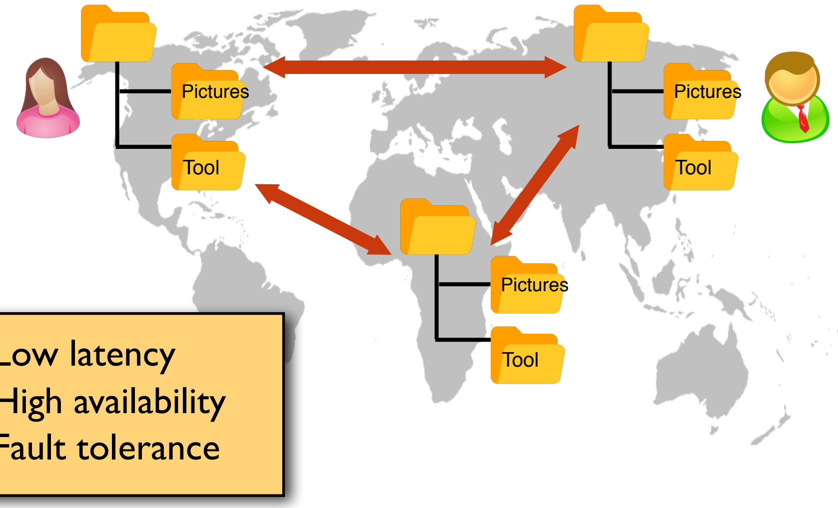
Co-design and Verification of an Available File System

Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster

File System Replication



- –Low latency
- –High availability
- –Fault tolerance

POSIX File Systems vs. Distribution

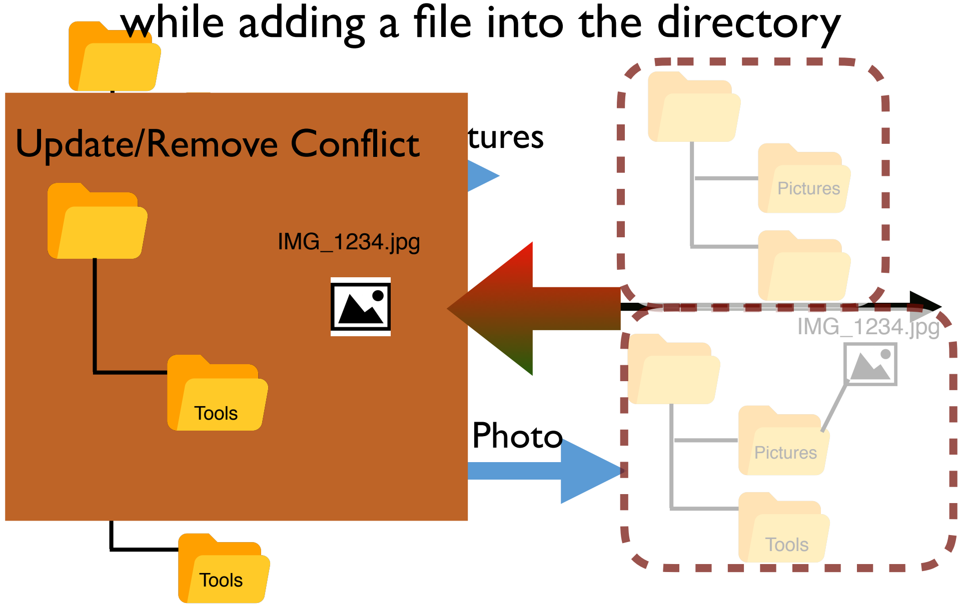
POSIX:

- Assumes operations occur in a total order
- Requires a synchronous, strong consistency model
- Synchronisation is costly and not available under partition
- In practice, concurrency conflicts are rare

Distribution:

- No synchronisation: processes an update locally, propagates effects to other replicas later.
- Weakens consistency and causes conflicts

Conflict Example= removing a directory while adding a file into the directory



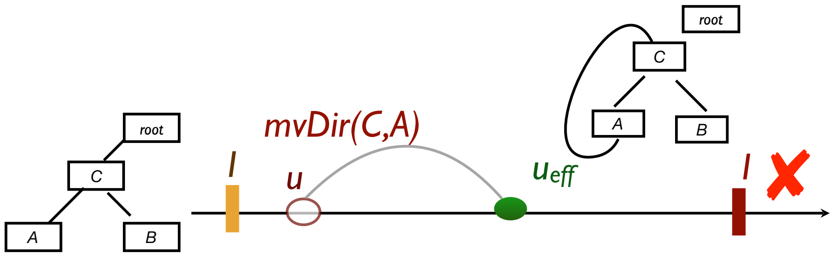
Safety

- Convergent: do replicas that delivered the same updates have the same state?
- Is the invariant preserved?
  - Sequential: single operation in isolation maintains the invariant
  - Concurrent execution maintains the invariant

Tree Invariant

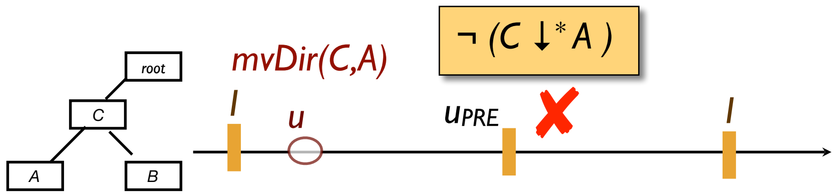
- Has a fixed root node
- *Root* is an ancestor of every node in the tree (reachability)
- Every node, which has a name has <u>exactly one</u> parent, except the *root*
- <u>No *cycle*</u> in the directory structure
- Unique names within a directory

Example= sequential move operation fails



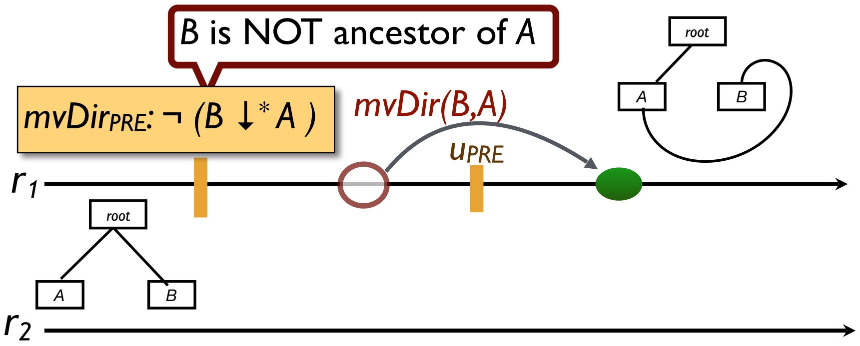
Example= do not move directory under self

