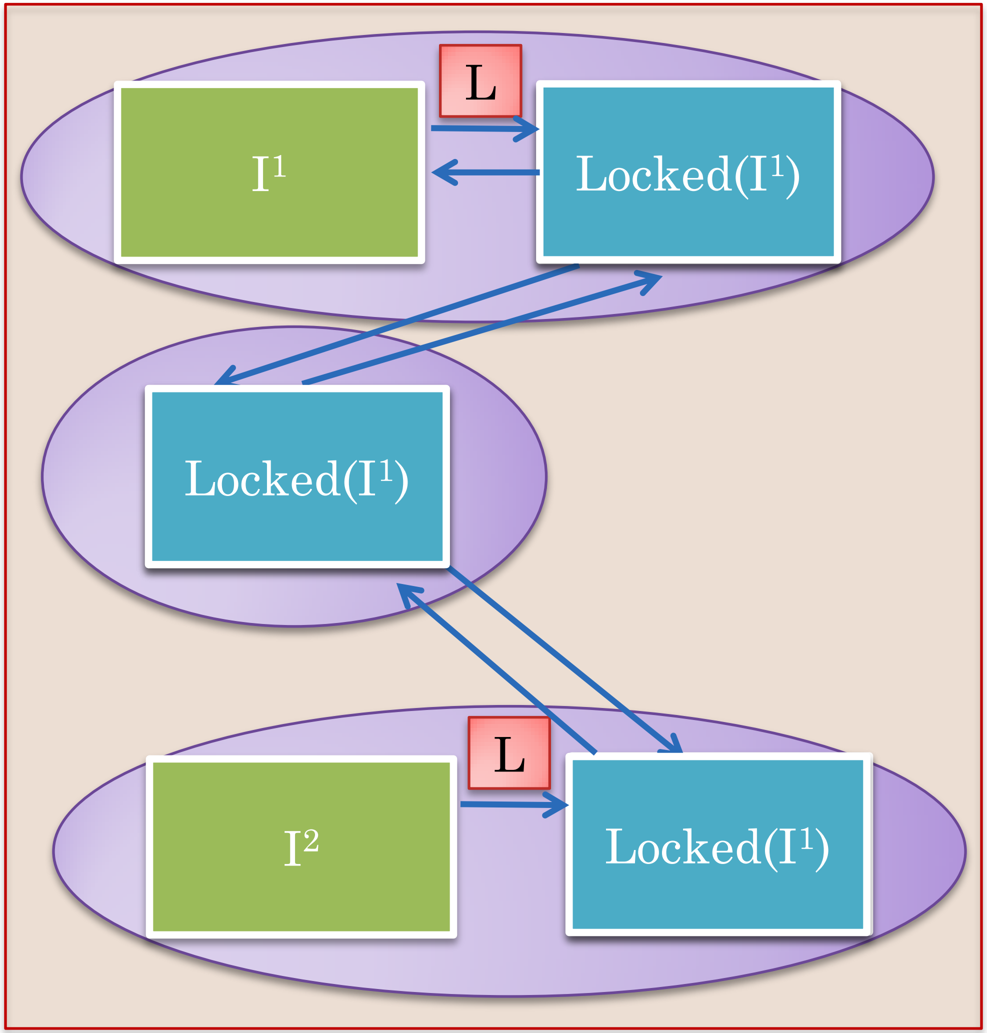
What if a message takes a long time to reach its destination?

Acquire (L) and Release(F) Messages

Concurrent Lock Request: Message to Second Locker Delayed

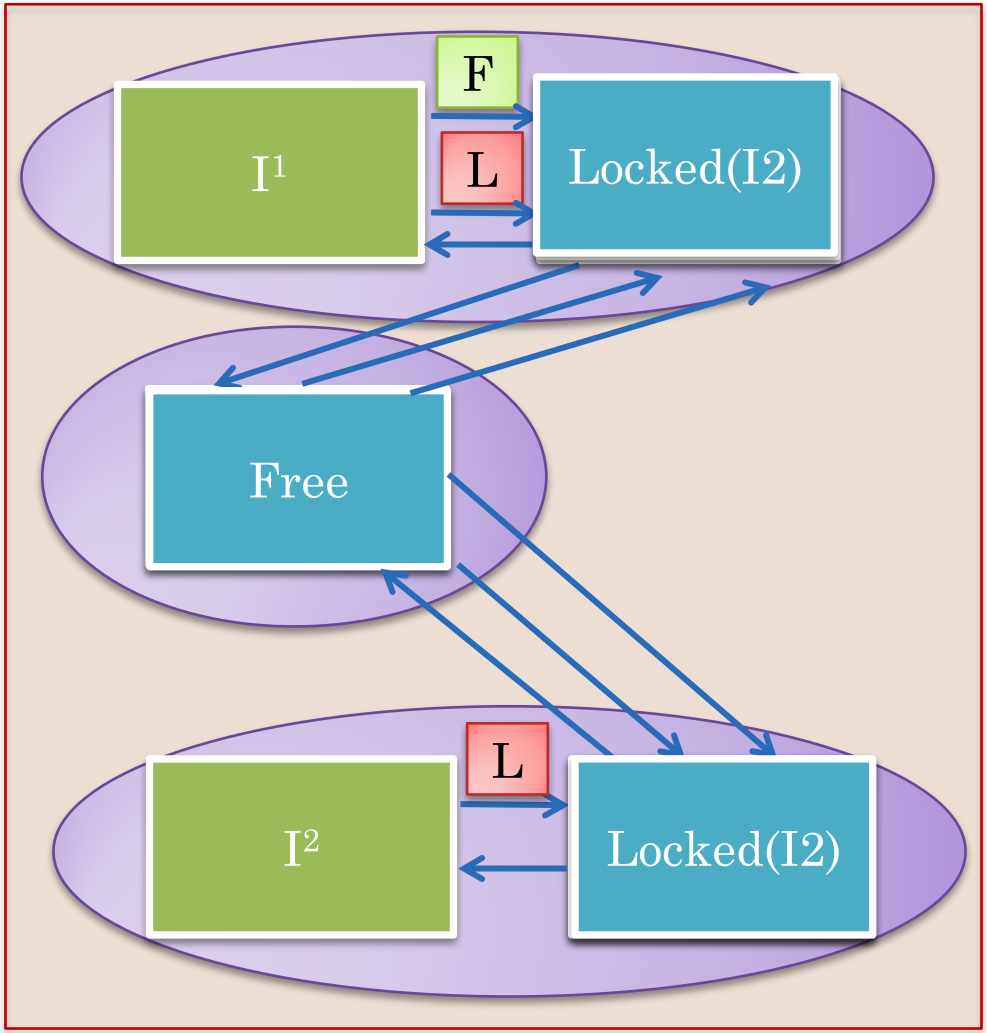


Concurrent Lock Request : Message to First Locker Delayed



At most one cache will make transition from free to locked

Concurrent Free/Lock Request

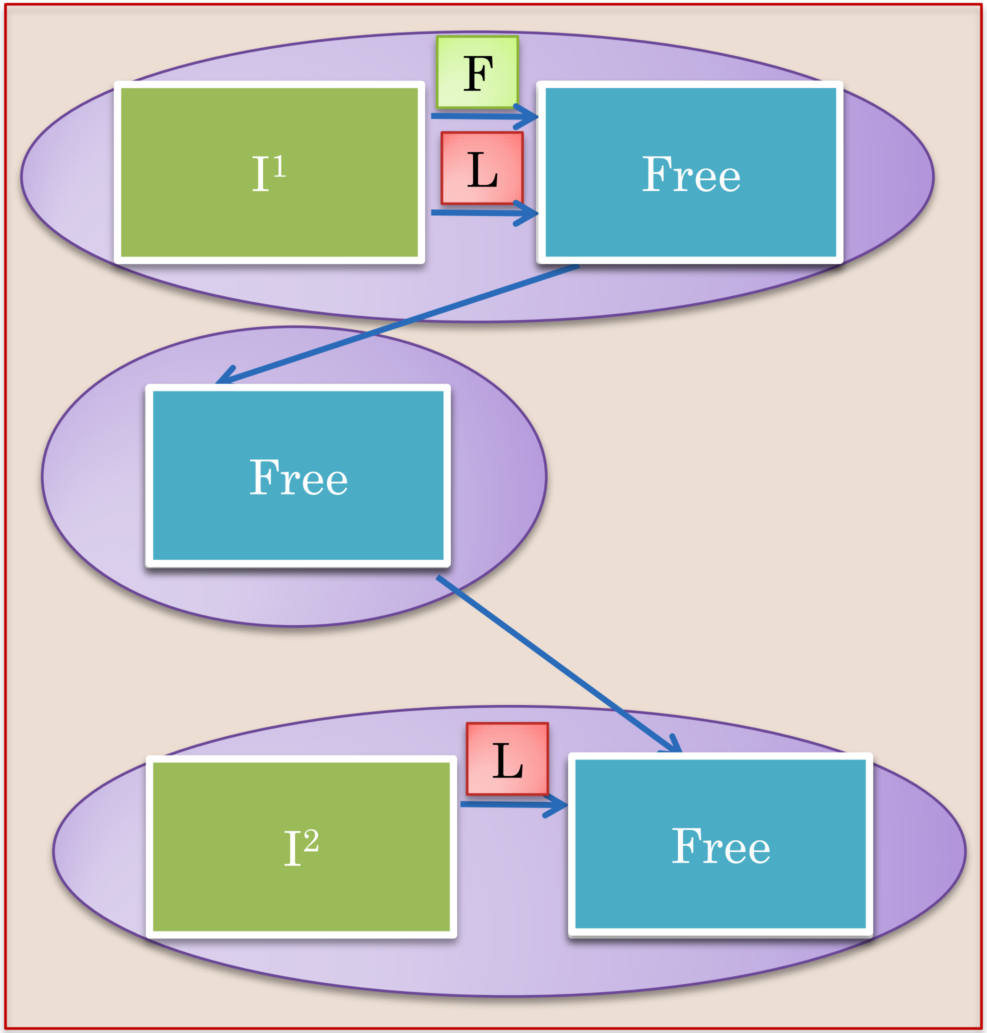


Caches are not consistent!

Conservative: Local cache needs to be invalidated after each write

Using application semantics for more concurrency?

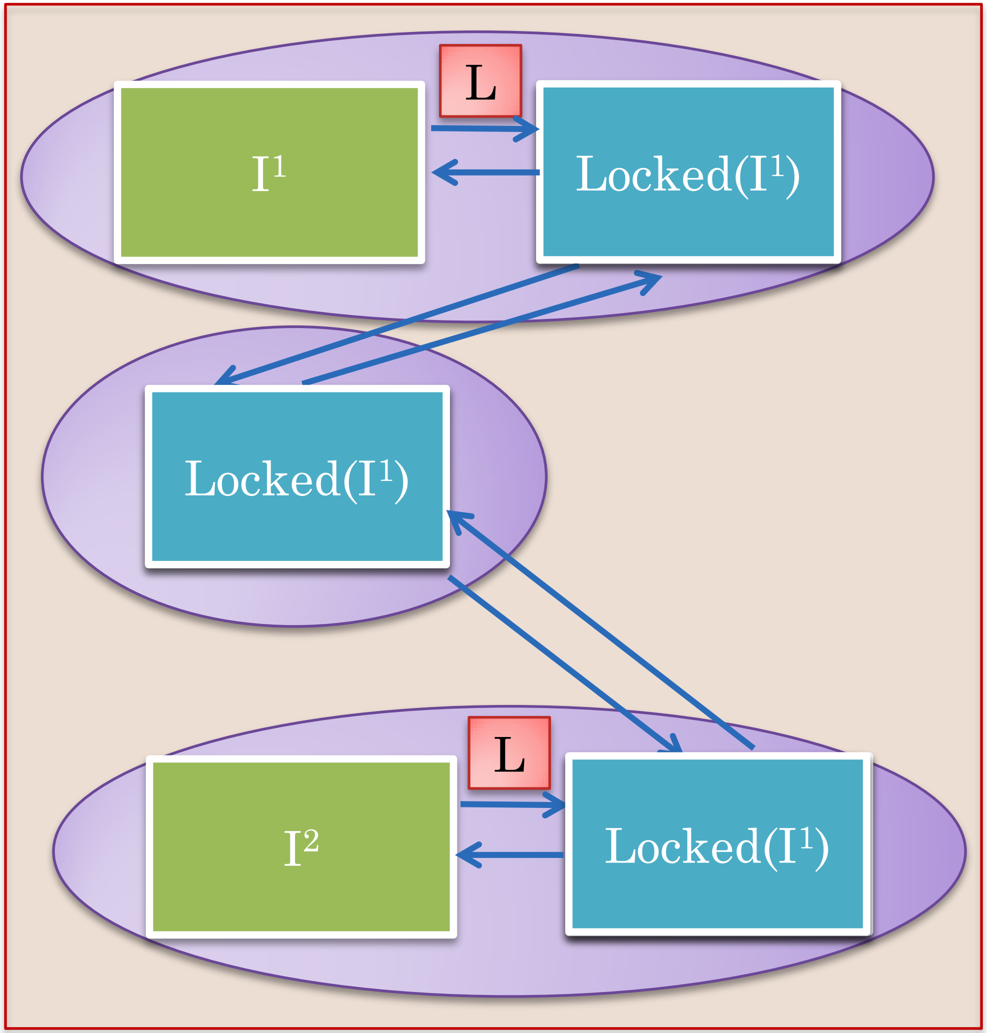
Immediate Freeing (Application Semantics)



Model cache is a proxy that forwards write (lock, release) operation without changing its data ✗

Release requests cause immediate freeing

IMMEDIATE LOCKING?



Weak/eventual consistency: pay the price

Optimistic locks: undo changes if lock request denied

May have received changes from others, must undo non last changes or block them

Others may have seen changes – must do distributed undo if changes sent

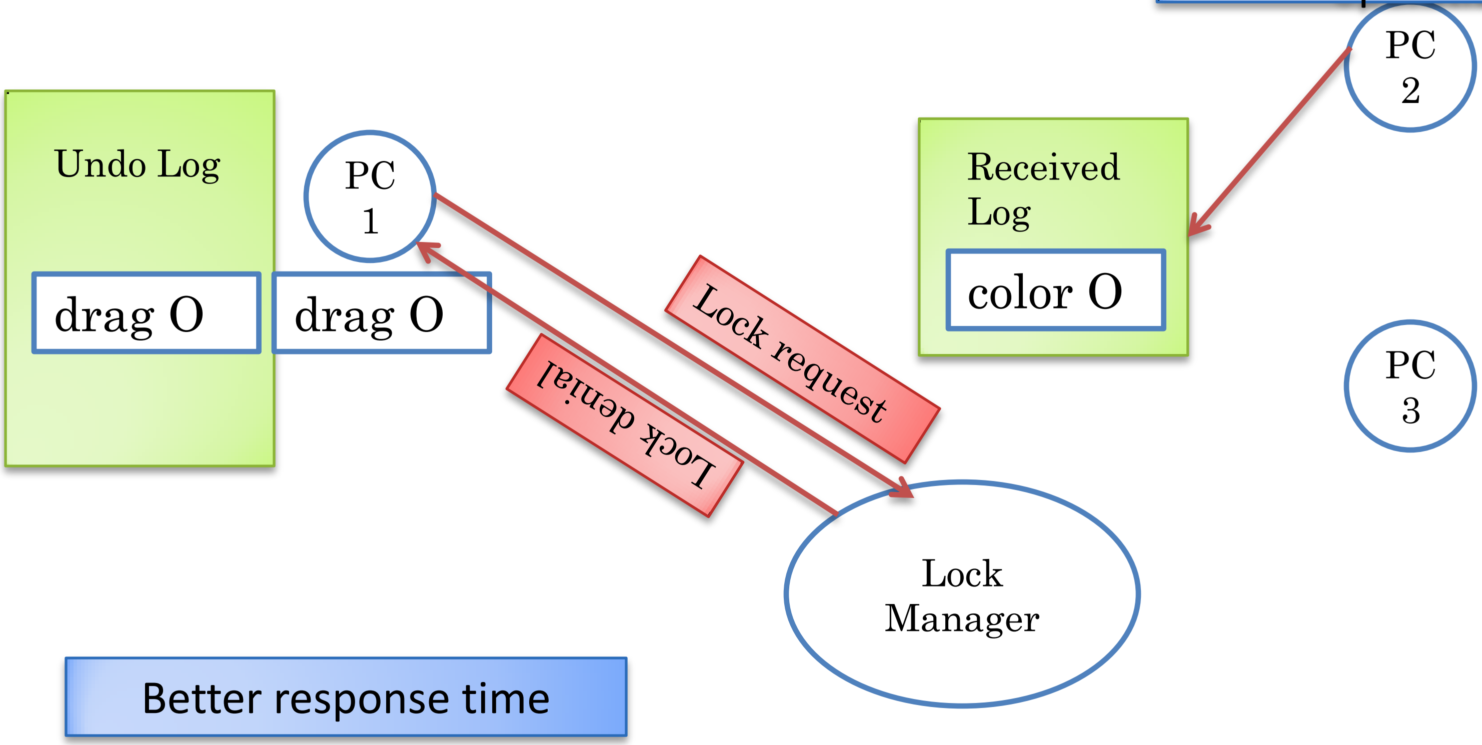
Optimistic Locking

1. Perform operation o and put it in undo log

2. Send permission to perform operation and defer performing received operations

3. Undo if lock request fails, and perform deferred received actions

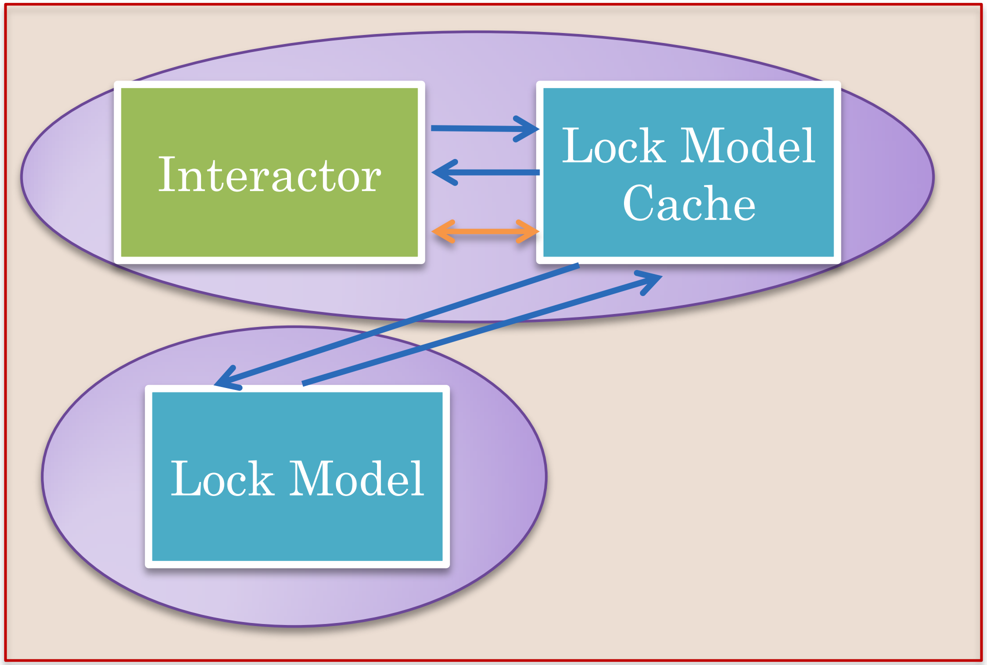
4. Othewrise, toOthers() send operation and perform deferred receive operations



Better response time

e

Distribution Unaware Interactor With Model Cache/Proxy

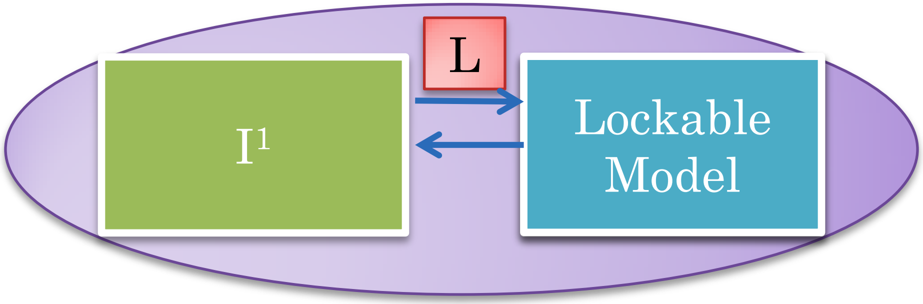


Model cache is a proxy that forwards lock operation without changing its data and forwards release request after changing its data

Read operations (checking lock) access cached data

Distributed vs software architecture

Single-User Pattern

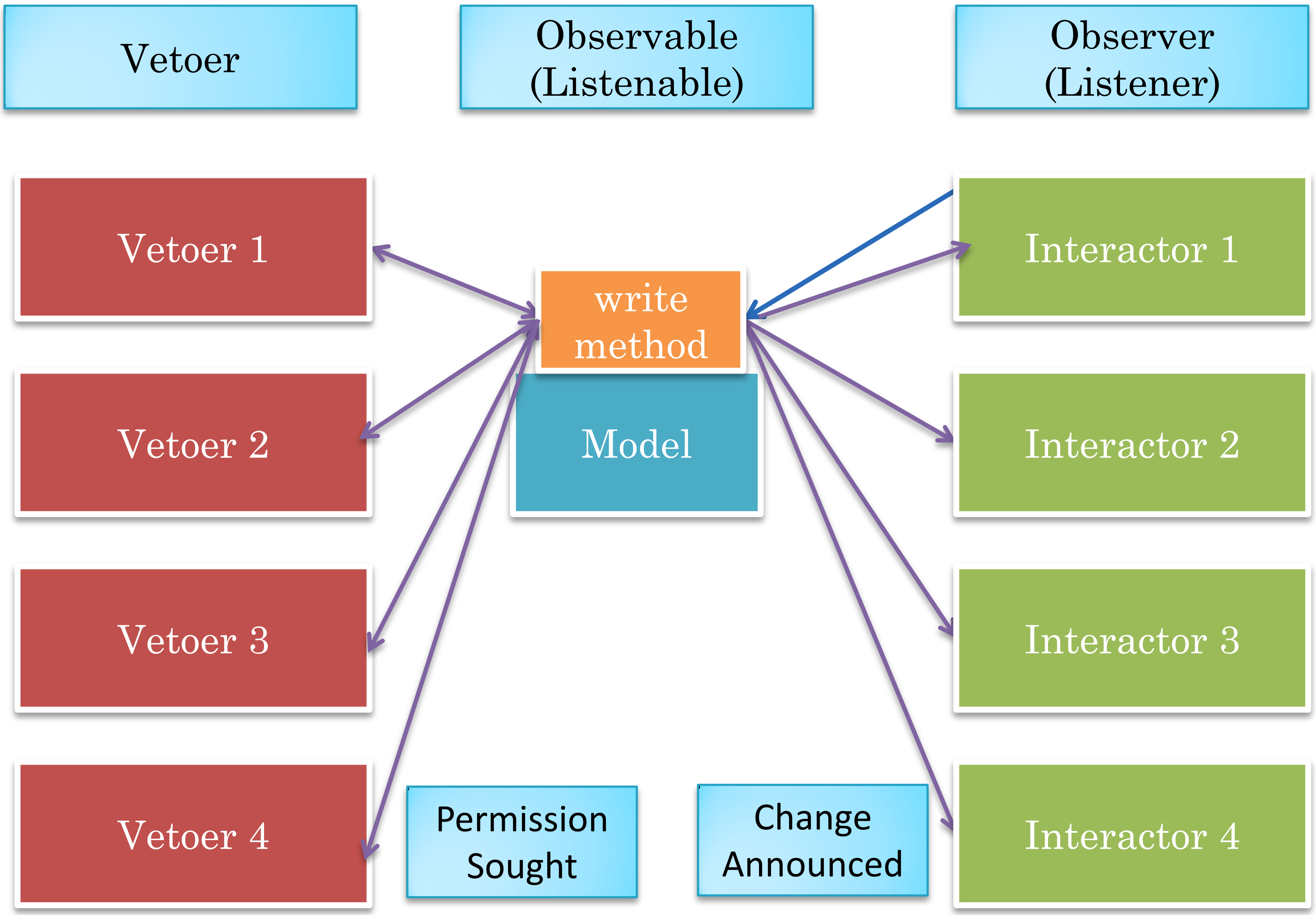


Put locking semantics in model?

May have more than one kind of concurrency controller (optimistic, pessimistic)

May have more than one controller (access, concurrency)

Vetoers vs Observers



Vetoer

Observable (Listenable)

Observer (Listener)

Vetoer 1

Vetoer 2

Vetoer 3

Vetoer 4

write method

Model

Interactor 1

Interactor 2

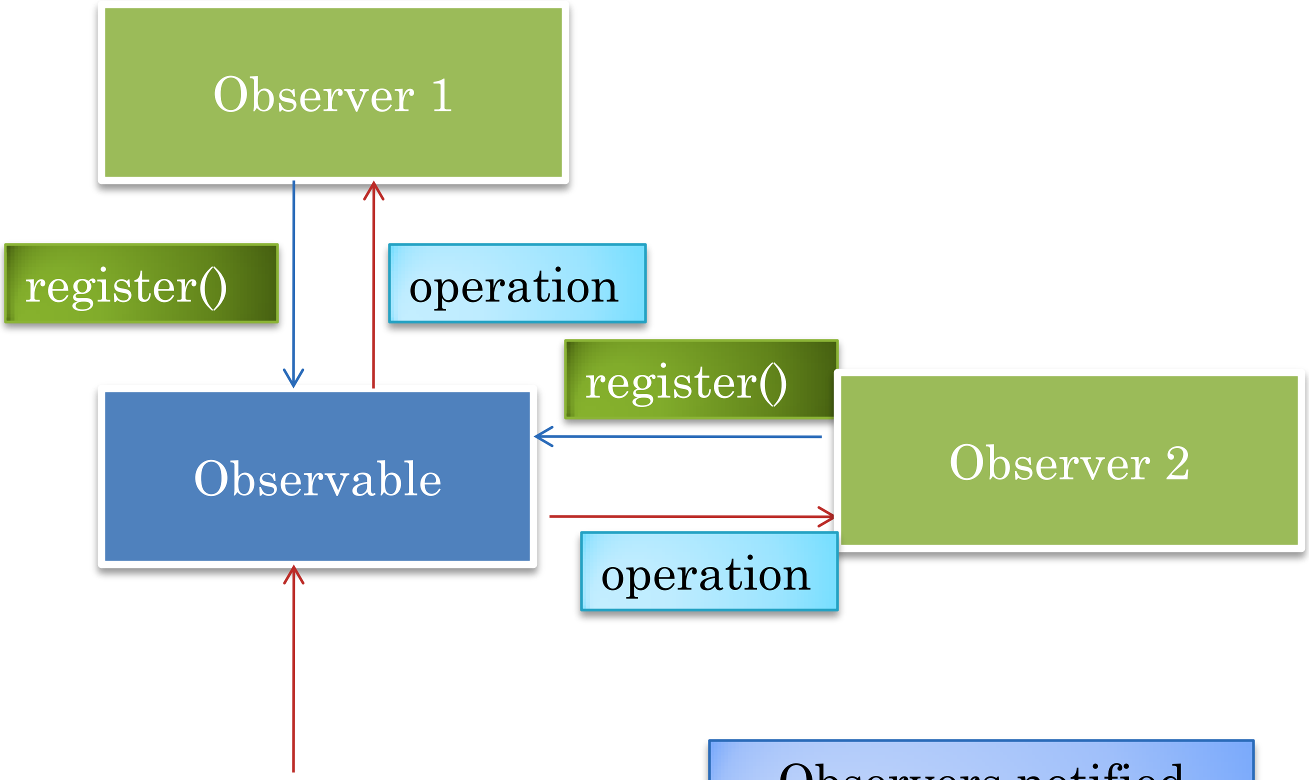
Interactor 3

Interactor 4

Permission Sought

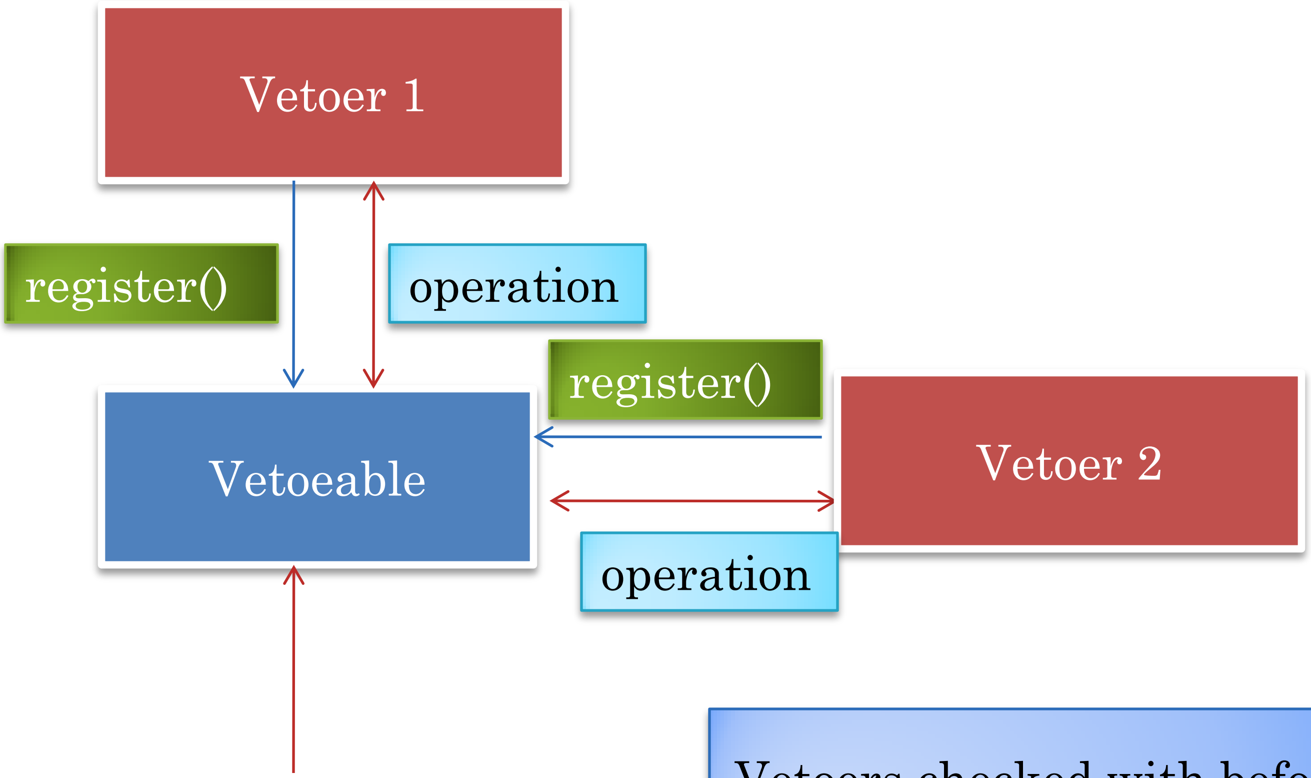
Change Announced

Observer vs. Vetoer



Observers notified after event processing done

OBSERVER VS. VETOER



Vetoers checked with before event processing done

Feedback, so notifier must wait in distributed implementation

Vetoers

- Like an observer, a vetoer can be registered with an object
- The object checks with each vetoer before making and announcing change
- If a singe vetoer rejects change, then it is not made or announced
- Java Beans comes with standard Vetoer interface

Vetoers (Review)

- Like an observer, a vetoer can be registered with an object
- The object checks with each vetoer before making and announcing change
- If a singe vetoer rejects change, then it is not made or announced
- Java Beans comes with standard Vetoer interface

Standard Java Vetoer Interface

```
public interface VetoableChangeListener {
   public void vetoableChange(PropertyChangeEvent evt)
       throws PropertyVetoException
}
```

Vetoing is not an exception (error)!

