UNIVERSITY OF
Southampton

UNIVERSITY OF Southampton

Distributed Databases

COMP3211 Advanced Databases

Dr Nicholas Gibbins – nmg@ecs.soton.ac.uk

2020-2021

UNIVERSITY OF Southampton

Overview

Fragmentation

- Horizontal (primary and derived), vertical, hybrid

Query processing

- Localisation, optimisation (semijoins)

Concurrency control

- Centralised 2PL, Distributed 2PL, deadlock

Reliability

- Two Phase Commit (2PC)

Replication

The CAP Theorem

UNIVERSITY OF Southampton

What is a distributed database?

A collection of sites connected by a communications network

Each site is a database system in its own right, but the sites have agreed to work together

A user at any site can access data anywhere as if data were all at the user's own site

DDBMS Principles

Local autonomy

The sites in a distributed database system should be autonomous or independent of each other

Each site should provide its own security, locking, logging, integrity, and recovery. Local operations use and affect only local resources and do not depend on other sites

UNIVERSITY OF Southampton

No reliance on a central site

A distributed database system should not rely on a central site, which may be a single point of failure or a bottleneck

Each site of a distributed database system provides its own security, locking, logging, integrity, and recovery, and handles its own data dictionary. No central site must be involved in every distributed transaction.

Continuous operation

A distributed database system should never require downtime

A distributed database system should provide on-line backup and recovery, and a full and incremental archiving facility. The backup and recovery should be fast enough to be performed online without noticeable detrimental affect on the entire system performance.

Location independence

Applications should not know, or even be aware of, where the data are physically stored; applications should behave as if all data were stored locally

Location independence allows applications and data to be migrated easily from one site to another without modifications.

Fragmentation independence

Relations can be divided into fragments and stored at different sites

Applications should not be aware of the fact that some data may be stored in a fragment of a table at a site different from the site where the table itself is stored.

Replication independence

Relations and fragments can be stored as many distinct copies on different sites

Applications should not be aware that replicas of the data are maintained and synchronized automatically.

Distributed query processing

Queries are broken down into component transactions to be executed at the distributed sites

Distributed transaction management

A distributed database system should support atomic transactions

Critical to database integrity; a distributed database system must be able to handle concurrency, deadlocks and recovery.

UNIVERSITY OF Southampton

Hardware independence

A distributed database system should be able to operate and access data spread across a wide variety of hardware platforms

A truly distributed DBMS system should not rely on a particular hardware feature, nor should it be limited to a certain hardware architecture.

University of Southampton

Operating system independence

A distributed database system should be able to run on different operating systems

Network independence

A distributed database system should be designed to run regardless of the communication protocols and network topology used to interconnect sites

Southampton UNIVERSITY OF

DBMS independence

An *ideal* distributed database system must be able to support interoperability between DBMS systems running on different nodes, even if these DBMS systems are unalike

All sites in a distributed database system should use common standard interfaces in order to interoperate with each other.

Distributed Databases vs. Parallel Databases

Distributed Databases

- Local autonomy
- No central site
- Continuous operation
- Location independence
- Fragmentation independence
- Replication independence
- Distributed query processing
- Distributed transactions
- Hardware independence
- Operating system independence
- Network independence
- DBMS independence

UNIVERSITY OF Southampton

Distributed Databases vs. Parallel Databases

Parallel Databases

- ~~Local autonomy~~
- No central site
- Continuous operation
- Location independence
- Fragmentation independence
- Replication independence
- Distributed query processing
- Distributed transactions
- ~~Hardware independence~~
- ~~Operating system independence~~
- ~~Network independence~~
- ~~DBMS independence~~

Fragmentation

Why Fragment?

Fragmentation allows:

- localisation of the accesses of relations by applications
- parallel execution (increases concurrency and throughput)

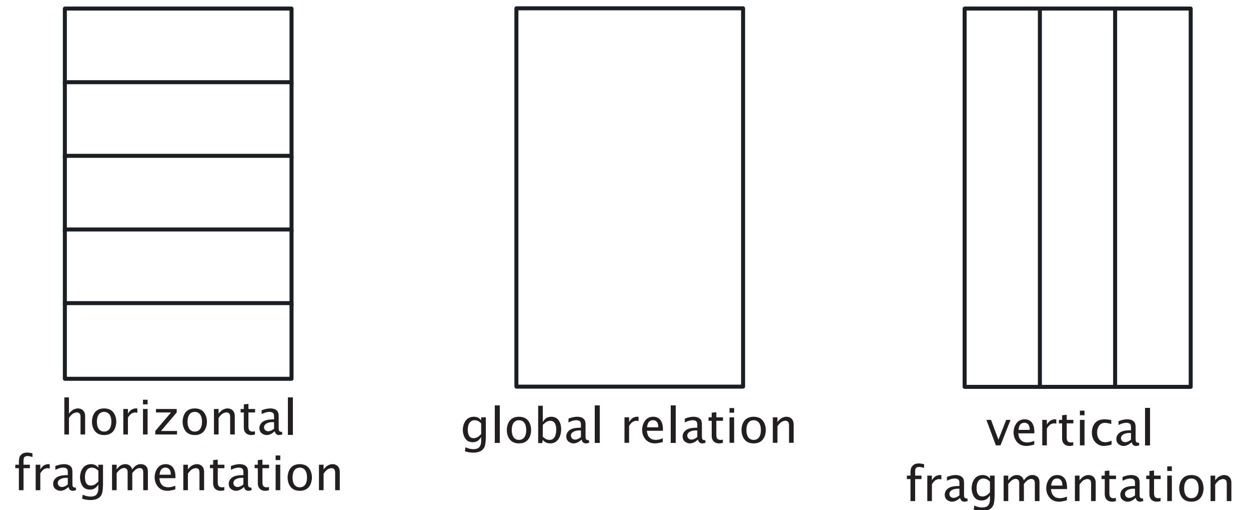
Fragmentation Approaches

Horizontal fragmentation

- Each fragment contains a subset of the tuples of the global relation

Vertical fragmentation

- Each fragment contains a subset of the attributes of the global relation



Decomposition

Relation R is *decomposed* into fragments $F_R = \{R_1, R_2, \ldots, R_n\}$

Decomposition (horizontal or vertical) can be expressed in terms of relational algebra expressions

Completeness

F_R is *complete* if each data item d_i in R is found in some R_j

UNIVERSITY OF Southampton

Reconstruction

R can be *reconstructed* if it is possible to define a relational operator ∇ such that $R = \nabla R_i$, for all $R_i \in F_R$

Note that ∇ will be different for different types of fragmentation

UNIVERSITY OF Southampton

Disjointness

F_R is *disjoint* if every data item d_i in each R_j is not in any R_k where $k \neq j$

Note that this is only strictly true for horizontal decomposition

For vertical decomposition, primary key attributes are typically repeated in all fragments to allow reconstruction; disjointness is defined on non-primary key attributes

UNIVERSITY OF Southampton

Horizontal Fragmentation

Each fragment contains a subset of the tuples of the global relation

Two versions:

- *Primary horizontal fragmentation*
  performed using a predicate defined on the relation being partitioned
- *Derived horizontal fragmentation*
  performed using a predicate defined on another relation

Primary Horizontal Fragmentation

Decomposition

$F_R = \{R_i : R_i = \sigma_{f_i}(R)\}$

where f_i is the *fragmentation predicate* for R_i

Reconstruction

$$R = \bigcup_{R_i \in F_R} R_i$$

Disjointness

F_R is disjoint if the simple predicates used in f_i are mutually exclusive

Completeness for primary horizontal fragmentation is beyond the scope of this lecture...

UNIVERSITY OF Southampton

Derived Horizontal Fragmentation

Decomposition

$F_R = \{R_i : R_i = R \ltimes S\}$

where $F_S = \{S_i : S_i = \sigma_{f_i}(S)\}$ and f_i are the fragmentation predicates for the primary horizontal fragmentation of S

Reconstruction

$$R = \bigcup_{R_i \in F_R} R_i$$

Completeness and disjointness for derived horizontal fragmentation are beyond the scope of this lecture...

Vertical Fragmentation

Decomposition

$F_R = \{R_i : R_i = \pi_{a_i}(R)\}$, where a_i is a subset of the attributes of R

Completeness

F_R is complete if each attribute of R appears in some a_i

Reconstruction

$R = \bowtie_K R_i$ for all $R_i \in FR$

where K is the set of primary key attributes of R

Disjointness

F_R is disjoint if each non-primary key attribute of R appears in at most one a_i

Hybrid Fragmentation

Horizontal and vertical fragmentation may be combined:

- Vertical fragmentation of horizontal fragments
- Horizontal fragmentation of vertical fragments

Query Processing

UNIVERSITY OF
Southampton

Localisation

Fragmentation expressed as relational algebra expressions

Global relations can be reconstructed using these expressions

- a *localisation program*

Naïvely, generate distributed query plan by substituting localisation programs for relations

- use reduction techniques to optimise queries

Reduction for Horizontal Fragmentation

Given a relation R fragmented as $F_R = \{R_1, R_2, \ldots, R_n\}$

Localisation program is $R = R_1 \cup R_2 \cup \cdots \cup R_n$

Reduce by identifying fragments of localised query that give empty relations

Two cases to consider:

- reduction with selection
- reduction with join

Horizontal Selection Reduction

Given horizontal fragmentation of R such that $R_j = \sigma_{p_j}(R)$:

$\sigma_p(R_j) = \emptyset$ if $\forall x \in R, \neg(p(x) \land p_j(x))$

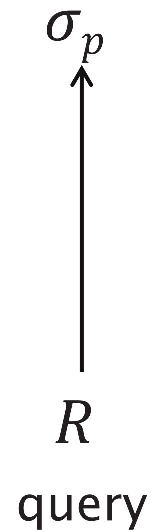
where p_j is the fragmentation predicate for R_j

Horizontal Selection Reduction

Given horizontal fragmentation of R such that $R_j = \sigma_{p_j}(R)$:

$\sigma_p(R_j) = \emptyset$ if $\forall x \in R, \neg(p(x) \land p_j(x))$

where p_j is the fragmentation predicate for R_j



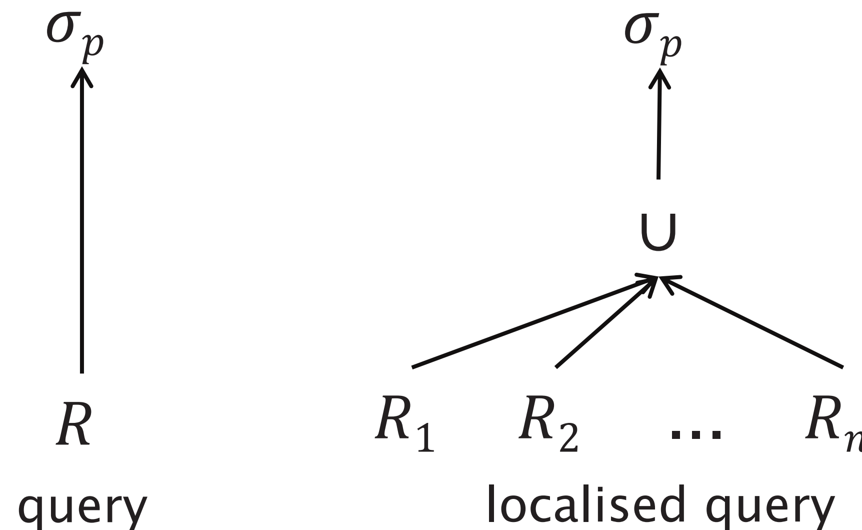
UNIVERSITY OF Southampton

Horizontal Selection Reduction

Given horizontal fragmentation of R such that $R_j = \sigma_{p_j}(R)$:

$\sigma_p(R_j) = \emptyset$ if $\forall x \in R, \neg(p(x) \wedge p_j(x))$

where p_j is the fragmentation predicate for R_j

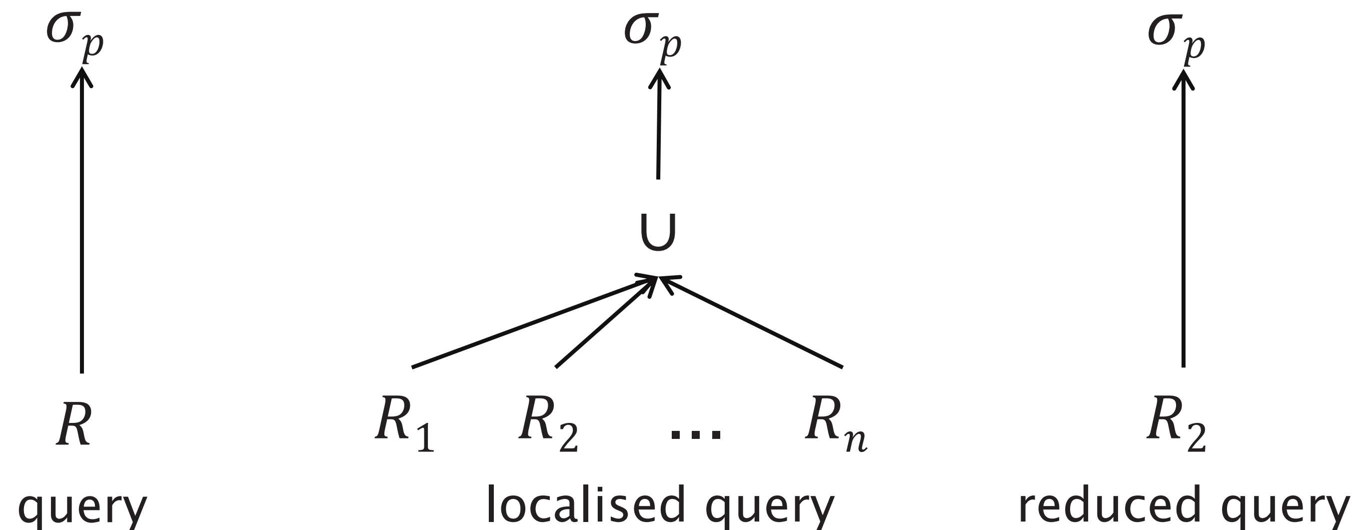


Horizontal Selection Reduction

Given horizontal fragmentation of R such that $R_j = \sigma_{p_j}(R)$:

$\sigma_p(R_j) = \emptyset$ if $\forall x \in R, \neg(p(x) \wedge p_j(x))$

where p_j is the fragmentation predicate for R_j



Horizontal Join Reduction

Recall that joins distribute over unions:

$$(R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

Given fragments R_i and R_j defined with predicates p_i and p_j :

$R_i \bowtie R_j = \emptyset$ if $\forall x \in R_i, \forall y \in R_j \neg(p_i(x) \wedge p_j(y))$

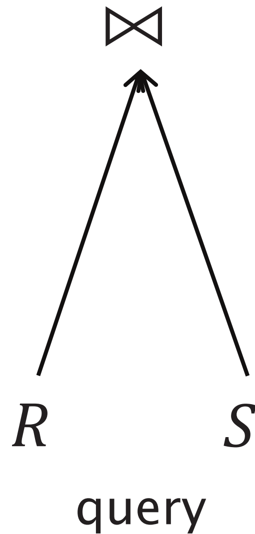
Horizontal Join Reduction

Recall that joins distribute over unions:

$$(R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

Given fragments R_i and R_j defined with predicates p_i and p_j :