C is NOT ancestor of *A*



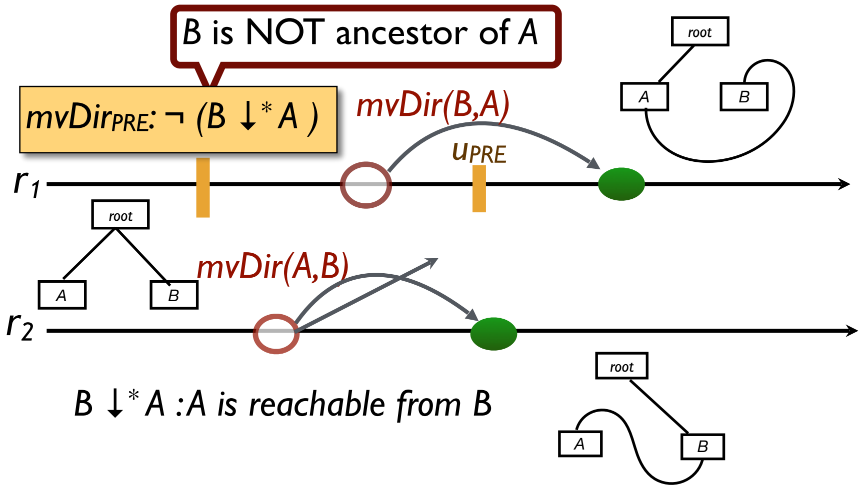
$C \downarrow^* A$: *C is reachable from A*

Example= concurrent moves fails



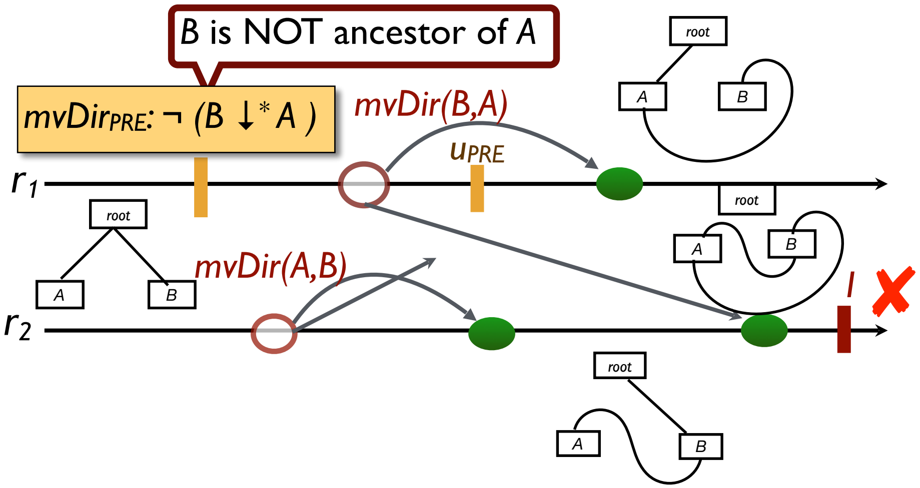
B ↓ A :A is reachable from B*

Example= concurrent moves fails



B ↓ A :A is reachable from B*

Example= concurrent moves fails



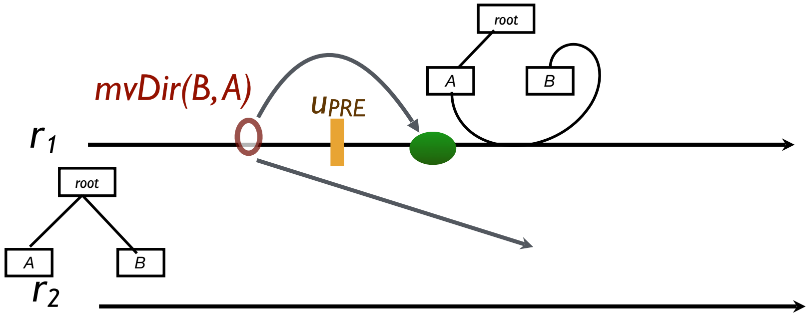
Concurrency Control

Tokens≈ concurrency control abstractions
Tokens = {τ, …}

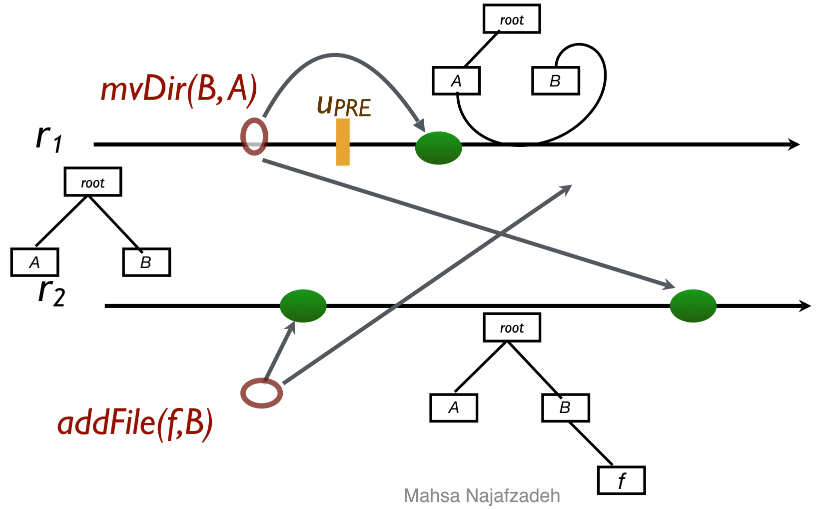
Conflict relation ⋈ ⊆ Tokens × Tokens
Example - mutual exclusion tokens:
Tokens = {τ}; τ ⋈ τ
An operation's generator may acquire a set of tokens