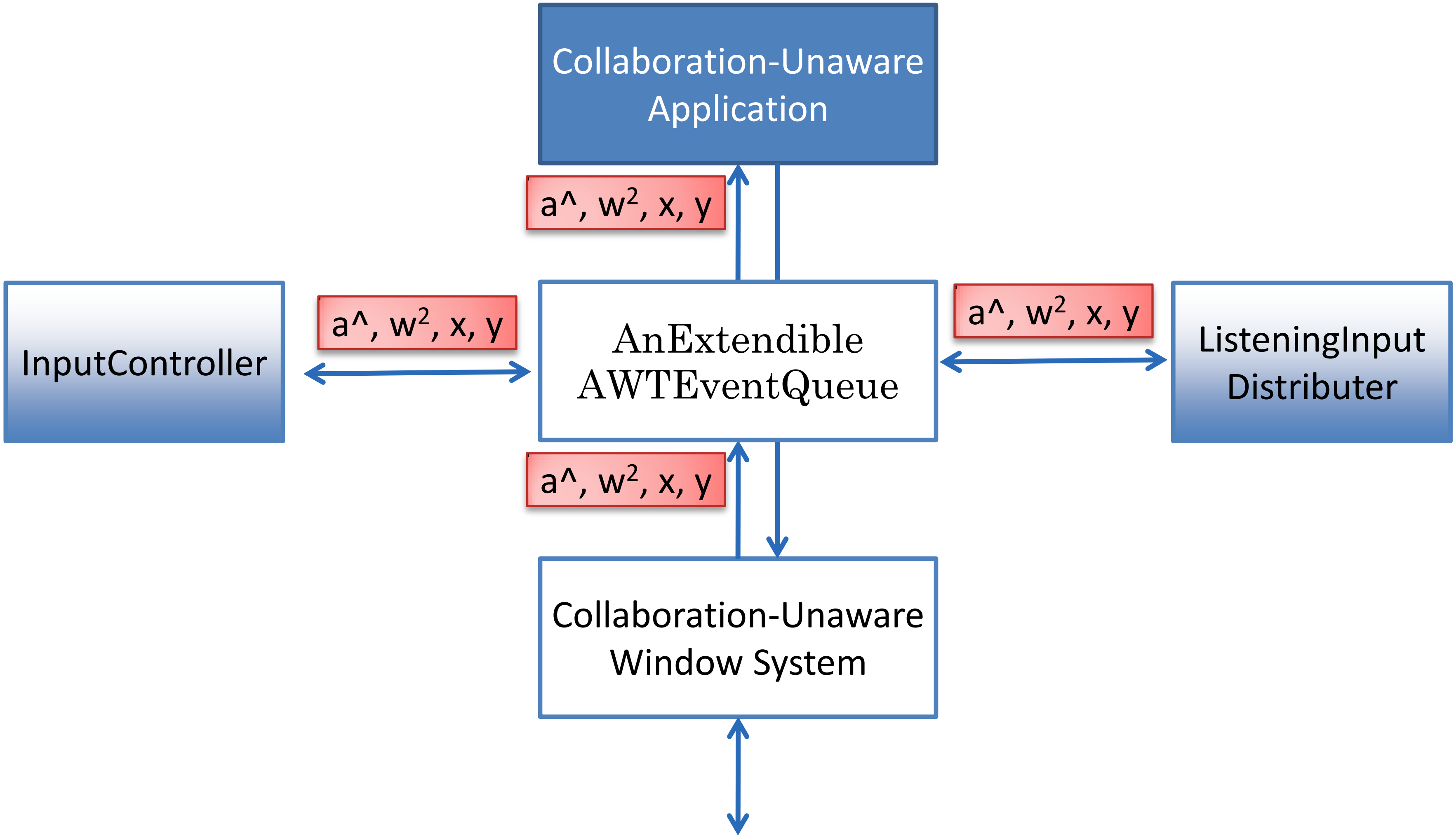
Better to return a Boolean value

Controlled Replicated History

```
public class AControlledReplicatedHistory<ElementType>
     extends AReplicatedSimpleList<ElementType>
     implements ControlledReplicatedHistory<ElementType>  {
  VetoableChangeSupport vetoableChangeSupport =
       new VetoableChangeSupport(this);
  public synchronized void replicatedAdd(ElementType aNewValue) {
    try {
       vetoableChangeSupport.fireVetoableChange(
           "IMHistory", null, aNewValue);
    } catch (PropertyVetoException e) {
       return;
    }
    super.replicatedAdd(aNewVal
  }
  public void addVetoableChangeListener(
             VetoableChangeListener listener) {
    vetoableChangeSupport.addVetoableChangeListener(listener);
  }
…
```

Fitting list add to property change – old value is null, property name could be also

Library Listenable Event Queue



Collaboration-Unaware Application

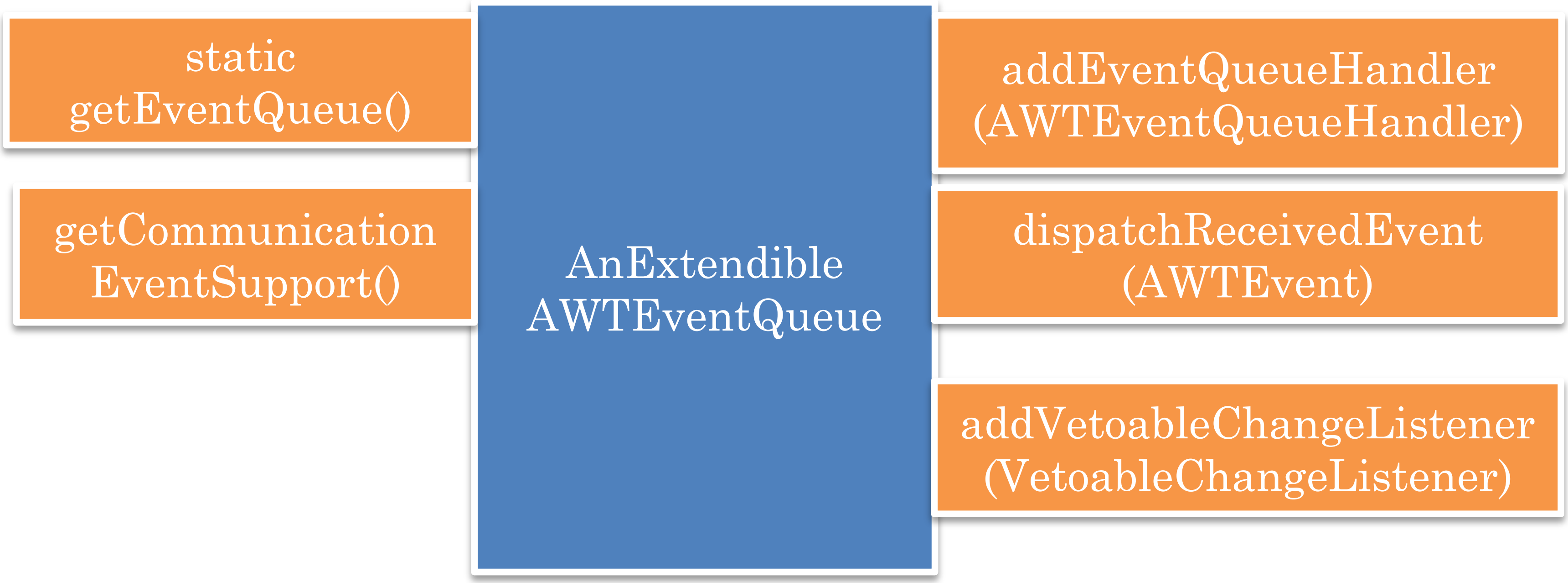
AnExtendible AWTEventQueue

InputController

ListeningInput Distributer

Collaboration-Unaware Window System

How to Intercept, Inject and Veto Window Events



static getEventQueue()

getCommunication EventSupport()

AnExtendible AWTEventQueue

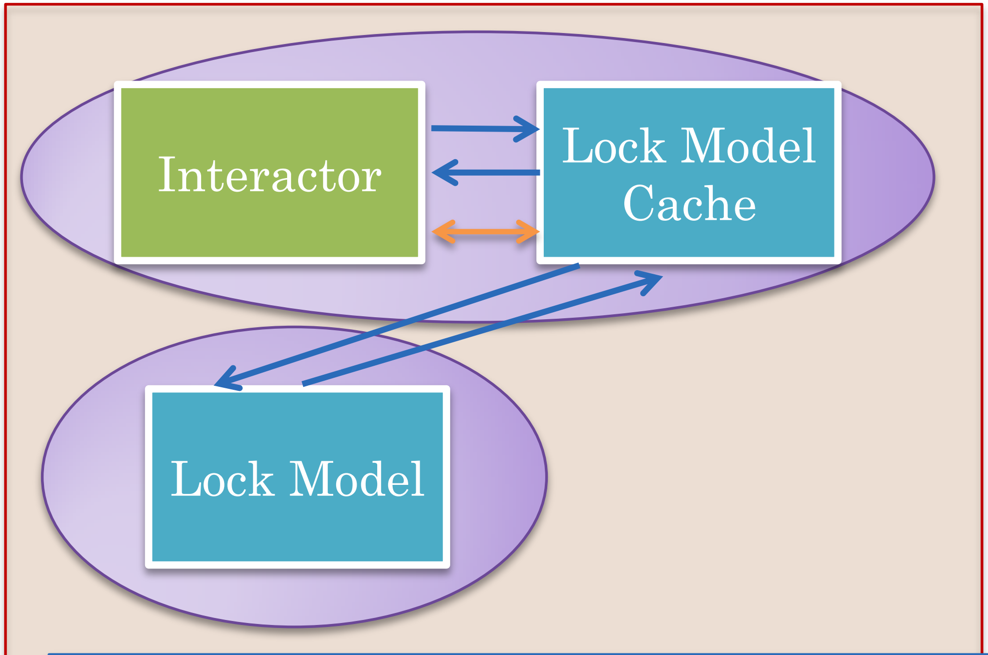
addEventQueueHandler (AWTEventQueueHandler)

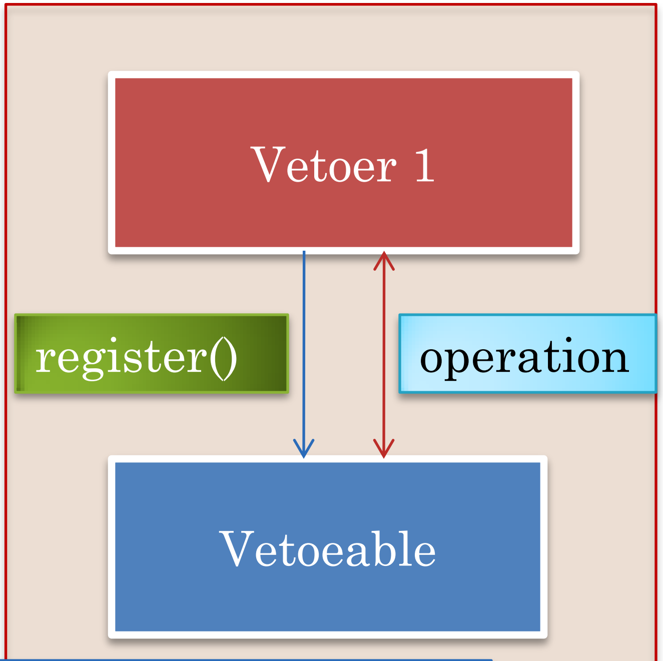
dispatchReceivedEvent (AWTEvent)

addVetoableChangeListener (VetoableChangeListener)

The property value of fired vetoable change is the AWTEvent and the property name is to be ignored

Distributed + Software Architecture





Local Model Cache = Vetoeable Model + Slave Model Vetoer

Lock Model = Master Lock Model

Assume each site has Slave Model Vetoer and one of these sites has Master Lock Model

Three relevant user operations: write, lock, release

Traceable Aware Slave UI Thread

Slave UI Thread (Vetoer)

For each vetoable write received from local user U

If not getLock(U), UserActionDenied

Slave UI Thread (Lock Grantor)

For each SlaveLockGrantRequestMade by local user U

if not locked(U), to all, SlaveLockGrantRequestSent

Slave UI Thread (Lock Releaser)

For each SlaveLockReleaseRequestMade by local user U

If locked(U), setLock(U, false), to all, SlaveLockReleaseRequestSent

Traceable Aware Master Receiving Threads

Master Receiving Thread

For each MasterLockRequestReceived

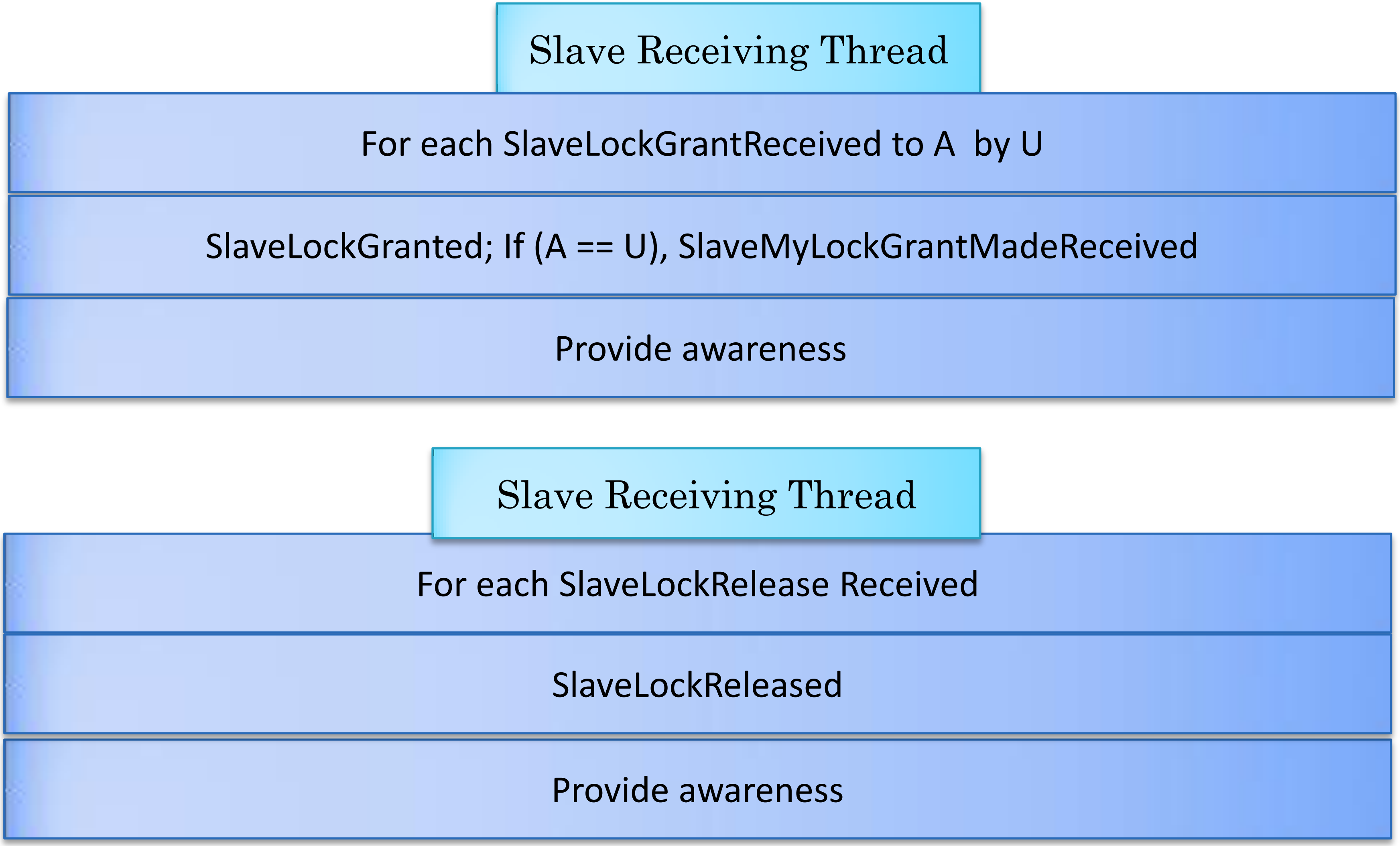
If not isLocked(), MasterLockGranted, to all, MasterLockGrantStatusSent

Master Receiving Thread

For each MasterLockReleaseRequestReceived from user U

If getLock(U), MasterLockReleased, to all, MasterLockReleaseStatusSent

Traceable Aware Slave Receiving Threads



Slave Receiving Thread

For each SlaveLockGrantReceived to A by U

SlaveLockGranted; If (A == U), SlaveMyLockGrantMadeReceived

Provide awareness

Slave Receiving Thread

For each SlaveLockRelease Received

SlaveLockReleased

Provide awareness

Summary

Concurrency Control and Transactions ✓

Simple Locking – One Lock – and its distributed implementation ✓

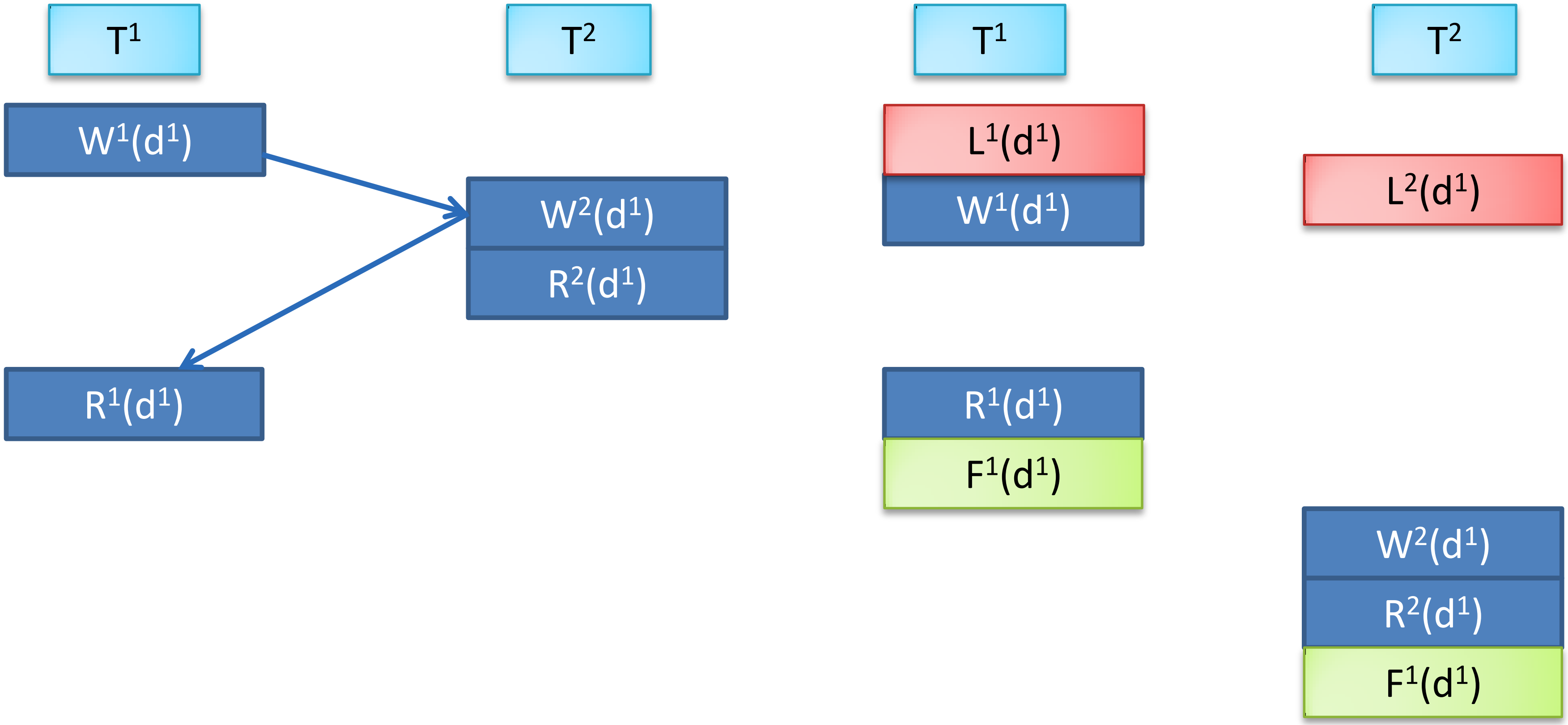
Multiple Locks

Multiple (Programmer-Defined) Lock Types

Alternatives to Locking

Nested transactions

Locking: One Item (Review)



T^1

$W^1(d^1)$

$R^1(d^1)$

T^2

$W^2(d^1)$

$R^2(d^1)$

T^1

$L^1(d^1)$

$W^1(d^1)$

$R^1(d^1)$

$F^1(d^1)$

T^2

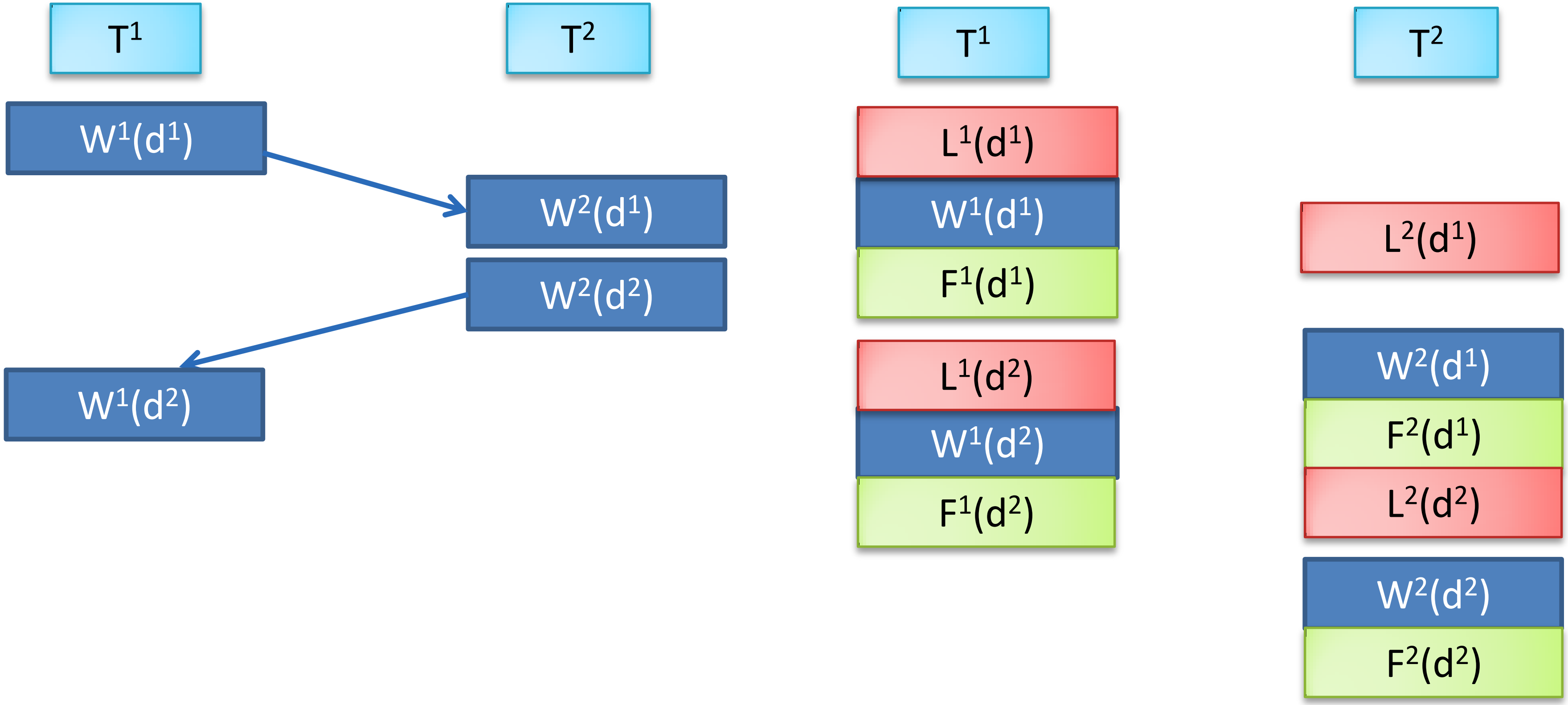
$L^2(d^1)$

$W^2(d^1)$

$R^2(d^1)$

$F^1(d^1)$

Locking Multiple Items in Same Order



T^1

$W^1(d^1)$

$W^1(d^2)$

T^2

$W^2(d^1)$

$W^2(d^2)$

T^1

$L^1(d^1)$

$W^1(d^1)$

$F^1(d^1)$

$L^1(d^2)$

$W^1(d^2)$

$F^1(d^2)$

T^2

$L^2(d^1)$

$W^2(d^1)$

$F^2(d^1)$

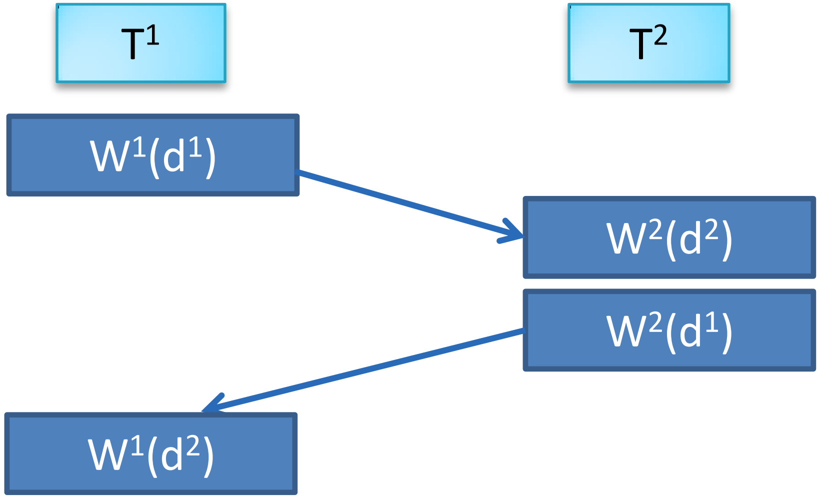
$L^2(d^2)$

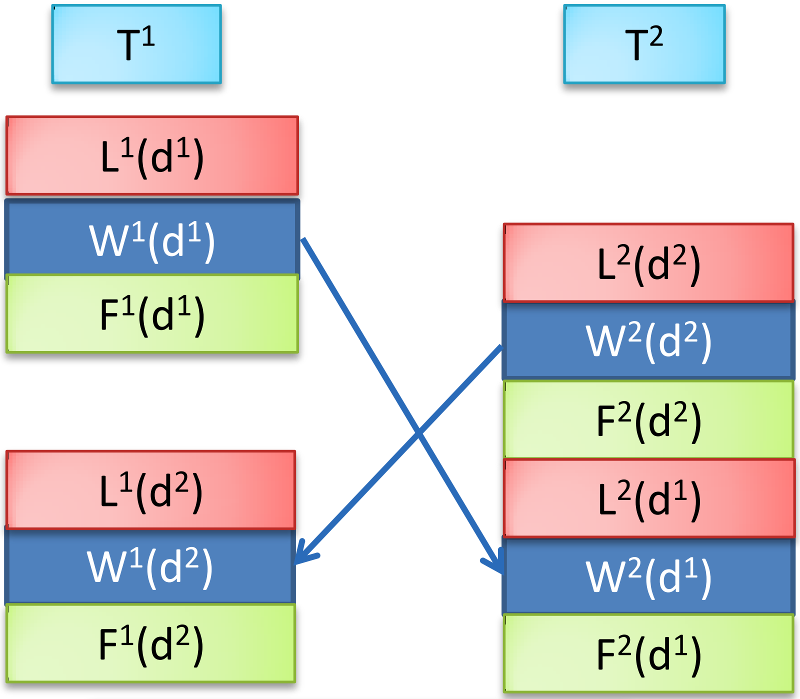
$W^2(d^2)$

$F^2(d^2)$

T^2 performs an operation on each object after T^1

Locking Multiple Items in Different Order





Two phase locking

Locks were freed too quickly!

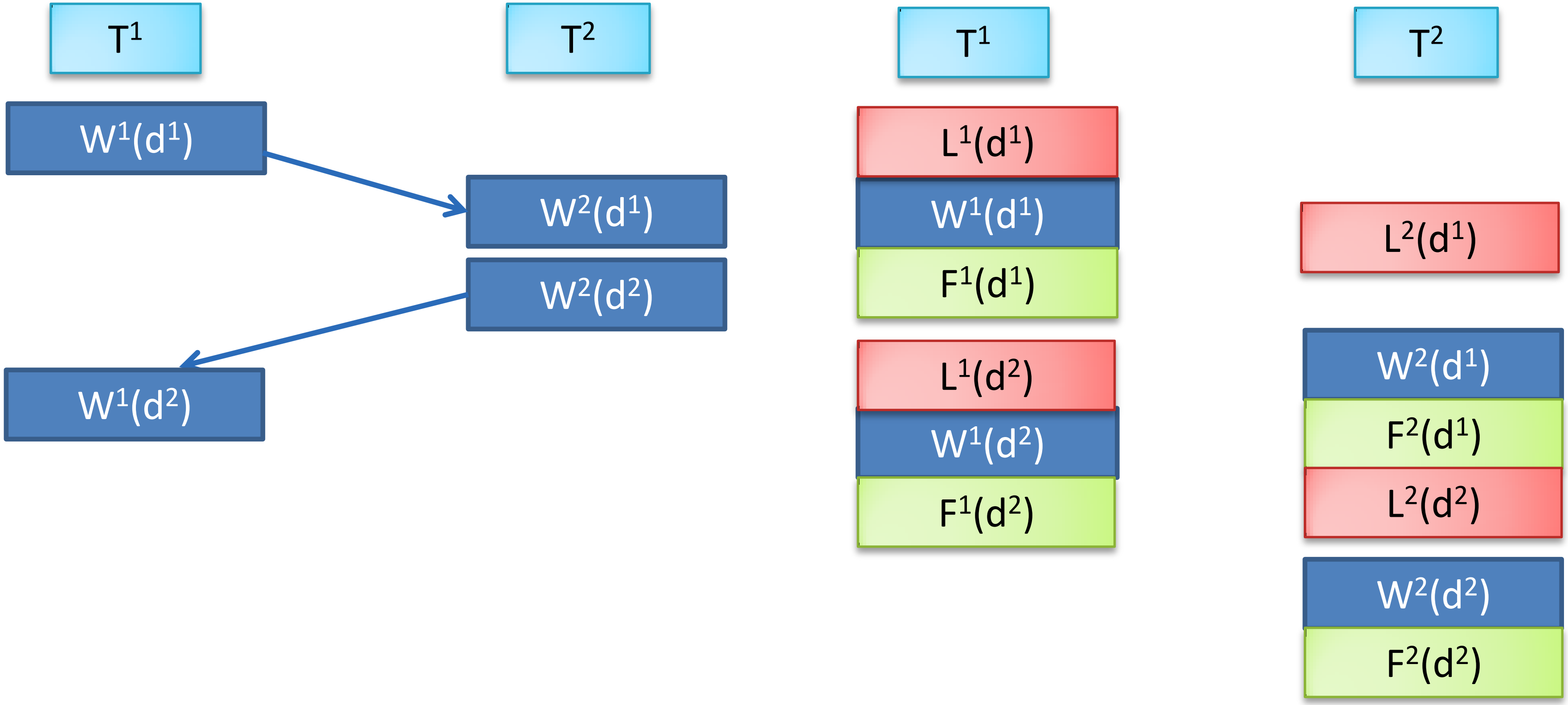
Get all locks before doing any operation?