$R_i \bowtie R_j = \emptyset$ if $\forall x \in R_i, \forall y \in R_j \neg(p_i(x) \wedge p_j(y))$



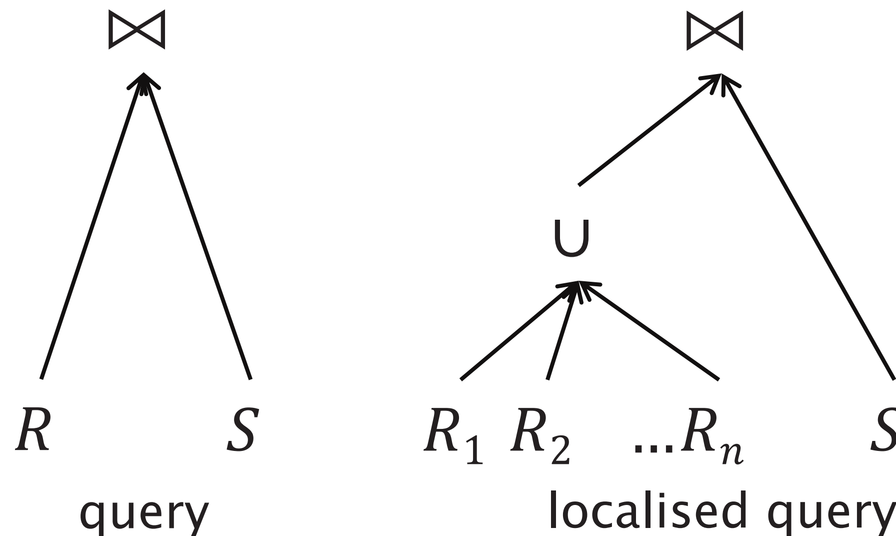
Horizontal Join Reduction

Recall that joins distribute over unions:

$$(R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

Given fragments R_i and R_j defined with predicates p_i and p_j :

$R_i \bowtie R_j = \emptyset$ if $\forall x \in R_i, \forall y \in R_j \neg(p_i(x) \wedge p_j(y))$



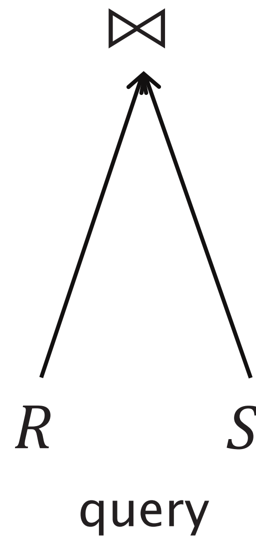
Horizontal Join Reduction

Recall that joins distribute over unions:

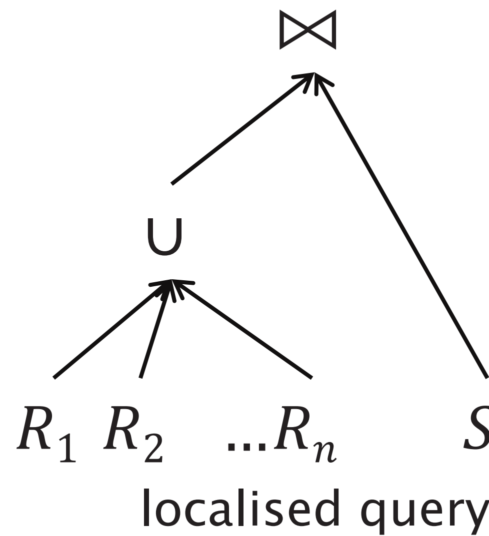
$(R_1 \cup R_2) \bowtie S \equiv (R_1 \bowtie S) \cup (R_2 \bowtie S)$

Given fragments R_i and R_j defined with predicates p_i and p_j :

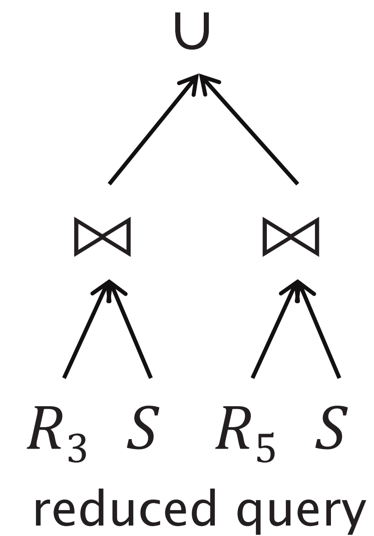
$R_i \bowtie R_j = \emptyset$ if $\forall x \in R_i, \forall y \in R_j \neg(p_i(x) \wedge p_j(y))$



query



localised query



reduced query

Reduction for Vertical Fragmentation

Given a relation R fragmented as $F_R = \{R_1, R_2, \dots, R_n\}$

Localisation program is $R = R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$

Reduce by identifying useless intermediate relations

One case to consider:

- reduction with projection

Vertical Projection Reduction

Given a relation R with attributes $A = \{a_1, a_2, \dots, a_n\}$ vertically fragmented as $R_i = \pi_{A_i}(R)$ where $A_i \subseteq A$

$\pi_{D,K}(R_i)$ is useless if $D \nsubseteq A_i$

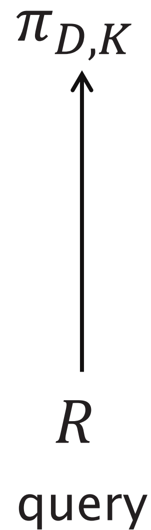
D is set of projection attributes

Vertical Projection Reduction

Given a relation R with attributes $A = \{a_1, a_2, \ldots, a_n\}$ vertically fragmented as $R_i = \pi_{A_i}(R)$ where $A_i \subseteq A$

$\pi_{D,K}(R_i)$ is useless if $D \nsubseteq A_i$

D is set of projection attributes

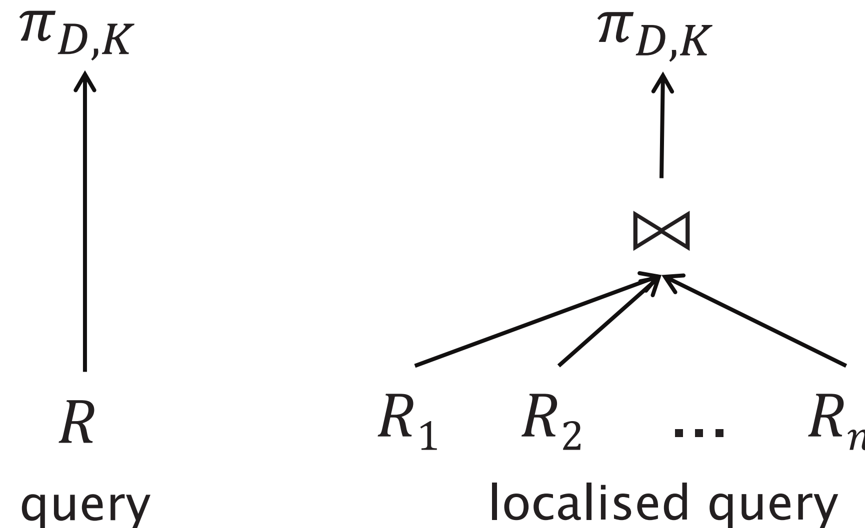


Vertical Projection Reduction

Given a relation R with attributes $A = \{a_1, a_2, \dots, a_n\}$ vertically fragmented as $R_i = \pi_{A_i}(R)$ where $A_i \subseteq A$

$\pi_{D,K}(R_i)$ is useless if $D \nsubseteq A_i$

D is set of projection attributes



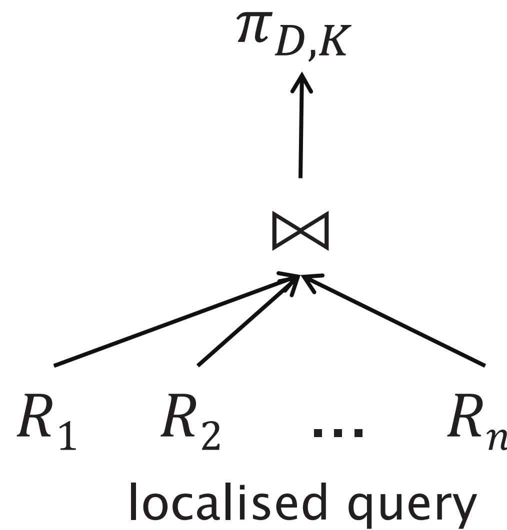
UNIVERSITY OF Southampton

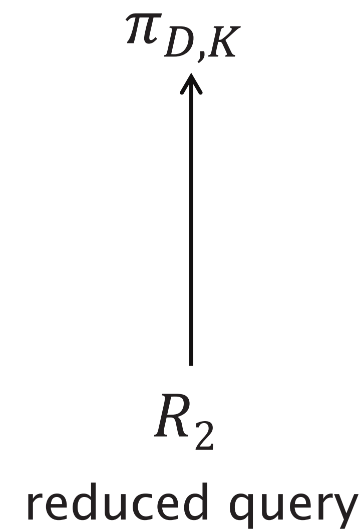
Vertical Projection Reduction

Given a relation R with attributes $A = \{a_1, a_2, \dots, a_n\}$ vertically fragmented as $R_i = \pi_{A_i}(R)$ where $A_i \subseteq A$

$\pi_{D,K}(R_i)$ is useless if $D \nsubseteq A_i$

D is set of projection attributes



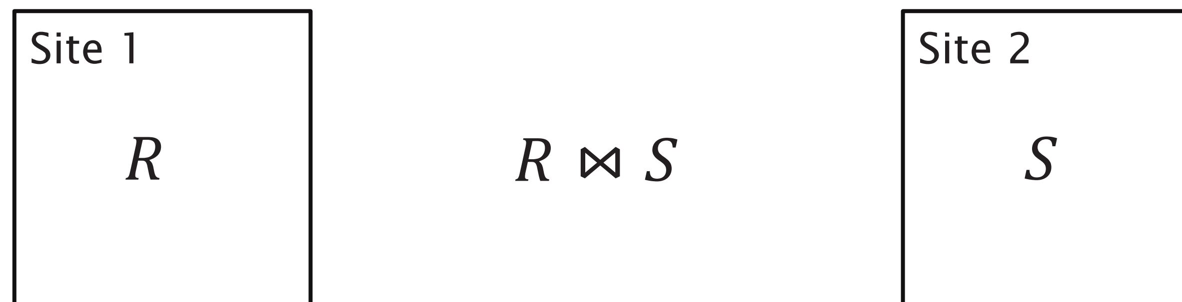


Distributed Joins

The Distributed Join Problem

We have two relations, R and S, each stored at a different site

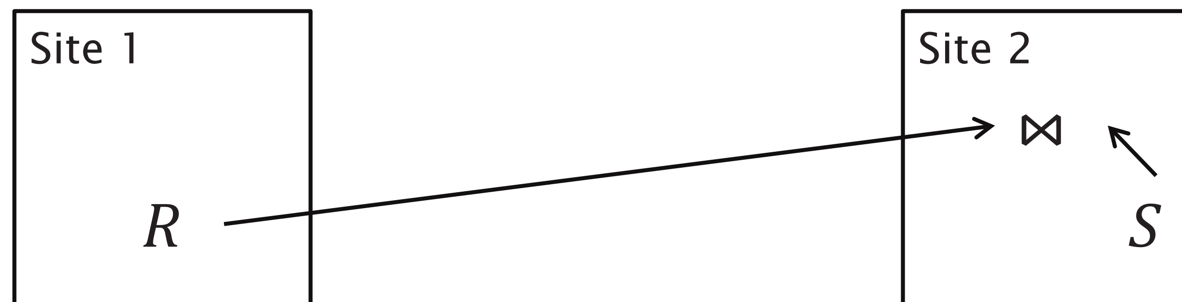
Where do we perform the join $R \bowtie S$?



The Distributed Join Problem

We can move one relation to the other site and perform the join there

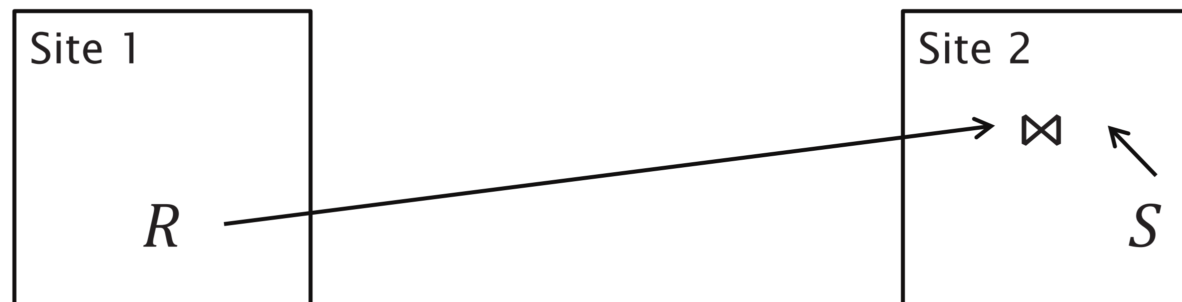
- CPU cost of performing the join is the same regardless of site
- Communications cost depends on the size of the relation being moved



The Distributed Join Problem

$Cost_{COM} = size(R) = cardinality(R) * length(R)$

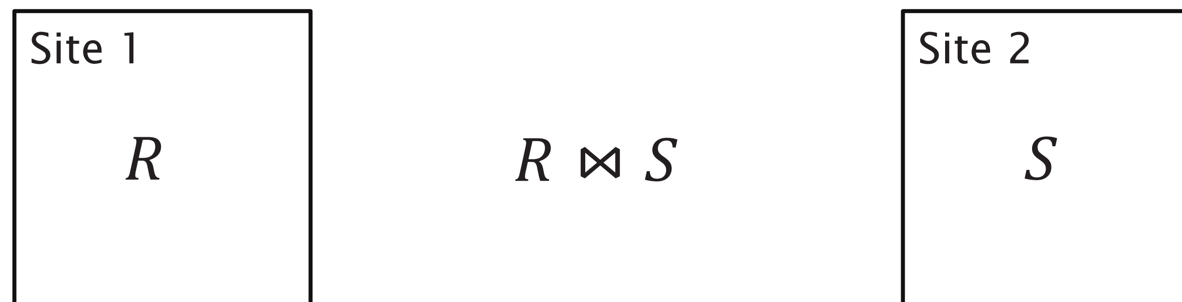
if $size(R) < size(S)$ then move R to site 2,
otherwise move S to site 1



Semijoin Reduction

We can further reduce the communications cost by only moving that part of a relation that will be used in the join

Use a semijoin...