Operations associated with conflicting tokens cannot be concurrent

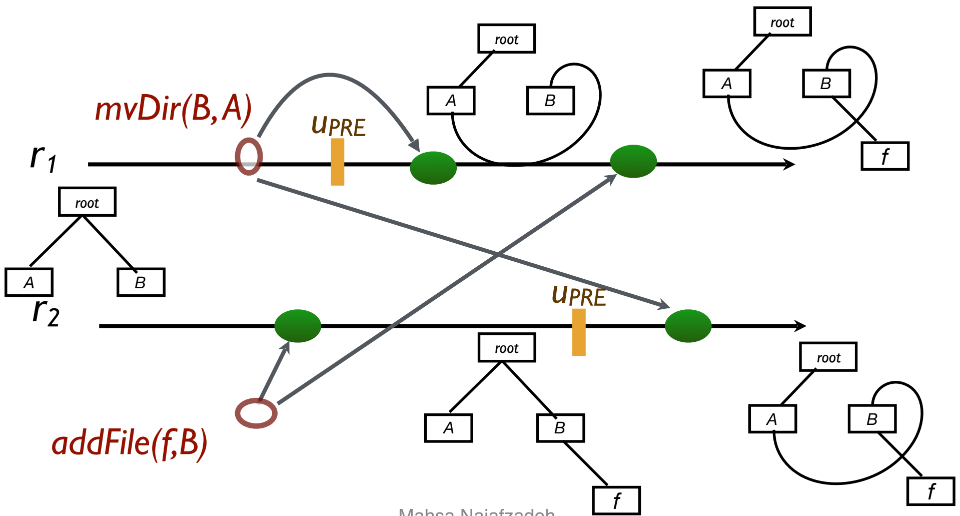
Example= moving a directory while updating its content is safe



Example= moving a directory while updating its content is ok



Example= moving a directory while updating its content is ok



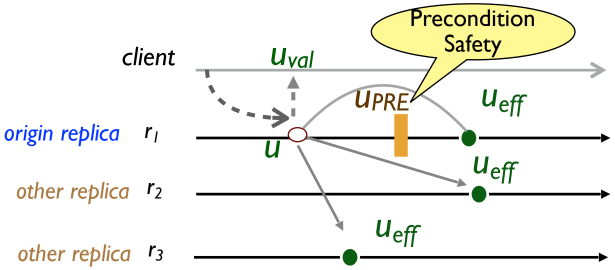
When is Synchronization Necessary?

- CAP theorem: Either (Strong) Consistency or Availability, not both, when Partitions occur
- This is a design trade-off

Our approach:
- Synchronize (CP) only operations where strictly necessary for safety
- Other operations are asynchronous (AP)

Safety = convergent + invariants

Model

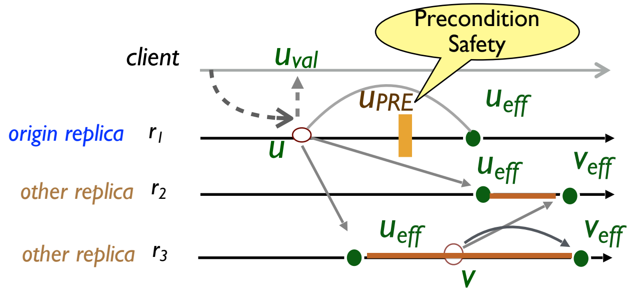


Generator (@origin) reads state from one copy and maps operation *u* to:

Return value: $u_{val} \in State \rightarrow Value$

Effects: $u_{eff} \in State \rightarrow (State \rightarrow State)$

Model

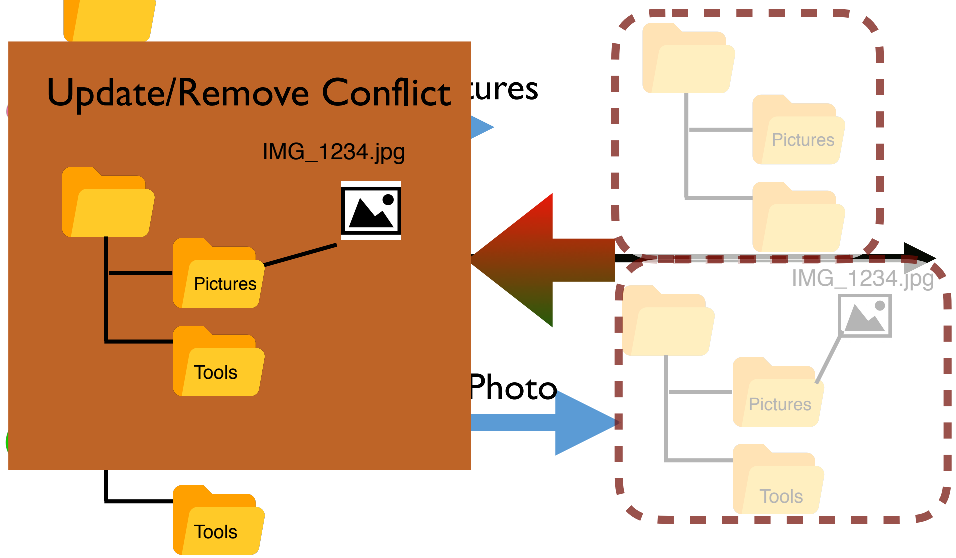


Deliver(@all replicas): causally dependent messages delivered in order

A Mostly-Available, Convergent and Correct File System Design

- Allows common file system operations can run without synchronization except for moves
- Maintains the tree invariant
- Guarantees convergence using replicated data types [Shapiro+ 2011]
  - Name conflicts:
    - Merge directories
    - Rename files
  - Update/Remove conflicts: add-wins directory

Add-wins directory= removing a directory while adding a file into the directory



CISE Analysis: Proves Application is Correct

- Rely-Guarantee reasoning for a causally-consistent system with only polynomial complexity
- Consists of three analysis rules:

  Effector Safety:
  Every effect in isolation execution maintains the invariant I (sequential safety)