Early binding and keeps locks for longer than necessary

A transaction has a growing phase when locks are added and not released

Then it has a shrinking phase when locks are released but not freed

Non Two Phase in Same Order



T^1

$W^1(d^1)$

$W^1(d^2)$

T^2

$W^2(d^1)$

$W^2(d^2)$

T^1

$L^1(d^1)$

$W^1(d^1)$

$F^1(d^1)$

$L^1(d^2)$

$W^1(d^2)$

$F^1(d^2)$

T^2

$L^2(d^1)$

$W^2(d^1)$

$F^2(d^1)$

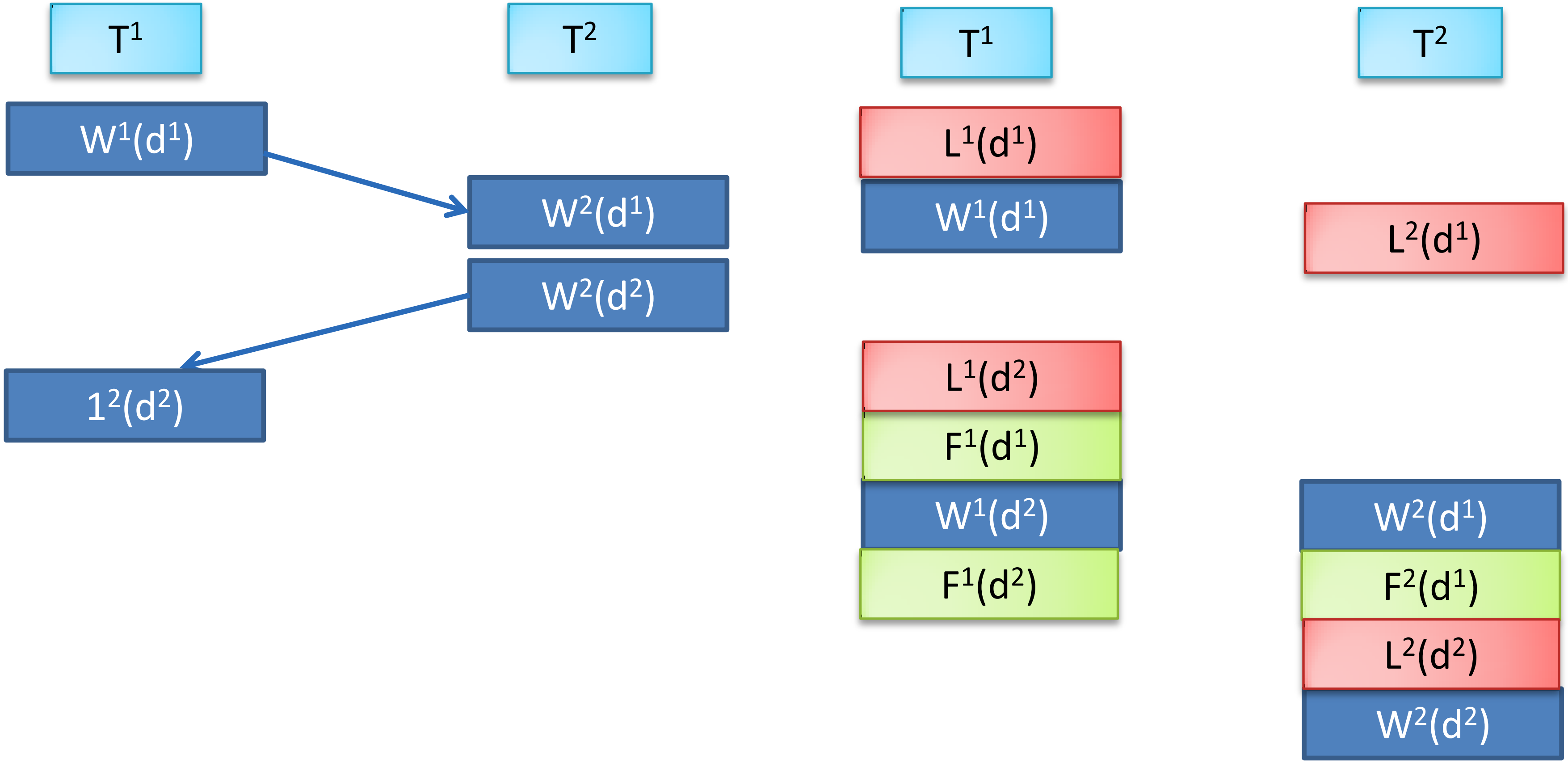
$L^2(d^2)$

$W^2(d^2)$

$F^2(d^2)$

Locks shrink and then grow

Two Phase Locking in Same Order



T^1

T^2

$W^1(d^1)$

$W^2(d^1)$

$W^2(d^2)$

$1^2(d^2)$

T^1

T^2

$L^1(d^1)$

$W^1(d^1)$

$L^2(d^1)$

$L^1(d^2)$

$F^1(d^1)$

$W^1(d^2)$

$F^1(d^2)$

$W^2(d^1)$

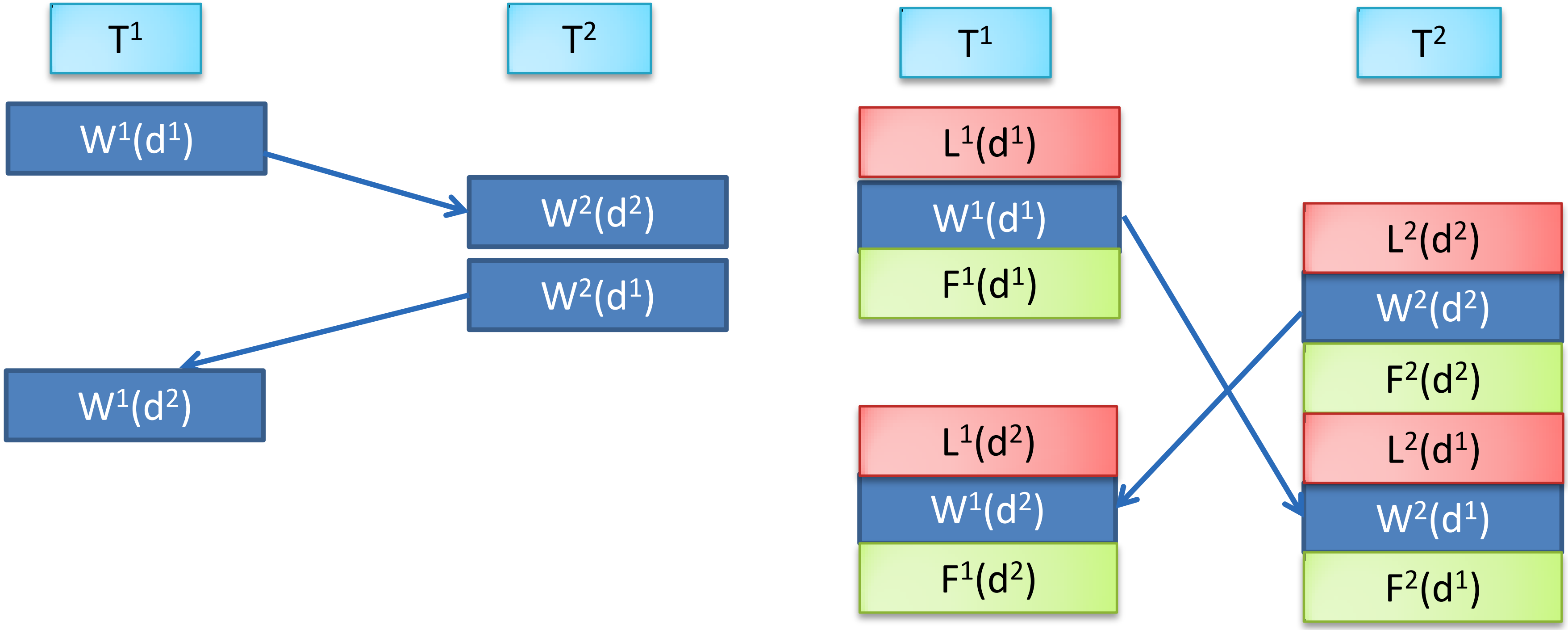
$F^2(d^1)$

$L^2(d^2)$

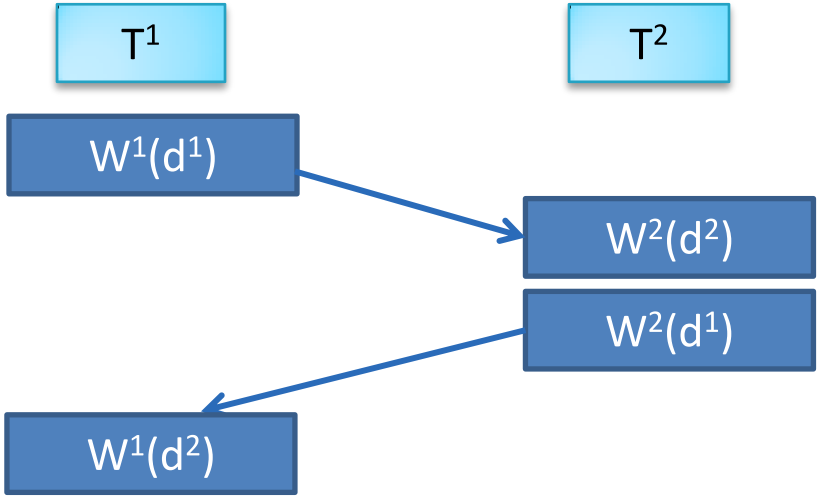
$W^2(d^2)$

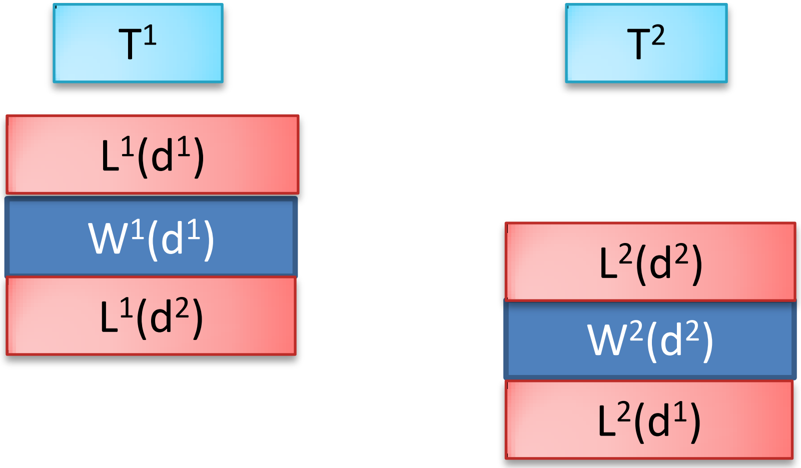
D^1 freed after all locks gathered but before end of transaction

Non Two Phase Different Order



Two-Phase Locking Different Order





Non serializable schedules lead to deadlocks

Need deadlock detection schemes

Two phase locking

A transaction has a growing phase when locks are added and not released

Then it has a shrinking phase when locks are released but not freed

Proof that 2PL → Serializability

Transaction graph: T^1 has edge to T^2 if T^2 performs some (non commuting) operation after some operation performed by T^1

Non-serializable == Cycles in transaction graph

Cycles in transaction graph under 2PL will lead to deadlocks

Proof by Contradiction

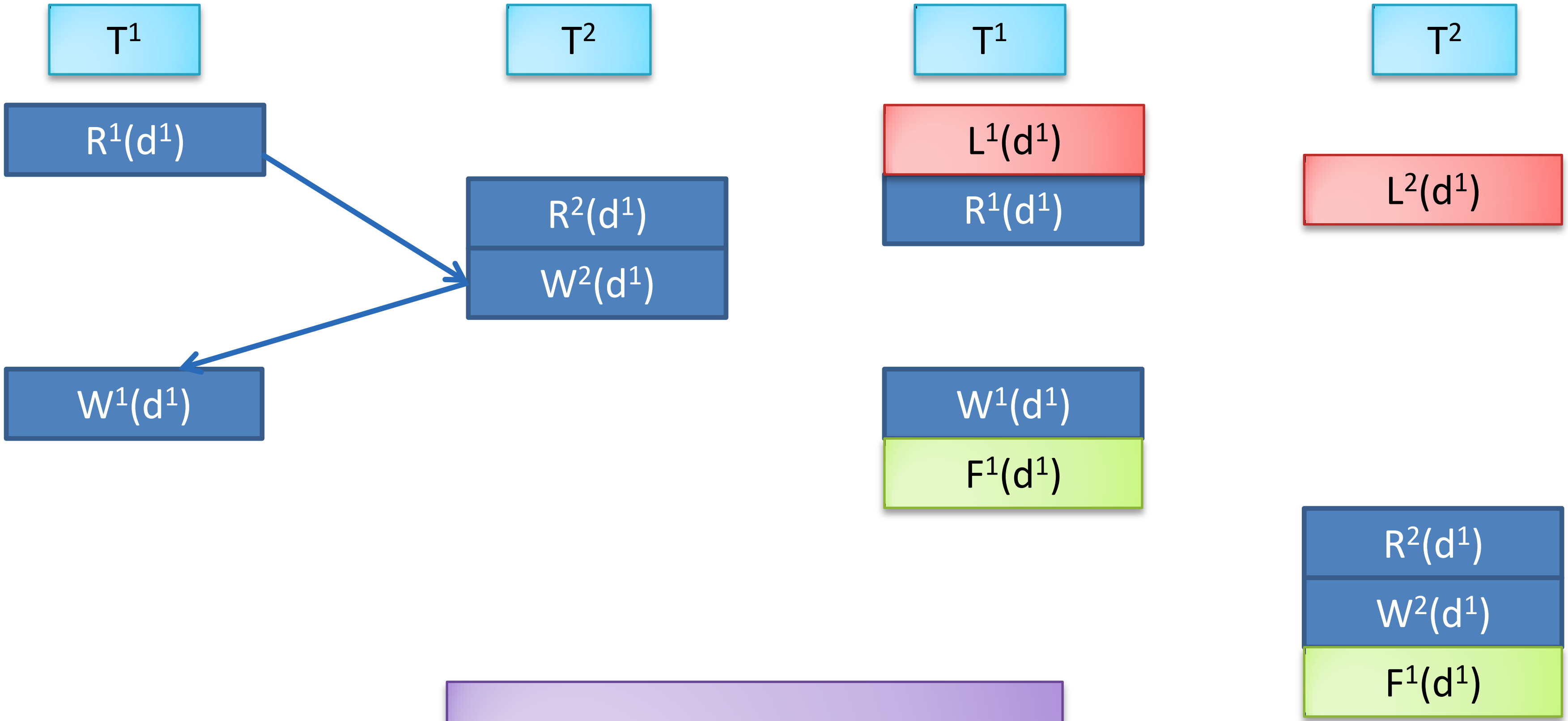
There is a cycle but no deadlock

Cycle: T^1 accessed d^1 before T^2, and T^2 accessed d^2 before T^1

No deadlock: T^1 had both locks before T^2 had any locks (or vice versa)

No deadlock: No cycle

Locking: One Item



T^1

T^2

$R^1(d^1)$

$R^2(d^1)$

$W^2(d^1)$

$W^1(d^1)$

T^1

T^2

$L^1(d^1)$

$R^1(d^1)$

$L^2(d^1)$

$W^1(d^1)$

$F^1(d^1)$

$R^2(d^1)$

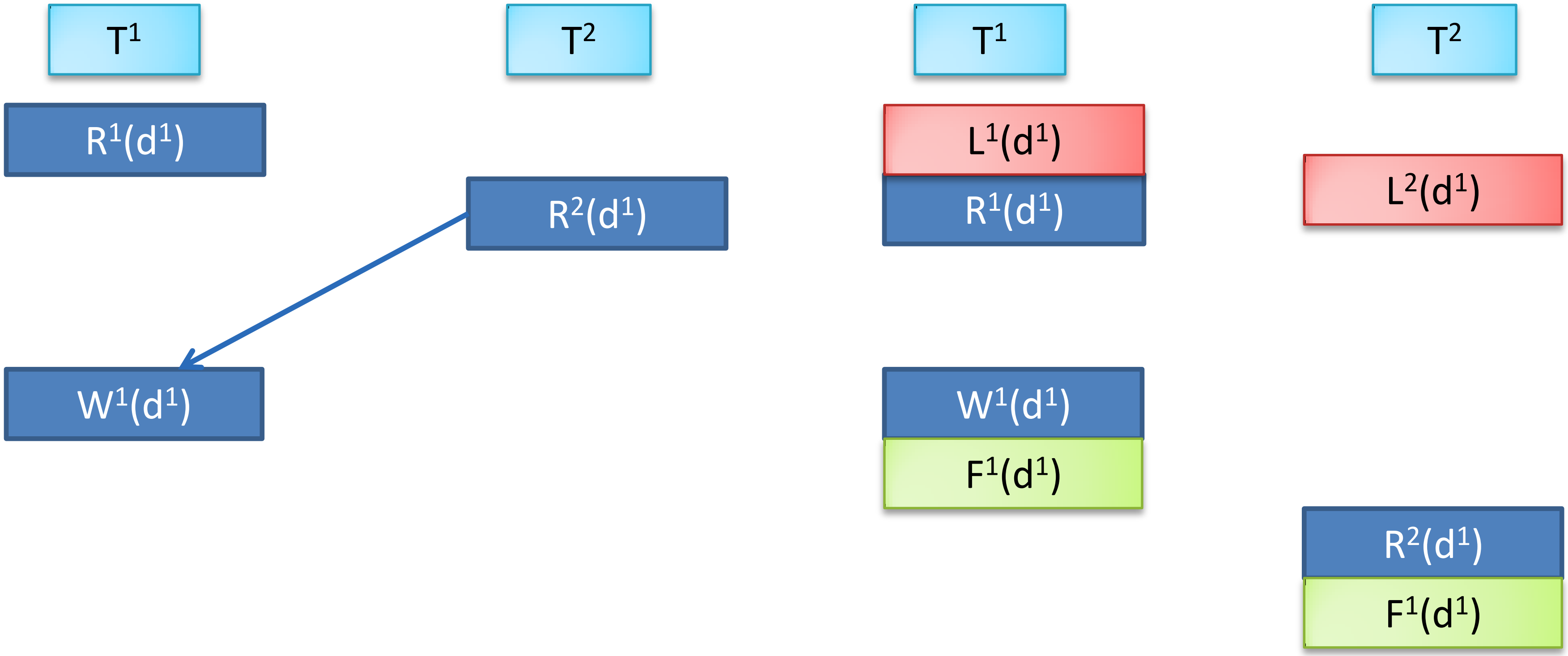
$W^2(d^1)$

$F^1(d^1)$

Single lock for read and write?

T^2 's read unnecessarily delayed

Locking: One Item



T^1

$R^1(d^1)$

$W^1(d^1)$

T^2

$R^2(d^1)$

T^1

$L^1(d^1)$

$R^1(d^1)$

$W^1(d^1)$

$F^1(d^1)$

T^2

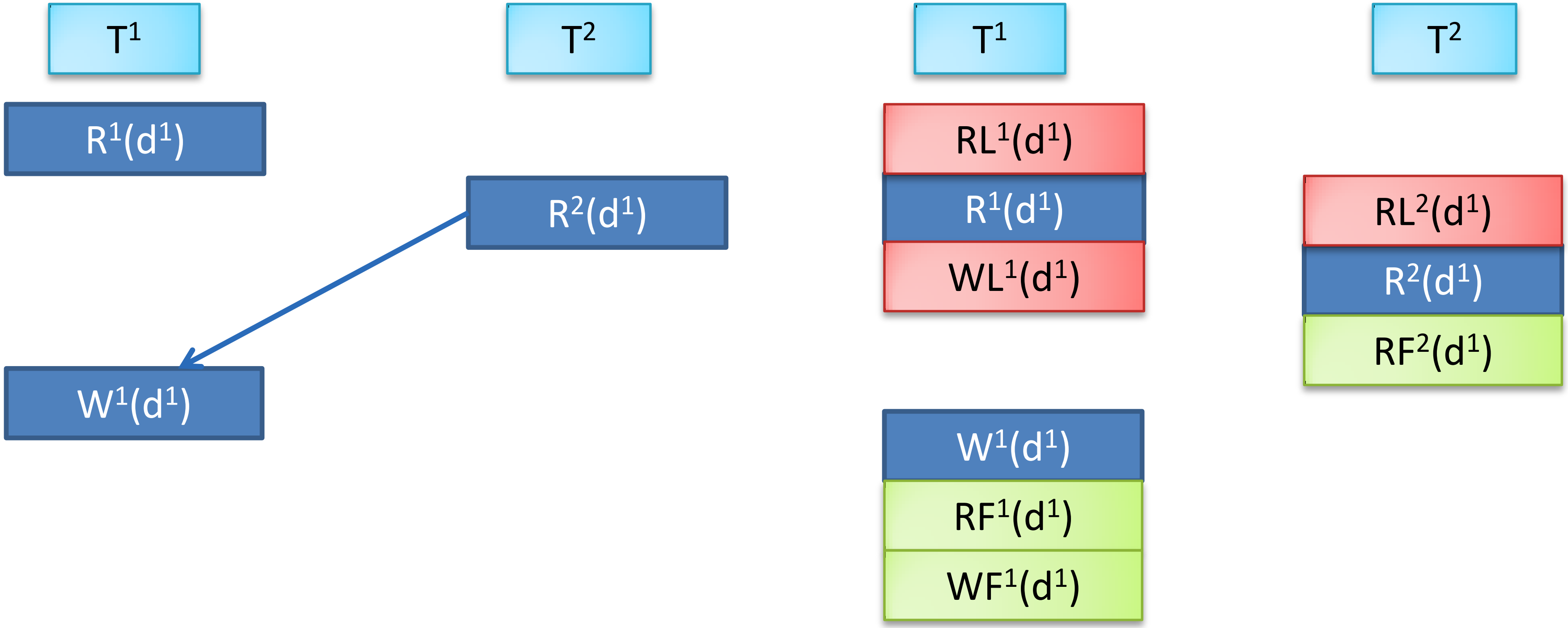
$L^2(d^1)$

$R^2(d^1)$

$F^1(d^1)$

T2 unnecessarily delayed

Type-Specific Locks



Concurrent reads allowed

Concurrent read and write not allowed

Read/Write Locks

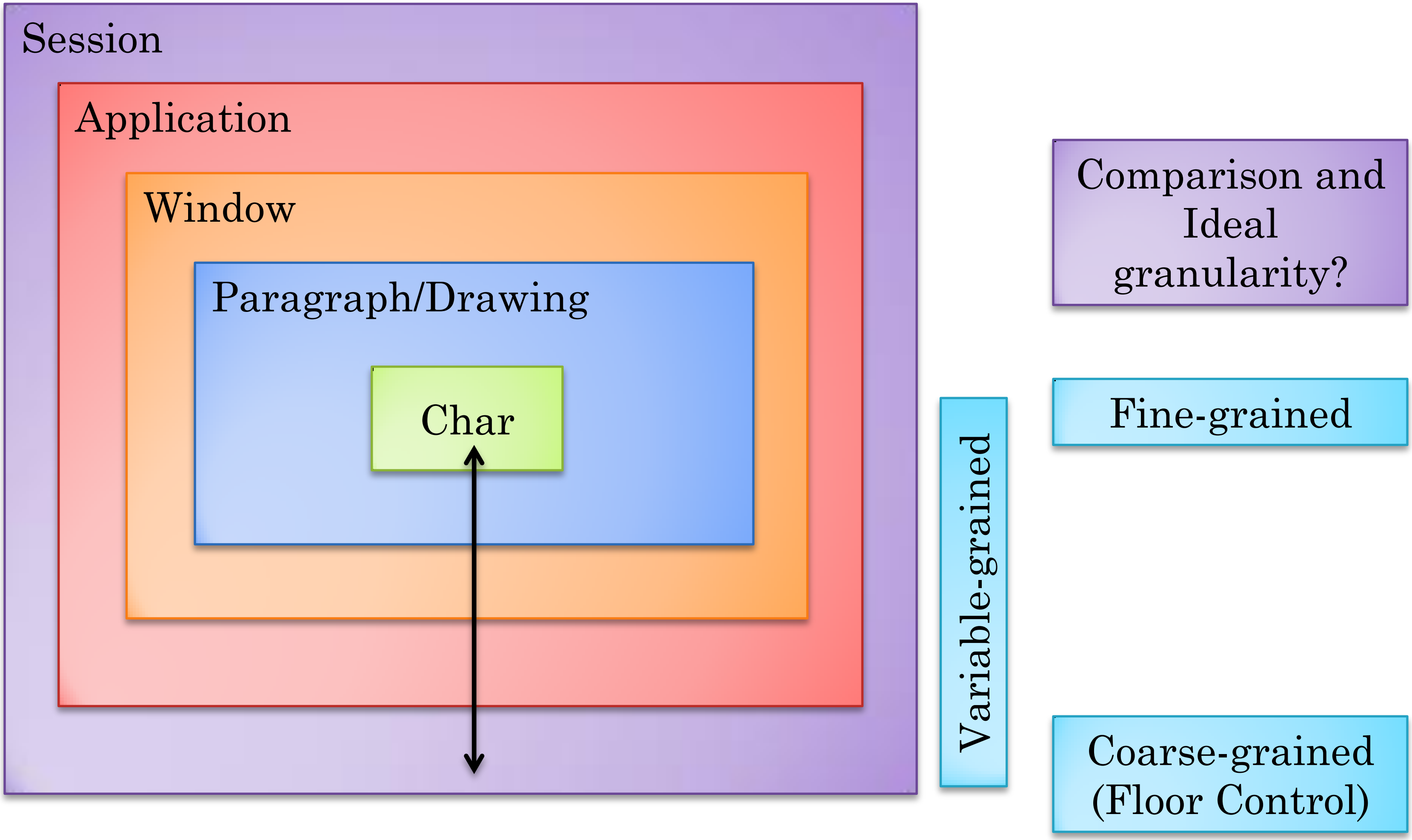
Data Item D	Read Locked	Write Locked	Unlocked
Read Lock	Yes	No	Yes
Write Lock	No	No	Yes

S(hared)/(E)x(clusive) Locks

	S	X
S	Yes	No
X	No	No

More compact representation

LOCK GRANULARITY



Session

Application

Window

Paragraph/Drawing

Char

Variable-grained

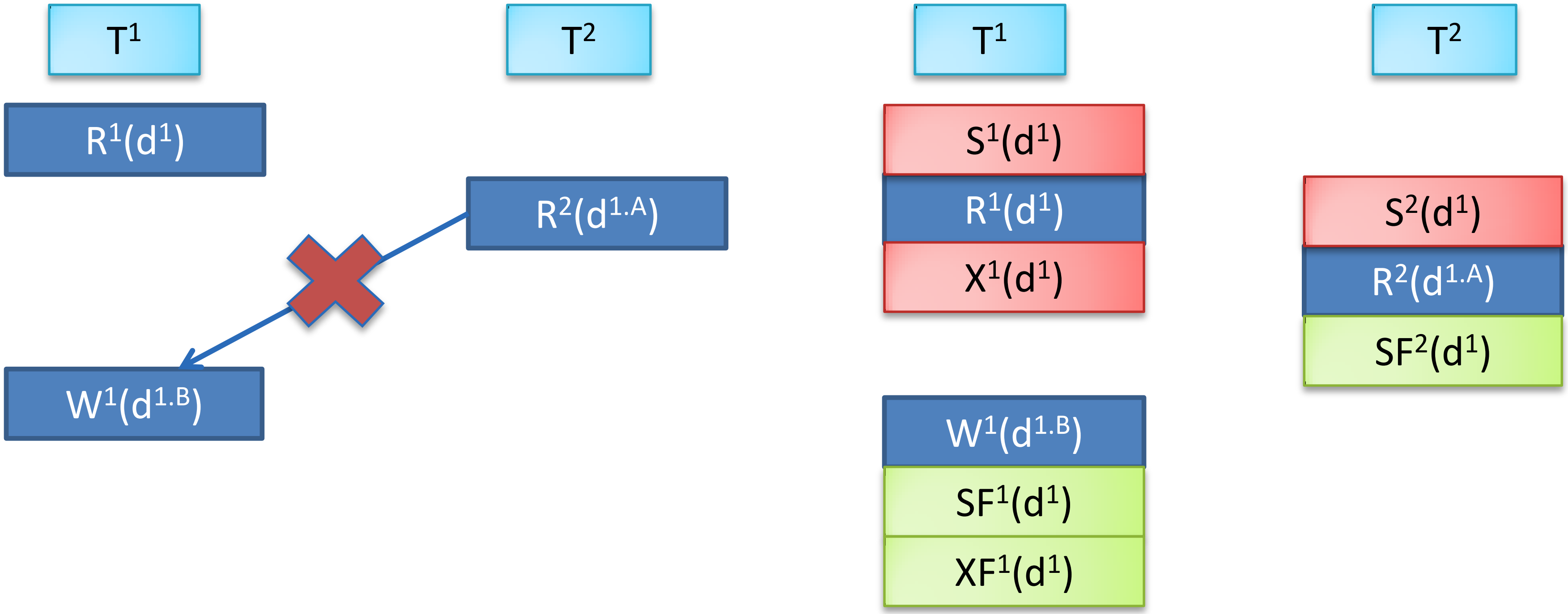
Comparison and Ideal granularity?

Fine-grained

Coarse-grained (Floor Control)

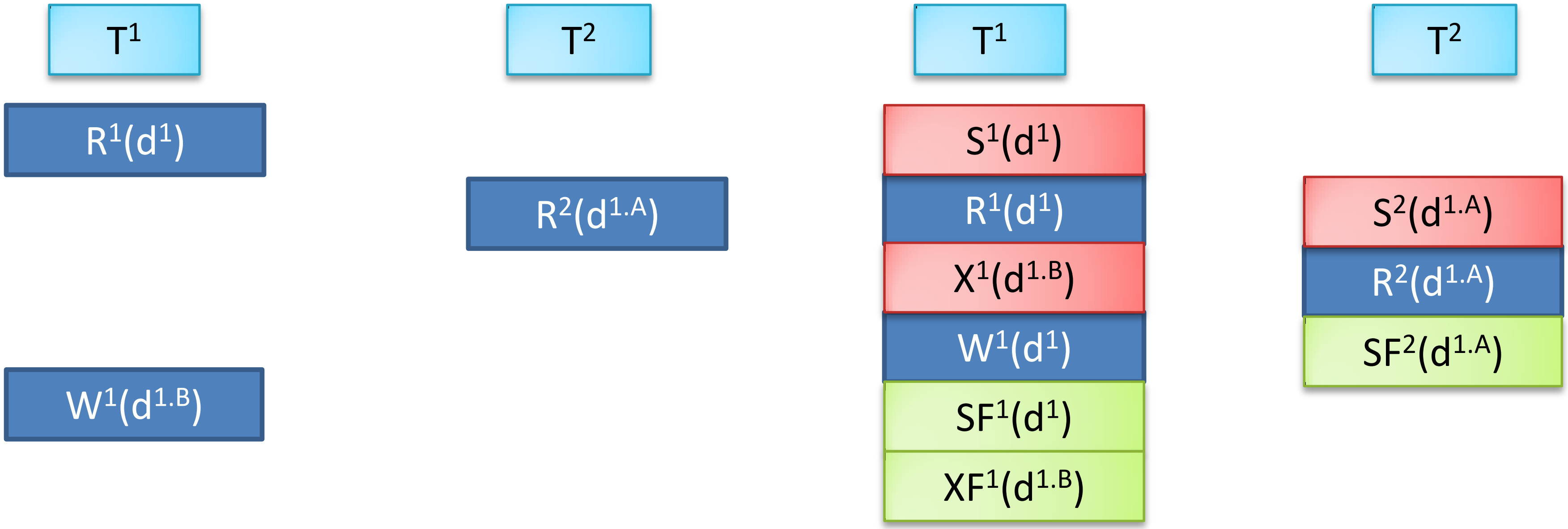
Finer Control → More Concurrency → More Lock/Unlock Operations → More Locking Overhead

Fixed-Grain Locking



T^1 unnecessarily waits for T^2 to finish write

Variable-Grained Hierarchical Locking



T^1

$R^1(d^1)$

$W^1(d^{1.B})$

T^2

$R^2(d^{1.A})$

T^1

$S^1(d^1)$

$R^1(d^1)$

$X^1(d^{1.B})$

$W^1(d^1)$

$SF^1(d^1)$

$XF^1(d^{1.B})$

T^2

$S^2(d^{1.A})$

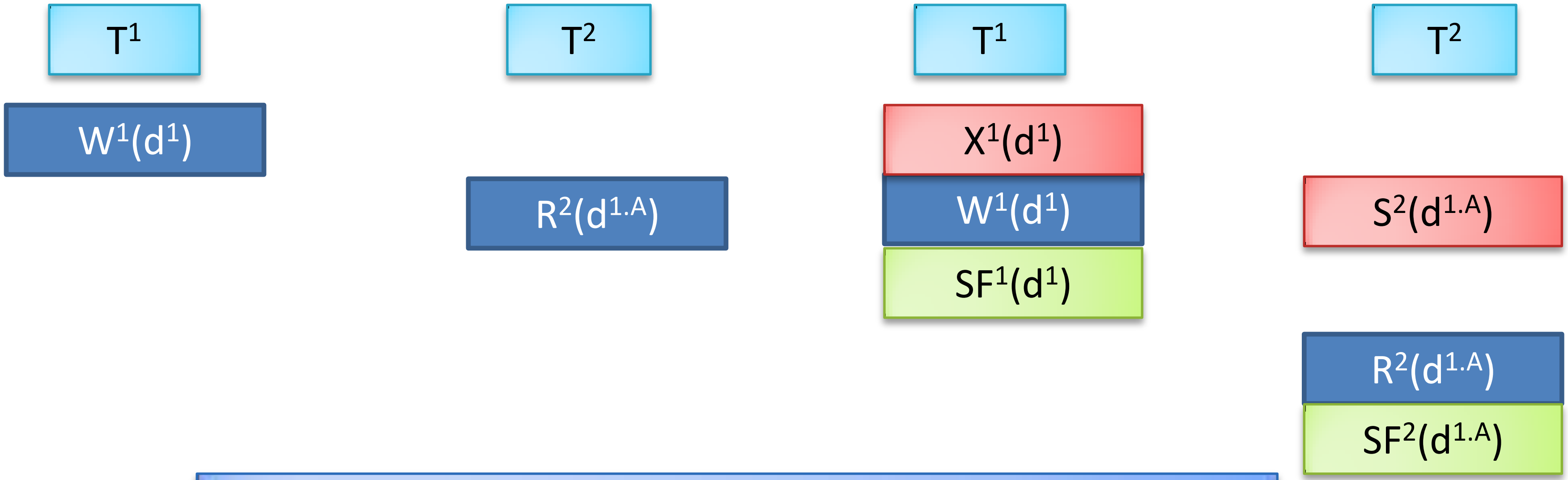
$R^2(d^{1.A})$

$SF^2(d^{1.A})$

More concurrency

Each lock in a tree independent, look only at lock at your level?