Semijoins

Recall that $R \ltimes_p S \equiv \pi_R(R \bowtie_p S)$

where p is a predicate defined over R and S

π_R projects out only those attributes from R

$size(R \ltimes_p S) < size(R \bowtie_p S)$

$R \bowtie_p S \equiv (R \ltimes_p S) \bowtie_p S$

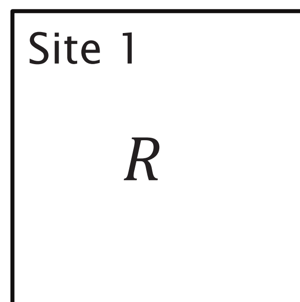
$R \bowtie_p S \equiv R \bowtie_p (R \rtimes_p S)$

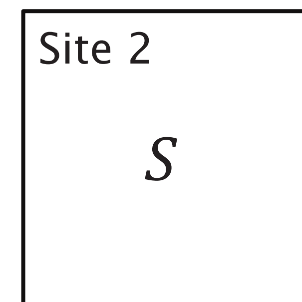
$R \bowtie_p S \equiv (R \ltimes_p S) \bowtie_p (R \rtimes_p S)$

Semijoin Reduction

$R \ltimes_p S \equiv \pi_R(R \bowtie_p S)$
$R \ltimes_p S \equiv \pi_R(R \bowtie_p \pi_p(S))$

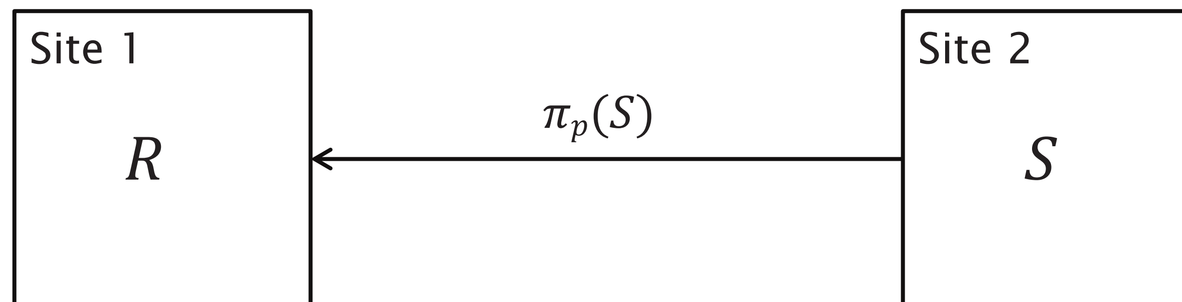
where $\pi_p(S)$ projects out from S only the attributes used in predicate p





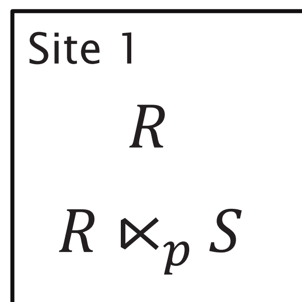
Semijoin Reduction, step 1

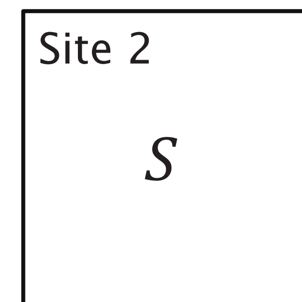
Site 2 sends $\pi_p(S)$ to site 1



Semijoin Reduction, step 2

Site 1 calculates $R \ltimes_p S \equiv \pi_R(R \bowtie_p \pi_p(S))$

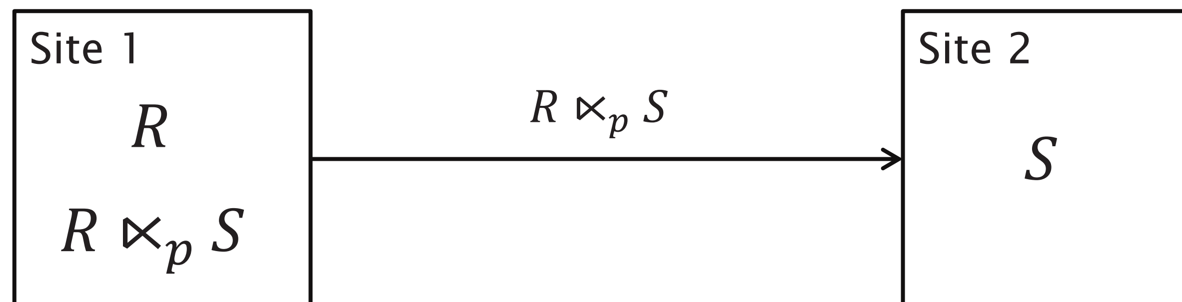




University of Southampton

Semijoin Reduction, step 3

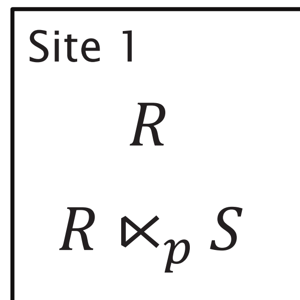
Site 1 sends $R \ltimes_p S$ to site 2

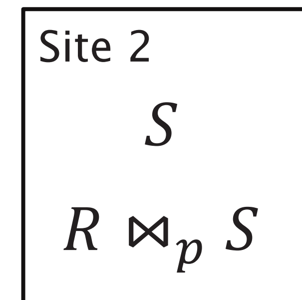


UNIVERSITY OF Southampton

Semijoin Reduction, step 4

Site 2 calculates $R \bowtie_p S \equiv (R \ltimes_p S) \bowtie_p S$

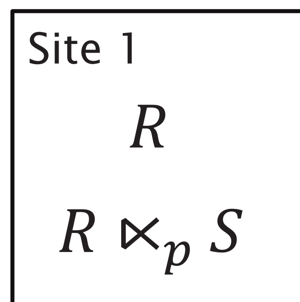


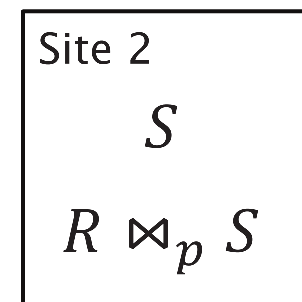


Semijoin Reduction

$Cost_{COM} = size(\pi_p(S)) + size(R \ltimes_p S)$

This approach is better if $size(\pi_p(S)) + size(R \ltimes_p S) < size(R)$





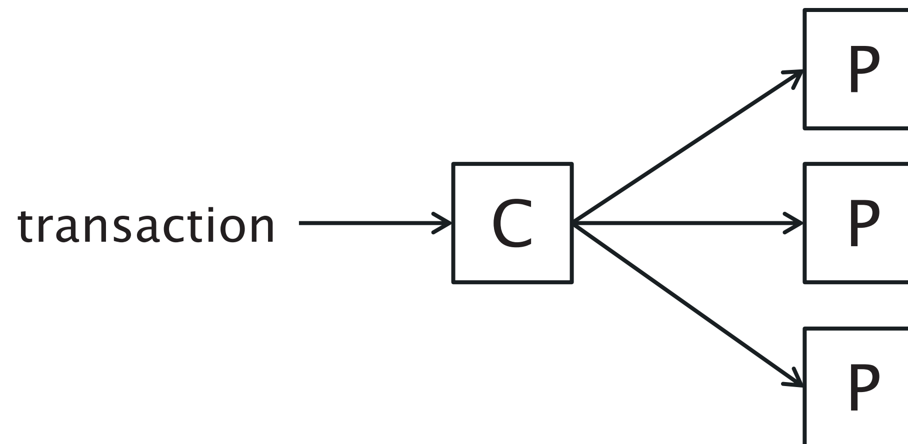
Concurrency Control

UNIVERSITY OF Southampton

Distributed Transactions

Transaction processing may be spread across several sites in the distributed database

- The site from which the transaction originated is known as the *coordinator*
- The sites on which the transaction is executed are known as the *participants*



Distribution and ACID

Non-distributed databases aim to maintain isolation

- Isolation: A transaction should not make updates externally visible until committed

Distributed databases commonly use two-phase locking (2PL) to preserve isolation

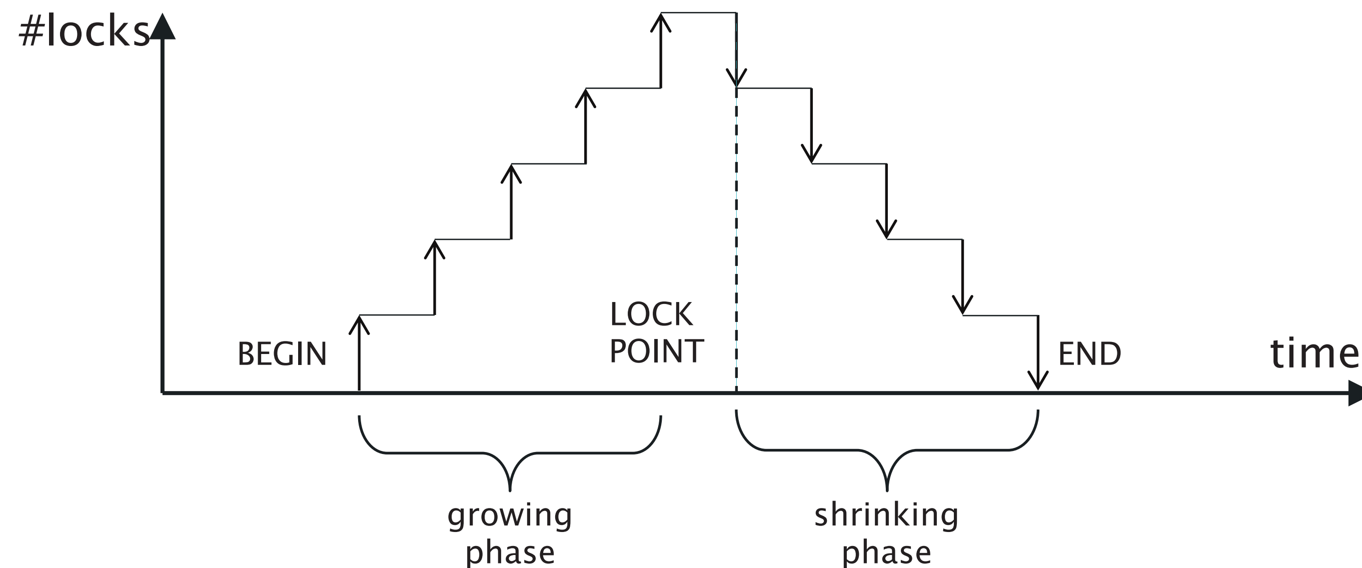
- 2PL ensures serialisability, the highest isolation level

Two-Phase Locking

Two phases:

- Growing phase: obtain locks, access data items
- Shrinking phase: release locks

Guarantees serialisable transactions



Distribution and Two-Phase Locking

In a non-distributed database, locking is controlled by *a lock manager*

Two main approaches to implementing two-phase locking in a distributed database:

- Centralised 2PL (C2PL)
  Responsibility for lock management lies with a single site
- Distributed 2PL (D2PL)
  Each site has its own lock manager

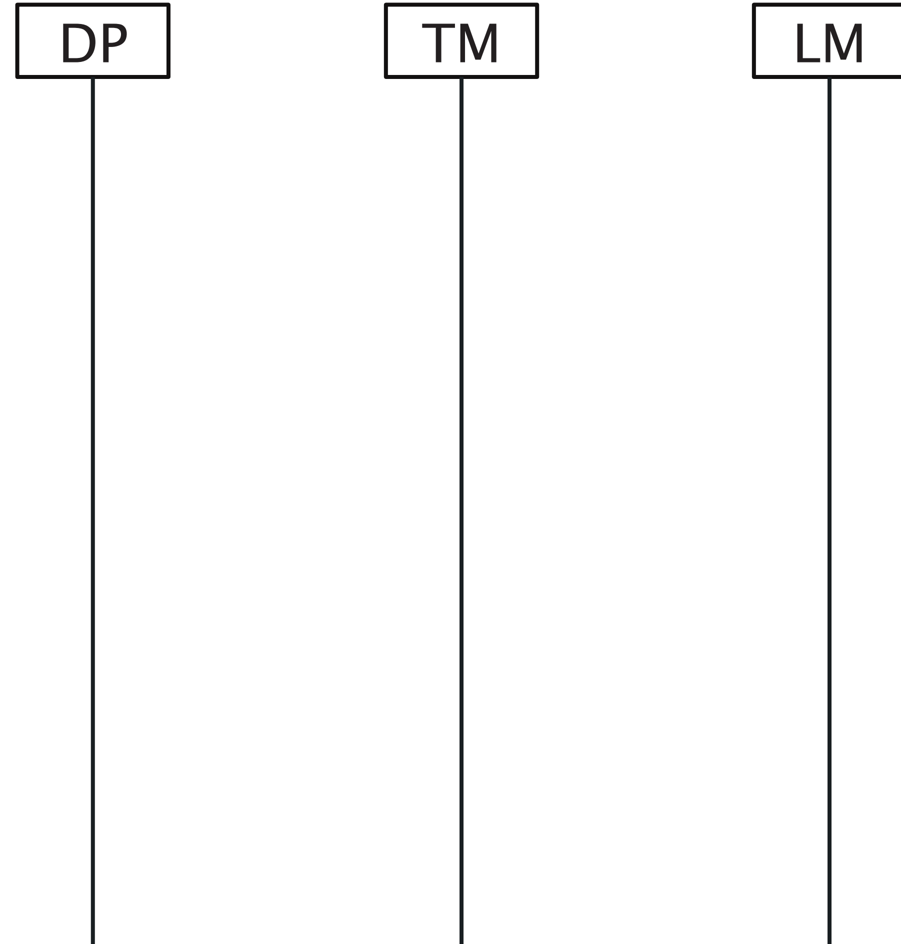
UNIVERSITY OF Southampton

Centralised Two-Phase Locking (C2PL)

Coordinating site runs *transaction manager* TM

Participant sites run *data processors* DP

Lock manager LM runs on central site



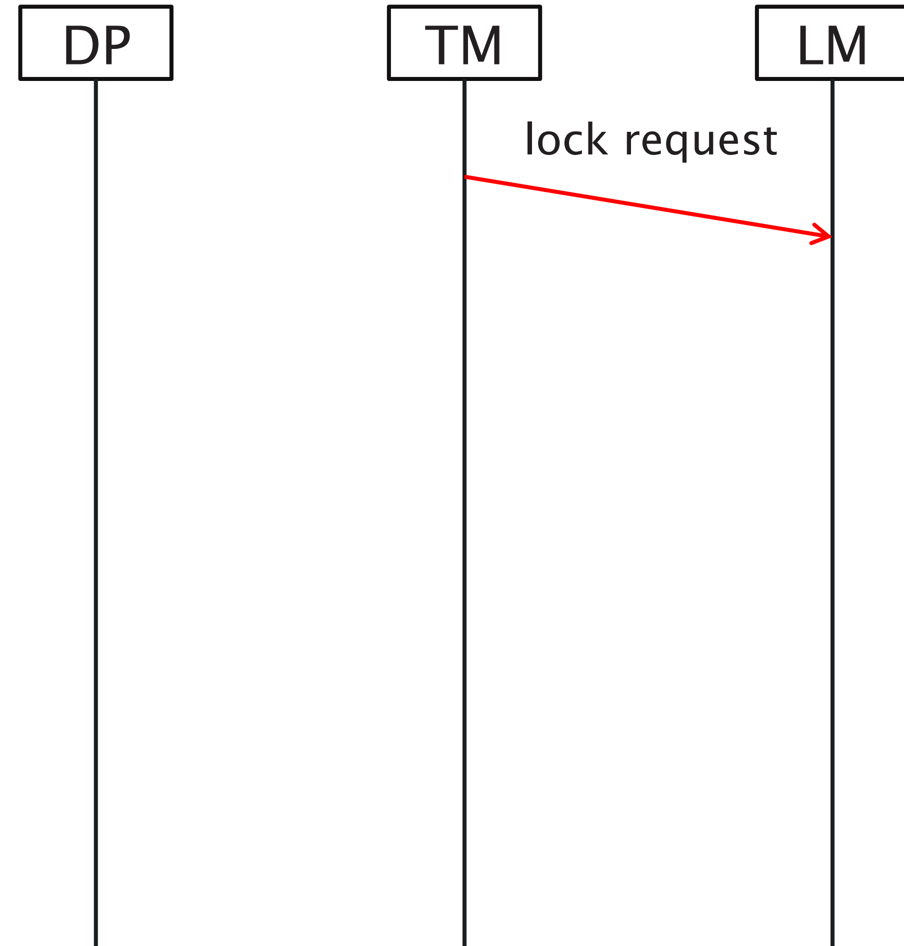
Centralised Two-Phase Locking (C2PL)

