Distributed Optimistic Algorithm

- Assumptions
    1. Synchronized clocks
    2. MTD (max, trans, delay) can be defined
- *Step* 1: Read
- *Step* 2: Compute
- *Step* 3: Transaction is broadcasted to all nodes at time $\pi(V_i)$ (time when computation finishes and T_i is ready for validation)
- *Step* 4: At time $\pi(V_i)$ + MTD, all nodes start validation of T_i. (Note $\pi(V_i)$ is attached to T_i) and if T_i reaches before $\pi(V_i)$ + MTD, it must wait

Distributed Optimistic Algorithm

- Step 5:
  IF validation succeeds, all nodes write S(wi)
  ELSE all nodes except "X" ignore T_i
  At node X, T_i is restarted and repeated until T_i validates

THEOREM:

The dist. opt. algorithm produces only correct histories at each node and all histories are identical.

PROOF:

ONLY correct histories are produced. Because of Theorem 1
ELSE UPDATE $S(R_i)$ and repeat from step 2

Centralized Optimistic Algorithm

A node(C) is chosen as central node

CASE 1: Validation takes place only at central node When T_i arrives at a node "X"

1. Read $S(R_i)$
2. Execute (compute) and get $S(w_i)$

   Note $S(w_i)$ is semantic write set (actual)

   Locking may require syntactic (potential) write set

   $S'(w_i)$ $S(w_i)$ leq $S'(w_i)$

- T_i goes to node C (if $X \neq C$)
- If T_i succeeds, send write set to all nodes

Centralized Optimistic Algorithm

CASE 2: Validation takes place at local node and then at central node

1. Same
2. Same
3. T_i validates at X
4. IF successful, T_i commits at X and is sent to C
5. ELSE UPDATE $S(R_i)$ and repeat from step 2
6. If successful at C, send write set to all nodes

   ELSE UPDATE $S(R_i)$ at C and execute at C and repeat validation until successful.

Centralized Optimistic

CASE 1: Validation takes place only at central node only

CASE 2: Validation takes place at local node and then central node

Distributed Optimistic

Validation takes place at all nodes after a delay of MTD (Max. transmission Delay)

When to synchronize (assert concurrency control)

- First access to an entity

  (locking, pessimistic validation)
- At each access

  (granularity level)
- After all accesses and before commitment

  (optimistic validation)

Information needed for synchronization

- Locks on database entities
  (system R, INGRES, Rosenkrantz,...)
- Time stamps on database entities
  (Thomas, Reed,...)
- Time stamps on transactions
  (Kung, SDD-1, Schlageter,...)

OBSERVATIONS

- Time stamps more fundamental than locking
- Time stamps carry more information
- Time stamp checking costs more than checking locks

T_1 $\qquad\qquad$ T_2

T_{11}: $X \leftarrow X + 1$ $\qquad$ T_{21}: $X \leftarrow X + 1$

T_{12}: $X \leftarrow 2 * X$

History

Serial $\qquad$ $T_1\ T_2$ or $T_2\ T_1$

$\downarrow$ $\qquad$ $\downarrow$

$f_{12}(f_{11}(f_{21}(x)))$

f: Herbrand fn. $\qquad$ $f_{21}(f_{12}(f_{11}(x)))$

non serializable $\qquad$ T_{11}, T_{21}, T_{12}

$f_{12}(f_{21}(f_{11}(x)))$

So given interpretation of f_{ij}'s allows us to include histories which are not allowed by SERIALIZABILITY and hence allows us higher concurrency