Ancestor Dependence



T^1

$W^1(d^1)$

T^2

$R^2(d^{1.A})$

T^1

$X^1(d^1)$

$W^1(d^1)$

$SF^1(d^1)$

T^2

$S^2(d^{1.A})$

$R^2(d^{1.A})$

$SF^2(d^{1.A})$

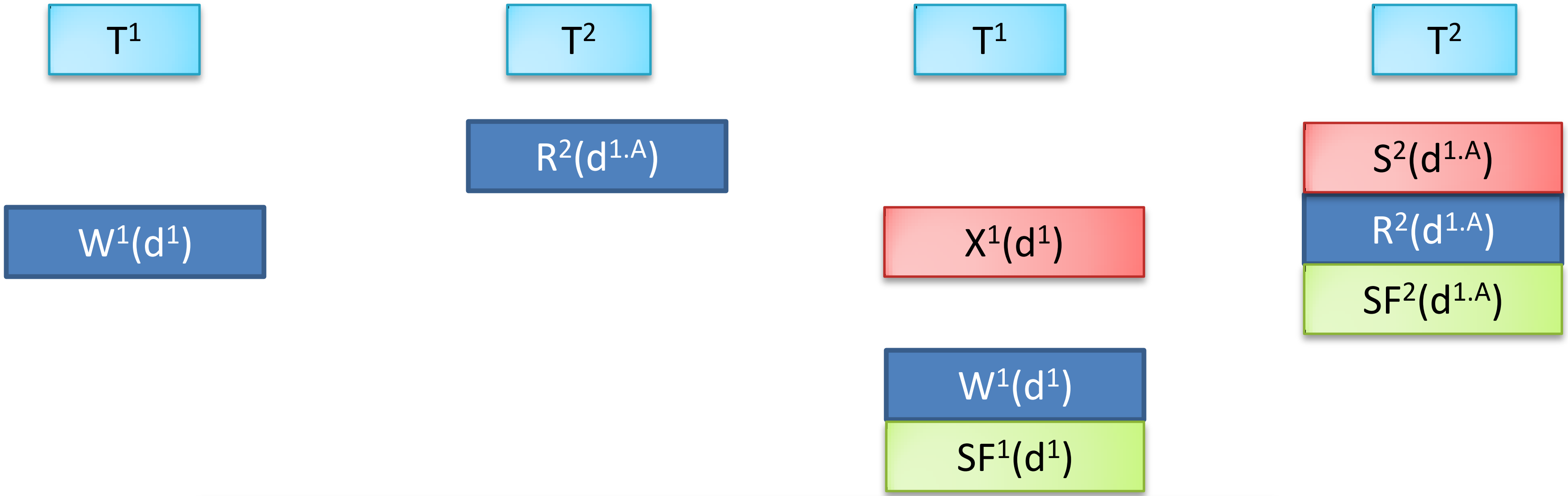
Lock operation must consider lock at ancestor nodes

Search cost?

# searches ~ height of tree - O(h))

Descendent dependence?

Descendent Dependence



T^1

$W^1(d^1)$

T^2

$R^2(d^{1.A})$

T^1

$X^1(d^1)$

$W^1(d^1)$

$SF^1(d^1)$

T^2

$S^2(d^{1.A})$

$R^2(d^{1.A})$

$SF^2(d^{1.A})$

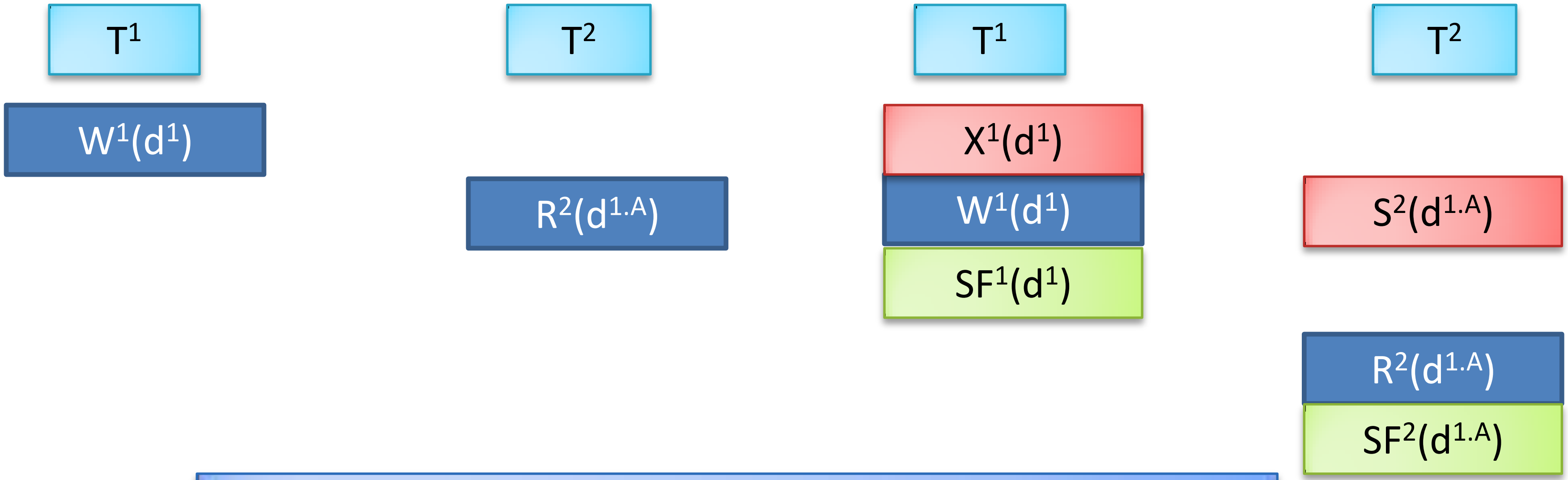
Lock operation must consider lock at descendent nodes

# searches ~ nodes in tree - $O(2^h)$

Assuming a node contains information only about locks at that node

Trade space for time?

Ancestor Dependence (Review)



T^1 $W^1(d^1)$

T^2 $R^2(d^{1.A})$

T^1 $X^1(d^1)$ $W^1(d^1)$ $SF^1(d^1)$

T^2 $S^2(d^{1.A})$ $R^2(d^{1.A})$ $SF^2(d^{1.A})$

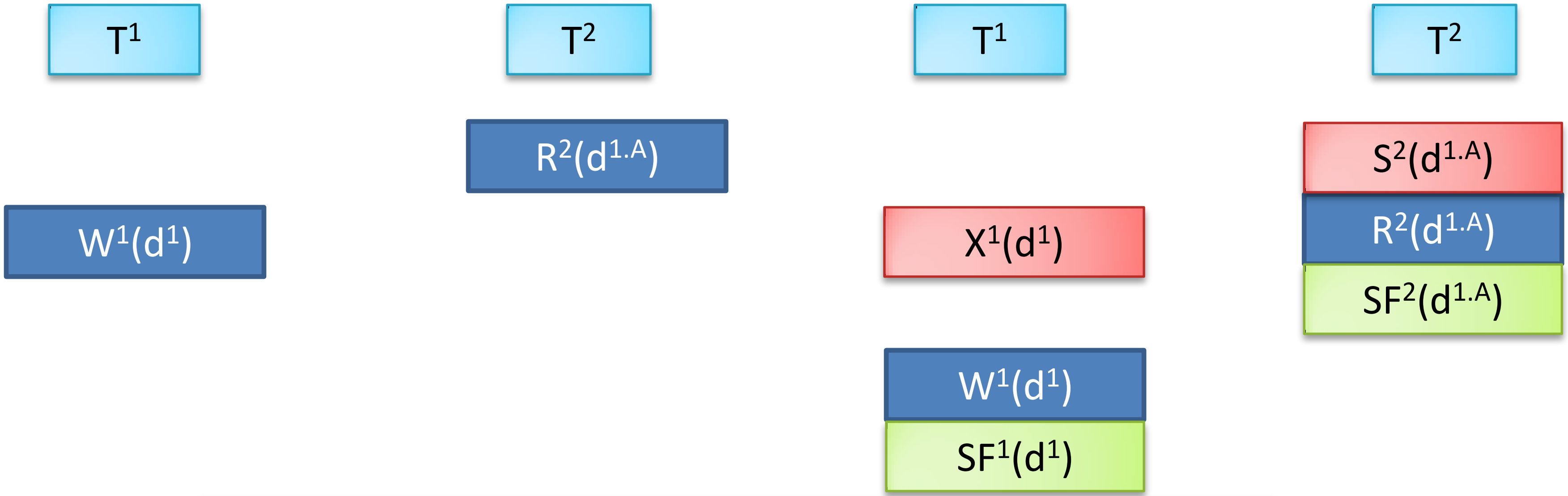
Lock operation must consider lock at ancestor nodes

Search cost?

# searches ~ height of tree - O(h))

Descendent dependence?

Descendent Dependence (Review)



T^1

$W^1(d^1)$

T^2

$R^2(d^{1.A})$

T^1

$X^1(d^1)$

$W^1(d^1)$

$SF^1(d^1)$

T^2

$S^2(d^{1.A})$

$R^2(d^{1.A})$

$SF^2(d^{1.A})$

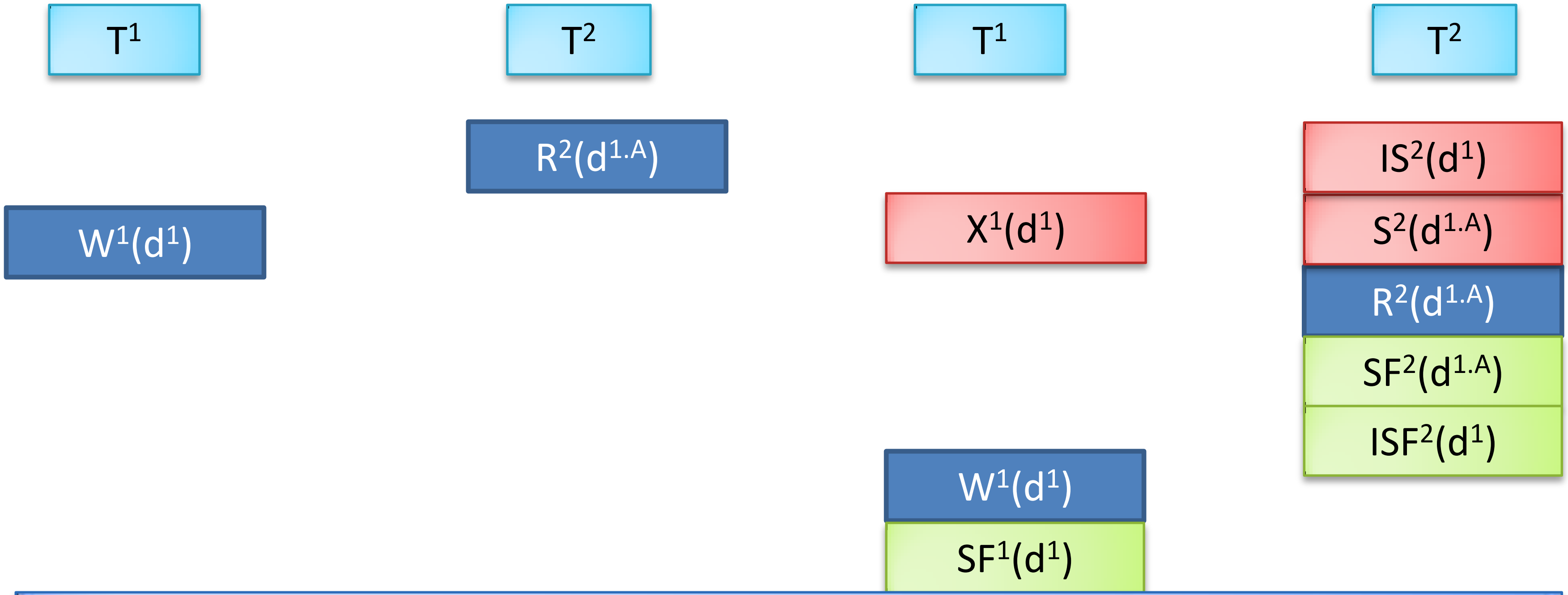
Lock operation must consider lock at descendent nodes

# searches ~ nodes in tree - O (2^h)

Assuming a node contains information only about locks at that node

Trade space for time?

Intention Locks



Intention lock: a flag (synthesized attribute) in each ancestor of a locked node indicating the kind of lock, associated with a reference count incremented/decremented by lock and free operations

Synthesized attribute: An attribute of a node that is a function of a descendent(IS)

Inherited attribute: An attribute of a node that is a function of an ancestor(S)

S(HARED)/(E)X(CLUSIVE) LOCKS

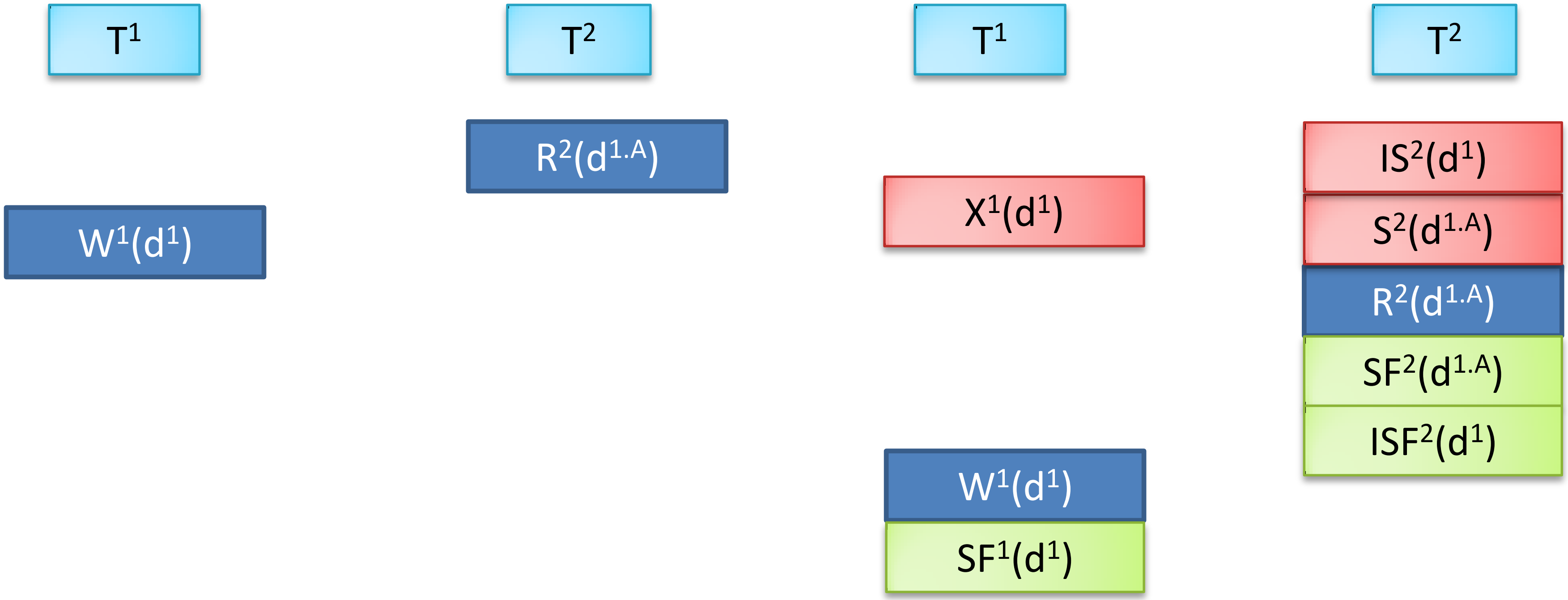
	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

IS: some descendent of the node will have a shared lock

IX: some descendent of the node will have an exclusive lock

SIX: shared lock on this node and an exclusive lock on some descendent (inherited and synthesized attribute)

Intention Locks



T^1

$W^1(d^1)$

T^2

$R^2(d^{1.A})$

T^1

$X^1(d^1)$

$W^1(d^1)$

$SF^1(d^1)$

T^2

$IS^2(d^1)$

$S^2(d^{1.A})$

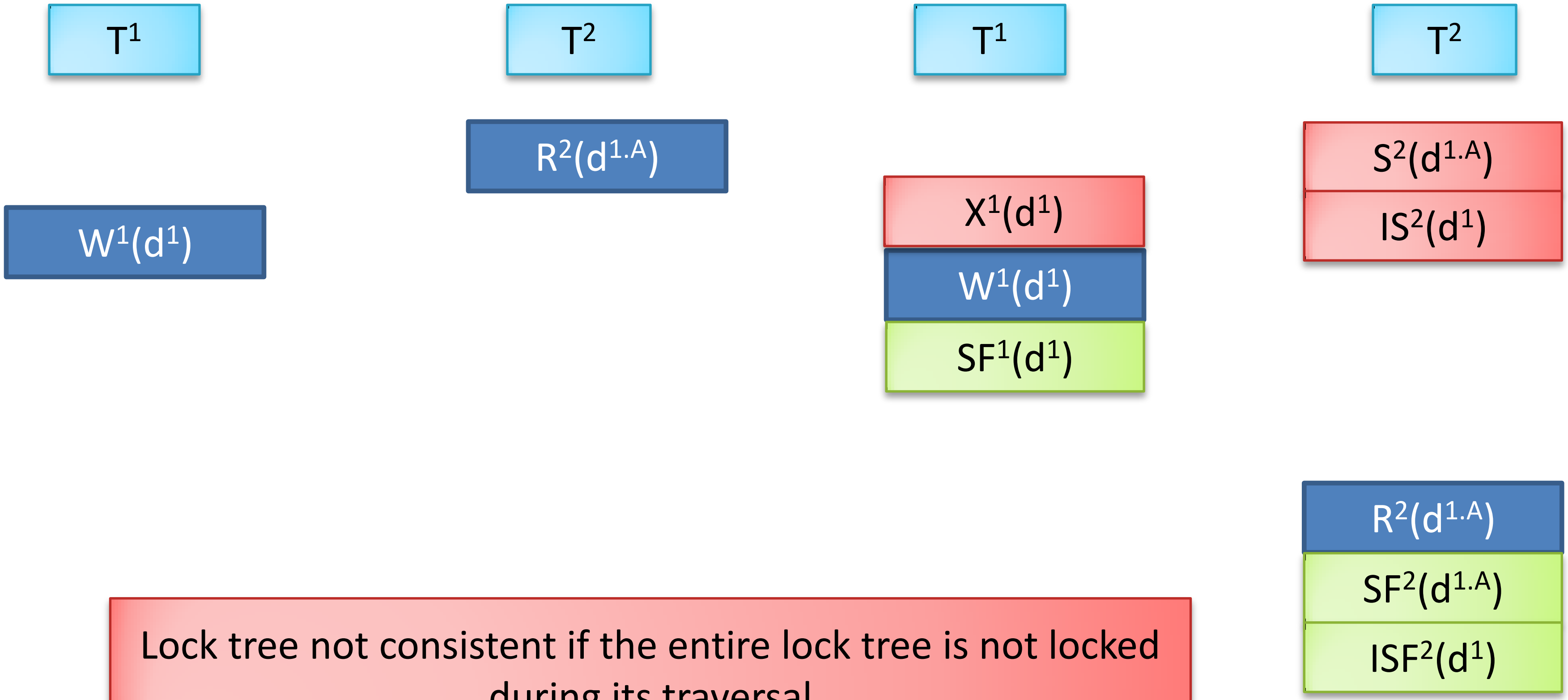
$R^2(d^{1.A})$

$SF^2(d^{1.A})$

$ISF^2(d^1)$

Re-order intention and shared locks?

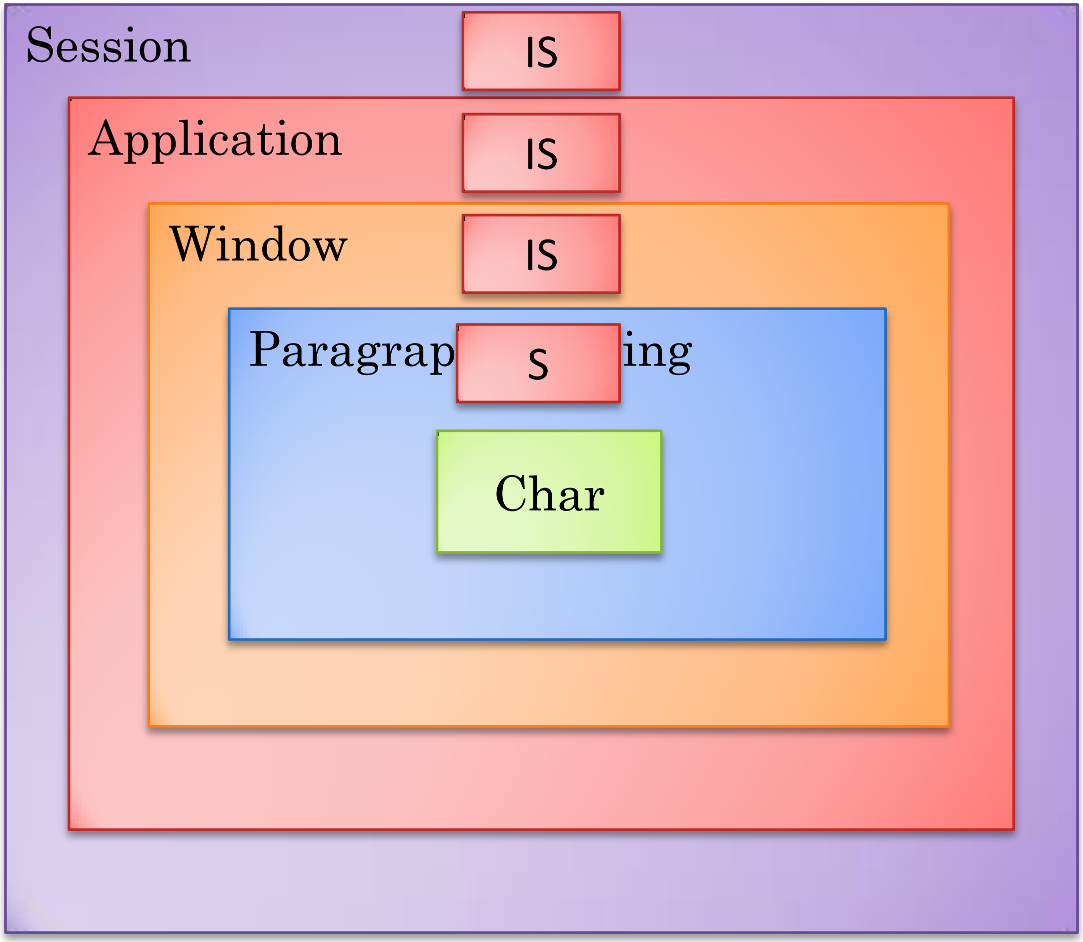
Intention Locks



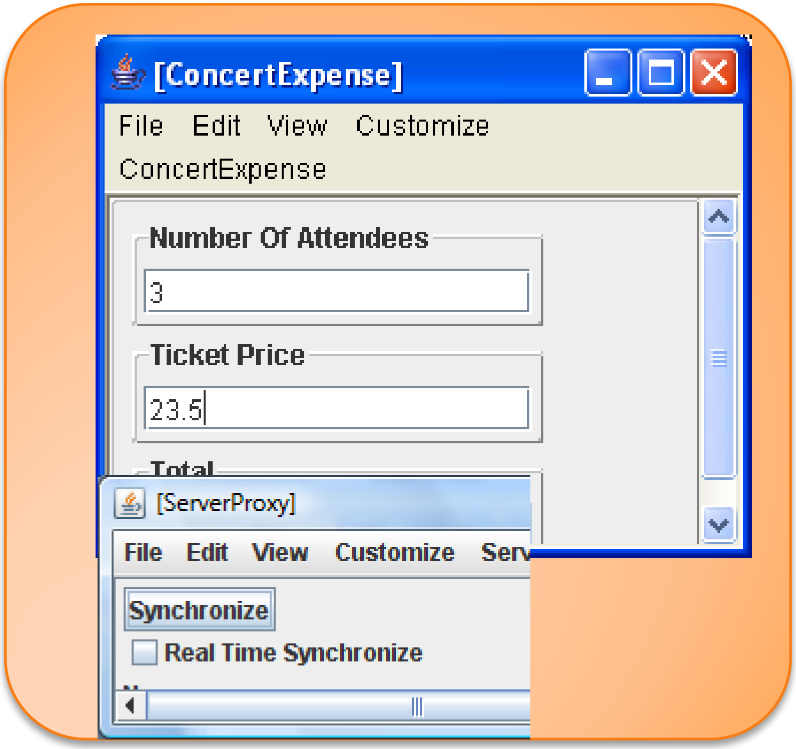
Lock tree not consistent if the entire lock tree is not locked during its traversal

Earlier transaction delayed and can lead to unecessary deadlocks

Locking/Unlocking Order



Shared Model Systems

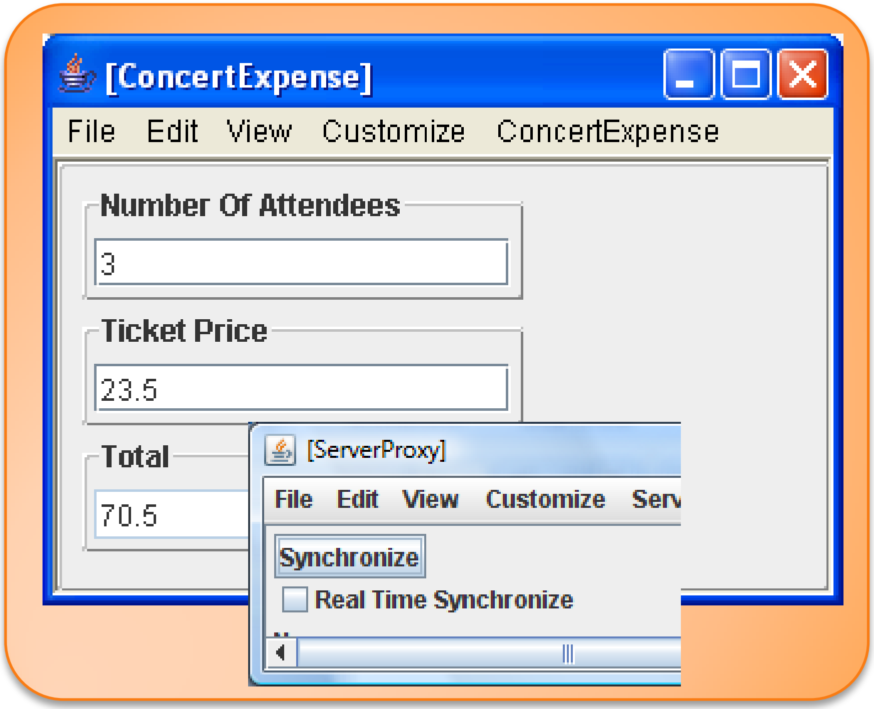

[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize

Number Of Attendees
3

Ticket Price
23.5

Total

□ Real Time Synchronize


[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize

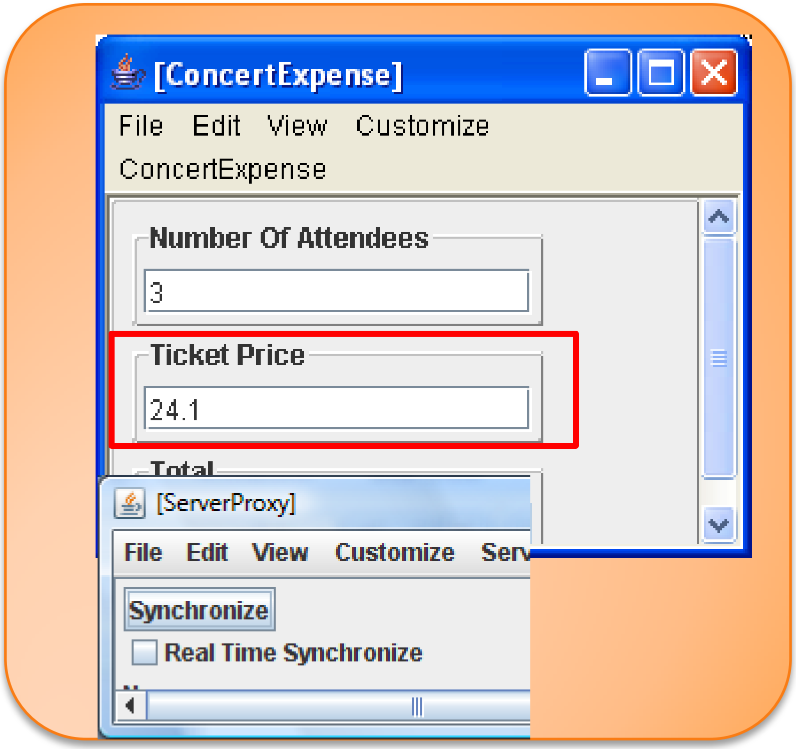
Number Of Attendees
3

Ticket Price
23.5

Total
70.5

□ Real Time Synchronize

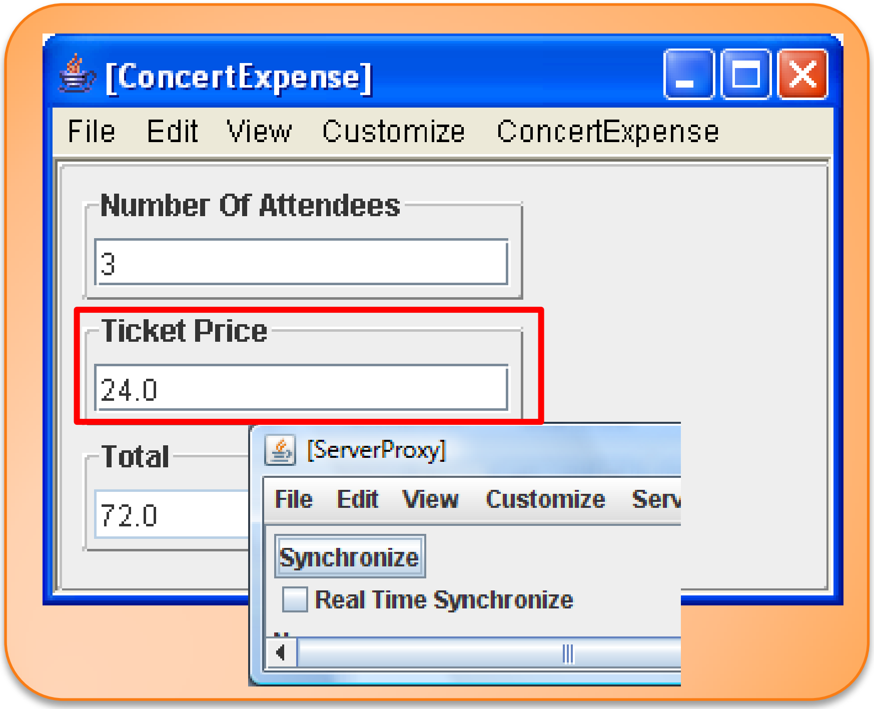
Shared Model Systems


[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize
□ Real Time Synchronize

Number Of Attendees: 3

Ticket Price: 24.1

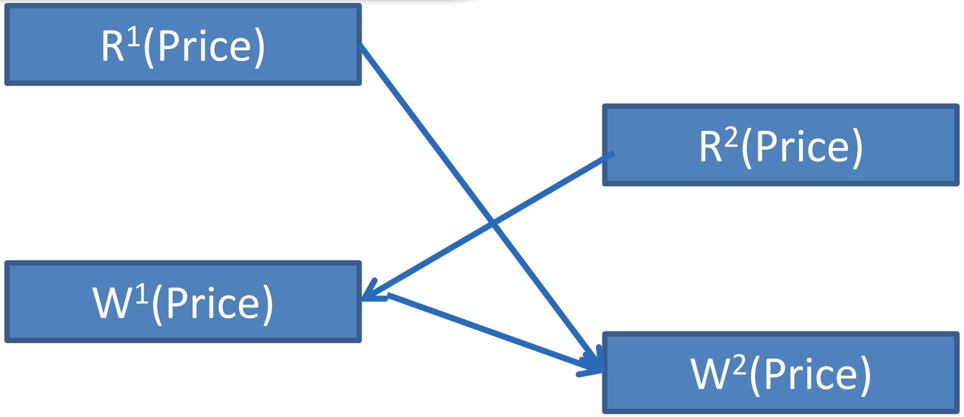
Total


[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize
□ Real Time Synchronize

Number Of Attendees: 3

Ticket Price: 24.0

Total: 72.0



R^1(Price)

R^2(Price)

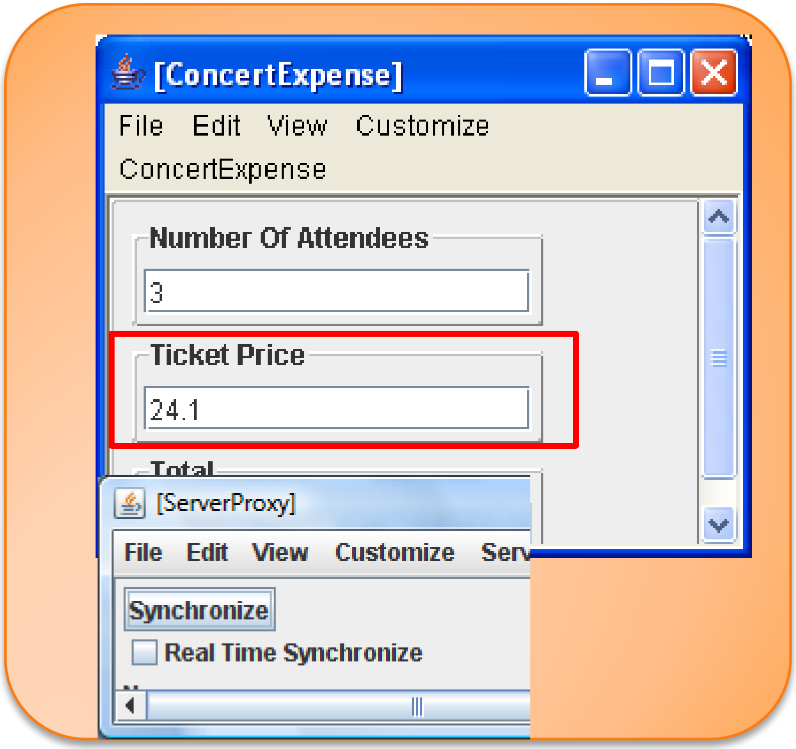
W^1(Price)

W^2(Price)

What does time line mean here?