Coordinating site runs *transaction manager* TM

Participant sites run *data processors* DP

Lock manager LM runs on central site

1. TM requests locks from LM



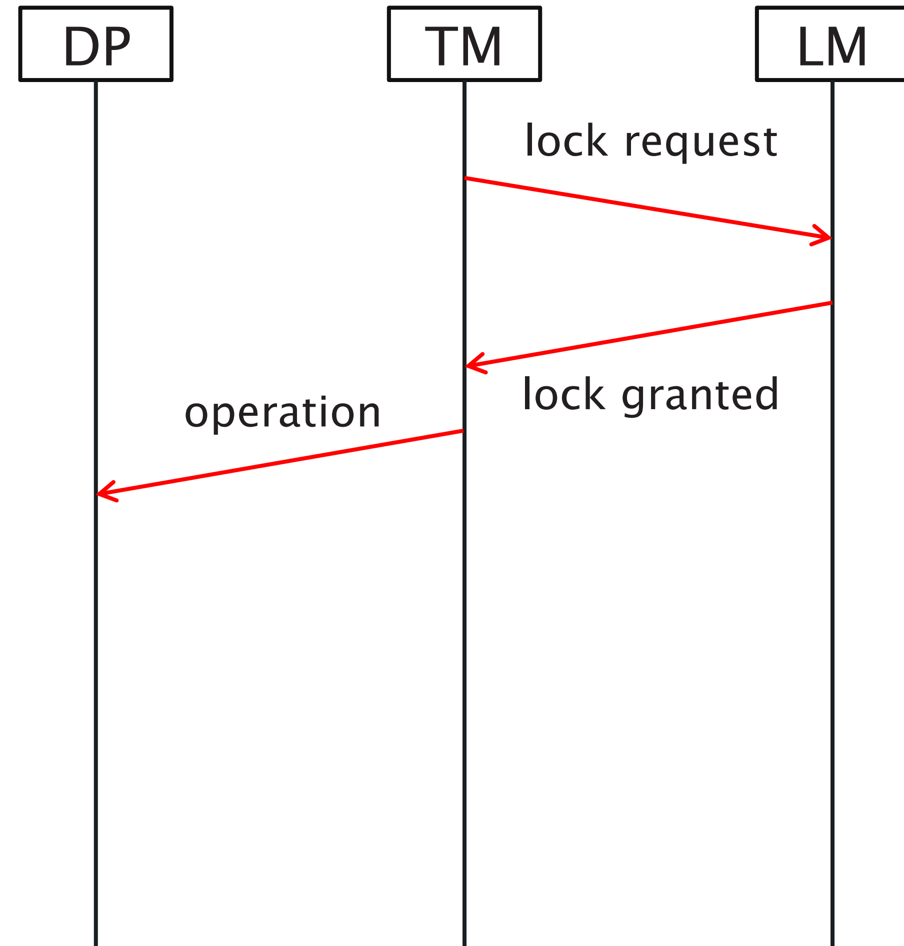
Centralised Two-Phase Locking (C2PL)

Coordinating site runs *transaction manager* TM

Participant sites run *data processors* DP

Lock manager LM runs on central site

1. TM requests locks from LM
2. If granted, TM submits operations to processors DP



UNIVERSITY OF Southampton

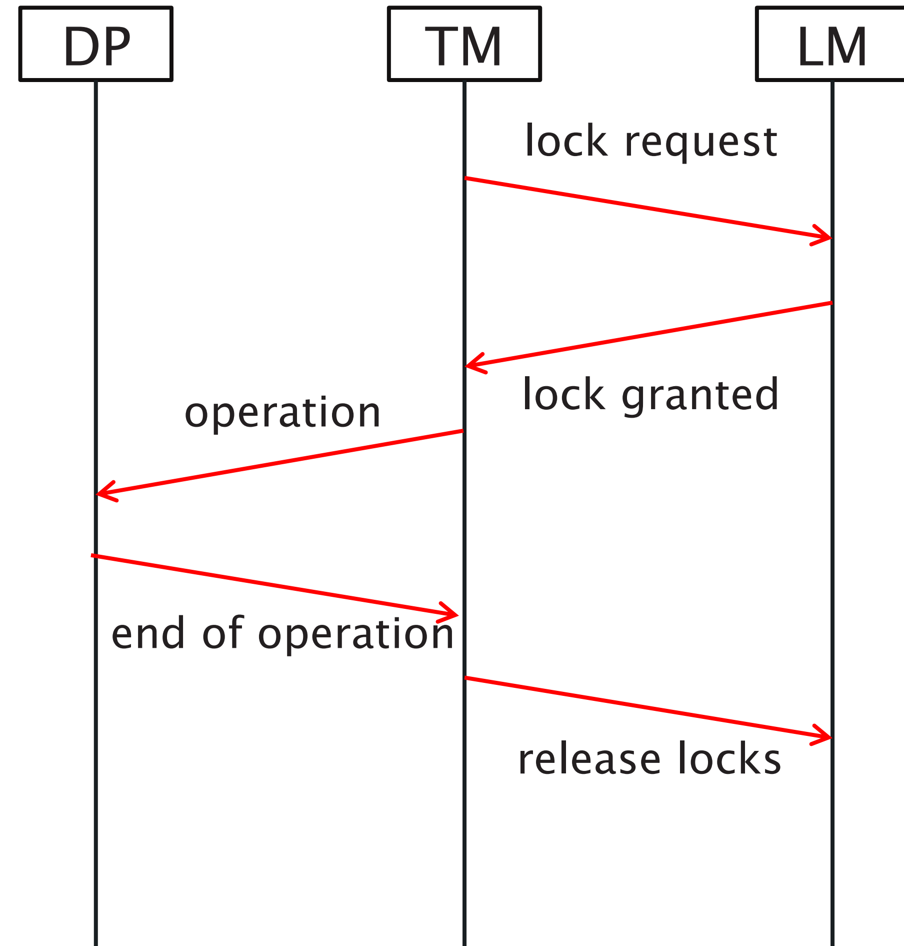
Centralised Two-Phase Locking (C2PL)

Coordinating site runs *transaction manager* TM

Participant sites run *data processors* DP

Lock manager LM runs on central site

1. TM requests locks from LM
2. If granted, TM submits operations to processors DP
3. When DPs finish, TM sends message to LM to release locks



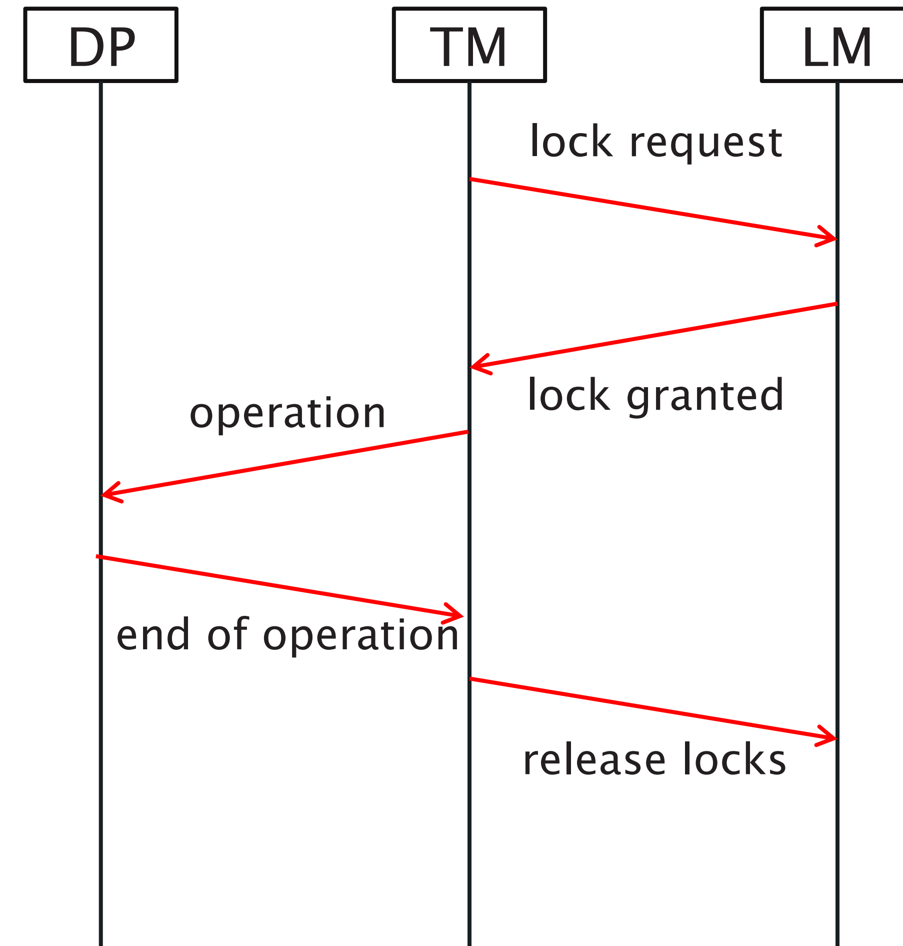
Centralised Two-Phase Locking (C2PL)

LM is a single point of failure

- less reliable

LM is a bottleneck

- affects transaction throughput

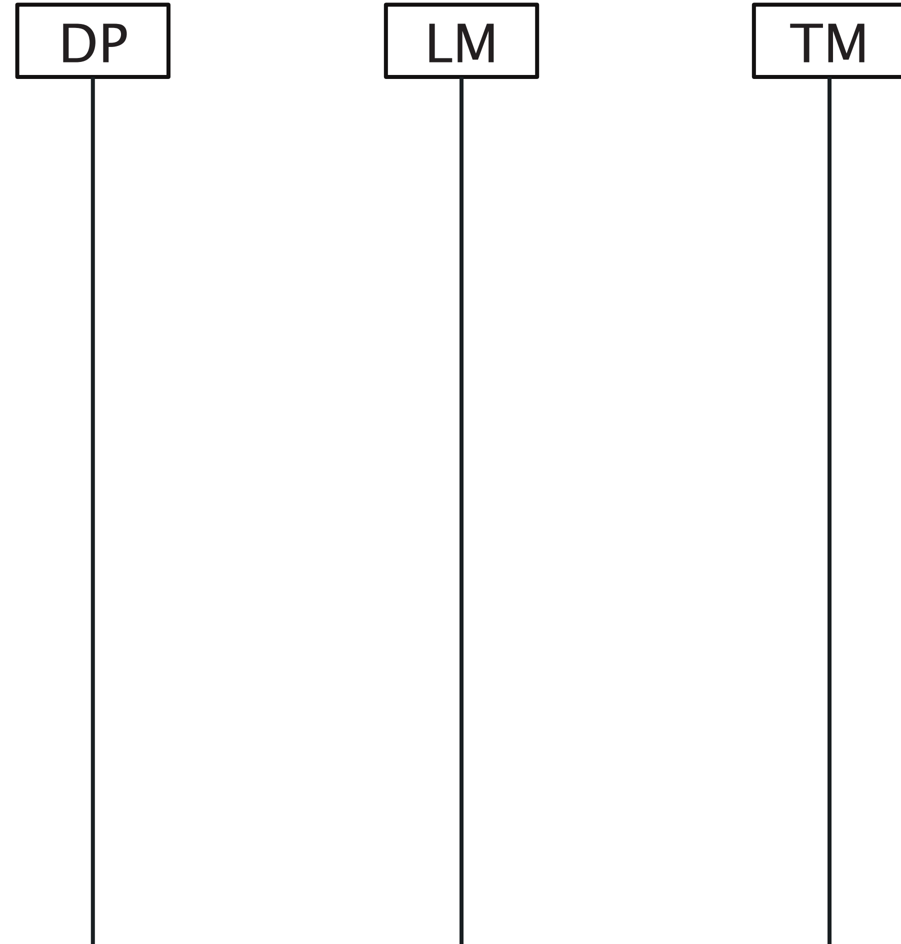


UNIVERSITY OF Southampton

Distributed Two-Phase Locking (D2PL)

Coordinating site C runs TM

Each participant runs both an LM and a DP

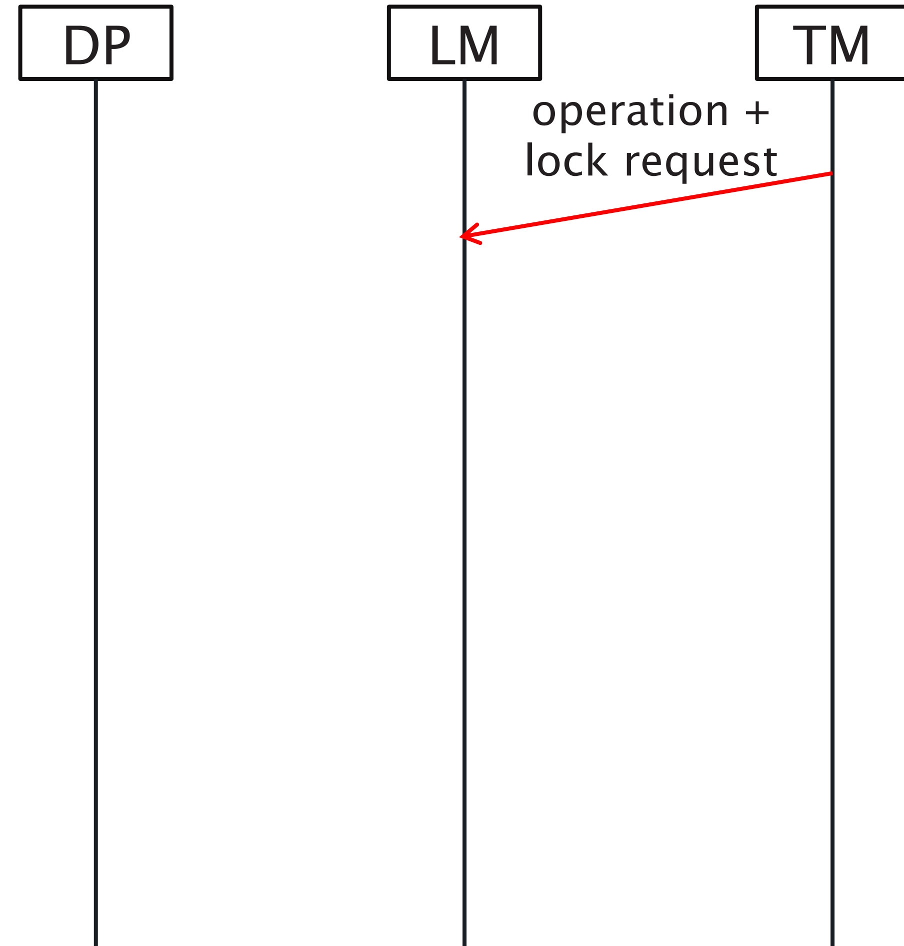


Distributed Two-Phase Locking (D2PL)