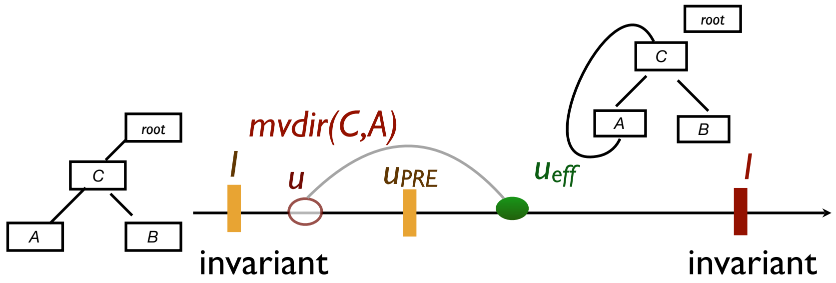
  Commutativity:
  Concurrent operations commute (convergence)

  Stability:
  Preconditions are stable under concurrency (concurrent safety)

If satisfied: the invariant I is guaranteed in every possible execution

[Gotsman et al. POPL 2016 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems]

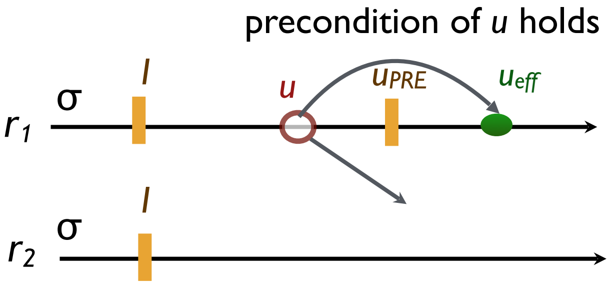
Effector Safety: Example= move requires precondition



- *do not move directory under self*

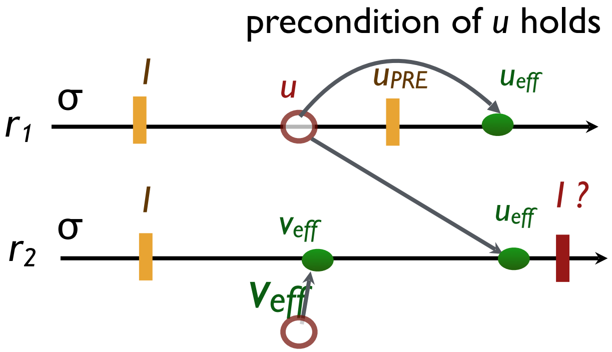
Stability Rule: precondition is stable under concurrent effect

1. Effector Safety: u_{eff} preserves I when executed in any state satisfying u_{PRE}



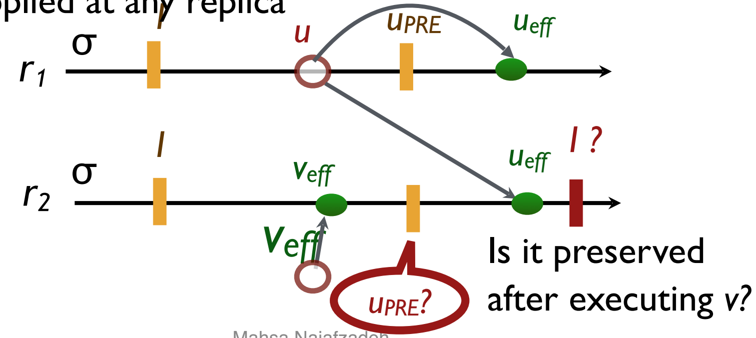
Stability Rule: precondition is stable under concurrent effect

1. Effector Safety: u_{eff} preserves I when executed in any state satisfying u_{PRE}



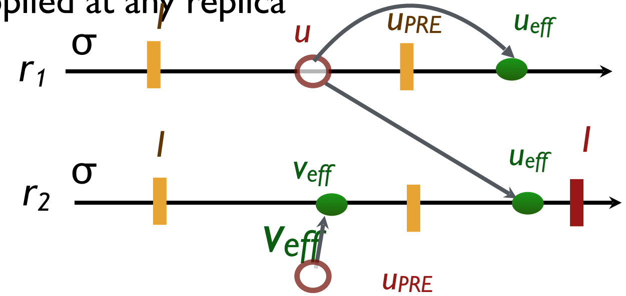
Stability Rule: precondition is stable under concurrent effect

1. Effector Safety: u_{eff} preserves I when executed in any state satisfying u_{PRE}
2. Precondition Stability: u_{PRE} will hold when u_{eff} is applied at any replica

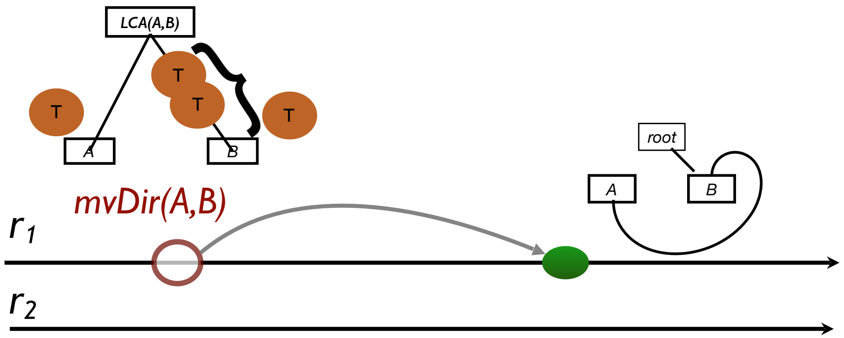


Stability Rule: precondition is stable under concurrent effect

1. Effector Safety: u_{eff} preserves I when executed in any state satisfying u_{PRE}
2. Precondition Stability: u_{PRE} will hold when u_{eff} is applied at any replica