Sync should be a first class operation known to the transaction system

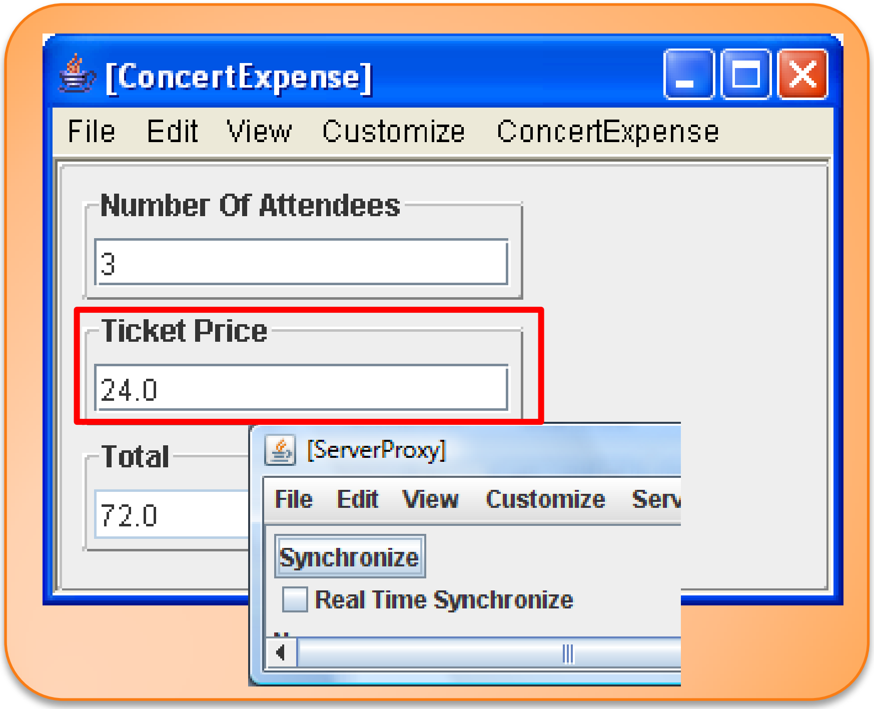
Alternative Read Modeling


[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize
□ Real Time Synchronize

Number Of Attendees
3

Ticket Price
24.1

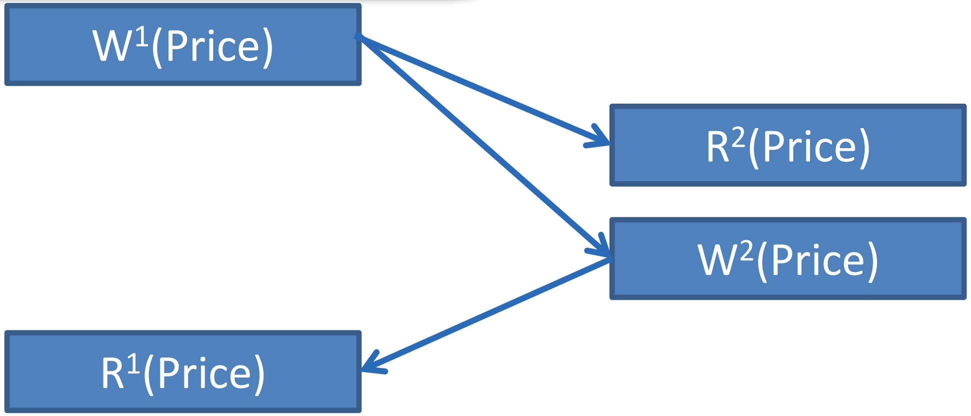
Total


[ConcertExpense]
File Edit View Customize ConcertExpense
[ServerProxy]
File Edit View Customize Serv
Synchronize
□ Real Time Synchronize

Number Of Attendees
3

Ticket Price
24.0

Total
72.0



W^1(Price)

R^2(Price)

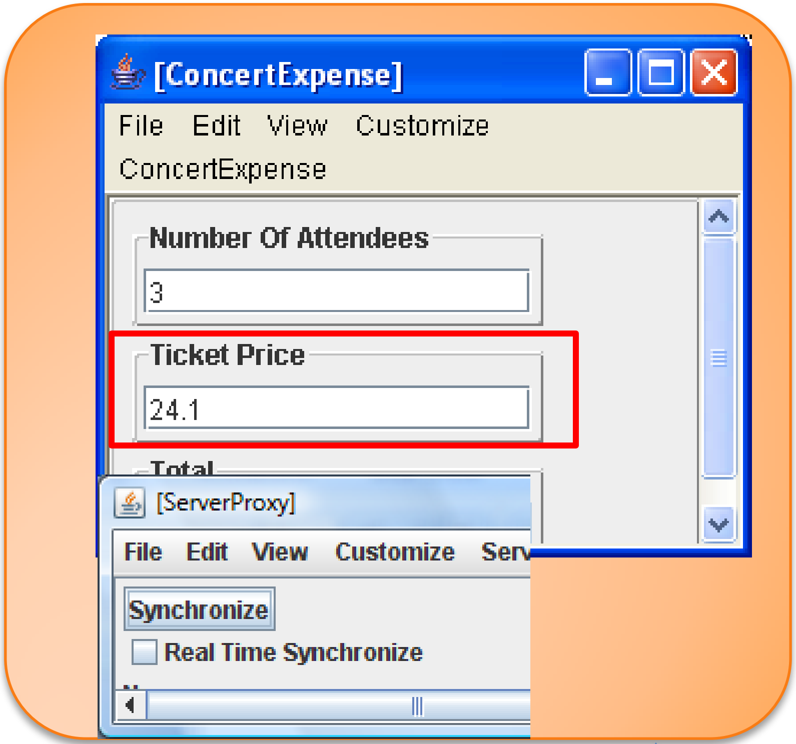
W^2(Price)

R^1(Price)

Neither transaction reads value of the other or overwrites until synchronize (commit) occurs

No incremental sharing

READ, VALIDATION, WRITE PHASE


[ConcertExpense]

File Edit View Customize ConcertExpense

Number Of Attendees: 3

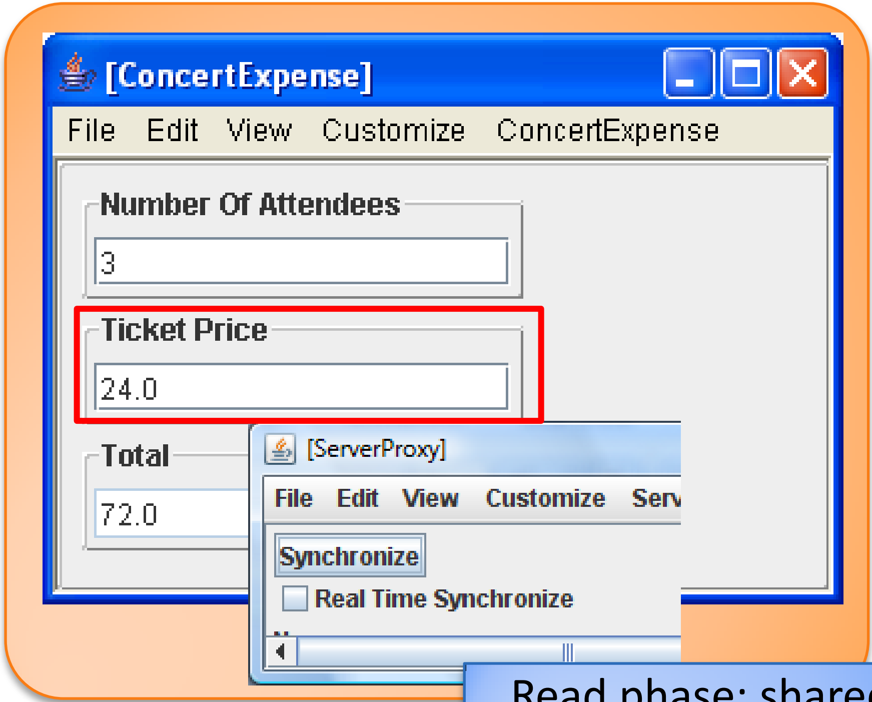
Ticket Price: 24.1

Total

[ServerProxy]

File Edit View Customize Serv
Synchronize

□ Real Time Synchronize


[ConcertExpense]

File Edit View Customize ConcertExpense

Number Of Attendees: 3

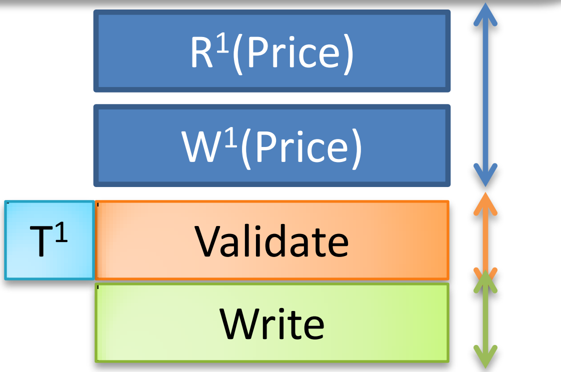
Ticket Price: 24.0

Total: 72.0

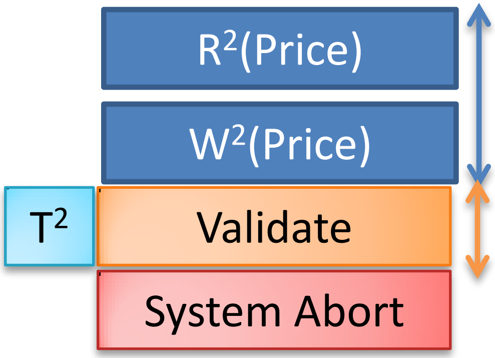
[ServerProxy]

File Edit View Customize Serv
Synchronize

□ Real Time Synchronize



Validation rules?



Read phase: shared object read but not written

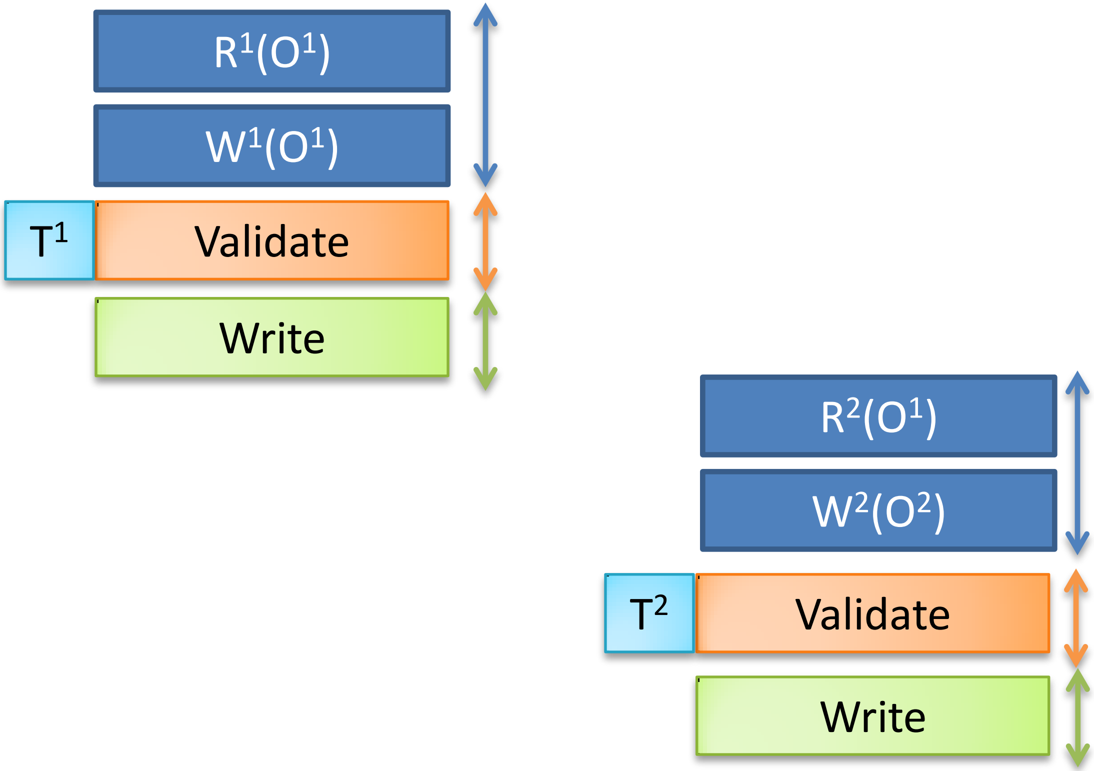
Validation phase, assign time stamps and decide commit or abort

Write phase

Optimistic Transaction Rules

- Optimistic concurrency control divides a transaction into a read phase, a validation phase, and a writing phase
- Read phase: transaction reads shared items, and performs writes on local buffers, with no checking taking place
- Validation phase: the system assigns time stamps to transactions, and assumes transactions are serialized in order of these timestamps
- Write phase, the local writes of validated transactions are made global.
- If a transaction is not validated wrt to another transaction, one of them is aborted

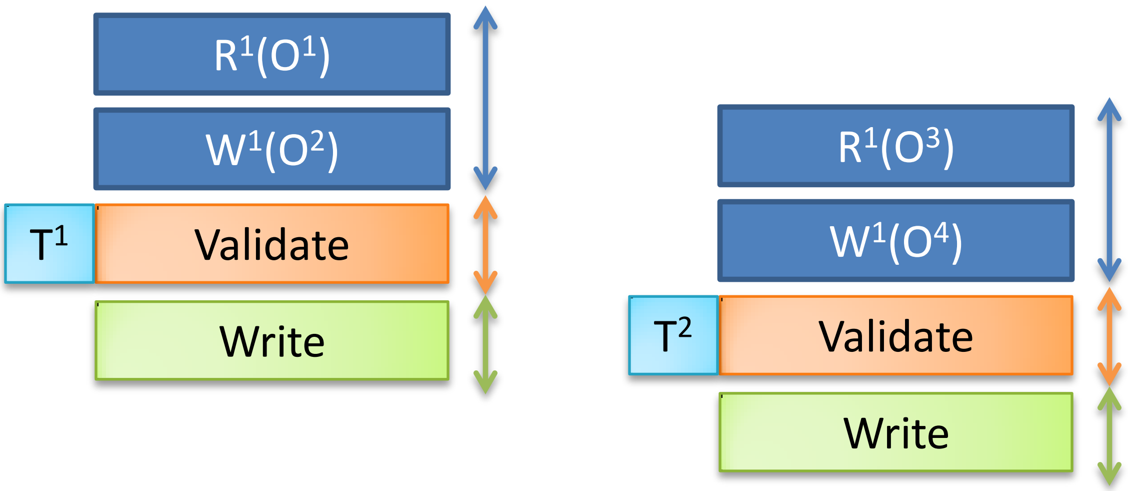
Validation Alternative



Transaction T^i is validated wrt to T^j, $j > i$, T^i finishes its write phase before T^j begins its read phase

Equivalent of locks on same object serializing access

Validation Rules

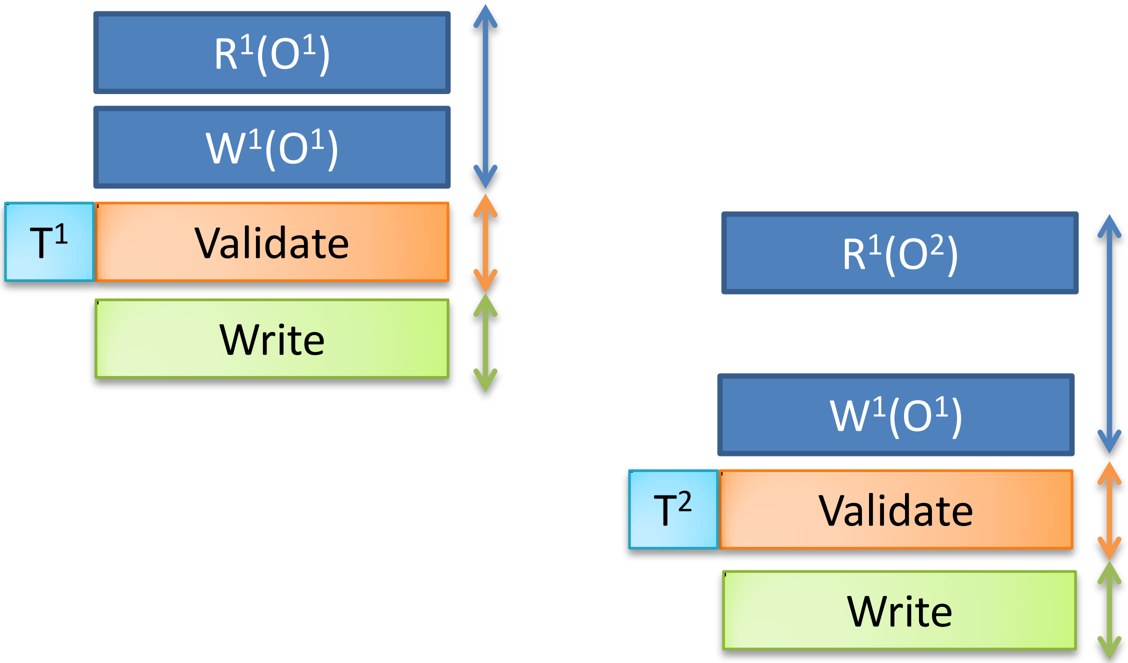


Transaction T^i is validated wrt T^j, $j > i$, T^j does not read or write any items written by T^l

Equivalent of different locks on different objects

Concurrent operations on same sets of object?

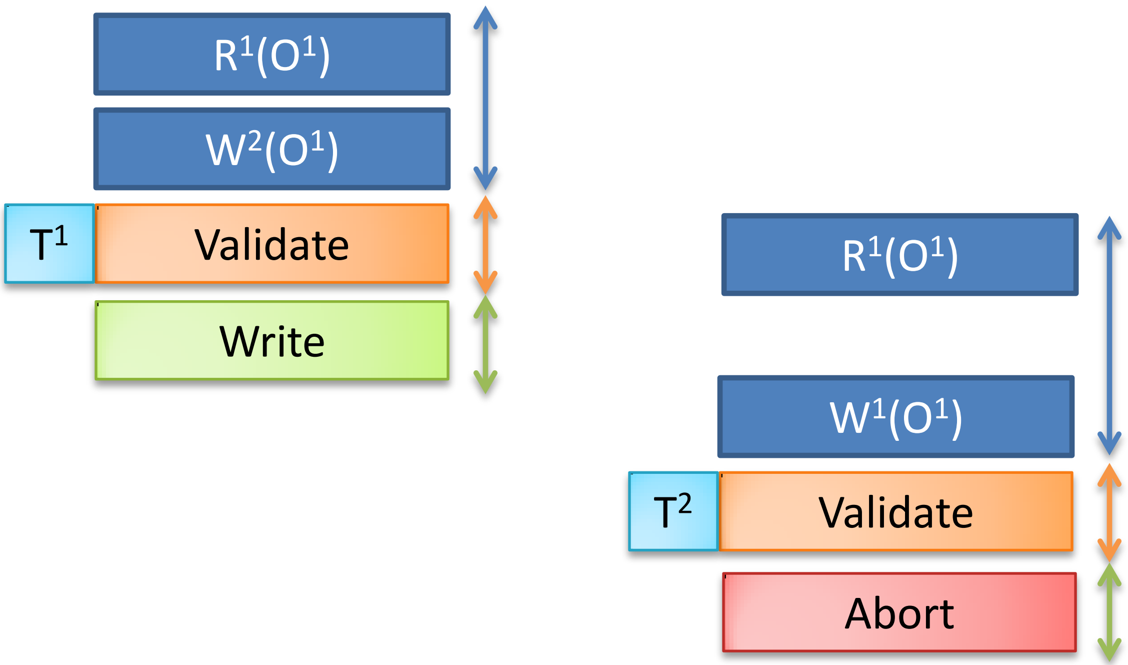
Validation Rules



Transaction T^i is validated if wrt T^j, $j > i$, if T^J does not read any of the items written by T^I and transaction T^I finishes its write phase before transaction T^J begins its write phase.

Lack of incremental sharing does not make a difference when there is no R-W and W-R dependency

Validation Rules



$R^1(O^1)$

$W^2(O^1)$

T^1 Validate

Write

$R^1(O^1)$

$W^1(O^1)$

T^2 Validate

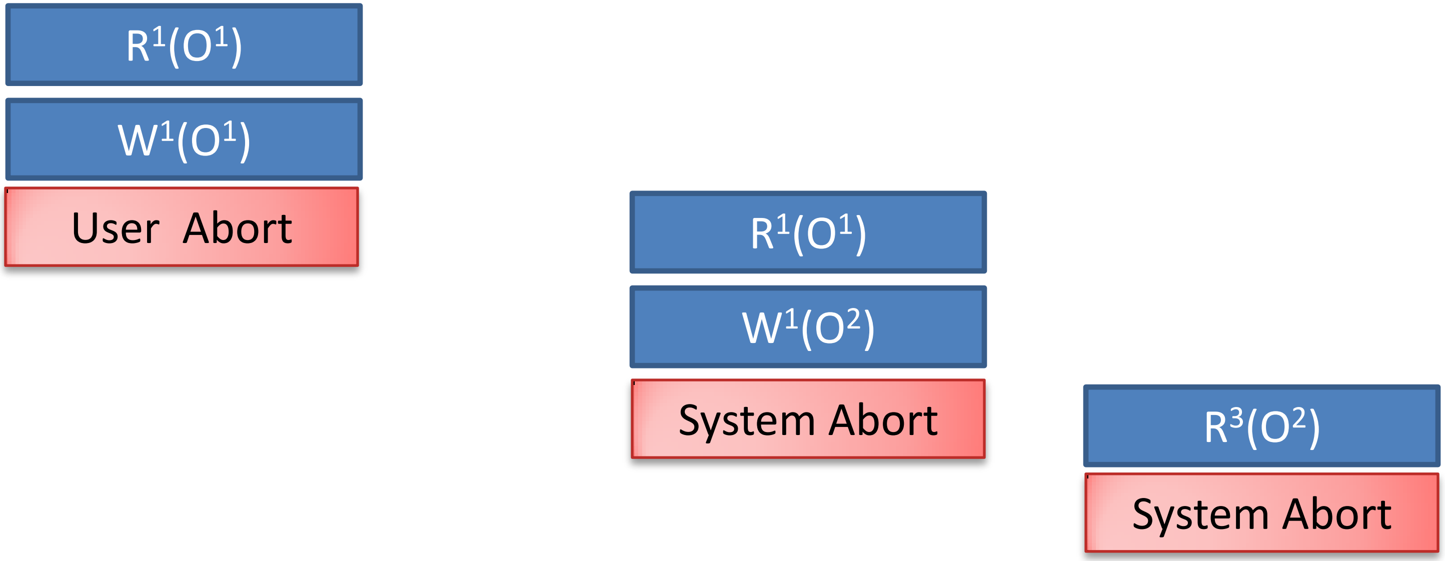
Abort

R-W dependencies (same as W-R dependencies) among concurrent transactions cause aborts

Because no incremental sharing

Locking would have allowed this schedule

Problems of Incremental Sharing



Cascaded abort because incremental results shared in pessimistic schemes

Problem would not occur in optimistic transactions or if no W-R dependencies from transaction aborted by user

In locking systems problem is avoided by keeping write lock until end of transaction

VALIDATION/CHECKING TIME

- Early
  - Pessimistic
- Late
  - Optimistic
- Merging

PESSIMISTIC VS. OPTIMISTIC CC

- Two alternatives to serializable transactions
- Pessimistic
  - Prevent conflicting operation before it is executed
  - Implies locks and possibly remote checking
- Optimistic
  - Abort conflicting operation after it executes
  - Involves replication, check pointing/compensating transactions

Early vs. late validation

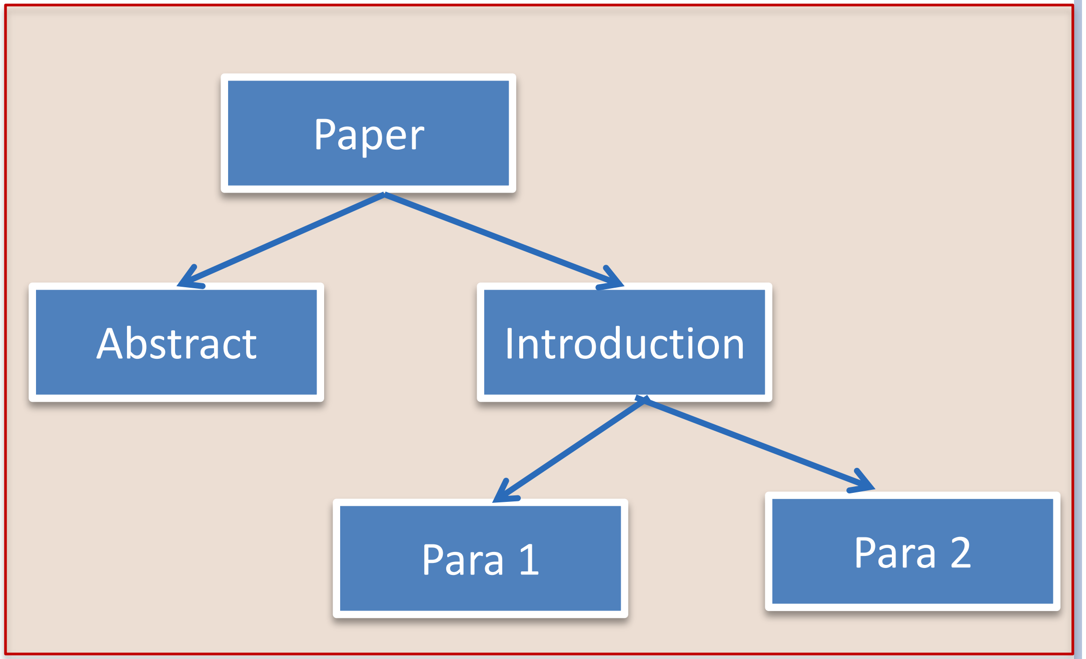
- Per-operation checking and communication overhead
- No compression possible.
- Prevents inconsistency.
- Tight coupling: incremental results shared
- Not functional if disconnected
  - Unless we lock very conservatively, limiting concurrency.

- No per-operation checking, communication overhead
- Compression possible.
- Inconsistency possible resulting in lost work.
- Allows parallel development.
- Functional when disconnected.