Coordinating site C runs TM

Each participant runs both an LM and a DP

1. TM sends operations and lock requests to each LM



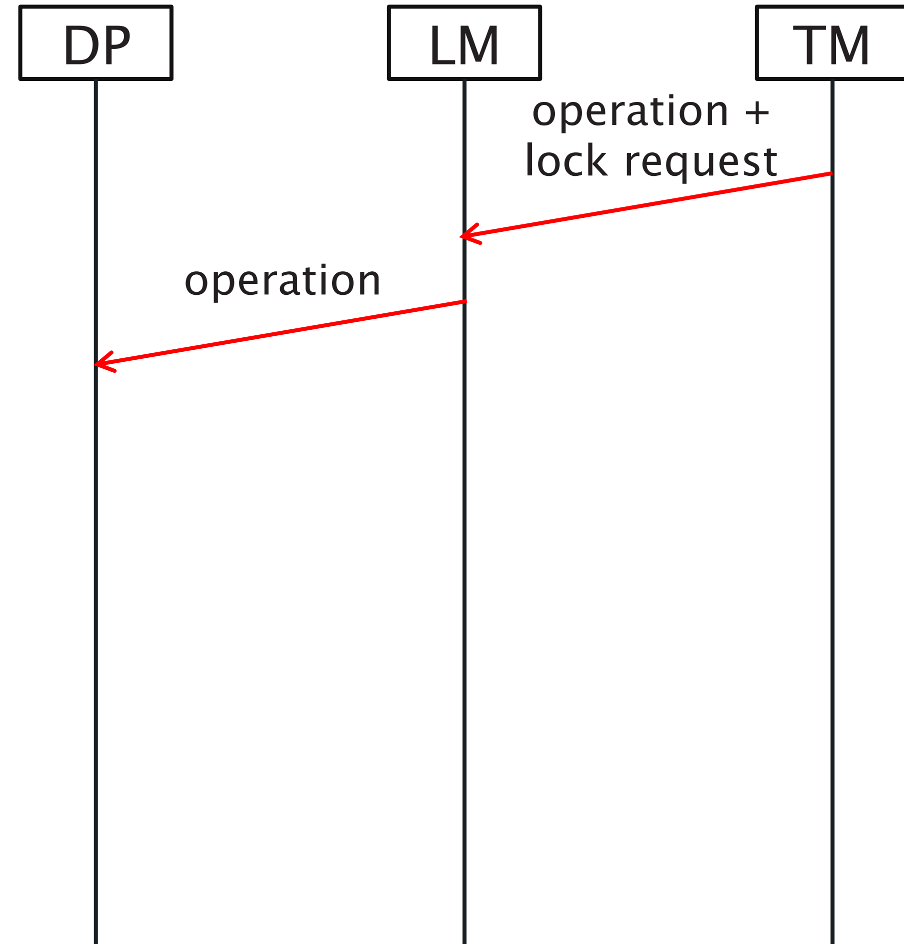
UNIVERSITY OF Southampton

Distributed Two-Phase Locking (D2PL)

Coordinating site C runs TM

Each participant runs both an LM and a DP

1. TM sends operations and lock requests to each LM
2. If lock can be granted, LM forwards operation to local DP

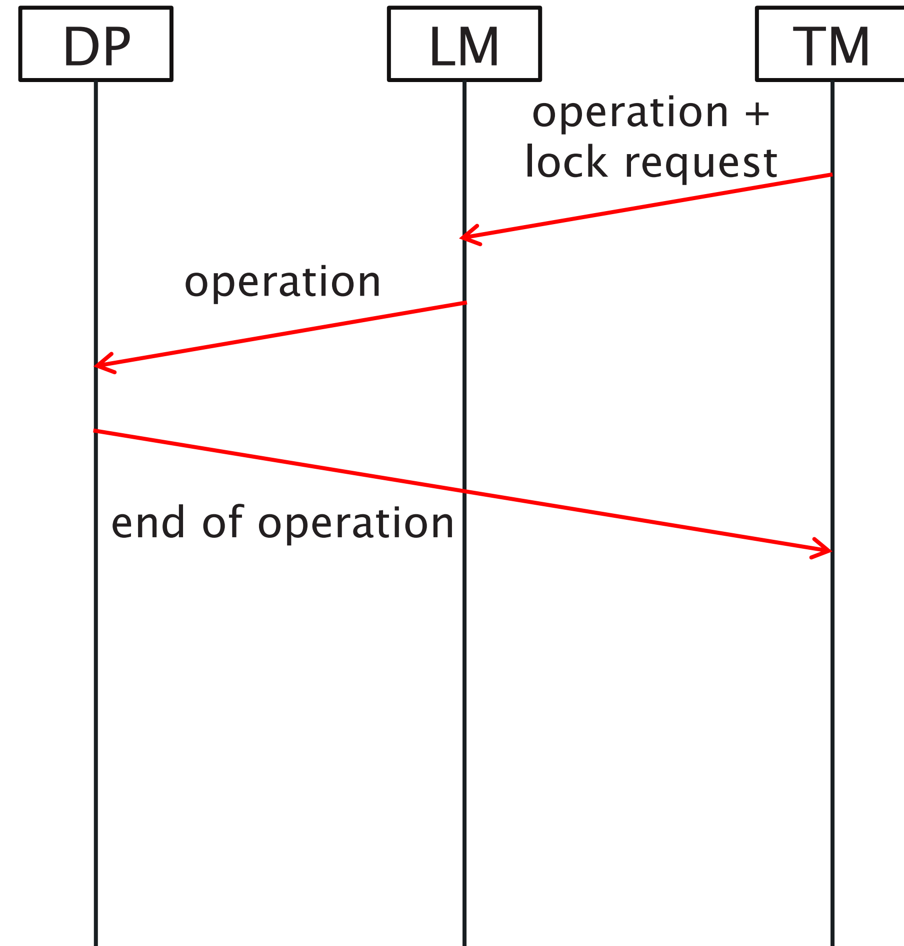


Distributed Two-Phase Locking (D2PL)

Coordinating site C runs TM

Each participant runs both an LM and a DP

1. TM sends operations and lock requests to each LM
2. If lock can be granted, LM forwards operation to local DP
3. DP sends "end of operation" to TM

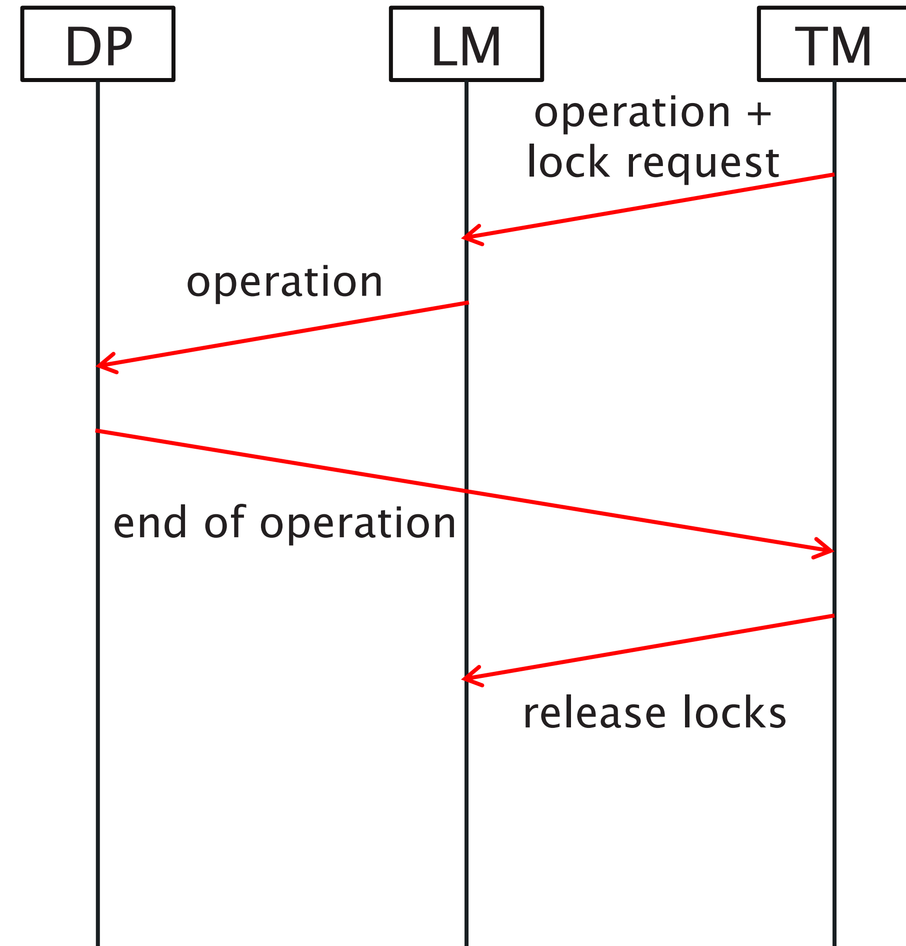


Distributed Two-Phase Locking (D2PL)

Coordinating site C runs TM

Each participant runs both an LM and a DP

1. TM sends operations and lock requests to each LM
2. If lock can be granted, LM forwards operation to local DP
3. DP sends “end of operation” to TM
4. TM sends message to LM to release locks

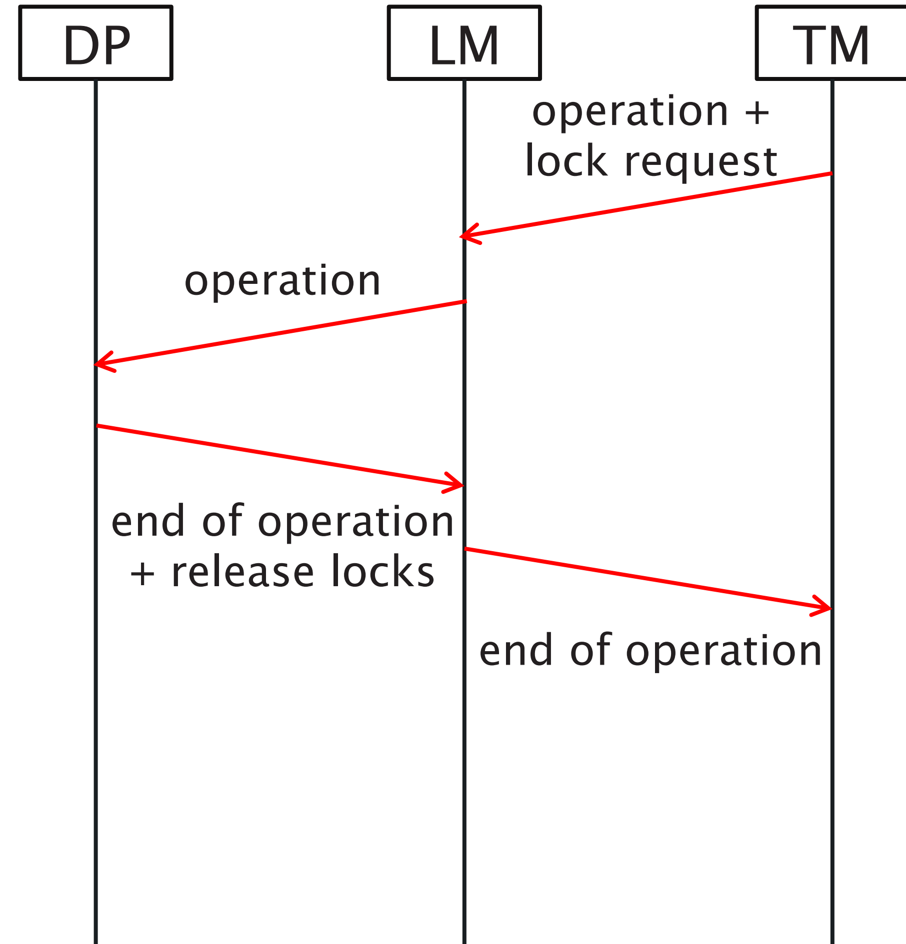


Distributed Two-Phase Locking (D2PL)

Variant:

DPs may send “end of operation” to their own LM

LM releases lock and informs TM



Deadlock

Deadlock exists when two or more transactions are waiting for each other to release a lock on an item

Three conditions must be satisfied for deadlock to occur:

- Concurrency: two transactions claim exclusive control of one resource
- Hold: one transaction continues to hold exclusively controlled resources until its need is satisfied
- Wait: transactions wait in queues for additional resources while holding resources already allocated

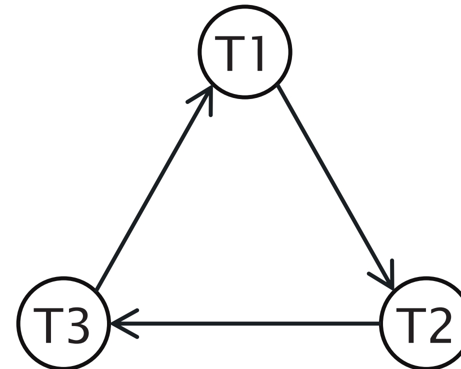
Wait-For Graph

Representation of interactions between transactions

Directed graph containing:

- A vertex for each transaction that is currently executing
- An edge from T1 to T2 if T1 is waiting to lock an item that is currently locked by T2

Deadlock exists iff the WFG contains a cycle



Distributed Deadlock

Two types of Wait-For Graph

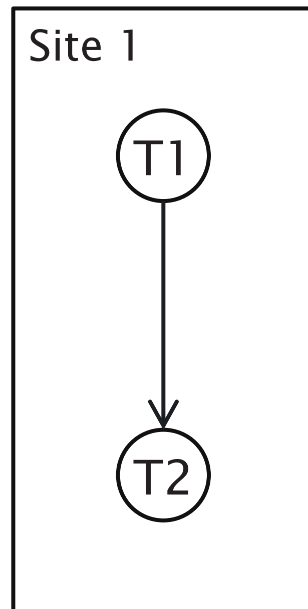
- Local WFG
  (one per site, only considers transactions on that site)
- Global WFG
  (union of all LWFGs)