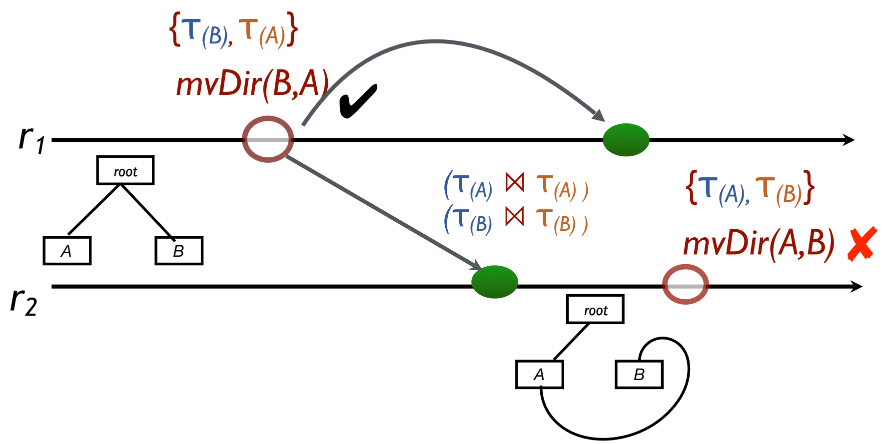


Necessary and Sufficient Concurrency Controls for Move



- Add tokens, avoid *mvDir* || *mvDir*
- A mutually exclusive token for each directory $d \in Dir$: $(\tau_{(d)} \bowtie \tau_{(d)})$

Example: avoid conflicting moves



Verification Results

Applications	#OP	#Tokens	#Invariants	Anomaly	Average Time(ms)
Sequential	7	7	1	NO	278
Concurrent	7	0	1	safety violation	1297
Fully-Asynchronous	7	0	1	duplication	2350
Mostly-Asynchronous	7	2	1	NO	1570

Conclusion

- A rigorous approach for modeling file system behavior for both centralized/synchronous and replicated asynchronous semantics
- Common operations except move to run without concurrency controls
- A hierarchical least-common ancestor concurrency control mechanism is necessary and sufficient for move operations

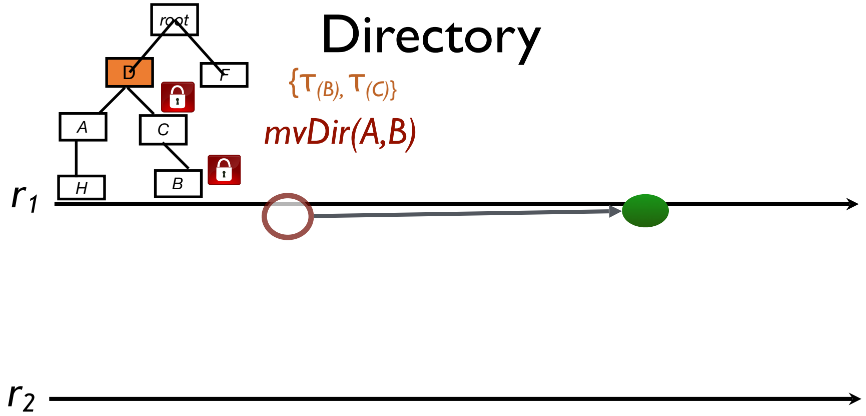
Future Work

- Translate the move concurrency controls into an efficient implementation
- Integrate hard links, devices, and mounts into model
- Reason about the file system behavior in the presence of failures