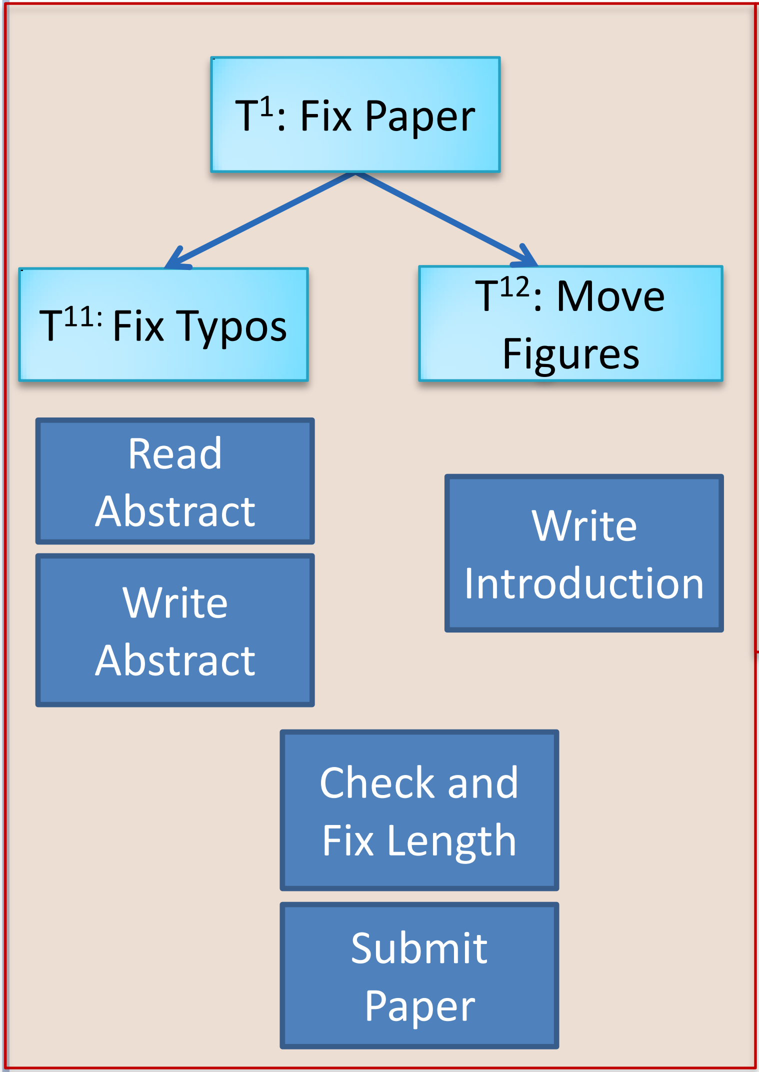
MERGING

- Like optimistic
  - Allow operation to execute without local checks
- But no aborts
  - Merge conflicting operations
  - E.g. insert 1,a | | insert 2, b = insert 1, a; insert 3, b | | insert 2, b; insert 1, a
- Serializability not guaranteed
  - Ignore reads
  - New transaction to replace conflicting transactions
  - Strange results possible
    - E.g. concurrent dragging of an object in whiteboard
- App-specific

Hierarchical Shared Objects



Hierarchical Transactions vs. Objects



T^{1}: Fix Paper

$T^{11:}$ Fix Typos

T^{12}: Move Figures

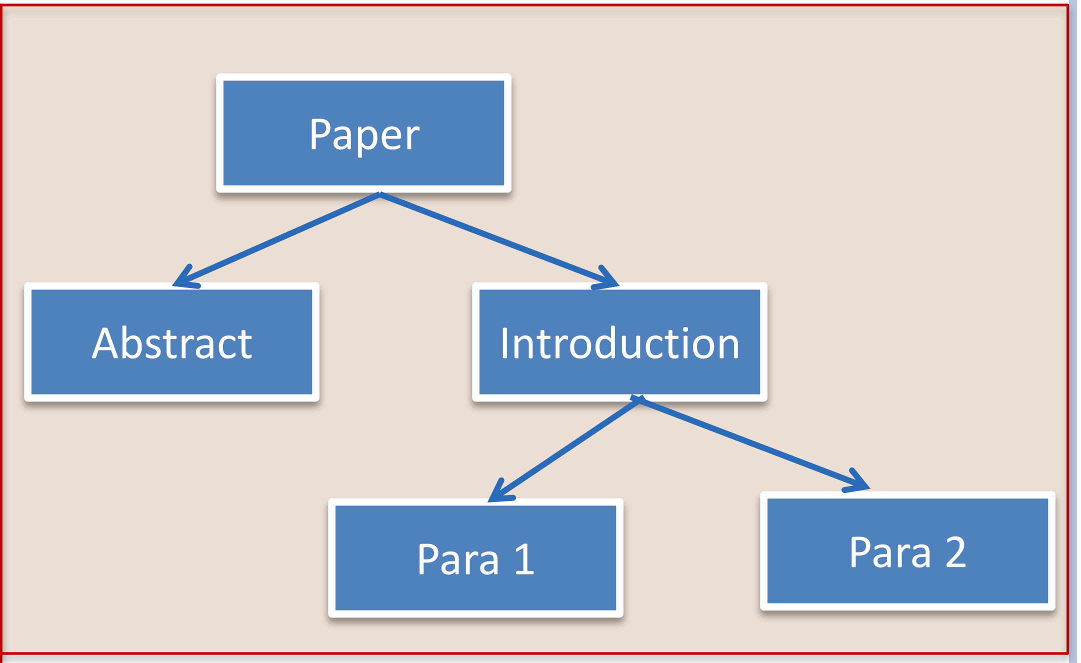
Read Abstract

Write Abstract

Write Introduction

Check and Fix Length

Submit Paper



Paper

Abstract

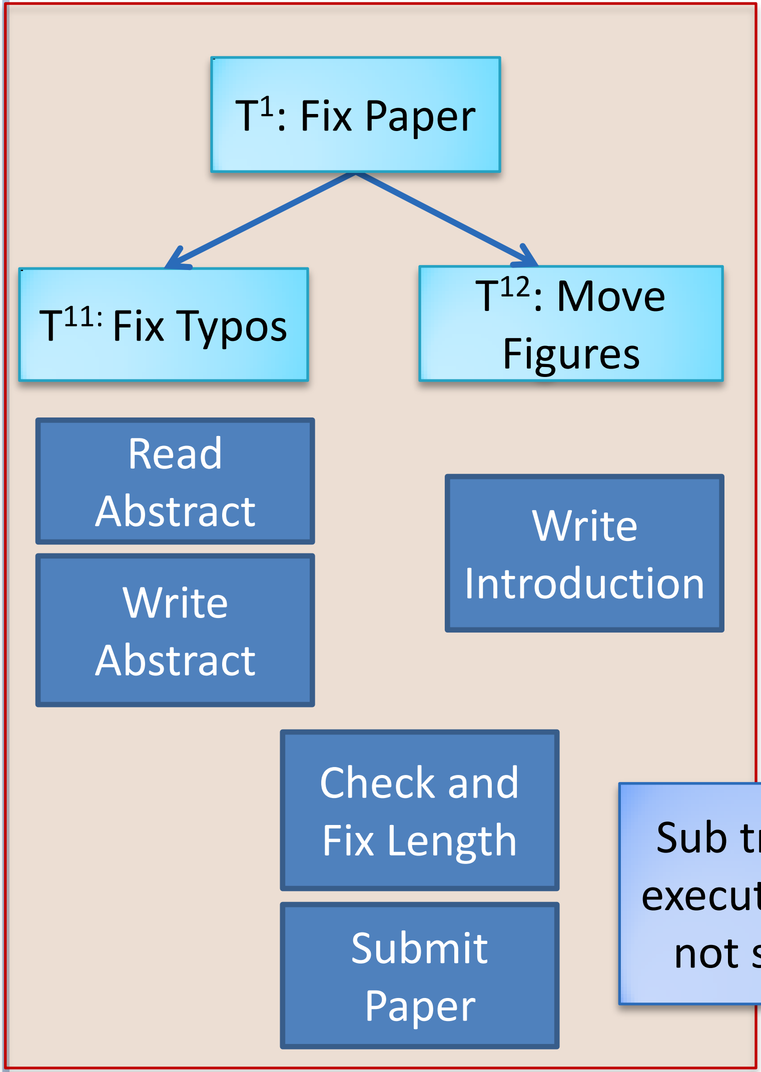
Introduction

Para 1

Para 2

The actions are hierarchical rather than the data

Hierarchical vs. Serial Transactions



T^1: Fix Paper

$T^{11:}$ Fix Typos

T^{12}: Move Figures

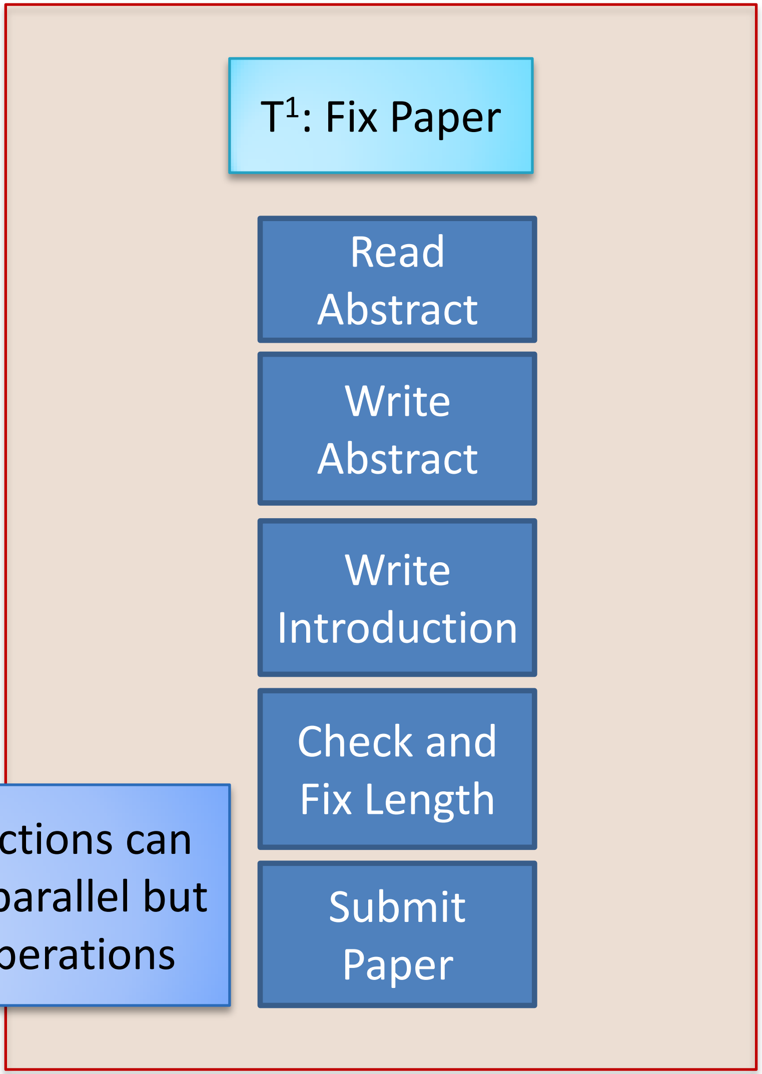
Read Abstract

Write Abstract

Write Introduction

Check and Fix Length

Submit Paper



T^1: Fix Paper

Read Abstract

Write Abstract

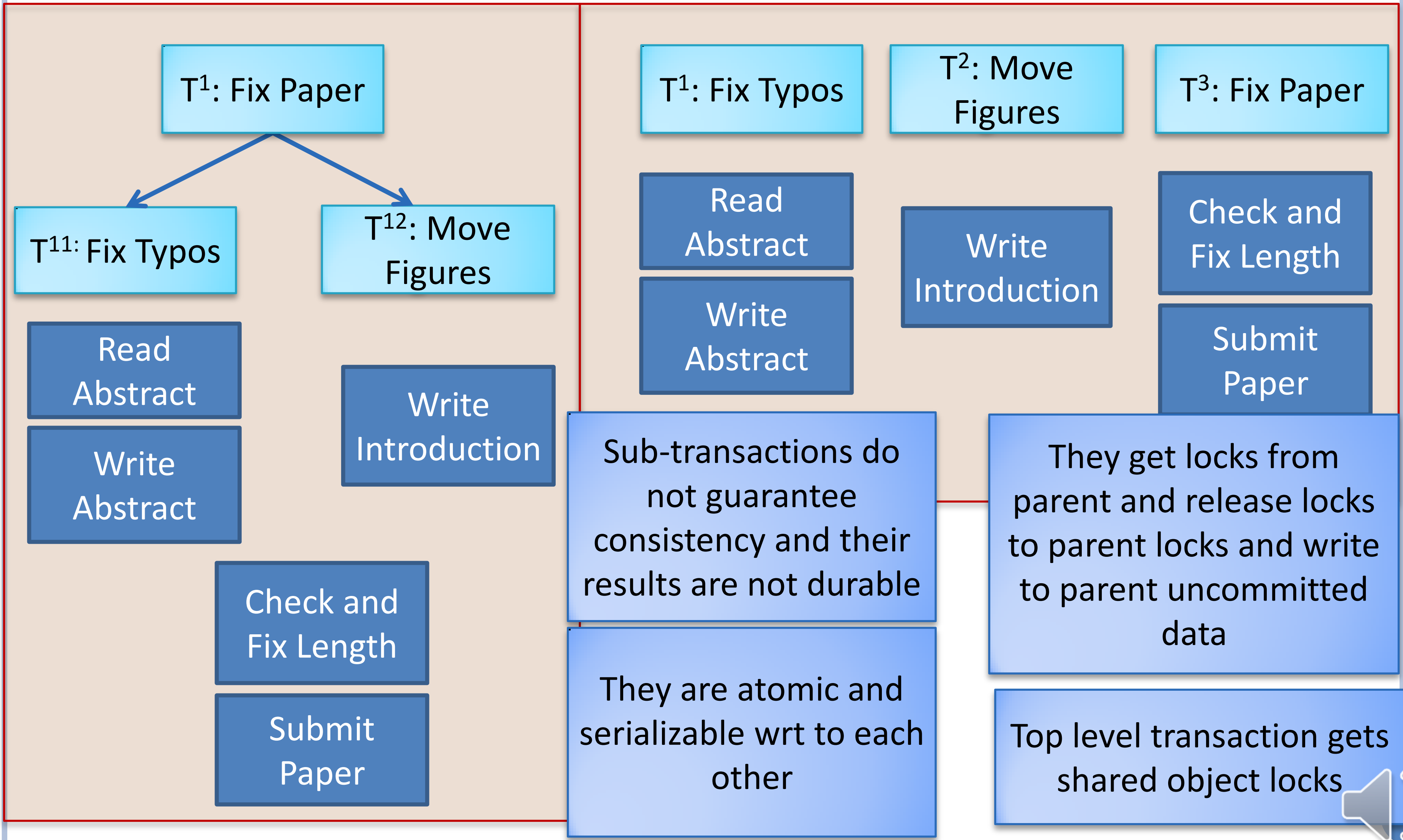
Write Introduction

Check and Fix Length

Submit Paper

Sub transactions can execute in parallel but not sub operations

Hierarchical vs. Flat Transactions



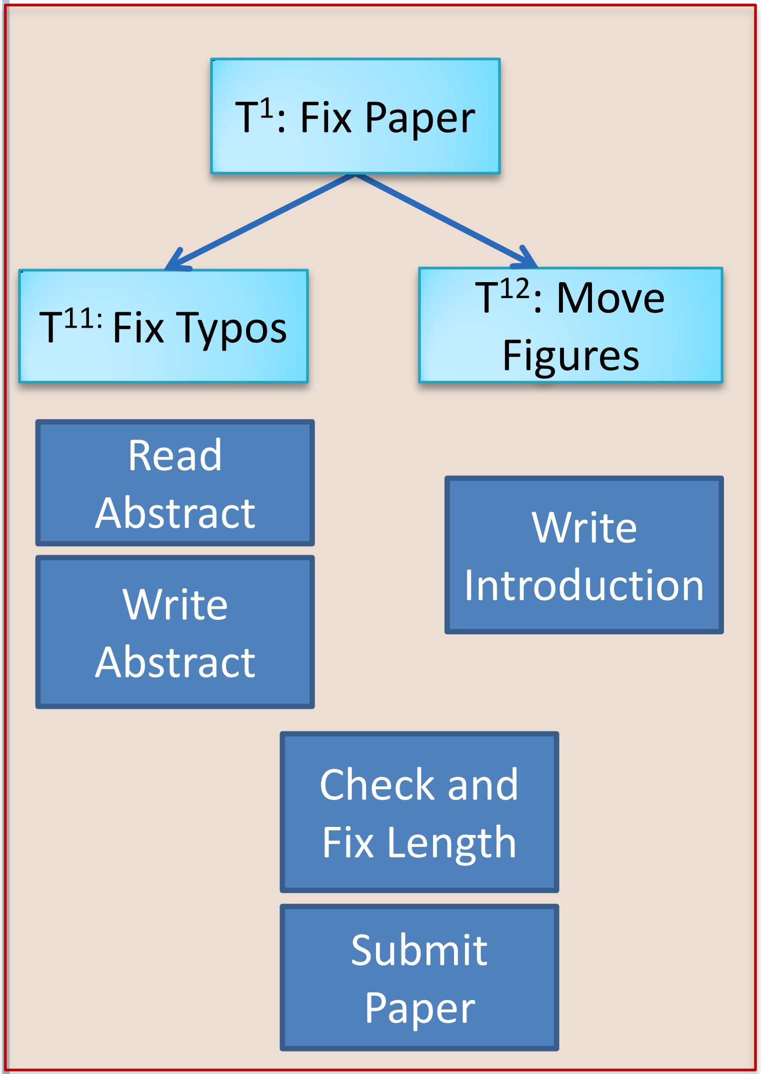
Sub-transactions do not guarantee consistency and their results are not durable

They are atomic and serializable wrt to each other

They get locks from parent and release locks to parent locks and write to parent uncommitted data

Top level transaction gets shared object locks

Concurrency of Parent



Parent may wait until sub-transactions finish

A la Java (Mesa) thread join

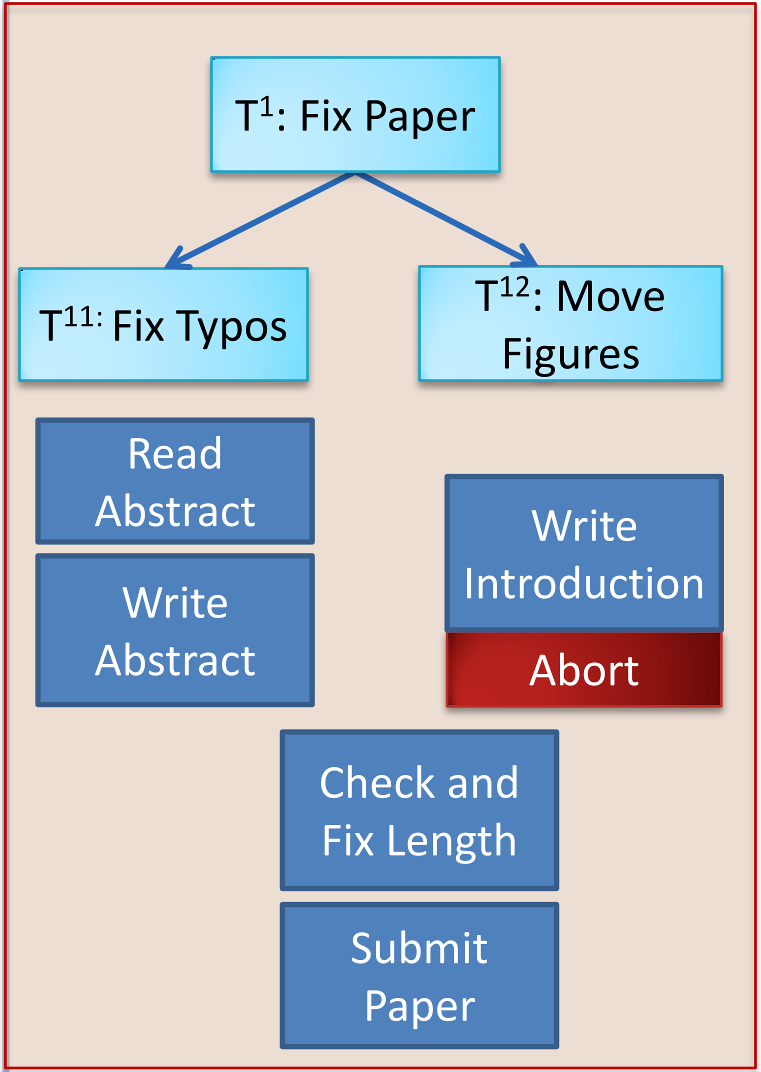
Needed in this example

Parent may execute in parallel

Subtransactions not serializable wrt to parent

Ignore parent locks (but not versa) and override parent writes

Abort Semantics



T1: Fix Paper

T11: Fix Typos

T12: Move Figures

Read Abstract

Write Abstract

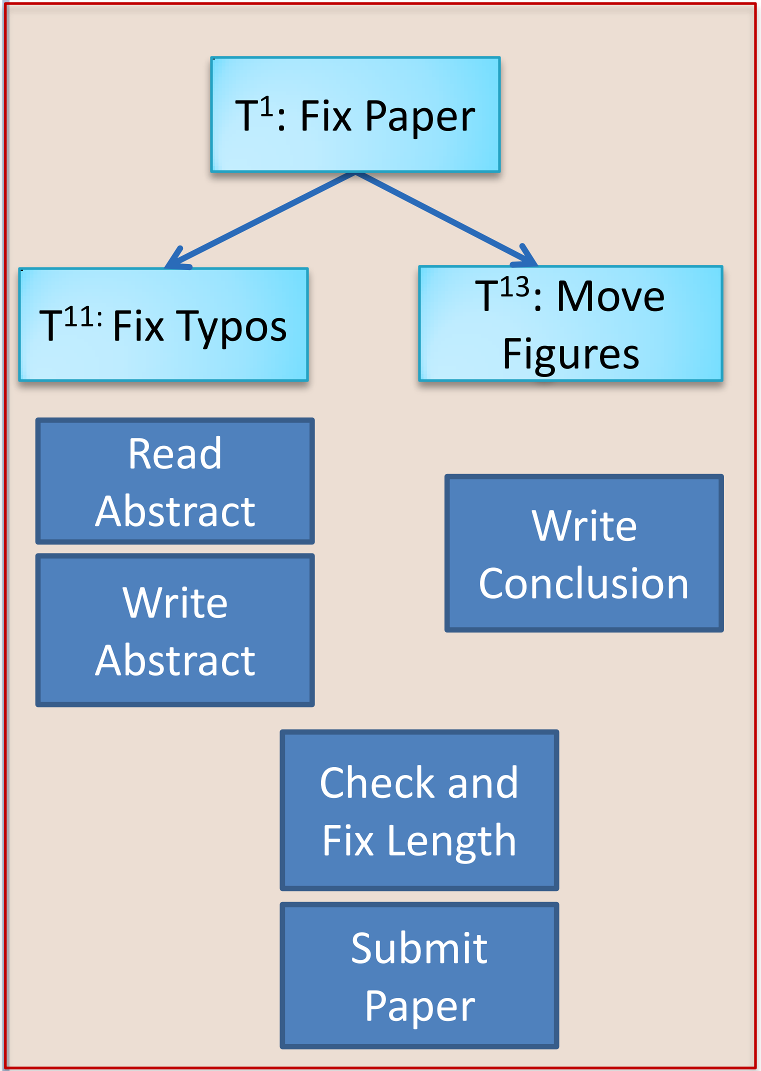
Write Introduction

Abort

Check and Fix Length

Submit Paper

Different Alternative Transaction

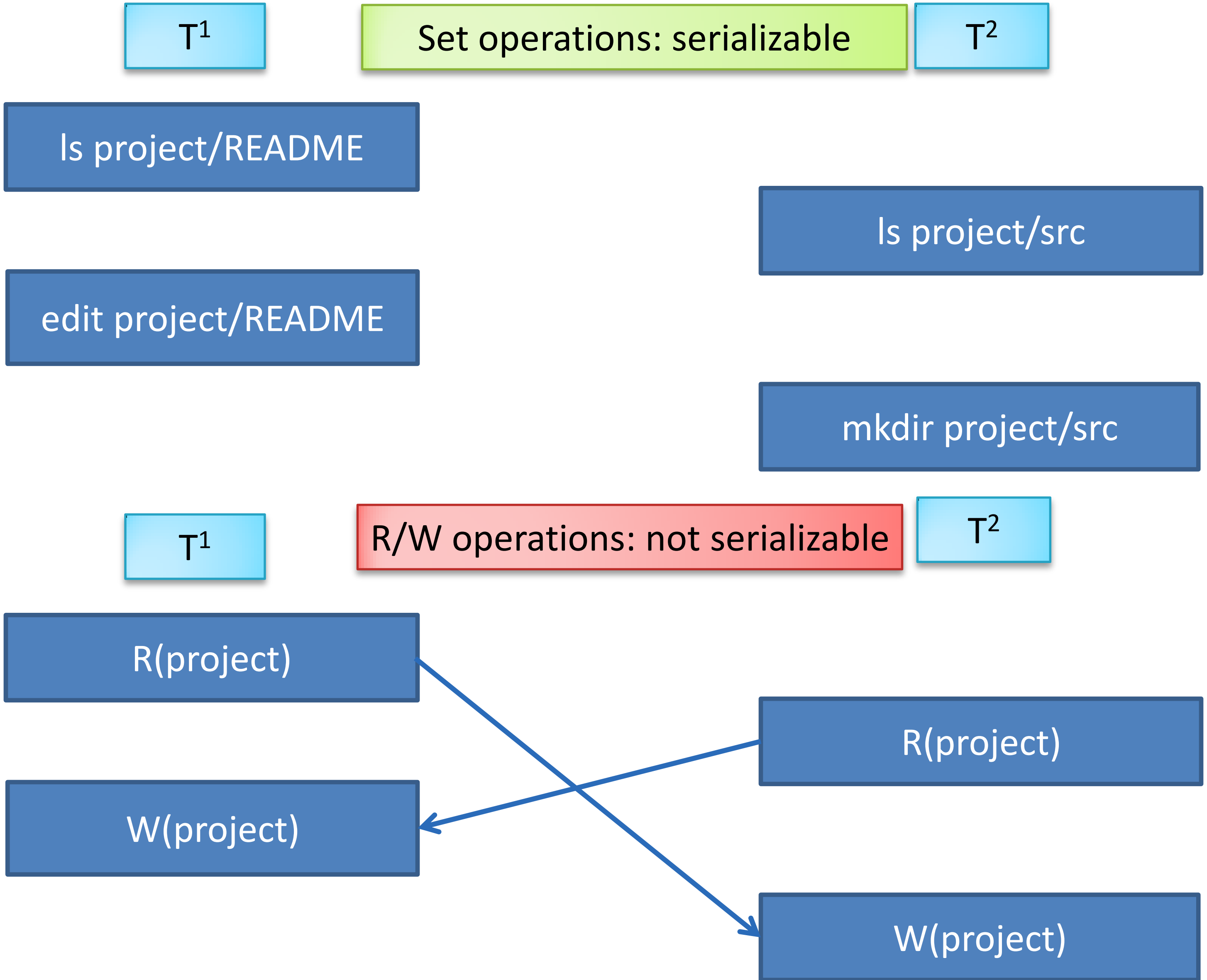


Child aborts do not abort parent transaction, a parent can try alternative transactions

NESTED TRANSACTIONS

- Like top-level, atomic and isolated wrt to siblings in transaction tree
- Not unit of consistency or durability
- Actions do not conflict with parent's transactions.
- In lock-based systems, can get a lock from parent in weaker mode and then release lock to parent
- In optimistic schemes they write to parent's data set
- Parent's actions conflict with child if parent executes in parallel
- Child abort does not abort the parent, which can try alternative sub-transactions

Type Specific Operations



T^1

Set operations: serializable

T^2

ls project/README

ls project/src

edit project/README

mkdir project/src

T^1

R/W operations: not serializable

T^2

R(project)

R(project)

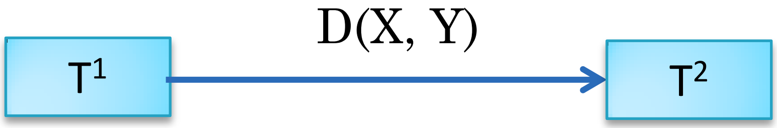
W(project)

W(project)

Type-Specific Serializability

- Modeling ls as read and mkdir as write leads to directory-independent, non-serializable case
- Using type-specific semantics leads to serializable case

Transaction Graph



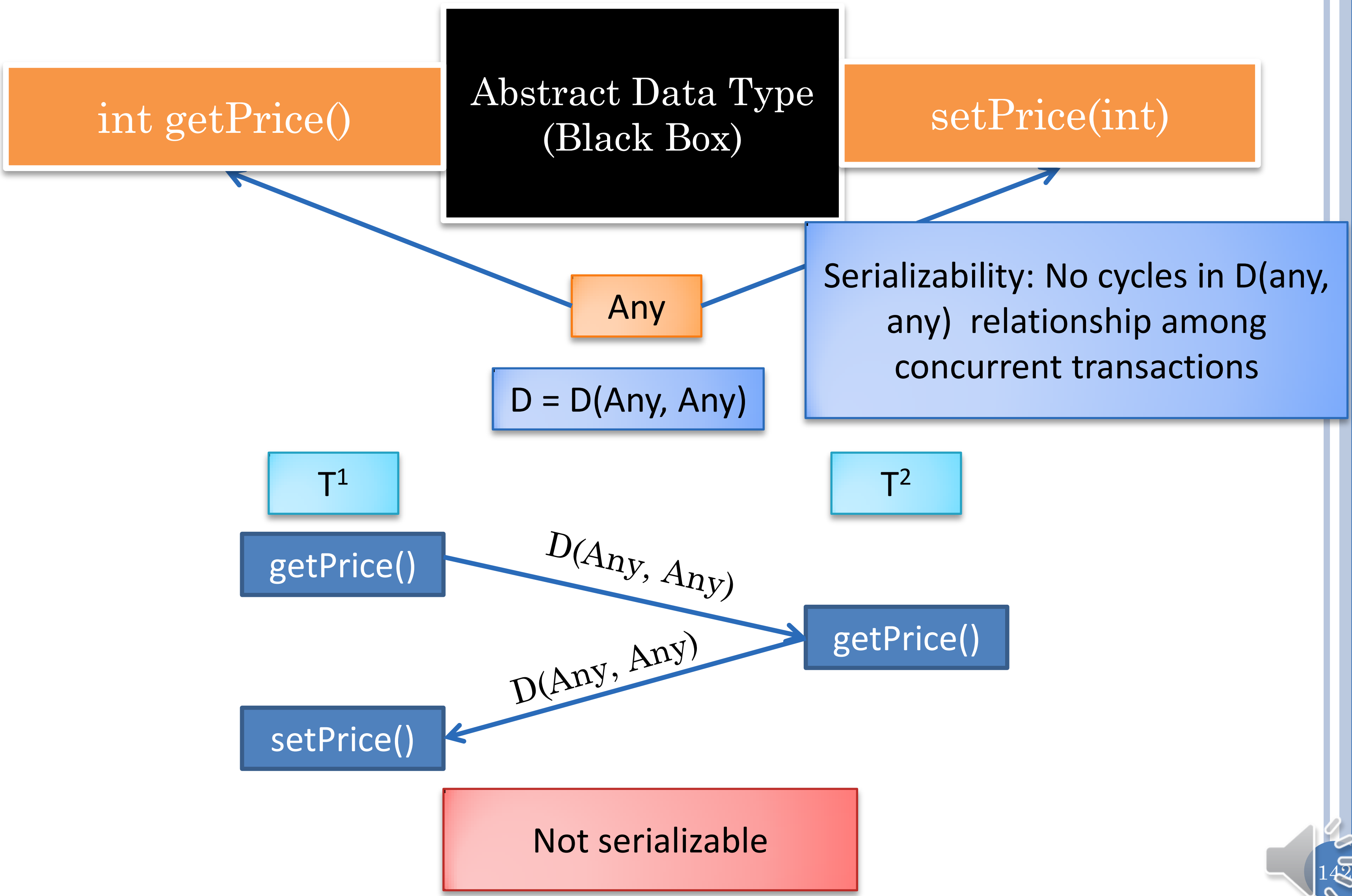
T^1

D(X, Y)