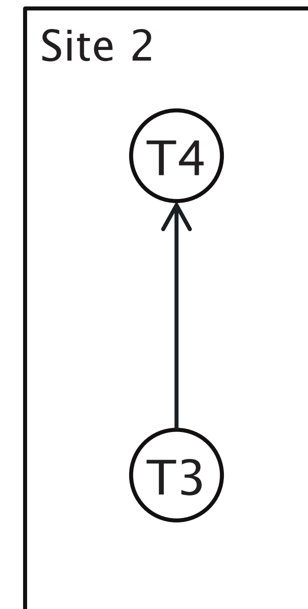
Deadlock may occur

- on a single site
  (within its LWFG)
- between sites
  (within the GWFG)

Distributed Deadlock Example

Consider the wait-for relationship T1→T2→T3→T4→T1
with T1, T2 on site 1 and T3, T4 on site 2





UNIVERSITY OF Southampton

Distributed Deadlock Example

Consider the wait-for relationship T1→T2→T3→T4→T1
with T1, T2 on site 1 and T3, T4 on site 2

UNIVERSITY OF Southampton

Managing Distributed Deadlock

Three main approaches:

1. Prevention
    - pre-declaration
2. Avoidance
    - resource ordering
    - transaction prioritisation
3. Detection and Resolution

Prevention

Guarantees that deadlocks cannot occur in the first place

1. Transaction pre-declares all data items that it will access
2. TM checks that locking data items will not cause deadlock
3. Proceed (to lock) only if all data items are available (unlocked)

Con: difficult to know in advance which data items will be accessed by a transaction

Avoidance

Two main sub-approaches:

1. Resource ordering
   - Concurrency controlled such that deadlocks won't happen
2. Transaction prioritisation
   - Potential deadlocks detected and avoided

Resource Ordering

All resources (data items) are ordered

Transactions always access resources in this order

Example:

- Data item A comes before item B
- Both transactions need to get locks on A and B
- All transactions must get a lock on A before trying for a lock on B
- No transaction will ever be left with a lock on B and waiting for a lock on A

UNIVERSITY OF Southampton

Transaction Prioritisation

Each transaction has a timestamp that corresponds to the time it was started: ts(T)

- Transactions can be prioritised using these timestamps

When a lock request is denied, use priorities to choose a transaction to abort

- WAIT-DIE and WOUND-WAIT rules

WAIT-DIE and WOUND-WAIT

T_i requests a lock on a data item that is already locked by T_j

The WAIT-DIE rule:

if $ts(T_i) < ts(T_j)$
 then T_i waits
 else T_i dies (aborts and restarts with same timestamp)

The WOUND-WAIT rule:

if $ts(T_i) < ts(T_j)$
 then T_j is wounded (aborts and restarts with same timestamp)
 else T_i waits

note: WOUND-WAIT pre-empts active transactions

Detection and Resolution

1. Study the GWFG for cycles (detection)
2. Break cycles by aborting transactions (resolution)

Selecting minimum total cost sets of transactions to abort is NP-complete

Three main approaches to deadlock detection:

- centralised
- hierarchical
- distributed

UNIVERSITY OF Southampton

Centralised Deadlock Detection

One site is designated as the deadlock detector (DD) for the system

Each site sends its LWFG (or changes to its LWFG) to the DD at intervals

DD constructs the GWFG and looks for cycles

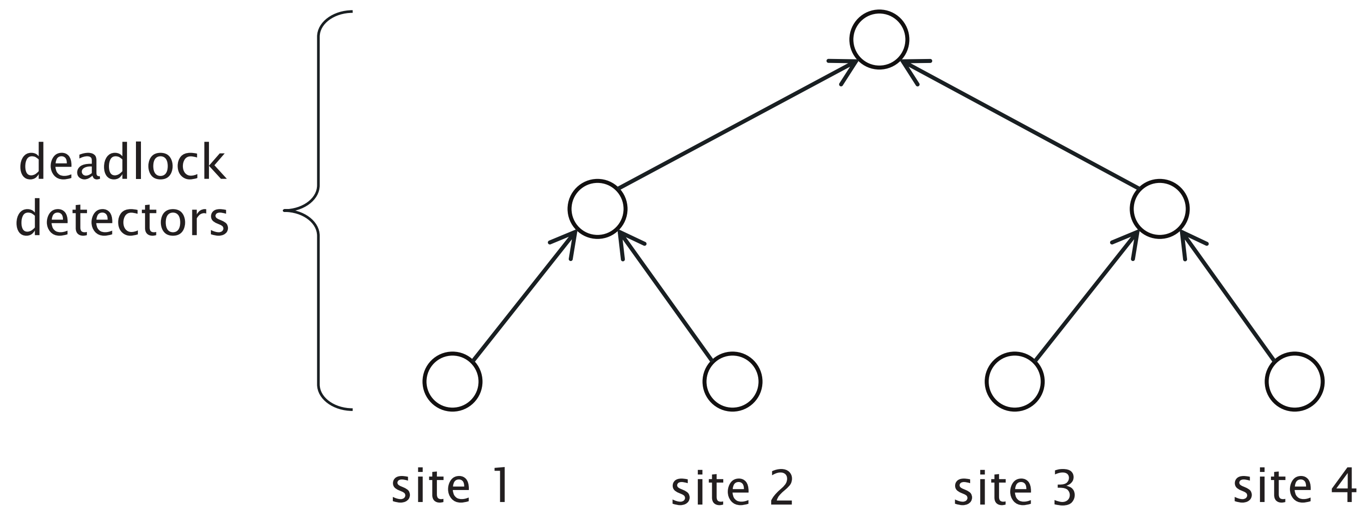
UNIVERSITY OF Southampton

Hierarchical Deadlock Detection

Each site has a DD, which looks in the site's LWFG for cycles

Each site sends its LWFG to the DD at the next level, which merges the LWFGs sent to it and looks for cycles

These DDs send the merged WFGs to the next level, etc

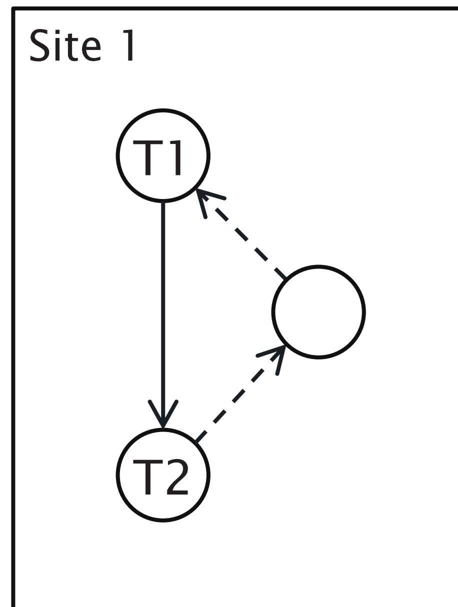


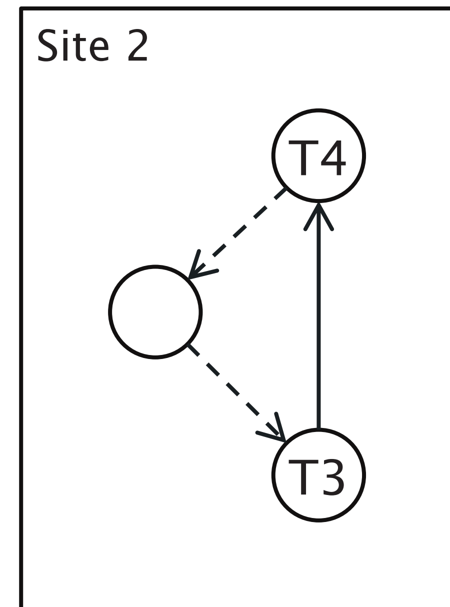
UNIVERSITY OF Southampton

Distributed Deadlock Detection

Responsibility for detecting deadlocks is delegated to sites

LWFGs are modified to show relationships between local transactions and remote transactions





Distributed Deadlock Detection

LWFG contains a cycle not involving external edges

- Local deadlock, resolve locally

LWFG contains a cycle involving external edges

- Potential deadlock – communicate to other sites
- Sites must then agree on a victim transaction to abort

Reliability

Distribution and ACID

Non-distributed databases aim to maintain atomicity and durability of transactions

- Atomicity: A transaction is either performed completely or not at all
- Durability: Once a transaction has been committed, changes should not be lost because of failure

As with parallel databases, distributed databases use the two-phase commit protocol (2PC) to preserve atomicity