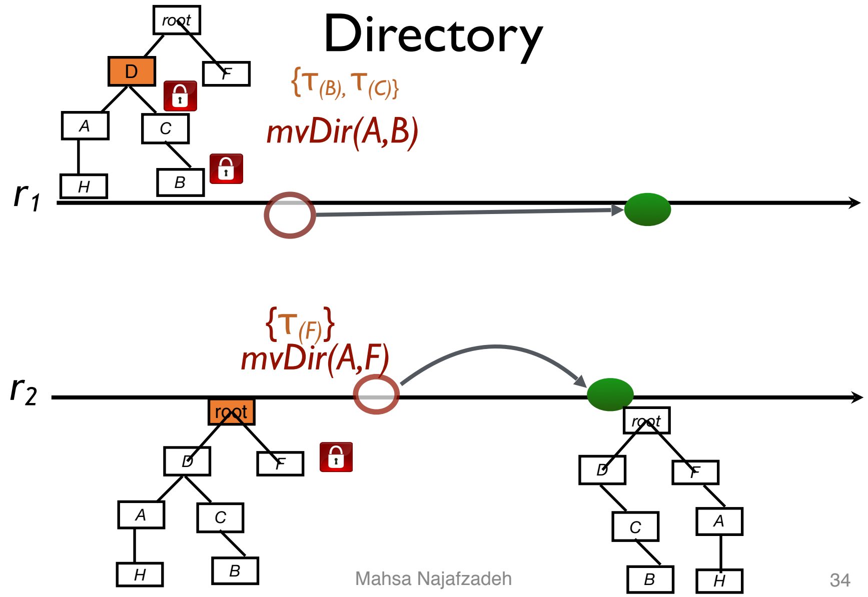
Q/A

Backup Slides

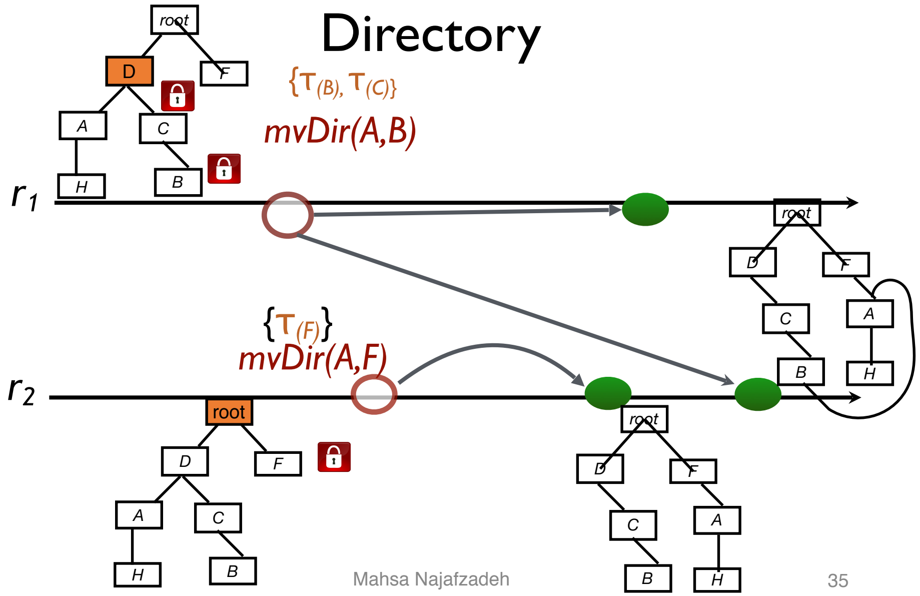
Removing Token Over Source Directory



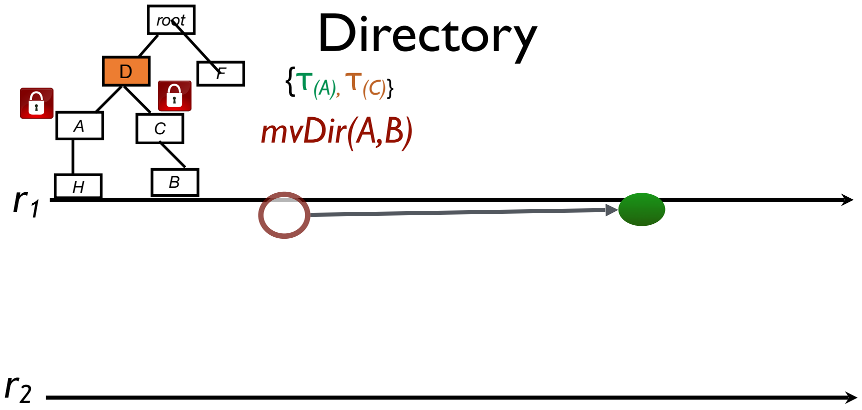
Removing Token Over Source Directory



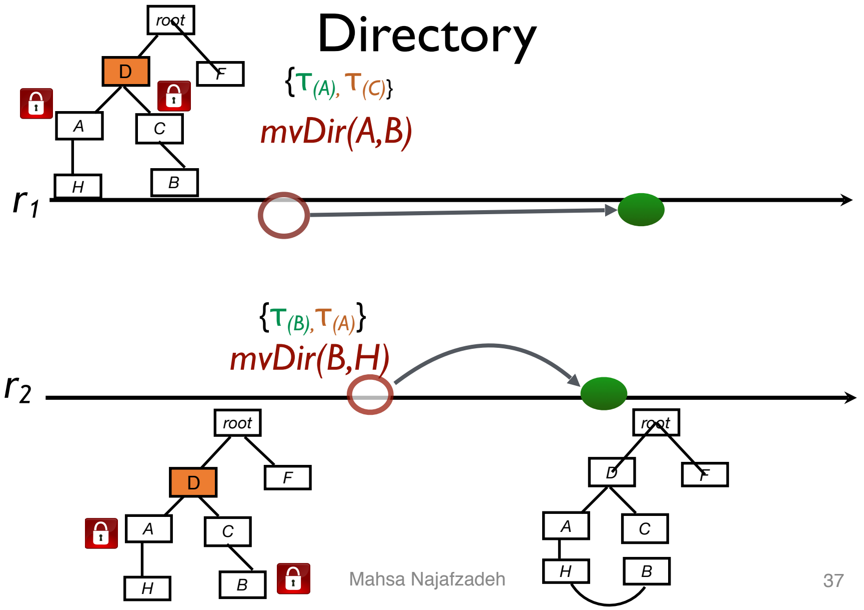
Removing Token Over Source Directory



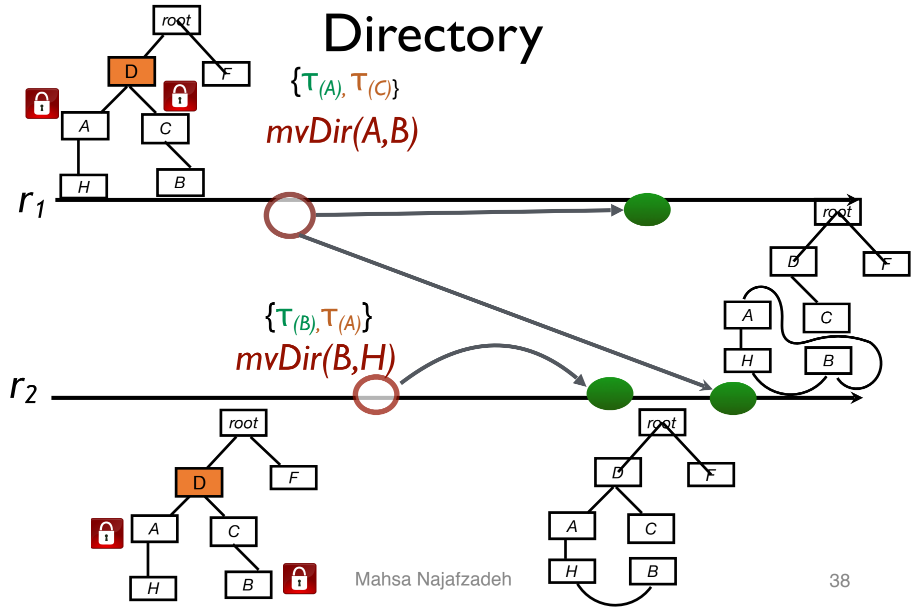
Removing Token Over Destination Directory



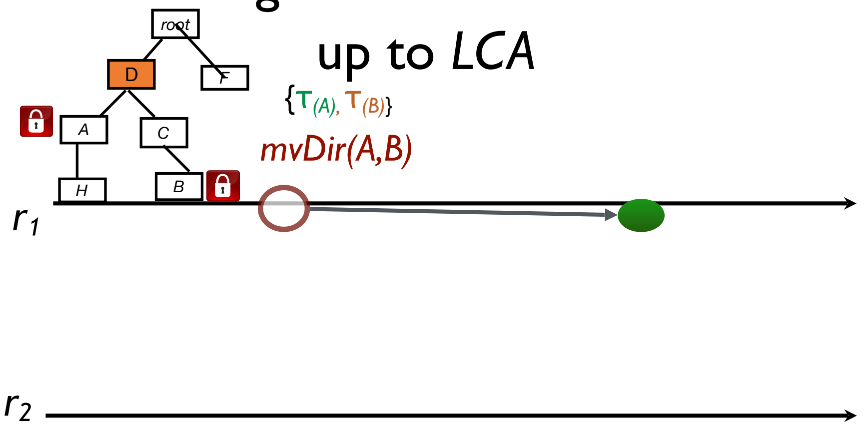
Removing Token Over Destination Directory