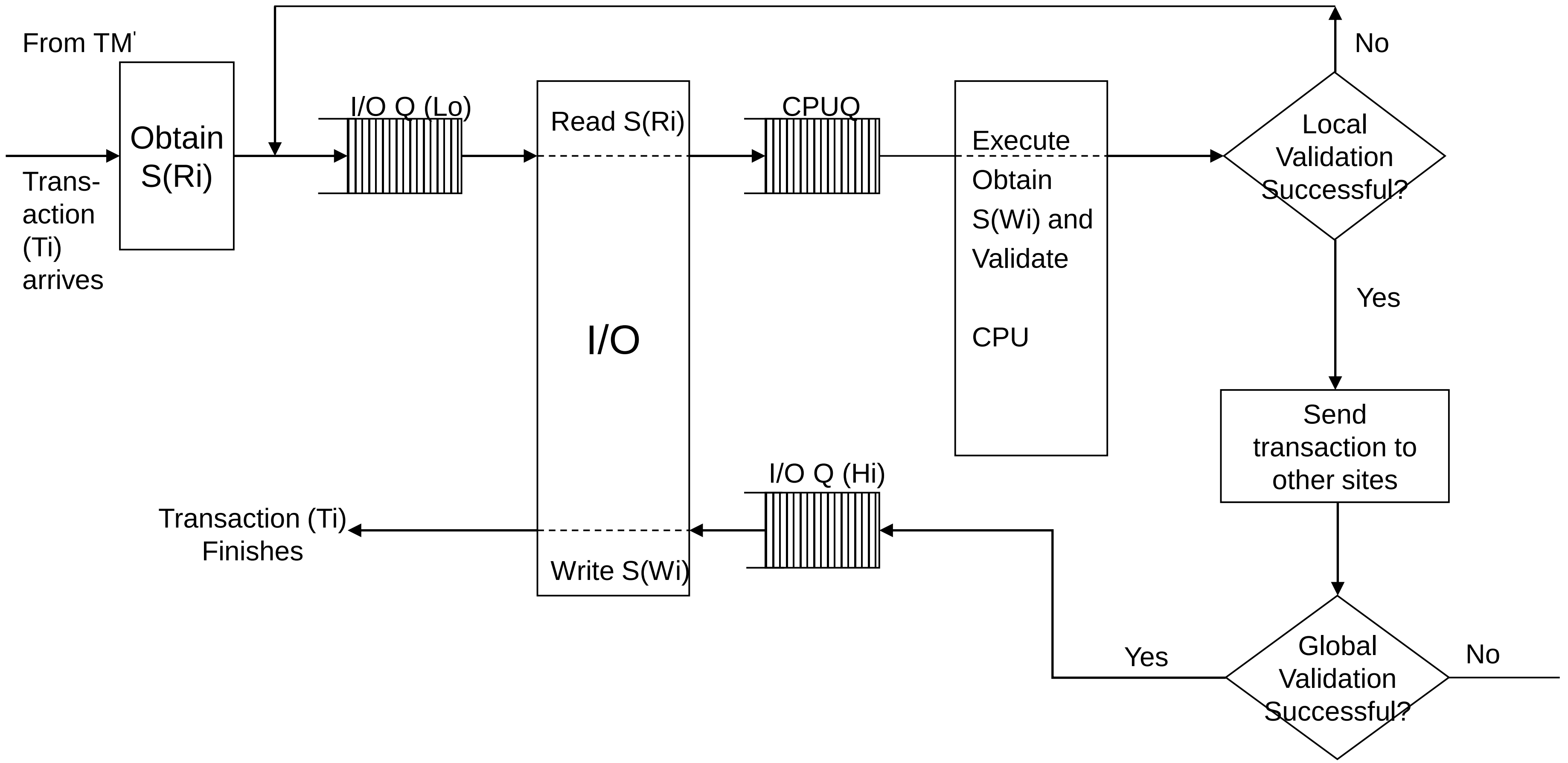
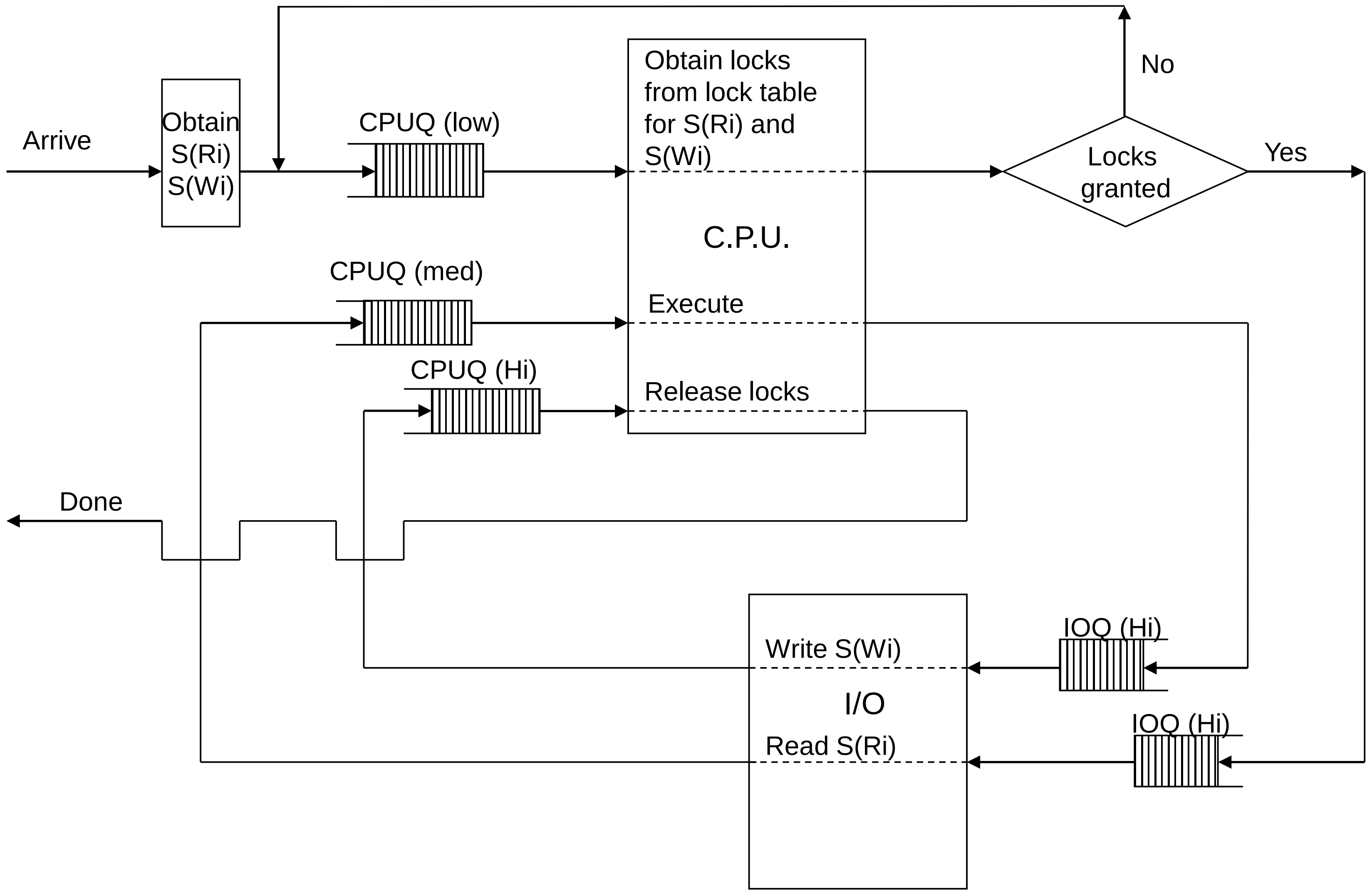