- Increased cost of communication may require a variant approach

Centralised 2PC

Communication only between the coordinator and the participants

- No inter-participant communication

Southampton UNIVERSITY OF

Centralised 2PC

Communication only between the coordinator and the participants

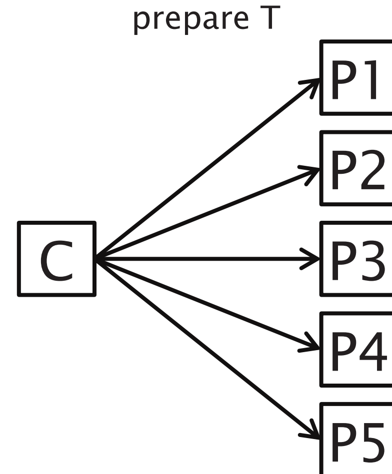
- No inter-participant communication

UNIVERSITY OF Southampton

Centralised 2PC

Communication only between the coordinator and the participants

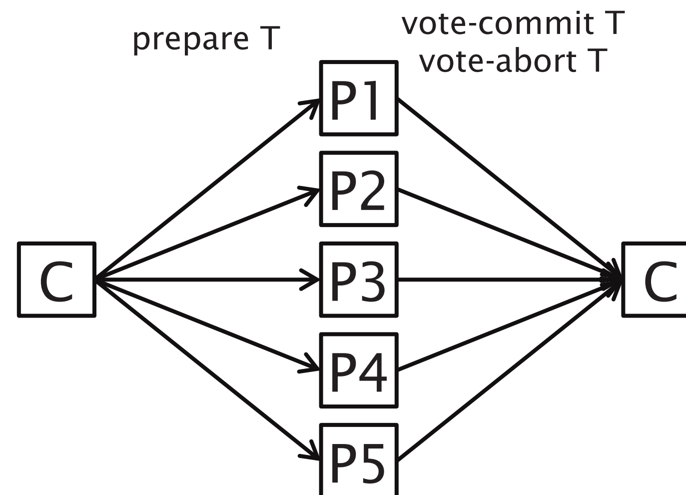
- No inter-participant communication



Centralised 2PC

Communication only between the coordinator and the participants

- No inter-participant communication

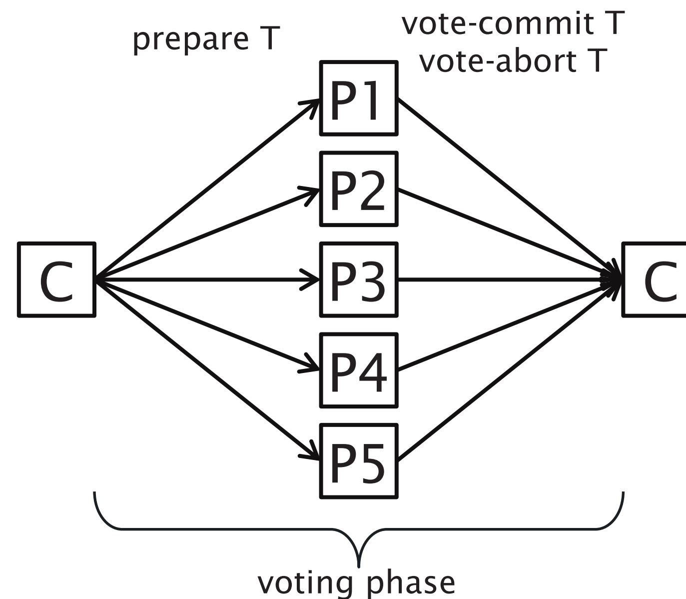


UNIVERSITY OF Southampton

Centralised 2PC

Communication only between the coordinator and the participants

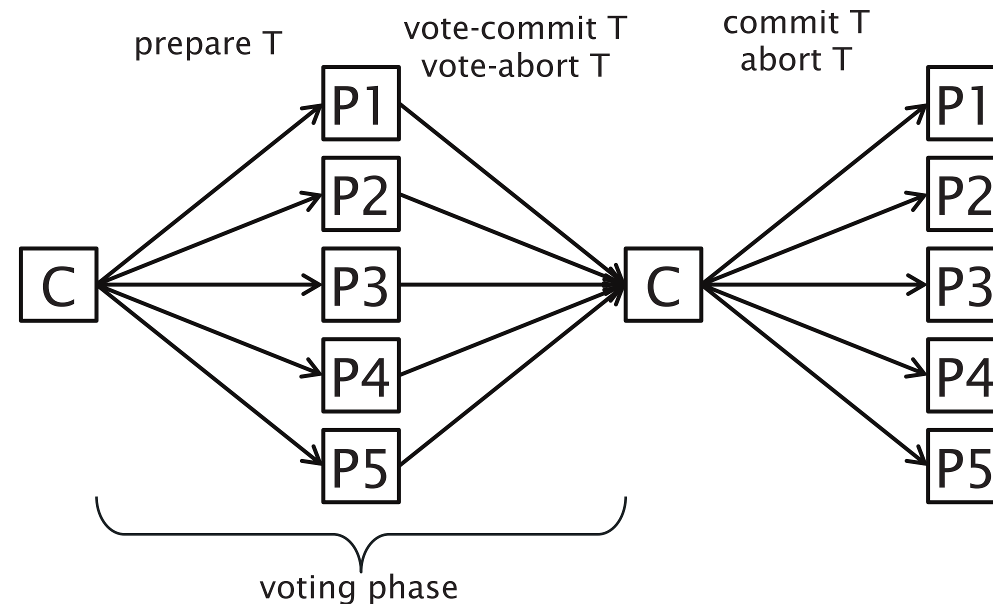
- No inter-participant communication



Centralised 2PC

Communication only between the coordinator and the participants

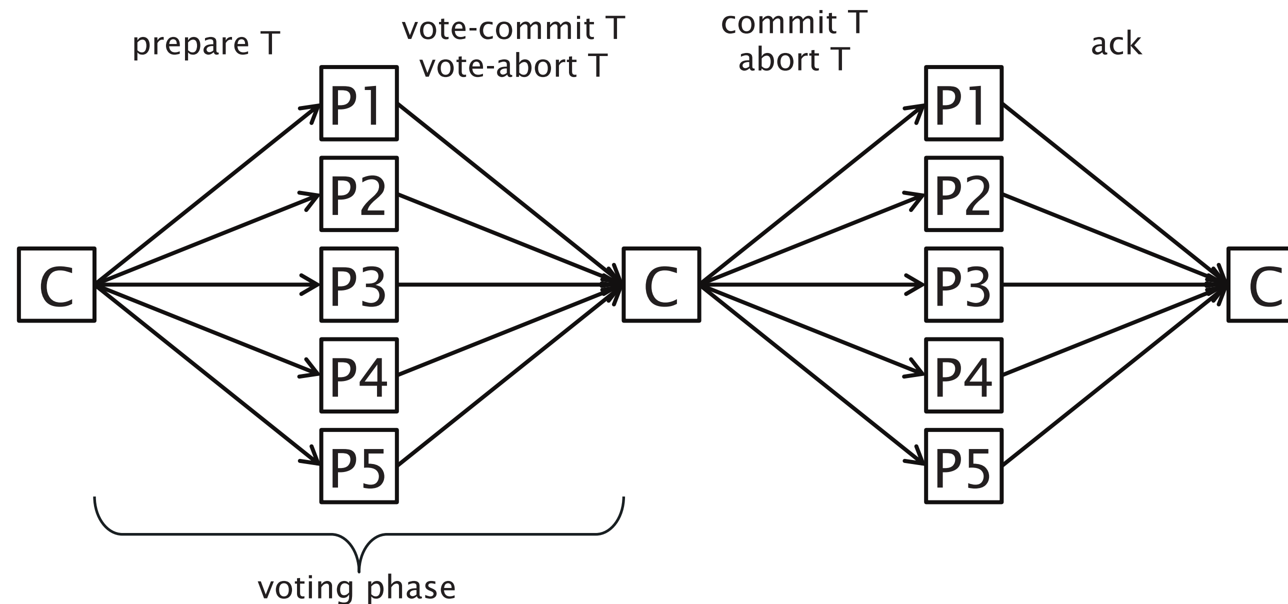
- No inter-participant communication



Centralised 2PC

Communication only between the coordinator and the participants

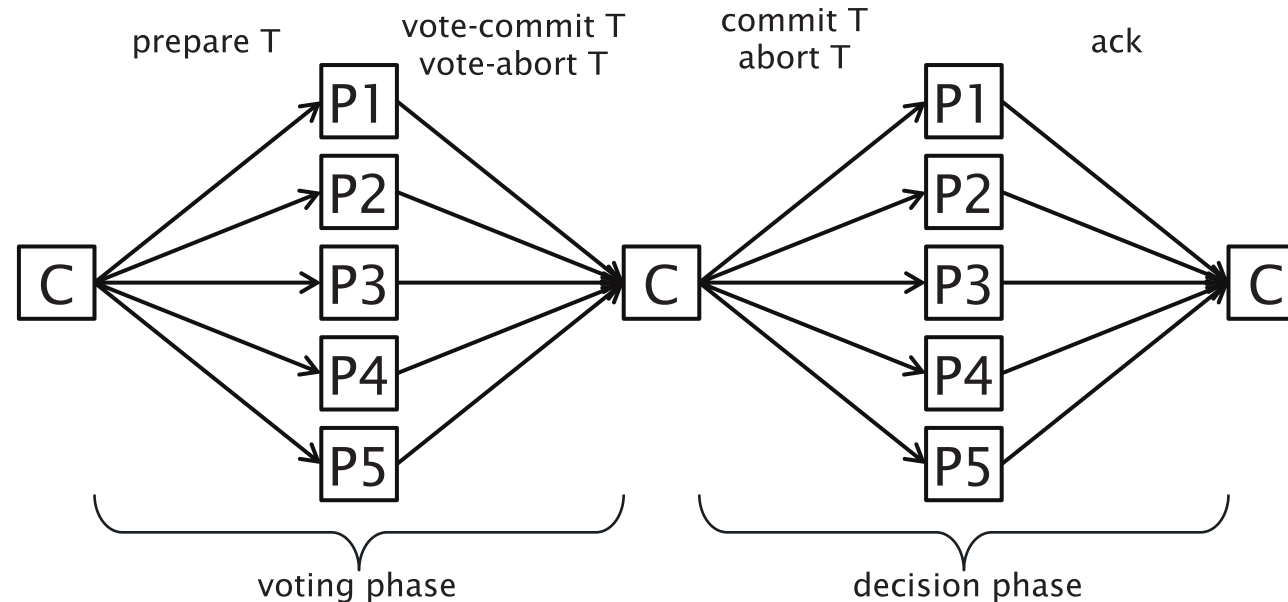
- No inter-participant communication



Centralised 2PC

Communication only between the coordinator and the participants

- No inter-participant communication



UNIVERSITY OF Southampton

Linear 2PC

- First phase from the coordinator to the participants
- Second phase from the participants to the coordinator
- Participants may unilaterally abort

UNIVERSITY OF Southampton

Linear 2PC

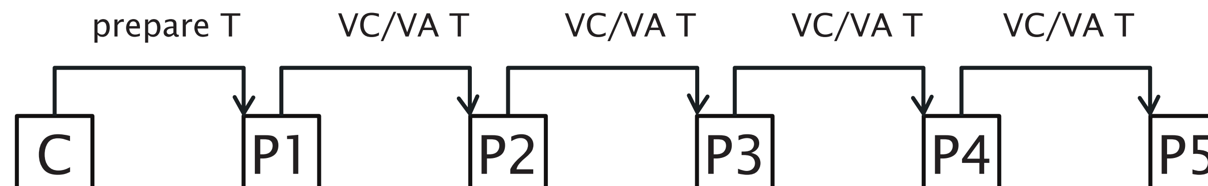
- First phase from the coordinator to the participants
- Second phase from the participants to the coordinator
- Participants may unilaterally abort

C

UNIVERSITY OF Southampton

Linear 2PC

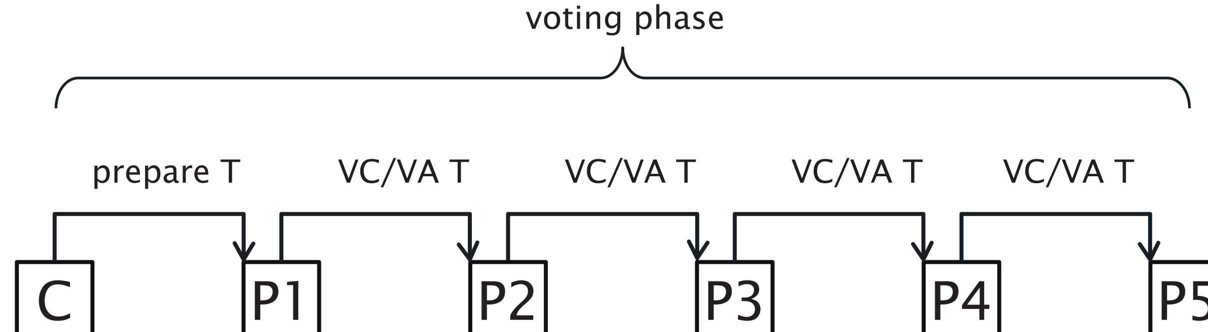
- First phase from the coordinator to the participants
- Second phase from the participants to the coordinator
- Participants may unilaterally abort



UNIVERSITY OF Southampton

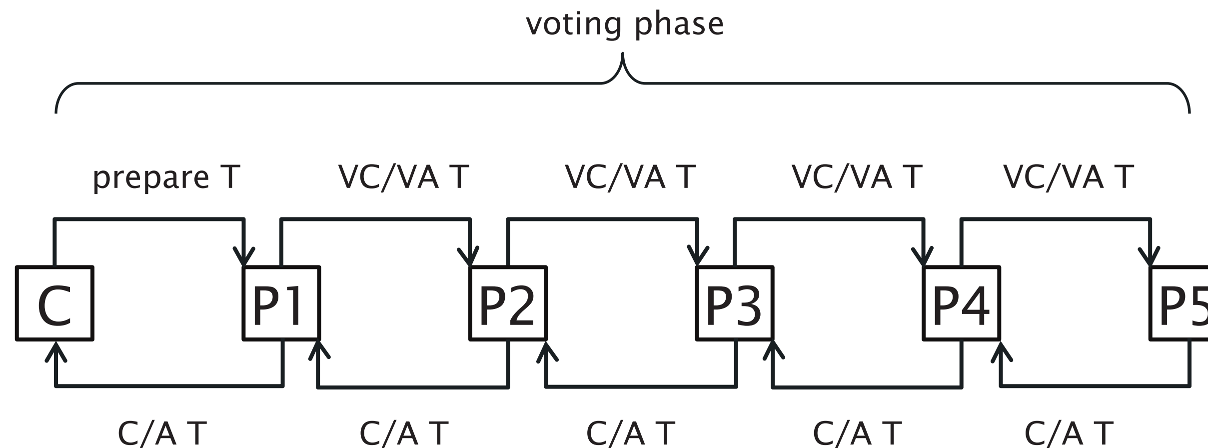
Linear 2PC

- First phase from the coordinator to the participants
- Second phase from the participants to the coordinator
- Participants may unilaterally abort



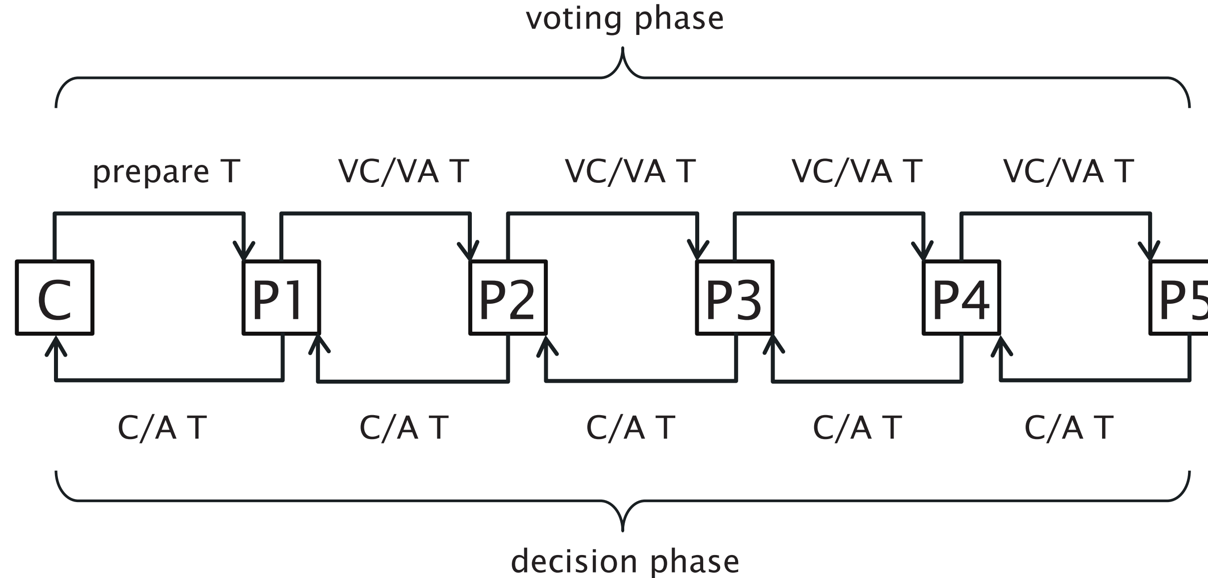
Linear 2PC

- First phase from the coordinator to the participants
- Second phase from the participants to the coordinator
- Participants may unilaterally abort



Linear 2PC

- First phase from the coordinator to the participants
- Second phase from the participants to the coordinator
- Participants may unilaterally abort



Distributed 2PC

- Participants send responses to coordinator plus all other participants
- Each participant can individually determine the global decision
- No need for second phase

UNIVERSITY OF Southampton

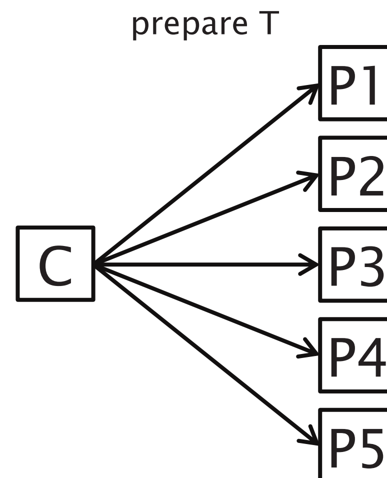
Distributed 2PC

- Participants send responses to coordinator plus all other participants
- Each participant can individually determine the global decision
- No need for second phase

UNIVERSITY OF Southampton

Distributed 2PC

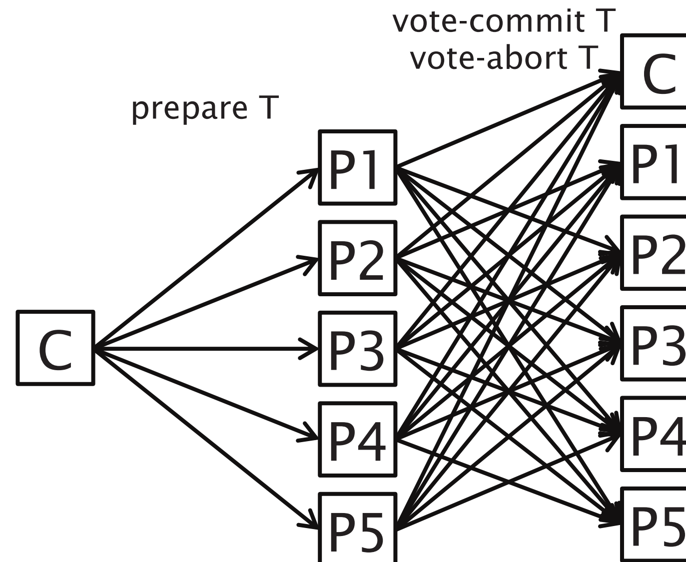
- Participants send responses to coordinator plus all other participants
- Each participant can individually determine the global decision
- No need for second phase



Southampton UNIVERSITY OF

Distributed 2PC

- Participants send responses to coordinator plus all other participants
- Each participant can individually determine the global decision
- No need for second phase



Comparison

- Linear 2PC involves fewer messages
- Centralised 2PC provides opportunities for parallelism
- Linear 2PC has worse response time performance
- Both Linear 2PC and Distributed 2PC require the coordinator to provide identities of all participants in the "prepare T" message

Replication

Replication

So far, we've assumed that there is a single copy of any given data item

We may wish to replicate data for several reasons:

- **System availability**
  Remove single points of failure
- **Performance**
  Reduce communications overhead by moving data closer to its access points
- **Scalability**
  Support growth in accesses while keeping response times acceptable

Replication

Need to distinguish a *logical data item* x from the *physical data items* $x_1 \ldots x_n$ that are its replicas

(note that not all data items may be replicated)

If the system provides *replication transparency*, transactions will issue read and write operations on the logical data items

Consistency

Transactions which access replicated data items are *global transactions*

- must be executed at multiple sites
- *local transactions* access only non-replicated data items

When a global transaction updates copies of a replicated data item on different sites, the values of the copies may be different at a given point in time

A replicated database is in a *mutually consistent* state if all copies of each of its data items have identical values

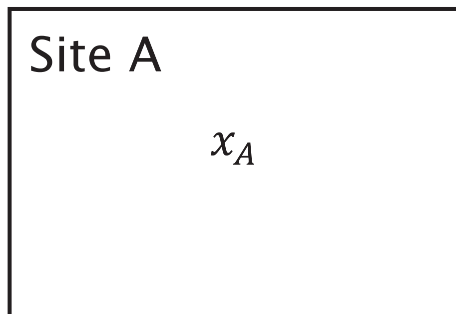
- **Strong mutual consistency:** all copies of a data item have the same value at the end of an update transaction
- **Weak mutual consistency:** all copies of a data item eventually have the same value (also known as *eventual consistency*)

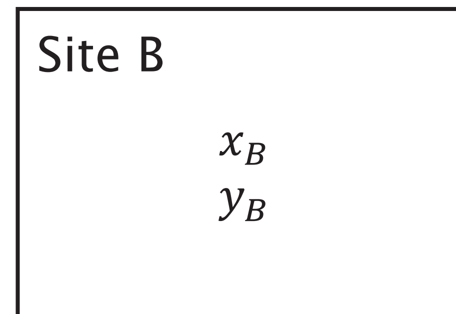
Consistency

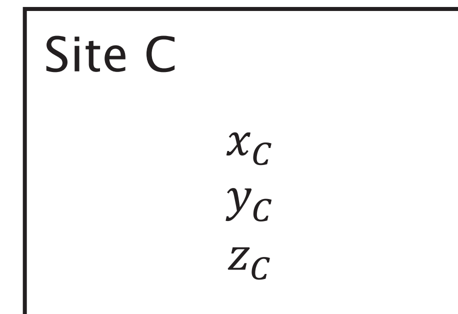
Mutual consistency and transactional consistency are not the same!