T^2

T^1 performs operation X before T^2 does operation Y

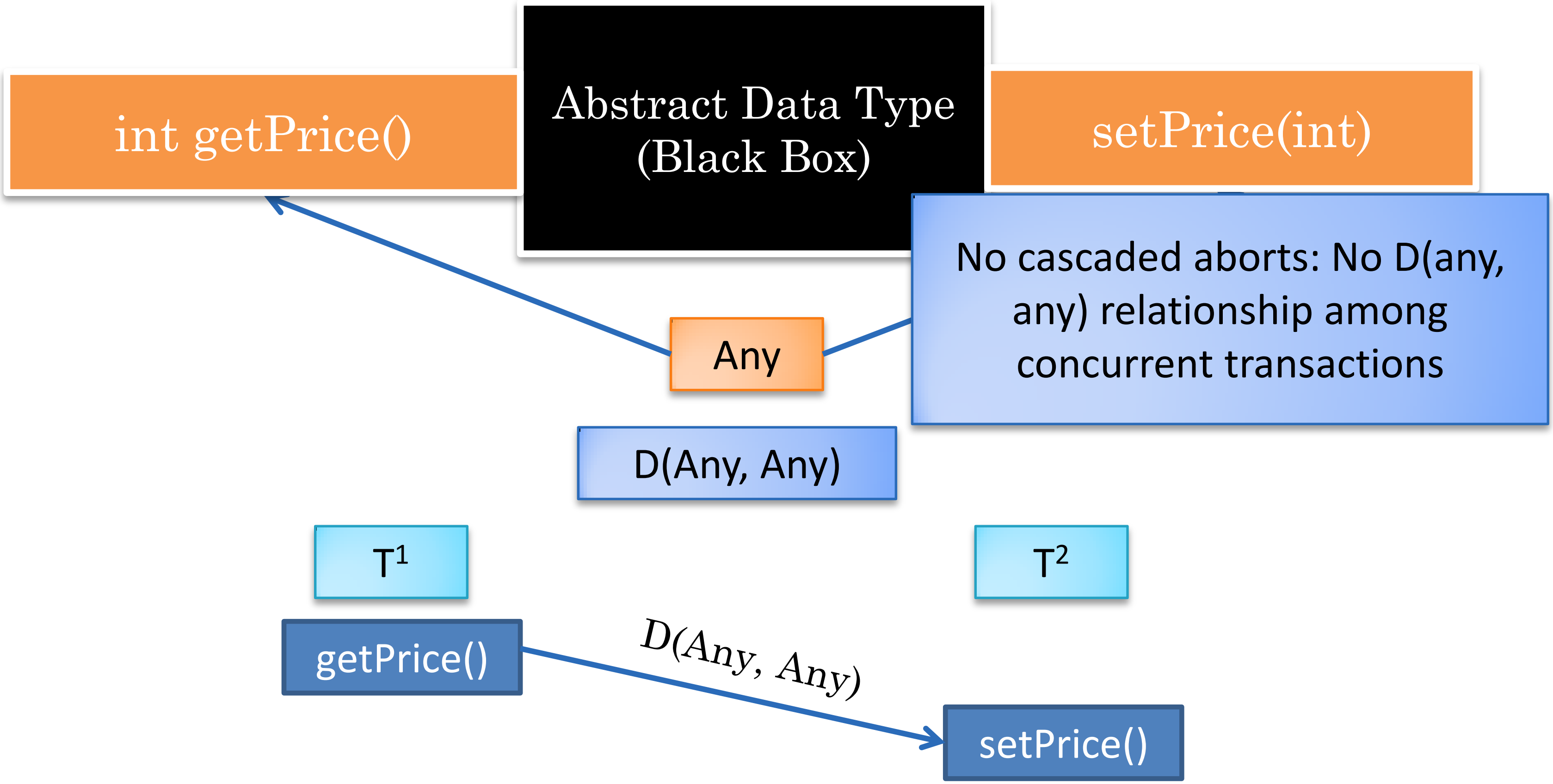
Modeling an Object as a Blackbox: Serializability



Serializability: No cycles in D(any, any) relationship among concurrent transactions

Not serializable

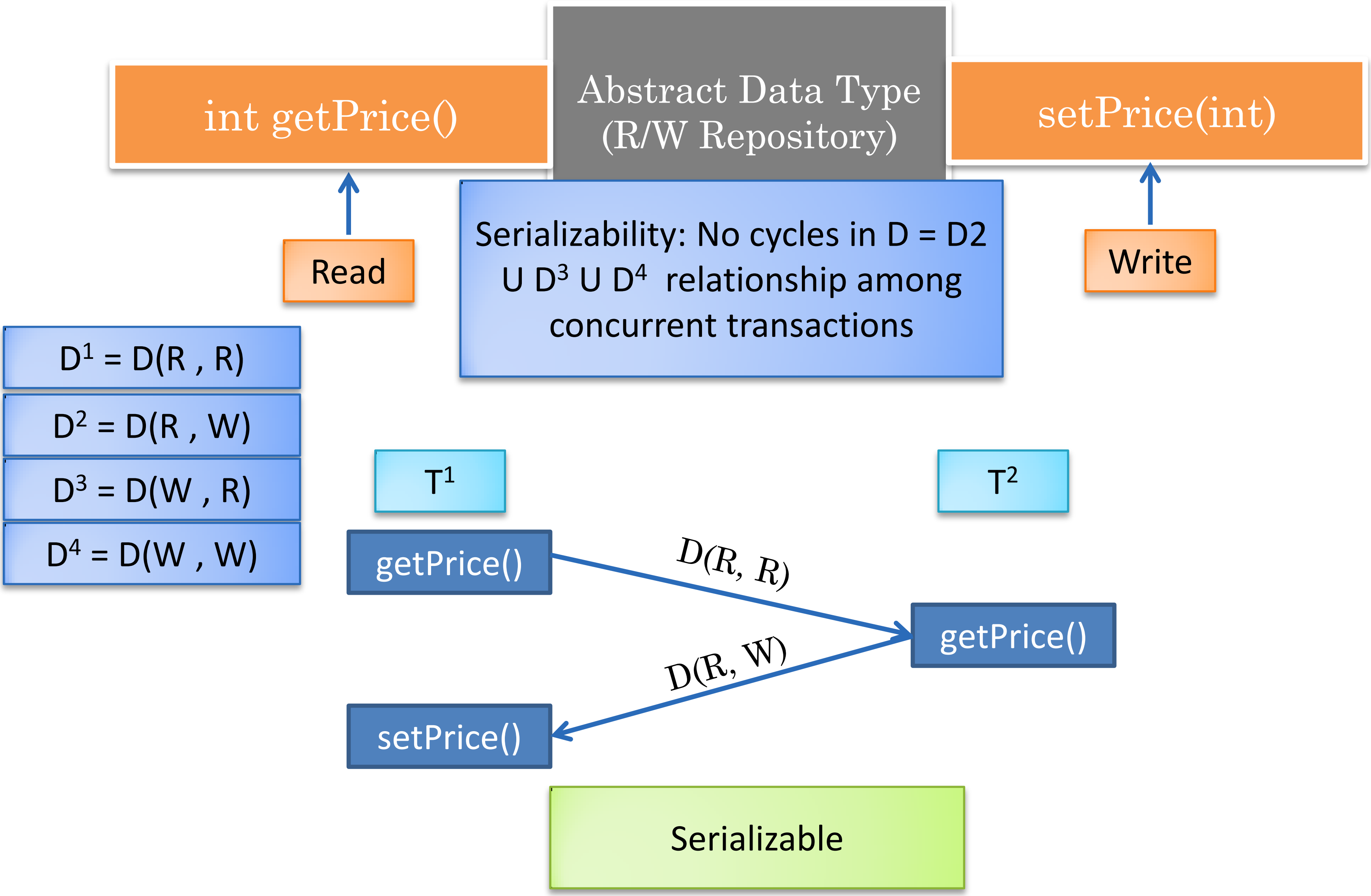
Modeling an Object as a Blackbox: Prevent Cascaded Aborts



No cascaded aborts: No D(any, any) relationship among concurrent transactions

Not allowed if cascaded aborts are to be avoided

Modeling an Object as a Read/Write Repository: Serializability



int getPrice()

Abstract Data Type (R/W Repository)

setPrice(int)

Read

Serializability: No cycles in D = D2 U D^3 U D^4 relationship among concurrent transactions

Write

$D^1 = D(R , R)$

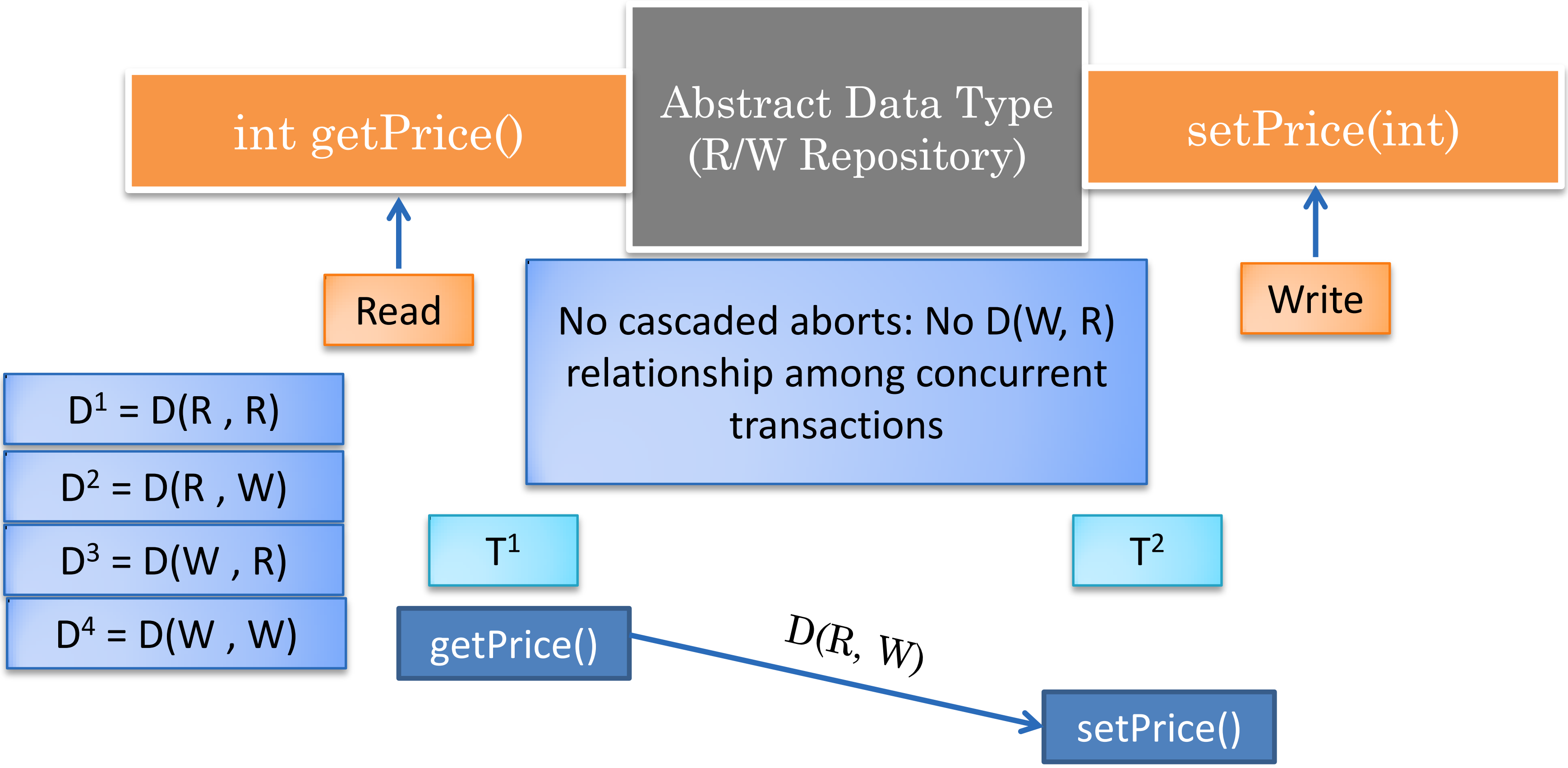
$D^2 = D(R , W)$

$D^3 = D(W , R)$

$D^4 = D(W , W)$

Serializable

Modeling an Object as a Blackbox: Prevent Cascaded Aborts



int getPrice()

Abstract Data Type (R/W Repository)

setPrice(int)

Read

Write

No cascaded aborts: No D(W, R) relationship among concurrent transactions

D^1 = D(R , R)

D^2 = D(R , W)

D^3 = D(W , R)

D^4 = D(W , W)

T^1

T^2

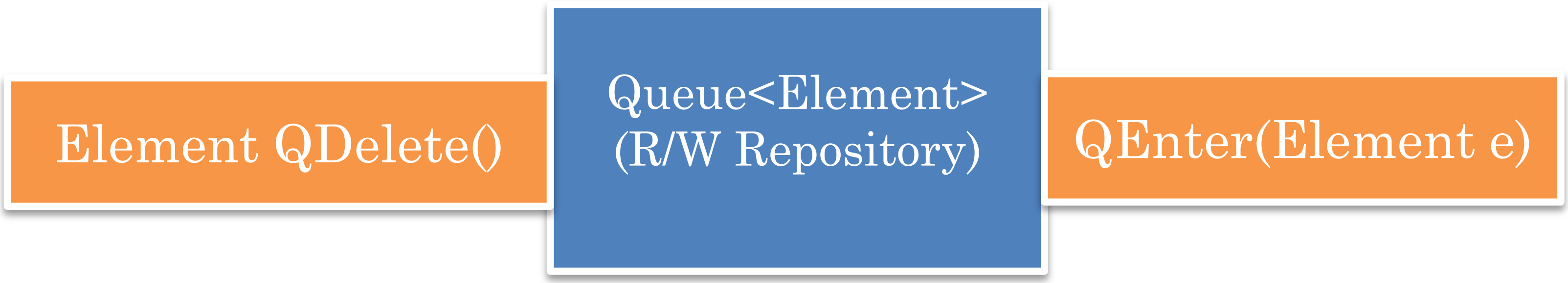
getPrice()

D(R, W)

setPrice()

Allowed if cascaded aborts are to be avoided

Queue



Element QDelete()

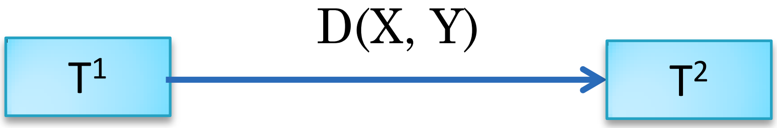
Queue<Element>
(R/W Repository)

QEnter(Element e)

E(X)	QEnter(X)
D(X)	X = QDelete()

Assume each element has a unique id assigned when it is entered into queue

Transaction Graph (Review)

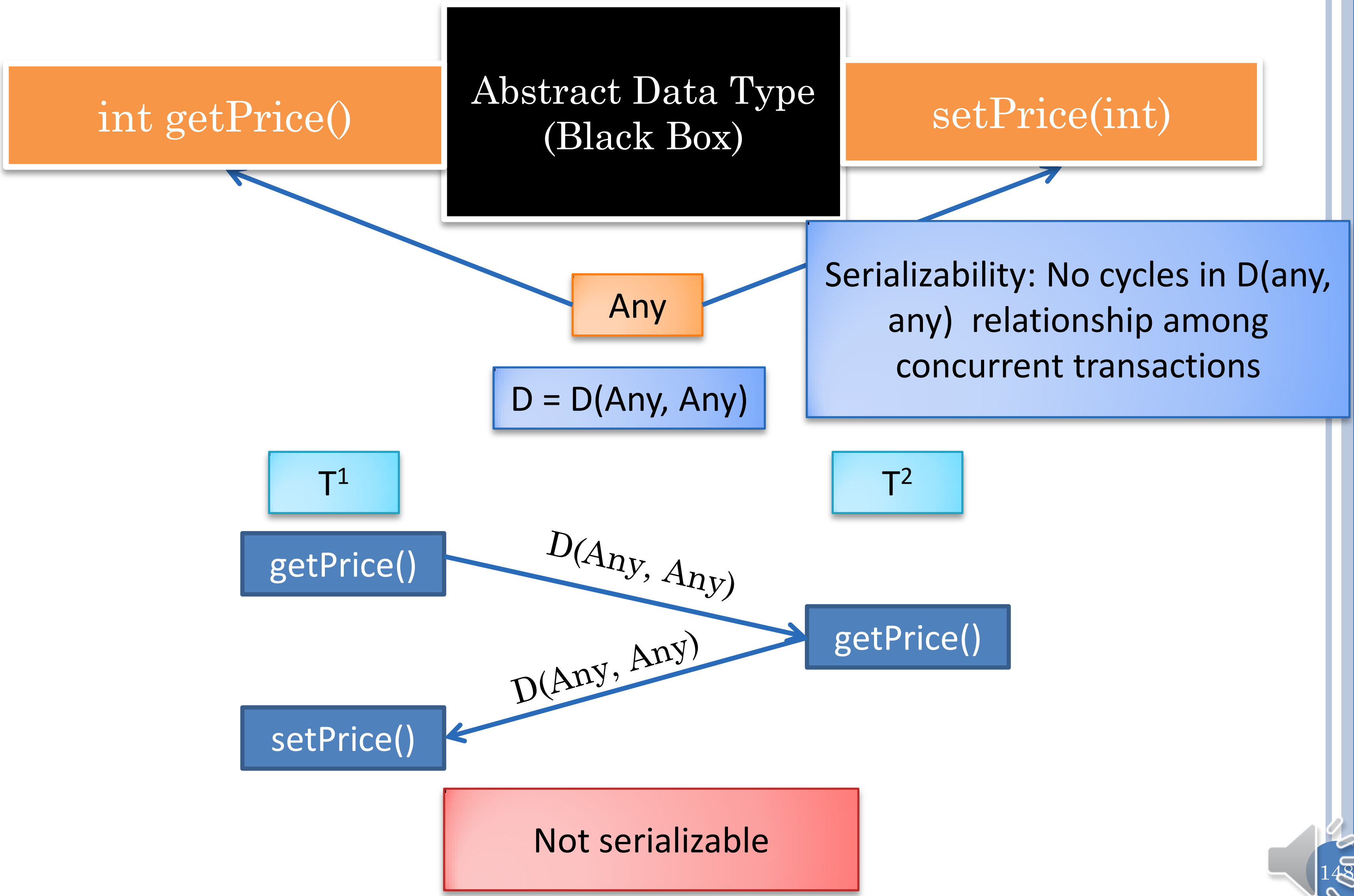


T^1 performs operation X before T^2 does operation Y

How to prevent non serializable transactions?

How to prevent cascaded aborts

Modeling an Object as a Blackbox: Serializability (Review)



Abstract Data Type (Black Box)

int getPrice()

setPrice(int)

Any

Serializability: No cycles in D(any, any) relationship among concurrent transactions

D = D(Any, Any)

T^1

T^2

getPrice()

D(Any, Any)

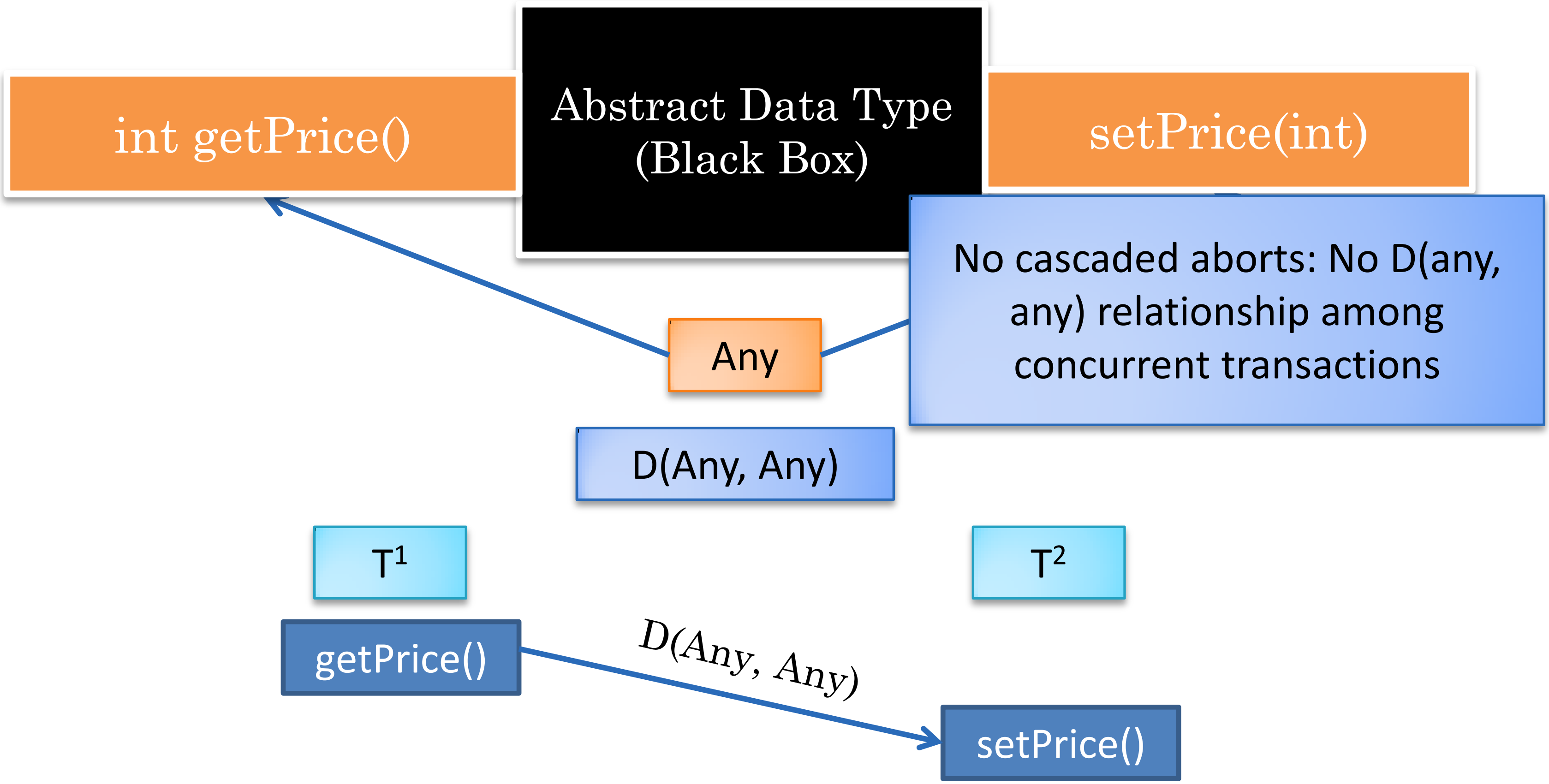
getPrice()

D(Any, Any)

setPrice()

Not serializable

Modeling an Object as a Blackbox: Prevent Cascaded Aborts (Review)



int getPrice()

Abstract Data Type (Black Box)

setPrice(int)

No cascaded aborts: No D(any, any) relationship among concurrent transactions

Any

D(Any, Any)

T1

T2

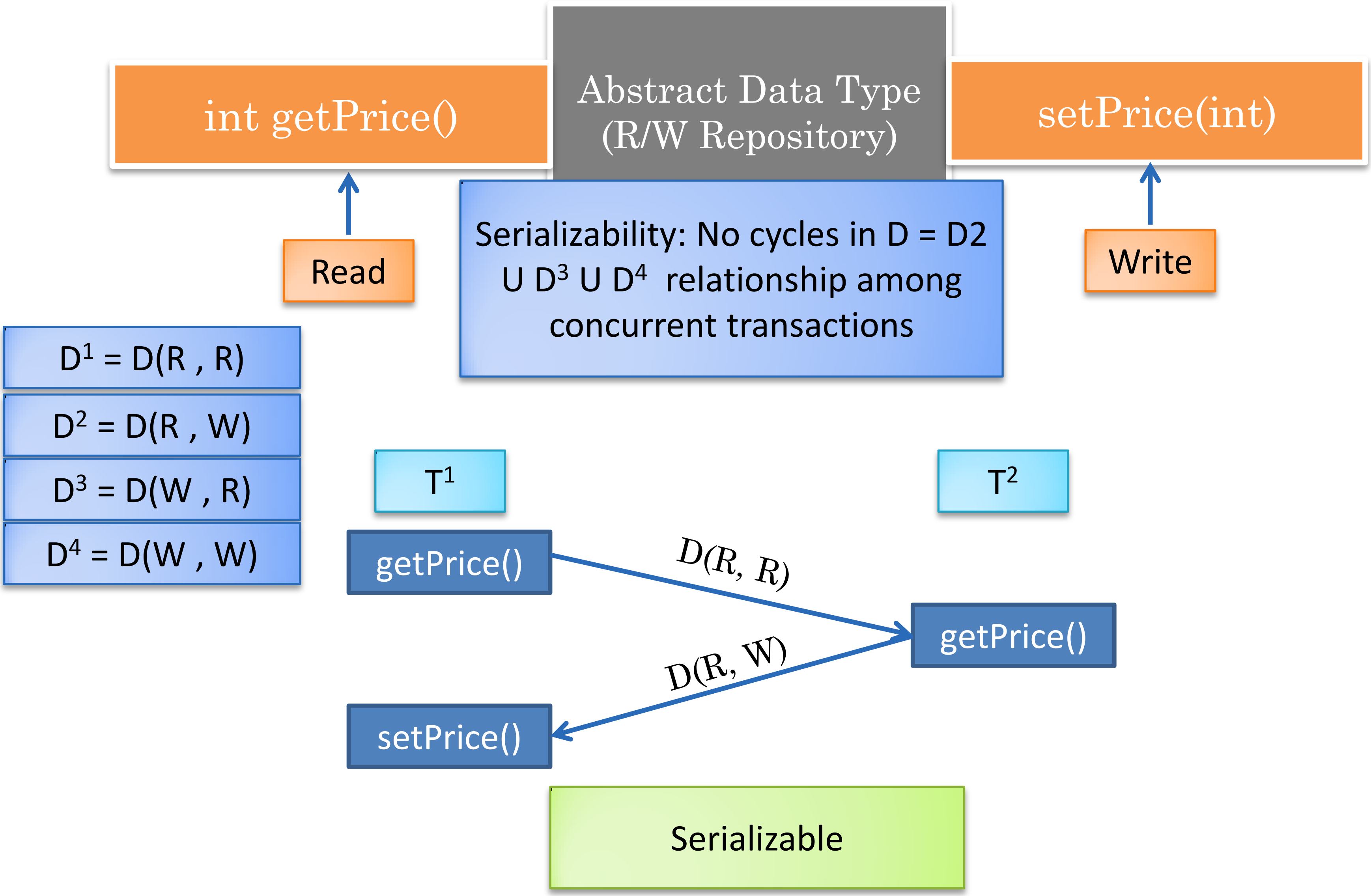
getPrice()

D(Any, Any)

setPrice()

Not allowed if cascaded aborts are to be avoided

Modeling an Object as a Read/Write Repository: Serializability (Review)



int getPrice()

Abstract Data Type (R/W Repository)

setPrice(int)

Read

Serializability: No cycles in D = D2 U D^3 U D^4 relationship among concurrent transactions

Write

D^1 = D(R , R)

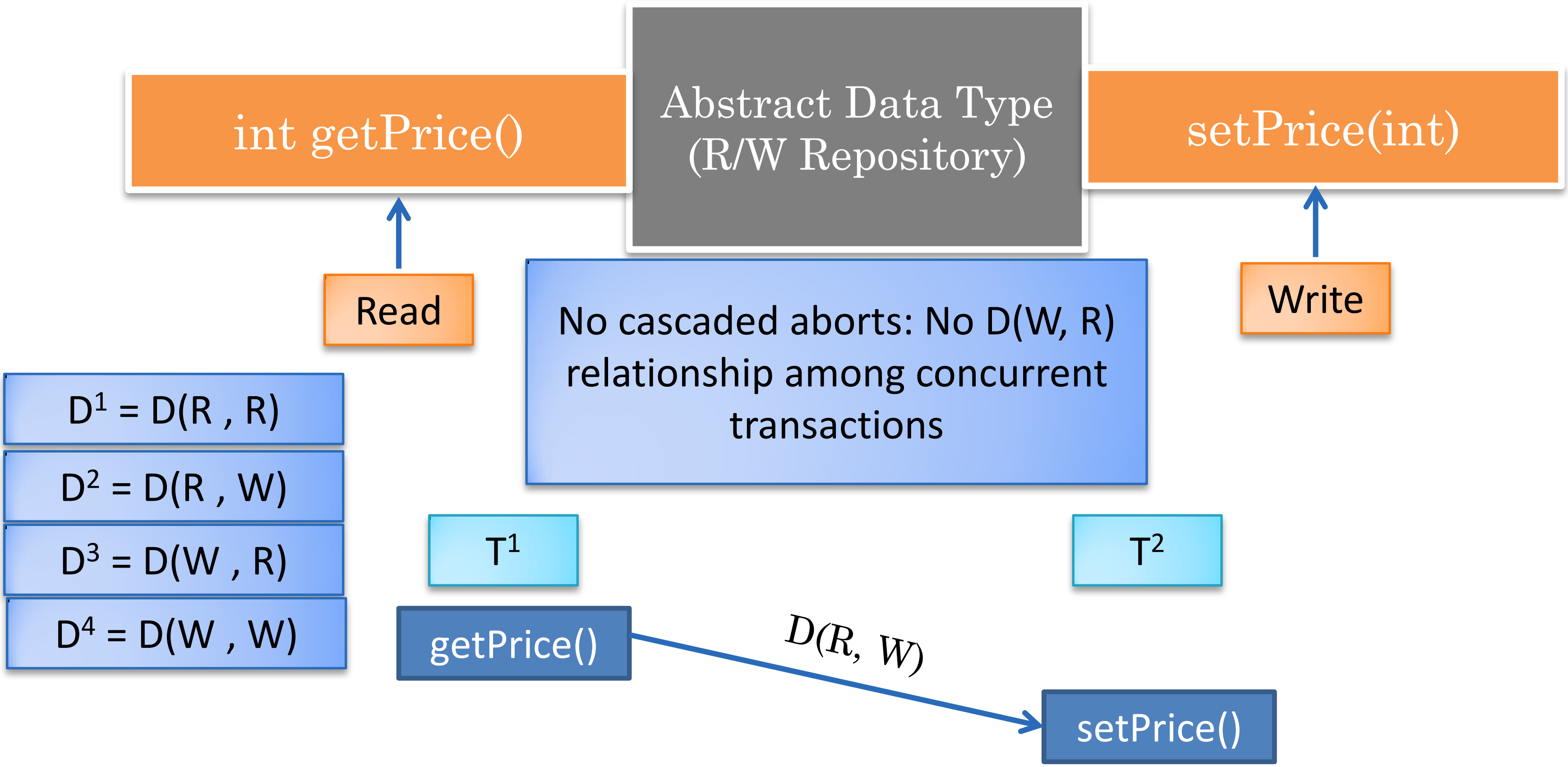
D^2 = D(R , W)

D^3 = D(W , R)

D^4 = D(W , W)

Serializable

Modeling an Object as a Blackbox: Prevent Cascaded Aborts (Review)



int getPrice()

Abstract Data Type (R/W Repository)

setPrice(int)

Read

Write

No cascaded aborts: No D(W, R) relationship among concurrent transactions

D^1 = D(R , R)

D^2 = D(R , W)

D^3 = D(W , R)

D^4 = D(W , W)

T^1

T^2

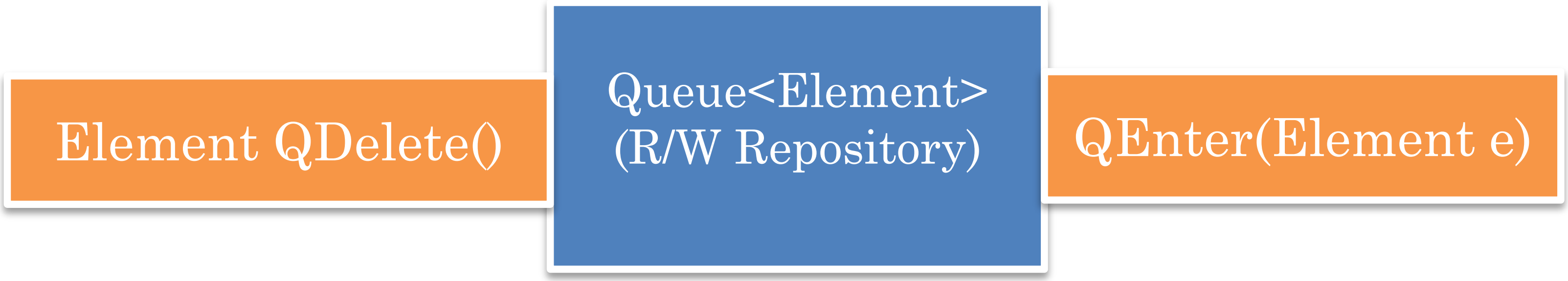
getPrice()

D(R, W)

setPrice()

Allowed if cascaded aborts are to be avoided

Queue (Review)



Element QDelete()