From TM'
Trans-
action
(Ti)
arrives
Obtain
S(Ri)
I/O Q (Lo)
Read S(Ri)
I/O
Write S(Wi)
CPUQ
Execute
Obtain
S(Wi) and
Validate
CPU
Local
Validation
Successful?
No
Yes
Send
transaction to
other sites
Global
Validation
Successful?
Yes
No
I/O Q (Hi)
Transaction (Ti)
Finishes

Figure 2



Locking Mechanism (Pessimistic)

Steps of a Transaction (Ti) Non-Locking Algorithm

1. The transaction (T_i) arrives in the system
2. The read $S'(R_i)$ and write $S'(W_i)$ set of the transaction is obtained. These sets are syntactic
3. The transaction goes to an I/O queue to obtain item values for read set $S'(R_i)$
4. The transaction goes to CPU queue and completes execution to obtain write set values. Also actual read set $S(R_i)$ and write set S(wi) are determined. These sets represent semantic information
5. The transaction's read sets are validated against other active transactions according consistency constraints (such as serializability)

Steps of a Transaction (Ti) … (cont)

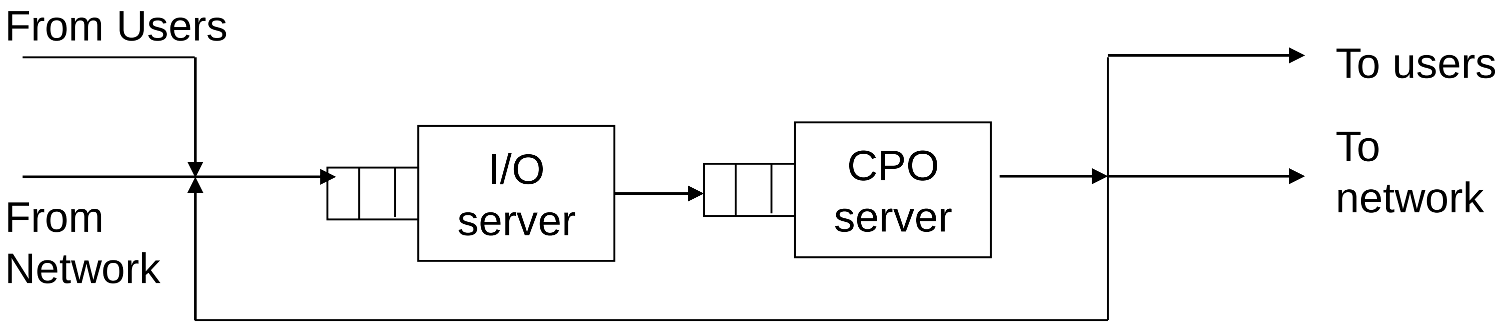
6. If validation fails due to conflict among transaction T_i and some other transaction T_j, then one of the transaction is required to repeat its execution. For example, if consistency constraint is "strongly serializable", then the transaction that arrived later (let us say T_i) is selected for re-execution. Moreover the conflict among T_i and T_j is resolved and the values of $S'(R_i)$ are updated with values from $S(W_j)$ at the time of validation. This is useful because T_i does not have to go and do its I/O once again.
7. The transaction is sent to CPU queue to do its computation.
8. The transaction T_i's write set is validated against write set of some transaction T_j (that has not completed but arrived before T_i). If conflict occurs, then T_i is delayed and writes after T_j writes in the database.

Steps of a Transaction (Ti) … (cont)

9. The transaction goes to an I/O queue and update its write set $S(W_i)$.
10. The transaction T_i waits in memory for validation against transactions that arrived in the interval between its arrival time and validation time.

Performance Techniques

- Complexity
- Analytical
- Simulation
- Empirical



Performance model at each node

Parameters

1. Arrival rate
2. Base set (size of write set/read
4. Size of database
5. Number of sets
6. Transmission delay
7. CPU time
8. I/O time
9. Retry delay
10. Read only trans/write & read trans ratio
11. Multiprogramming level
12. Degree of conflict