Transactional consistency requires serialisability

Consider the following example:







We have the following three transactions:

- T1: x=20; write(x); commit
- T2: read(x); y=x+y; write(y); commit
- T3: read(x); read(y); z=(x*y)/100; write(z); commit

Consistency

The histories for sites A,B,C are as follows:

- H_A = $write_1(x_A)$; $commit_1$
- H_B = $write_1(x_B)$; $commit_1$; $read_2(x_B)$; $write_2(y_B)$; $commit_2$
- H_C = $write_2(y_C)$; $commit_2$; $read_3(x_C)$; $read_3(y_C)$; $write_3(z_C)$; $commit_3$; $write_1(x_C)$; $commit_1$

Serialisation order in H_B is T1;T2, whereas that in H_C is T2;T3;T1

Global history is not serialisable (therefore transactional consistency not preserved)

However, $x_A=x_B=x_C$ and $y_B=y_C$ (therefore mutual consistency is preserved)

We can introduce the notion of *one-copy serialisability* (1SR):

- The effects of transactions on replicated data should be the same as if they had been performed serially on unreplicated data

Updates

Where are updates to databases first performed?

- **Centralised:** updates performed first on a master copy
  - **Single master:** single site holds the master copy of all data items
  - **Primary copy:** sites holding master copies of each data item may be different
- **Distributed:** updates applied first to any replica
  (also referred to as *multi-master* or *update anywhere*)

Update Propagation

How are updates propagated to all replicas?

- **Eager propagation:** changes are propagated during the lifetime of the global transaction
- **Lazy propagation:** changes may be propagated after the global transaction has committed

UNIVERSITY OF Southampton

Eager Update Propagation

When transaction commits, all replicas have same value: *strong mutual consistency*

- Updates to replicas are atomic, as part of transaction
- Doesn't matter which replicas get read subsequently – they're all the same

Read-one/write-all (ROWA) behaviour is characteristic of eager approaches

"during the lifetime" allows some flexibility:

- **Synchronous propagation**: apply update to all replicas when write is performed
- **Deferred propagation**: apply update to all replicas at the end of the transaction before commit (include updates in "prepare T" message as part of 2PC?)

Lazy Update Propagation

Transaction commits before changes are propagated

- Lower response times than with eager update propagation

Updates are subsequently propagated in *refresh transactions* which contain the update operations from the transaction

Aim for eventual consistency (weak mutual consistency)

Update Propagation

Centralised/distributed and eager/lazy are orthogonal concerns

Replication protocols exist for each combination

Replication transparency is a related concern that affects centralised approaches

Eager Centralised (Single Master) Limited Transparency

Read-only transactions are submitted to TM on local site

Update transactions must be submitted to the master site TM (limited transparency)

- read(x) operations performed on master copy x_m
- write(x) operations performed on master copy x_m, and then write(x) forwarded to all other sites (either deferred or synchronously)

Update execution at other sites needs to carry out conflicting updates in the same order as on the master—order by timestamps

Eager Centralised (Single Master) Full Transparency

For full replication transparency, need to submit all transactions to local site TM

Coordinating TM at application site

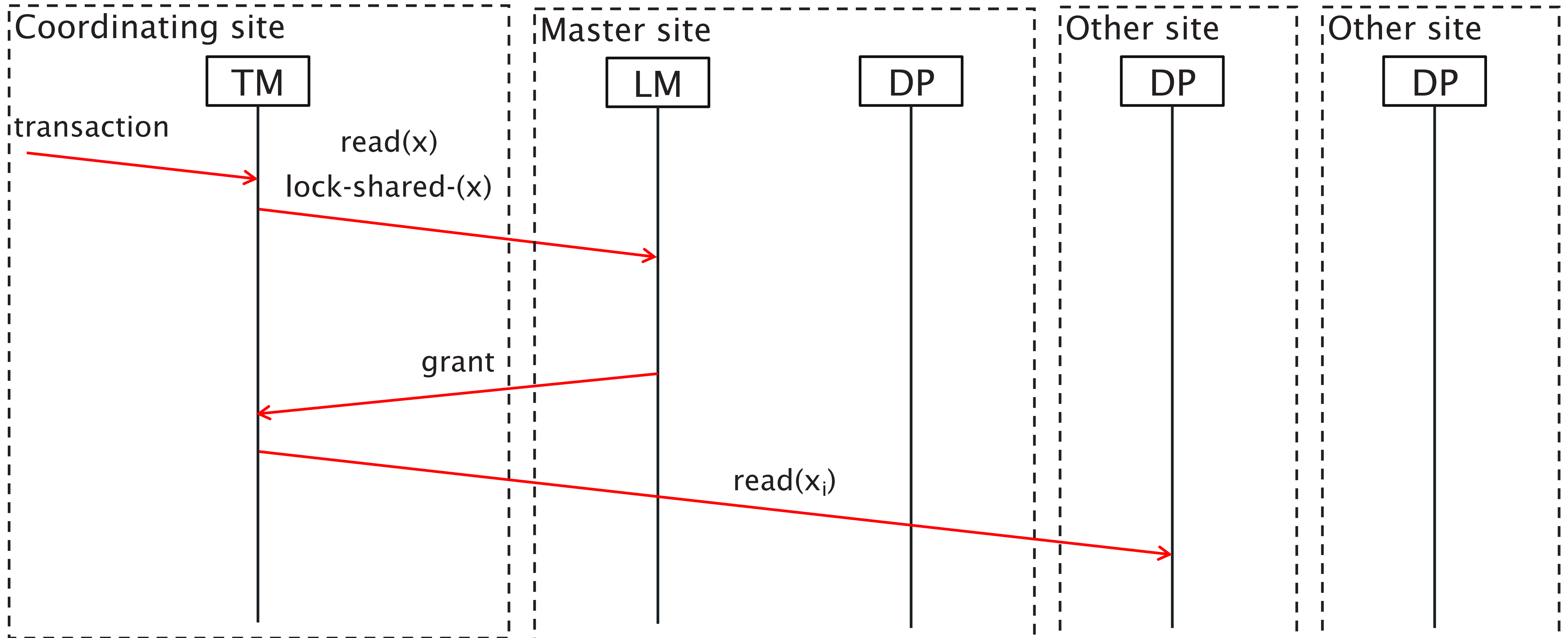
- Acts as coordinator for both read-only and update transactions

Simple approach: forward all operations to master site for execution

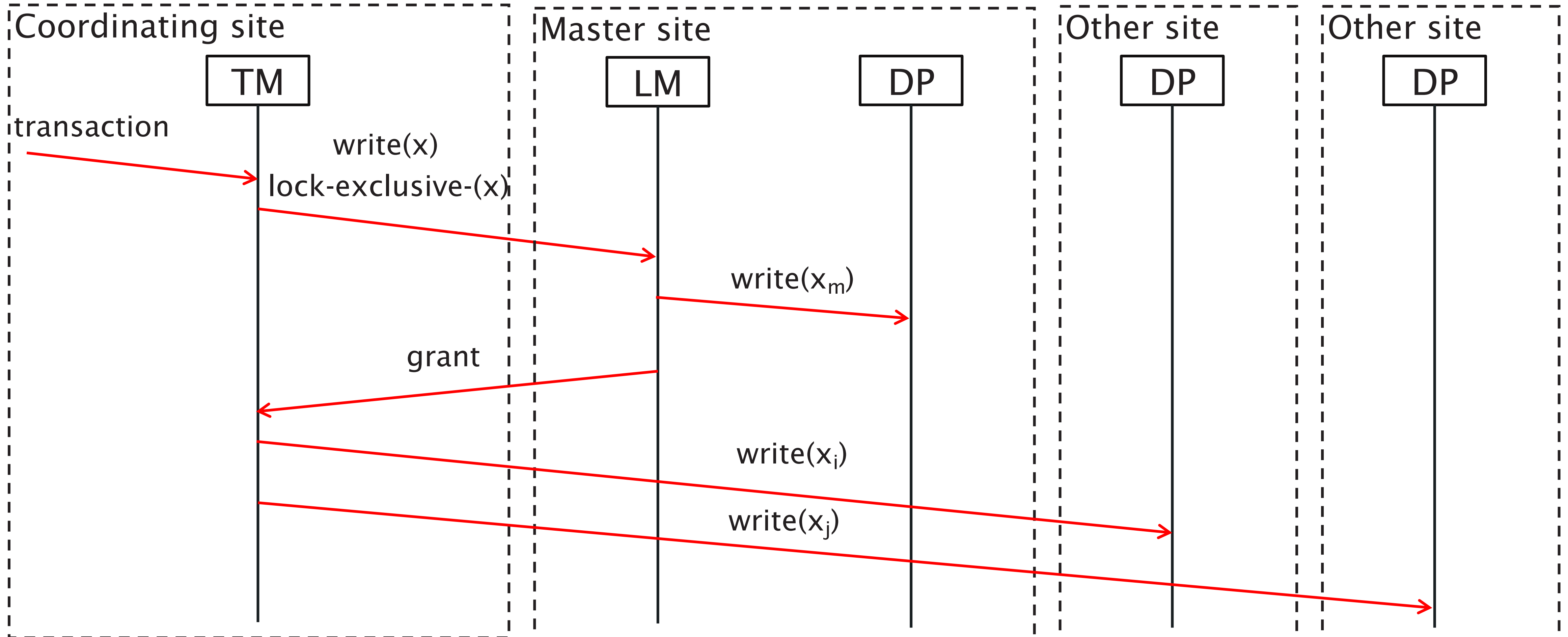
- Potential heavy load on master site
- Can we do better?

UNIVERSITY OF Southampton

Eager Centralised (Single Master) - Reads



Eager Centralised (Single Master) - Writes



Eager Centralised (Primary Copy) Full Transparency

Each replicated data item can have a different master (the primary copy)

- No single master to determine global serialisation order

Coordinating TM at application site

- Forward each operation to the primary site for that data item
- Primary site propagates operation to other sites

Requires that each site knows the location of the primary copy of each data item

LM on each site is responsible for the data items for which it has the primary copy

Eager Distributed

No master copies of data items

Updates can originate on any site

- Writes applied first to local replica, then propagated to other replicas
- Changes made by propagated write become permanent on commit

Concurrent conflicting writes must be executed in the same order at every site – use existing concurrency control techniques

UNIVERSITY OF Southampton

Lazy Centralised (Single Master) Limited Transparency

Read-only transactions are submitted to TM on local site

Update transactions must be submitted to the master site TM

- read(x) operations performed on master copy x_m
- write(x) operations performed on master copy x_m
- Once update transaction is committed, update is propagated to other sites as a *refresh transaction*

Ordering of refresh transactions preserved by using timestamps from master site

Lazy Centralised (Single Master) Full Transparency

Two potential problems:

1. Difficult to achieve a one-copy serialisable global history
2. A transaction may not see its own updates

UNIVERSITY OF Southampton

Problem 1: Set up

- A master site M and a second site S
- Data items x and y, replicated on both M and S
- Two transactions:
  - T1: read(x); write(y); commit (submitted to S)
  - T2: write(x); write(y); commit (submitted to M)

Problem 1: Execution trace

1. T1 submits $read_1(x)$ to site S, which executes it
2. T2 submits $write_2(x)$ to site M, which executes it
3. T2 submits $write_2(y)$ to site M, which executes it
4. T2 submits $commit_2$ to site M
5. Site M commits T2
6. T1 submits $write_1(y)$ to site S, which forwards it to M
7. Site M executes write1(x), and sends confirmation to S
8. T1 submits $commit_1$ to site S, which forwards it to M
9. Site M commits T1, and sends confirmation to S
10. Site S commits T1
11. Site M sends refresh transaction for T2 to site S, which executes and commits it
12. Site M sends refresh transaction for T1 to site S, which executes and commits it

Problem 1

This gives us the following histories on sites M and S:

H_M = $write_2(x_m)$; $write_2(y_m)$; $commit_2$; $write_{1R}(x_m)$; $commit_{1R}$

H_S = $read_1(x_S)$; $commit_1$; $write_{2R}(x_s)$; $write_{2R}(y_s)$; $commit_{2R}$; $write_{1R}(x_s)$; $commit_{1R}$

The resulting global history is non-1SR

For 1SR, refresh transactions must be executed in the same order as the original transactions are committed on the master site

One approach: use timestamps on transactions and data items to ensure that correct values are read.

Bernstein, P. et al (2006) Relaxed-currency serializability for middle-tier caching and replication. *Proceedings of the 2006 ACM SIGMOD international conference on management of data*. pp.599-610

University of Southampton

Problem 2: Set up

- A master site M and a second site S
- A data item x, replicated on both M and S
- One transaction:
  - T1: write(x); read(x); commit (submitted to S)

Problem 2: Execution trace

1. T1 submits $write_1(x)$ to site S, which forwards it to M
2. write1(x) is executed at site M, and confirmation sent back to S
3. T1 submits $read_1(x)$ to site S, which executes it
4. T1 submits $commit_1$ to site S, which forwards it to M
5. Site M commits T1 and sends confirmation sent back to S
6. Site S commits T1
7. Site M sends refresh transaction for T1 (containing write(x)) to site S
8. Site S executes the refresh transaction and commits it

(what is the value of x that T1 reads in step 3?)

Problem 2

Possible approach:

- Maintain a list of the updates that a transaction performs
- When a read() is executed, check list
- If data item being read has already been written during this transaction, execute read at mastere

Lazy Distributed

Execution on coordinating site is easy

- Make changes, send refresh transactions

Management and reconciliation of updates at other sites is difficult

UNIVERSITY OF Southampton

The CAP Theorem

The CAP Theorem

In any distributed system, there is a trade-off between:

- Consistency

  Each server always returns the correct response to each request

- Availability

  Each request eventually receives a response

- Partition Tolerance

  Communication may be unreliable (messages delayed, messages lost, servers partitioned into groups that cannot communicate with each other), but the system as a whole should continue to function

The CAP Theorem

CAP is an example of the trade-off between safety and liveness in an unreliable system

- Safety: nothing bad ever happens
- Liveness: eventually something good happens

We can only manage two of three from C, A, P

- Typically we sacrifice either availability (liveness) or consistency (safety)

UNIVERSITY OF Southampton

Next Lecture:
Data Warehousing