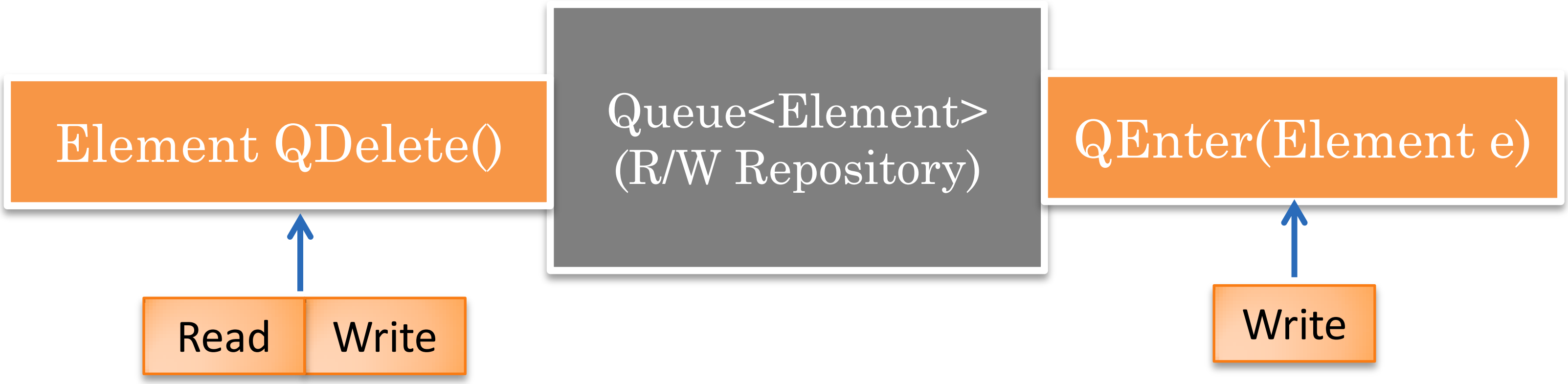
Queue<Element>
(R/W Repository)

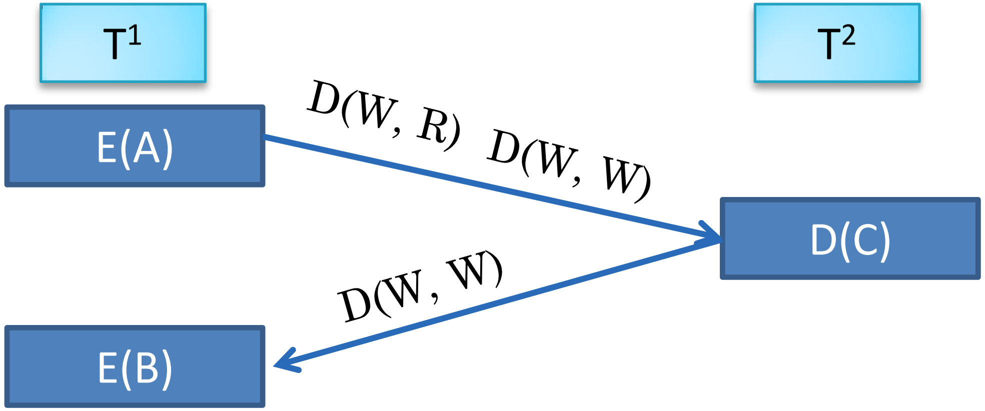
QEnter(Element e)

E(X)	QEnter(X)
D(X)	X = QDelete()

Assume each element has a unique id assigned when it is entered into queue

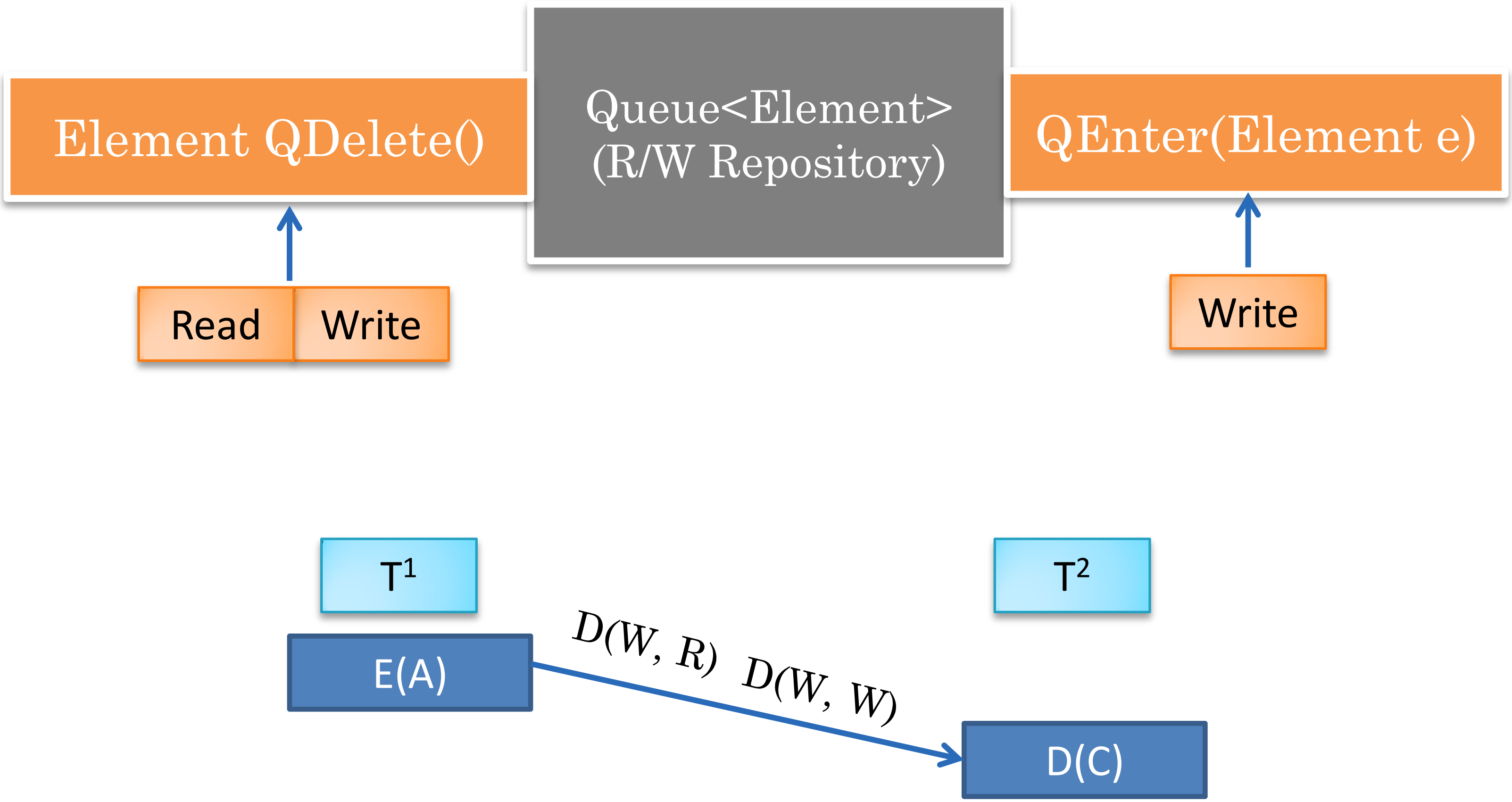
Modeling an Object as a Read/Write Repository: Serializability





Not serializable even though two transactions working at different ends of the queue

Modeling an Object as a Read/Write Repository: No Cascaded Aborts



Element QDelete()

Queue<Element>
(R/W Repository)

QEnter(Element e)

Read Write

Write

T^1

T^2

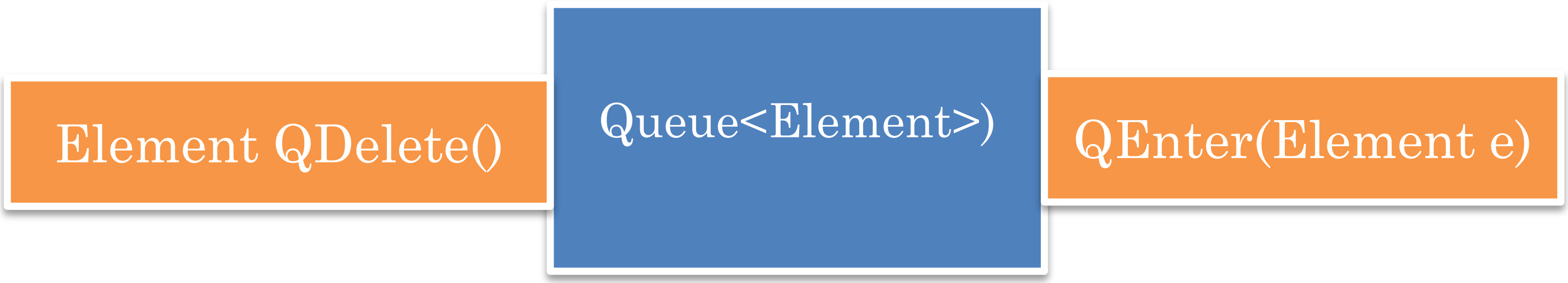
E(A)

D(W, R) D(W, W)

D(C)

Serializable but not allowed if cascaded aborts are to be avoided because of D(W, R) dependency

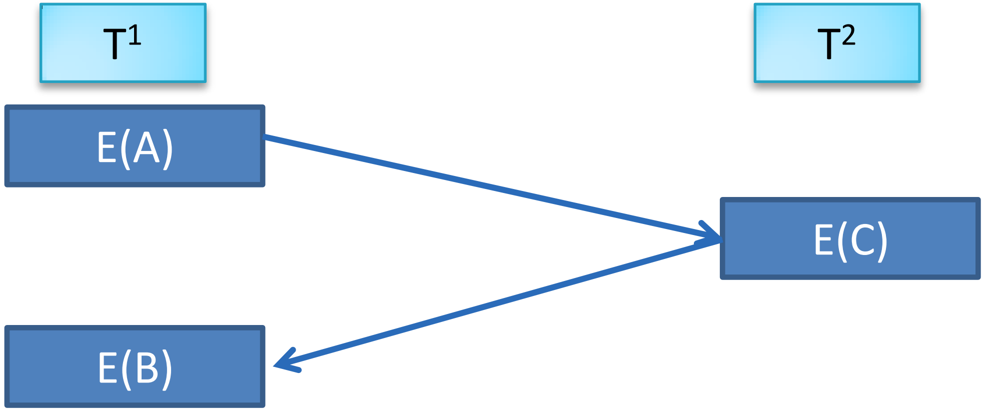
Supporting Queue Operations Directly



Element QDelete()

Queue<Element>)

QEnter(Element e)



T^1

T^2

E(A)

E(C)

E(B)

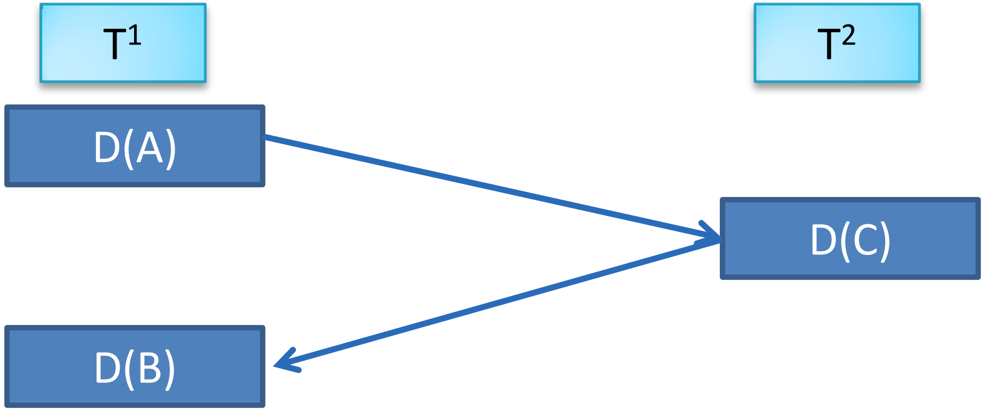
Not serializable because in a serial schedule all elements of one transaction will be queued before or after another

Supporting Queue Operations Directly

Element QDelete()

Queue<Element>)

QEnter(Element e)



T1

T2

D(A)

D(C)

D(B)

Not serializable because in a serial schedule all elements of one transaction will be dequeued before or after another

Modeling Queue Operations

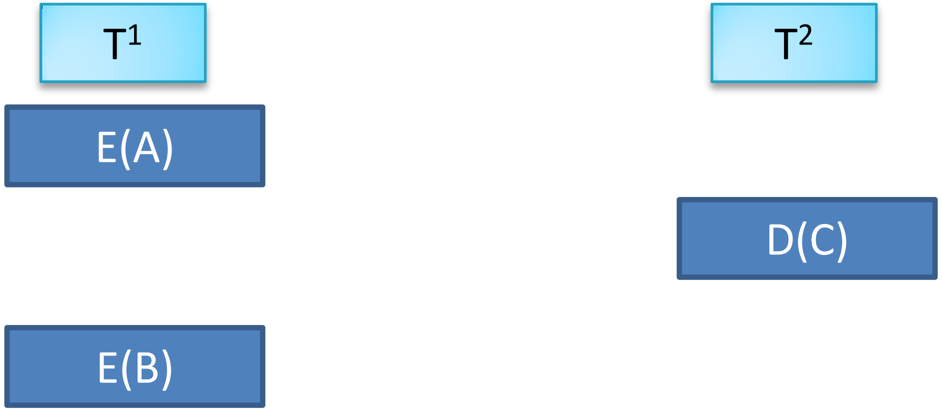
Element QDelete()

Queue<Element>)

QEnter(Element e)

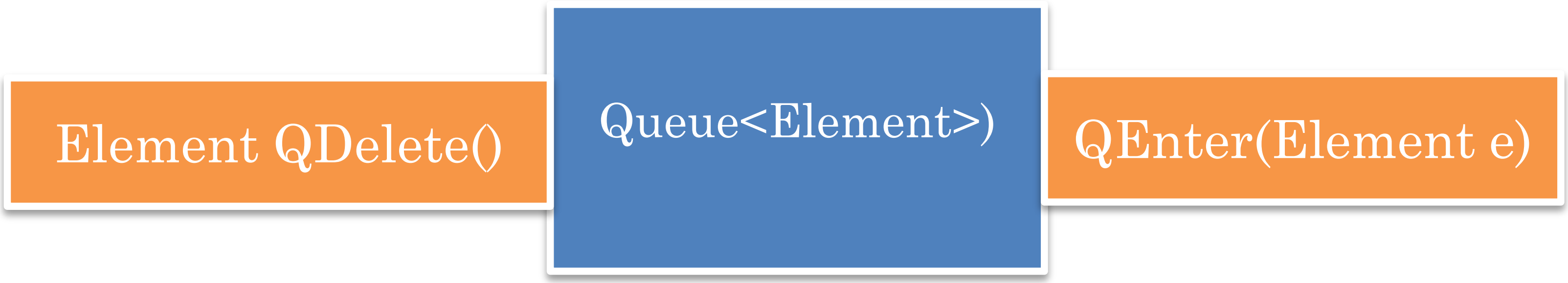
The element dequeued by T^2 is not an element queued by T^1

We know that based on argument and return values of transactions



Serializable because two transactions working at different ends of the queue

Supporting Queue Operations Directly



Element QDelete()

Queue<Element>)

QEnter(Element e)

D^1 = E(e) , E(e’)

D^2 = E(e) , D(e’)

D^3 = E(e) , D(e)

D^4 = D(e) , E(e’)

D^5= D(e) , D(e’)

Serializability?

Serializability: No cycles in D = D^1 U D^3 U D^5

Avoiding cascaded aborts?

No D^3 relationship in concurrent transactions

S(hared)/(E)x(clusive) Locks

	E(e)	D(e)	D(null)
E(e)	N/A	N/A	No
E(e’)	No	Yes	No
D(e)	No	N/A	No
D(e’)	Yes	No	No
D(null)	No	No	Yes

Type-Specific Locks

- Less information about operations available to locking system more conservative it is about commuting and dependent operations
- No information
  - Serializability: no interleaving operations on an object
  - Cascaded aborts: no concurrency
- R/W:
  - Serializability: No cycles in D(R,W), D(W,R), D(R,R)
  - No Cascaded aborts: No D(W,R) relationship among concurrent transactions
- Queue-specific:
  - Serializability: Cycles allowed in D(E(e), D(e')) and D(D(e), E(e'))
  - Cascaded aborts: No D(E(e), D(e)) relationship among concurrent transactions
  - A lock specifies kind of operation and element queued or dequeued
  - Kept until end of transaction

Concurrency Control Summary

- Transactions and ACID
- Isolation: serializability and cascaded aborts
- Explicit, implicit Locks
- Locking implementation: Two phase commit, cache incoherence
- Shared vs. exclusive locks
- Two phase locking
- Hierarchical locking and intention locks
- Type-specific dependencies and locking
- Optimistic transactions
- Optimistic locks
- Nested transactions

Extra Slides

S(HARED)/(E)X(CLUSIVE) LOCKS

	E(e)	D(e)
E(e)	N/A	No
E(e')	No	Yes
D(e)	No	N/A
D(e')	Yes	No

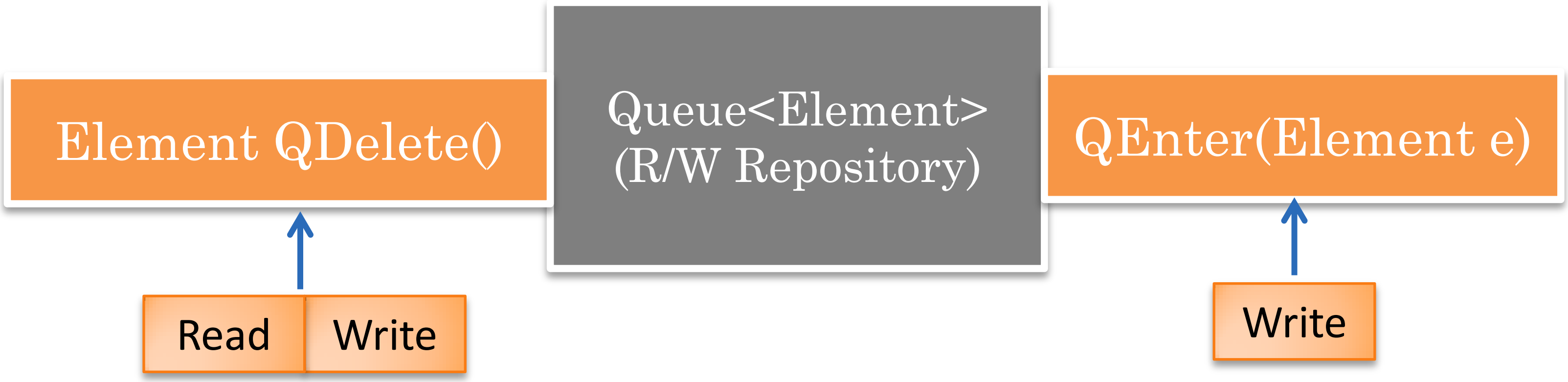
Locks held until end of transaction to:

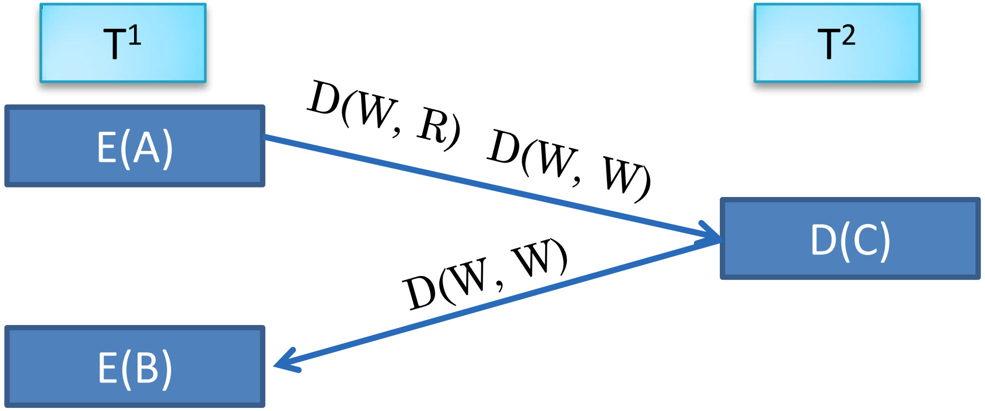
remember which elements have been added removed

prevent cascaded aborts

Not clear we need to remember elements added removed if we have null row and column

Modeling an Object as a Read/Write Repository: Serializability

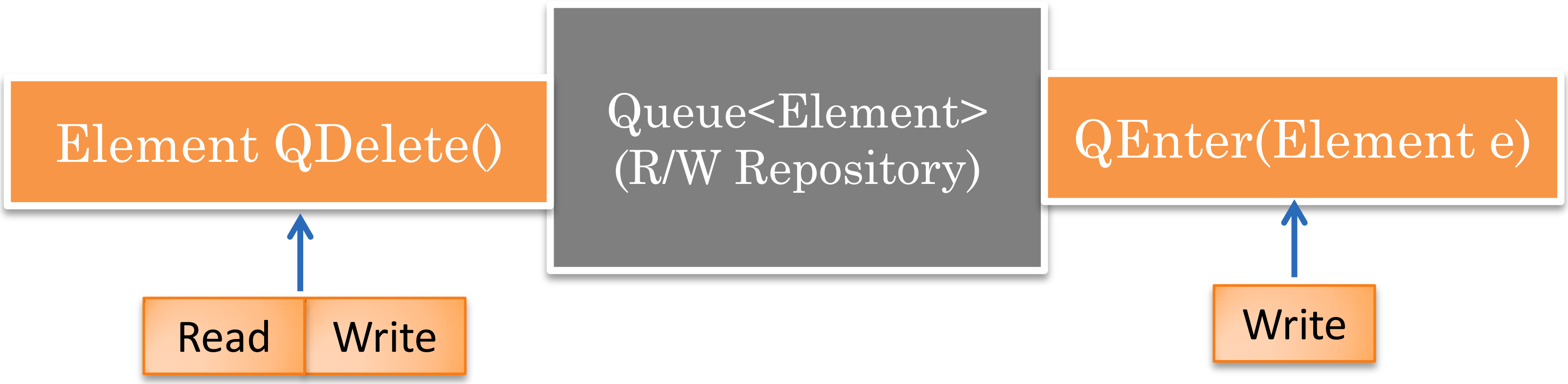


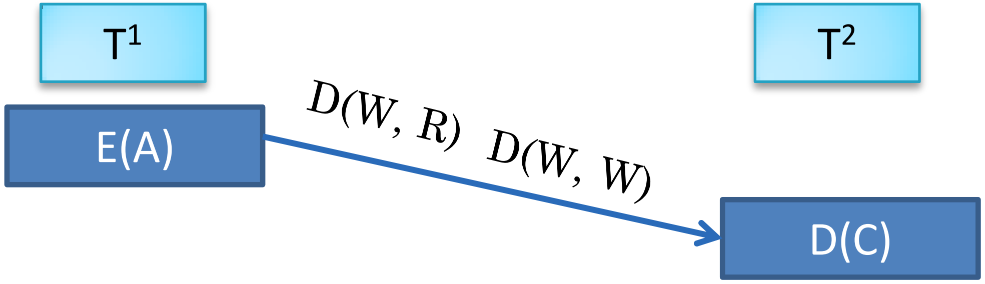


There is indeed a D(W,R) relation between T1 and T2 contrary to what the recording says

Not serializable even though two transactions working at different ends of the queue

Modeling an Object as a Read/Write Repository: No Cascaded Aborts

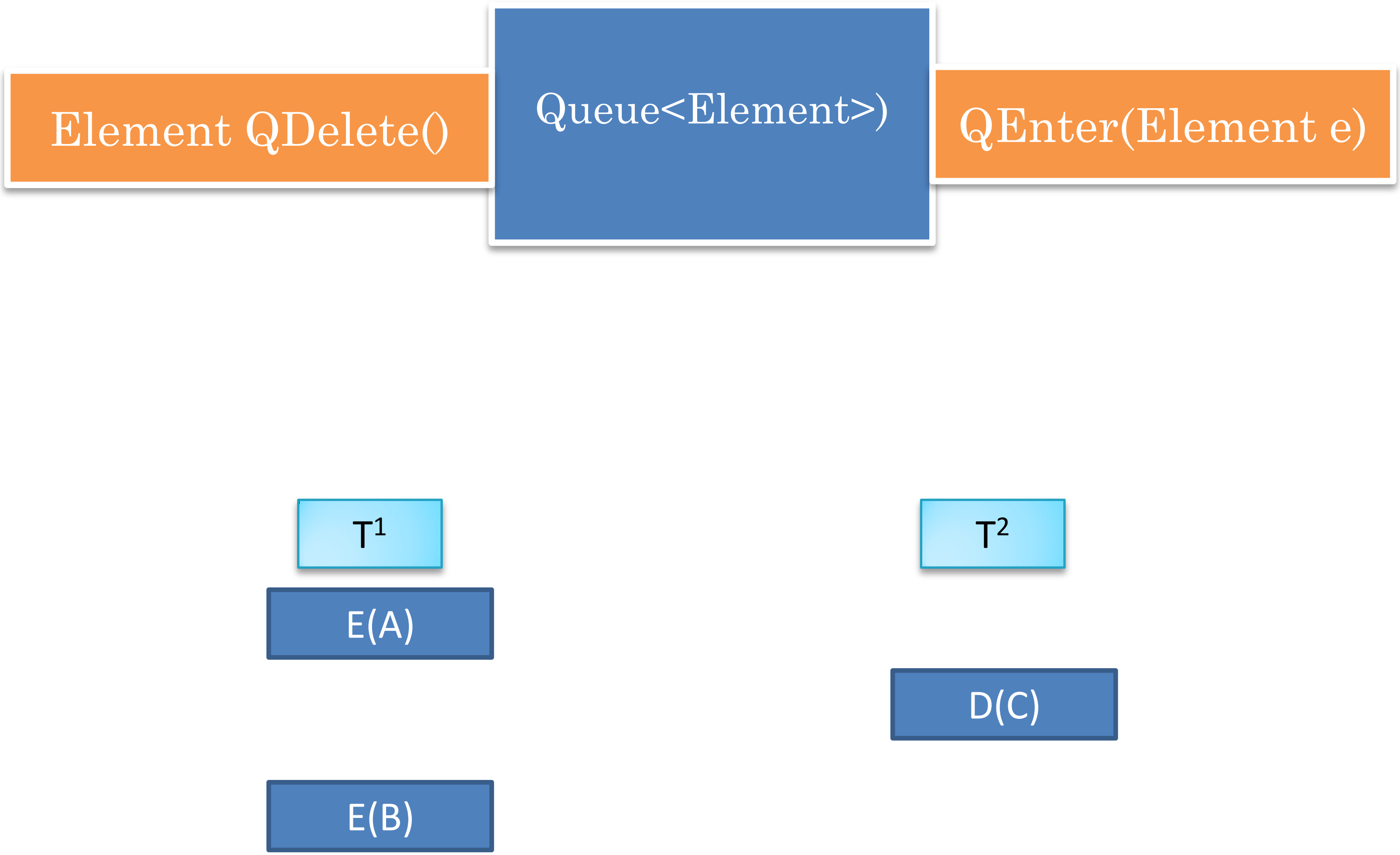




There is indeed a D(W,R) relation between T¹ and T² contrary to what the recording says and so under R/W semantics it wull be disallowed

Serializable but not allowed if cascaded aborts are to be avoided because of D(W, R) dependency

Supporting Queue Operations Directly



Element QDelete()

Queue<Element>)

QEnter(Element e)

T^1

T^2

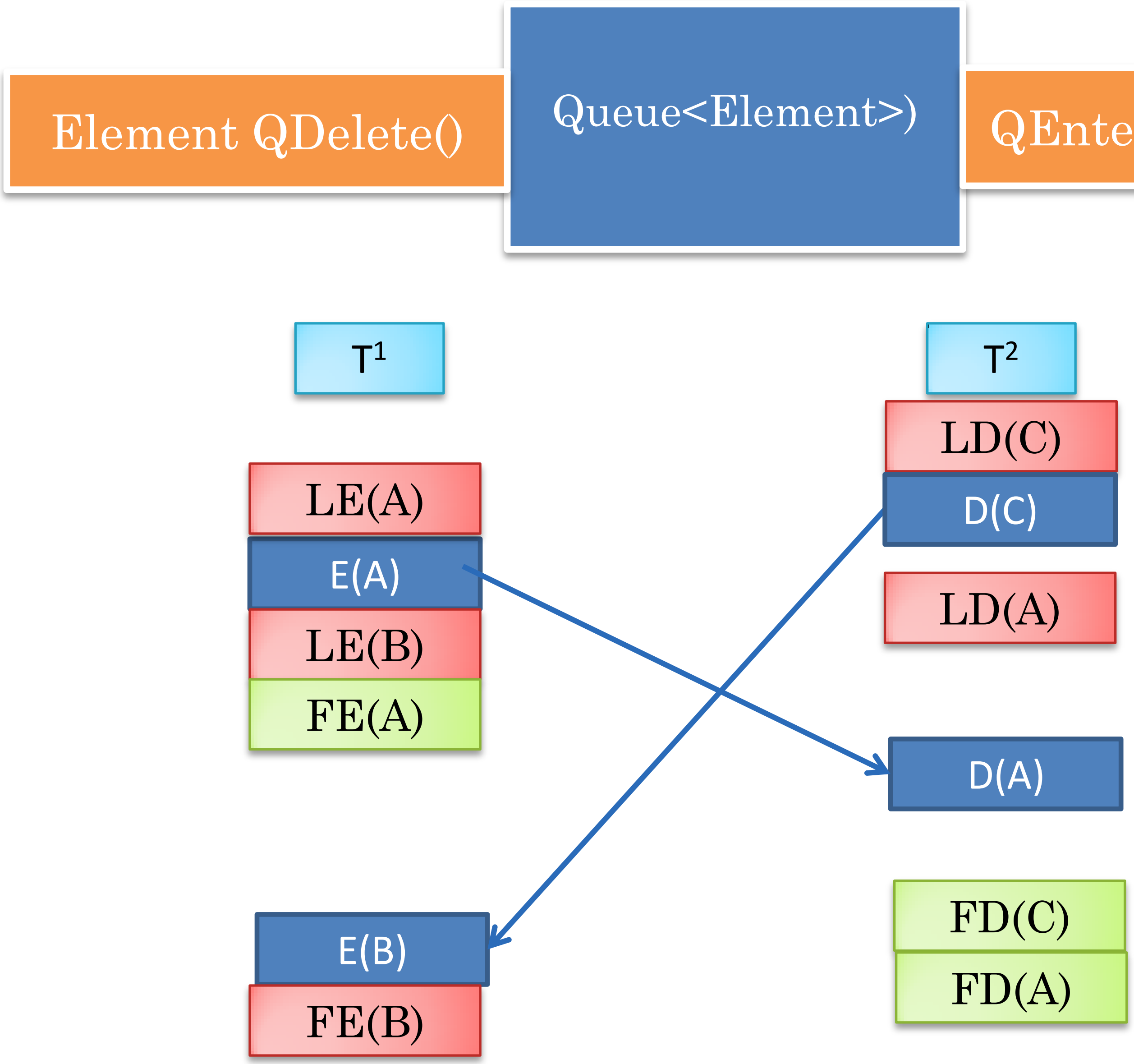
E(A)

D(C)

E(B)

Serializable because two transactions working at different ends of the queue

Not Holding Lock Until End of Transaction



Element QDelete()

Queue<Element>)

QEnter(Element e)

T^1

LE(A)

E(A)

LE(B)

FE(A)

E(B)

FE(B)

T^2

LD(C)

D(C)

LD(A)

D(A)

FD(C)

FD(A)

S(hared)/(E)x(clusive) Locks

	E(e)	D(e)	D(null)
E(e)	N/A	N/A	No
E(e’)	No	Yes	No
D(e)	No	N/A	No
D(e’)	Yes	No	No
D(null)	No	No	Yes

S(hared)/(E)x(clusive) Locks

	E(e)	D(e)	D(null)
E(e)	N/A	N/A	No
E(e’)	No	Yes	No
D(e)	No	N/A	N/A
D(e’)	Yes	No	N/A
D(null)	N/A	No	Yes

S(hared)/(E)x(clusive) Locks

	E(e)	D(e)	D(null)
E(e)	N/A	N/A	No
E(e’)	No	Yes	No
D(e)	No	N/A	N/A
D(e’)	Yes	No	N/A
D(null)	N/A	No	Yes

S(hared)/(E)x(clusive) Locks

	E(e)	D(e)	D(null)
E(e)	N/A	N/A	No
E(e’)	No	Yes	No
D(e)	No	N/A	Yes
D(e’)	Yes	No	Yes
D(null)	Yes	No	Yes

Assume dequeue is non blocking

S(hared)/(E)x(clusive) Locks

	E(e)	D(e)	D(null)
E(e)	N/A	N/A	No
E(e’)	No	Yes	No
D(e)	No	N/A	Yes
D(e’)	Yes	No	Yes
D(null)	Yes	No	Yes

Assume dequeue is non blocking