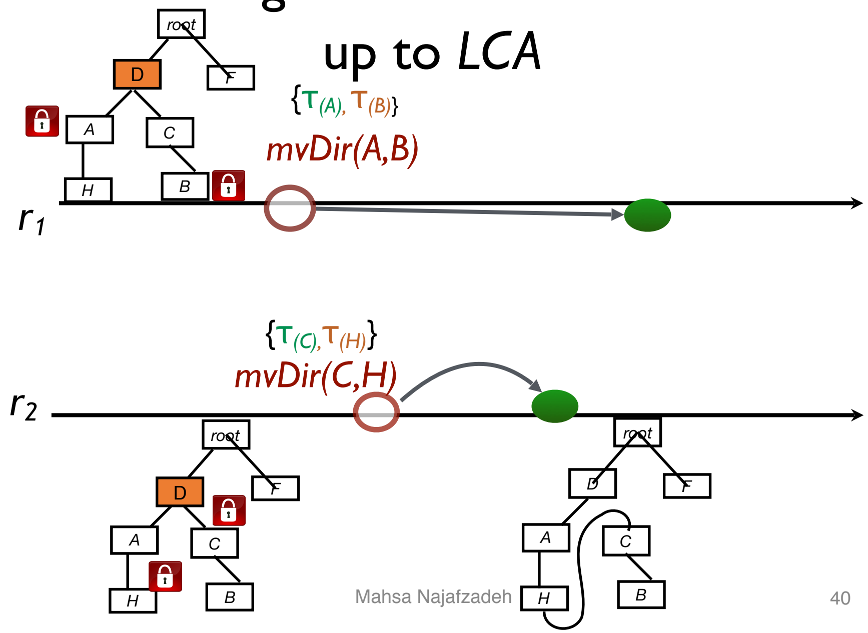
Removing Token Over Destination Directory



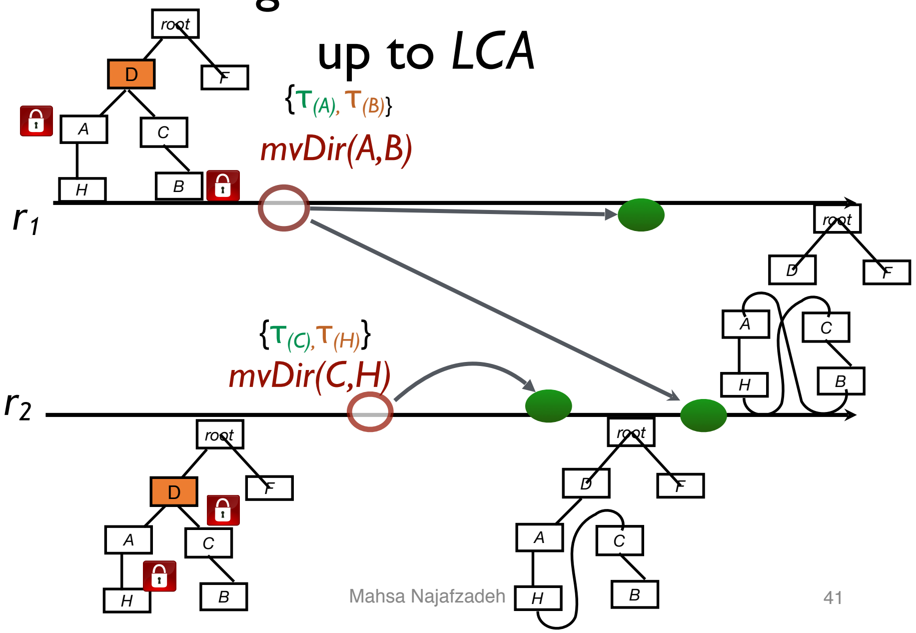
Removing Token Over Ancestors up to *LCA*



Removing Token Over Ancestors up to *LCA*



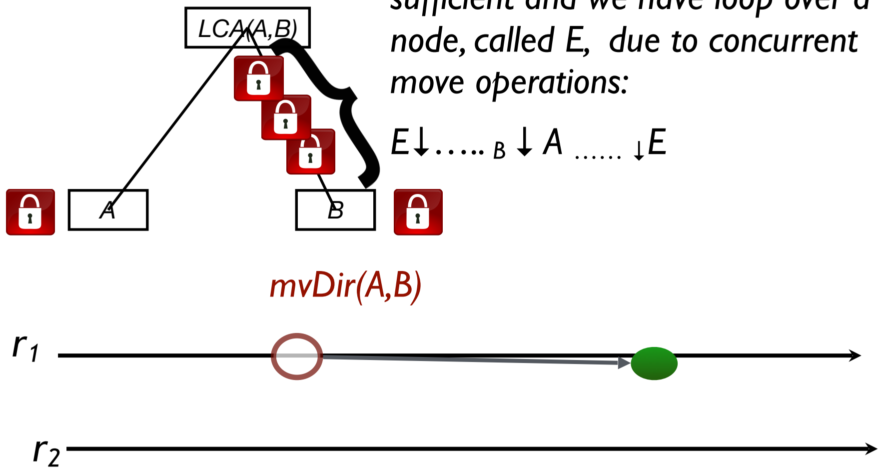
Removing Token Over Ancestors up to *LCA*



Intuition For Move Tokens

Assume that these tokens are not sufficient and we have loop over a node, called E, due to concurrent move operations:

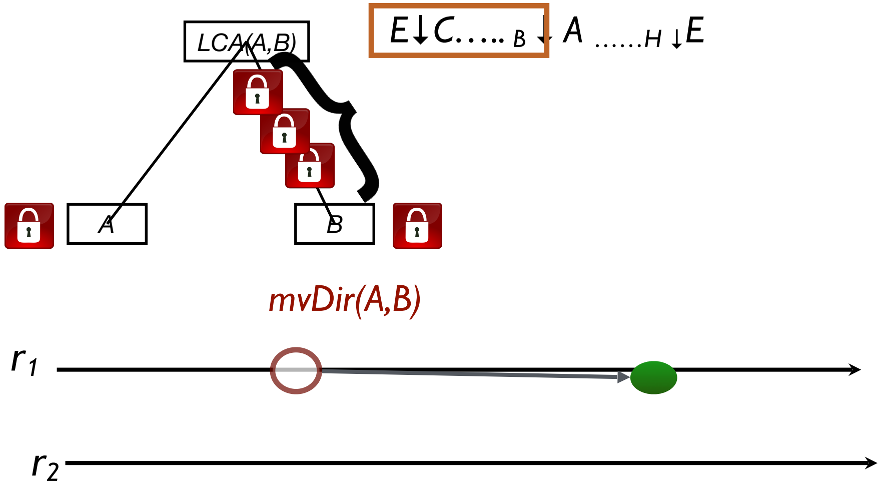
$E\downarrow \ldots_{B} \downarrow A \ldots\ldots {}_{\downarrow}E$



Intuition For Move Tokens

consider the left side of the loop

$\boxed{E\downarrow C\ldots_{B}} \downarrow A \ldots_{H} \downarrow E$



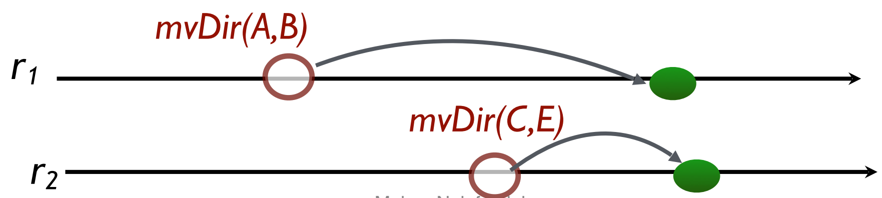
Intuition For Move Tokens

$\boxed{E\downarrow C\ldots_{B}} \downarrow A \ldots_{H} \downarrow E$

The left side implies that one of B's ancestors, called C, concurrently moves to E

mvDir(C,E):

Precondition: Directory *E* is not a descendent of *C*



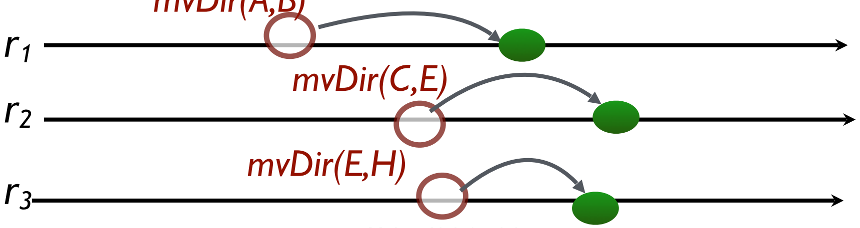
$E \downarrow C \ldots_{B} \downarrow \boxed{A \ldots_{H} \downarrow E}$

Now, consider the right side of loop

The right side implies that E concurrently moves to one of A's descendants, called H

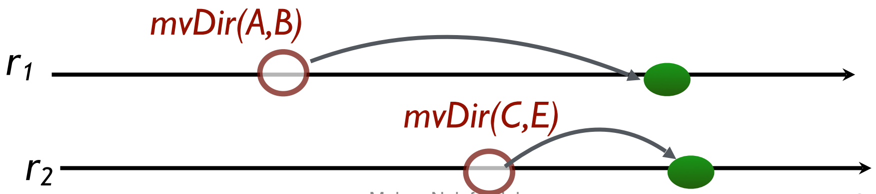
mvDir(E,H)

Tokens over directory *H* up to *LCA(H,E)*

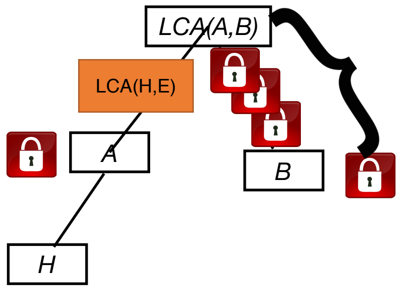


Intuition For Move Tokens

where is *LCA(H,E)*?



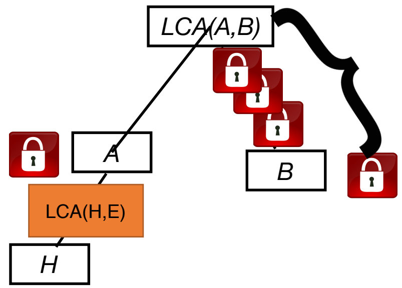
$E \downarrow C \ldots_{B} \downarrow \boxed{A \ldots_{H} \downarrow E}$



1) LCA(H,E) is located between A and LCA(A,B)

in this case moving E to H requires token over A that conflicts with tokens for moving A to B

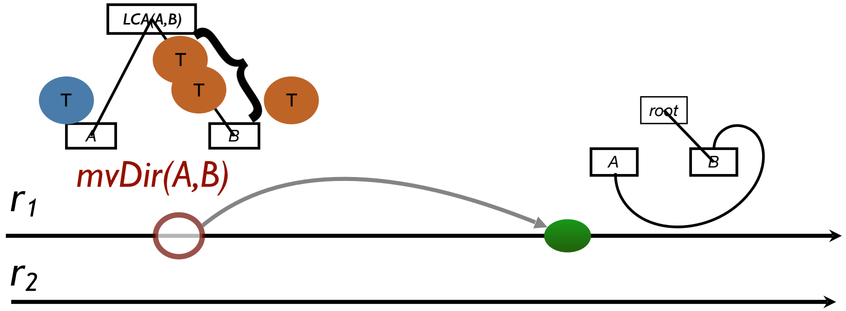
$E \downarrow C \ldots_{B} \downarrow \boxed{A \ldots_{H} \downarrow E}$



2) LCA(H,E) is located under A:

E is concurrently moved under A which is not possible because this move operation needs to acquire tokens conflicting with mvDir(A,B)

Exploiting More Parallelism



- Concurrent moves to the same destination directory
- Conflicting tokens for each directory $A \in Dir$:

source token $\tau_{s(A)}$ and *destination token* $\tau_{d(A)}$

$$(\tau_{s(A)} \bowtie \tau_{d(A)})$$