- Detection of mutual inconsistency in distributed systems (Parker, Popek, et. al.)
- Distributed system with replication for
  - reliability (availability)
  - efficient access
- Maintaining consistency of all copies
  - hard to do efficiently
- Handling discovered inconsistencies
  - not always possible
  - semantics-dependent

- Tradeoffs between

  degree of replication of objects versus access time of object

  availability of object (during partition)

  synchronization of updates

  (overhead of consistency)

- All objects should always be available.

- All objects should always be consistent.

  "Partitioning can destroy mutual consistency in the worst case".

Basic Design Issue:

Single failure must not affect entire system (robust, reliable).

- Previous work
  - Maintain consistency by:
    - Voting (majority consent)
    - Tokens (unique/resource)
    - Primary site (LOCUS)
    - Reliable networks (SDD-1)
    - Disk toking
- Prevent inconsistency at a cost does not address detection or resolution issues.
- Want to provide availability and correct propagation of updates.

- Detecting Inconsistency

  Network may continue to partition or partially merge for an unbounded time.

  Semantics also different with replication:

  naming, creation, deletion…

  names in on partition do not relate to entities in another

  Need globally unique system name, and user name(s).

  Must be able to use in partitions.

- System name consists of a
  - < Origin, Version > pair
  - Origin – globally unique creation name
  - Version – vector of modification history
- Two types of conflicts:
  - Name – two files have same user-name
  - Version – two incompatible versions of the same file. Different mod. Histories (not initial history)
- Conflicting files may be identical…

  Semantics of update determine action
- Detection of version conflicts
  - Timestamp – overkill
  - Version vector – "necessary + sufficient"
  - Update log – need global synchronization

Version vector approach

each file has a version vector

(S_i : u_i) pairs

S_i – site file is resident upon

u_i - # updates on that site

Example: < A:4, B:2; C:0; D:1 >

Compatible vectors:

one is at least as large as the other over all sites in vector

< A:1; B:2; C:4; D:3 > ← < A:0; B:2; C:2; D:3 >

< A:1; B:2; C:4; D:3 > ≠ < A:1; B:2; C:3; D:4 >

(< A:1; B:2; C:4; D:4 >)

Committed updates on site S_i update u_i by one

Deletion/Renaming are updates, leave zero-length file (?)

Remove file if all versions zero

Resolution on site S_i increments u_i to maintain consistency later.

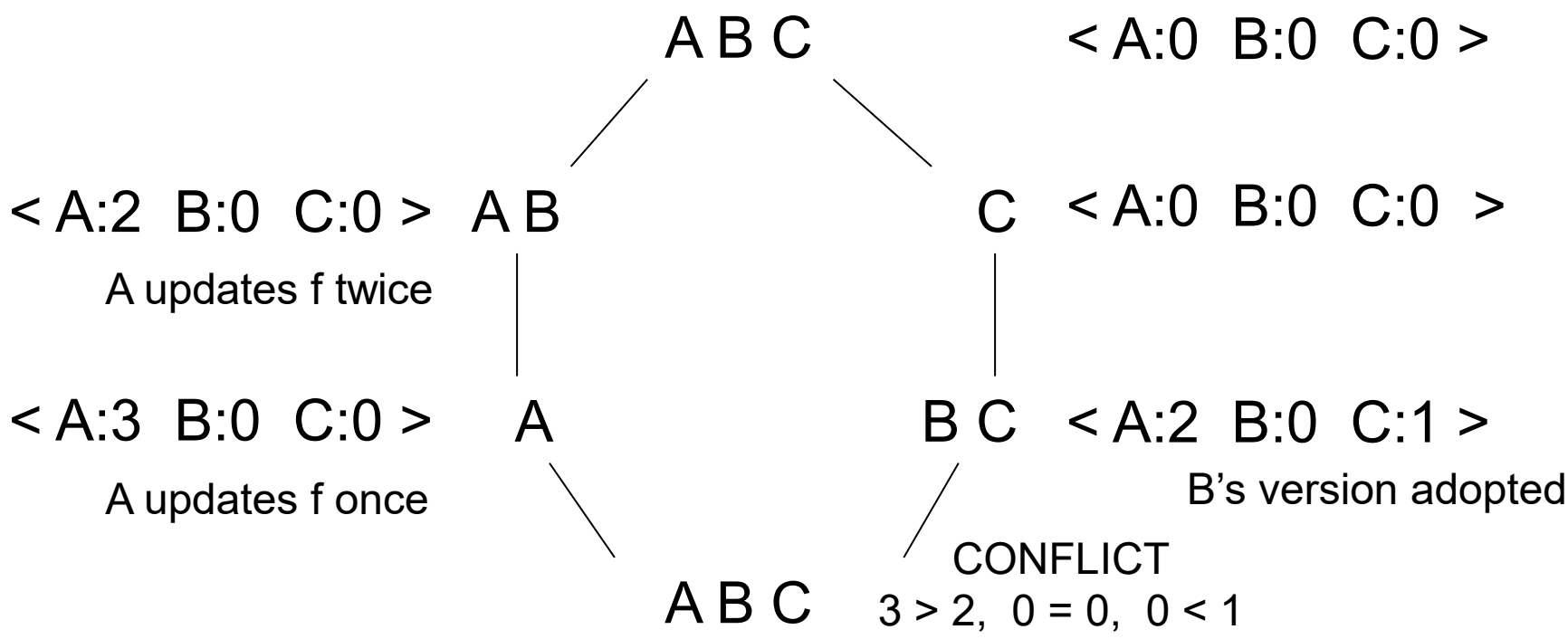
to Max S_i

Storing at new site makes vector longer by one site. Still compat.

(vectors may grow and shrink)

Inconsistency determined as early as possible.

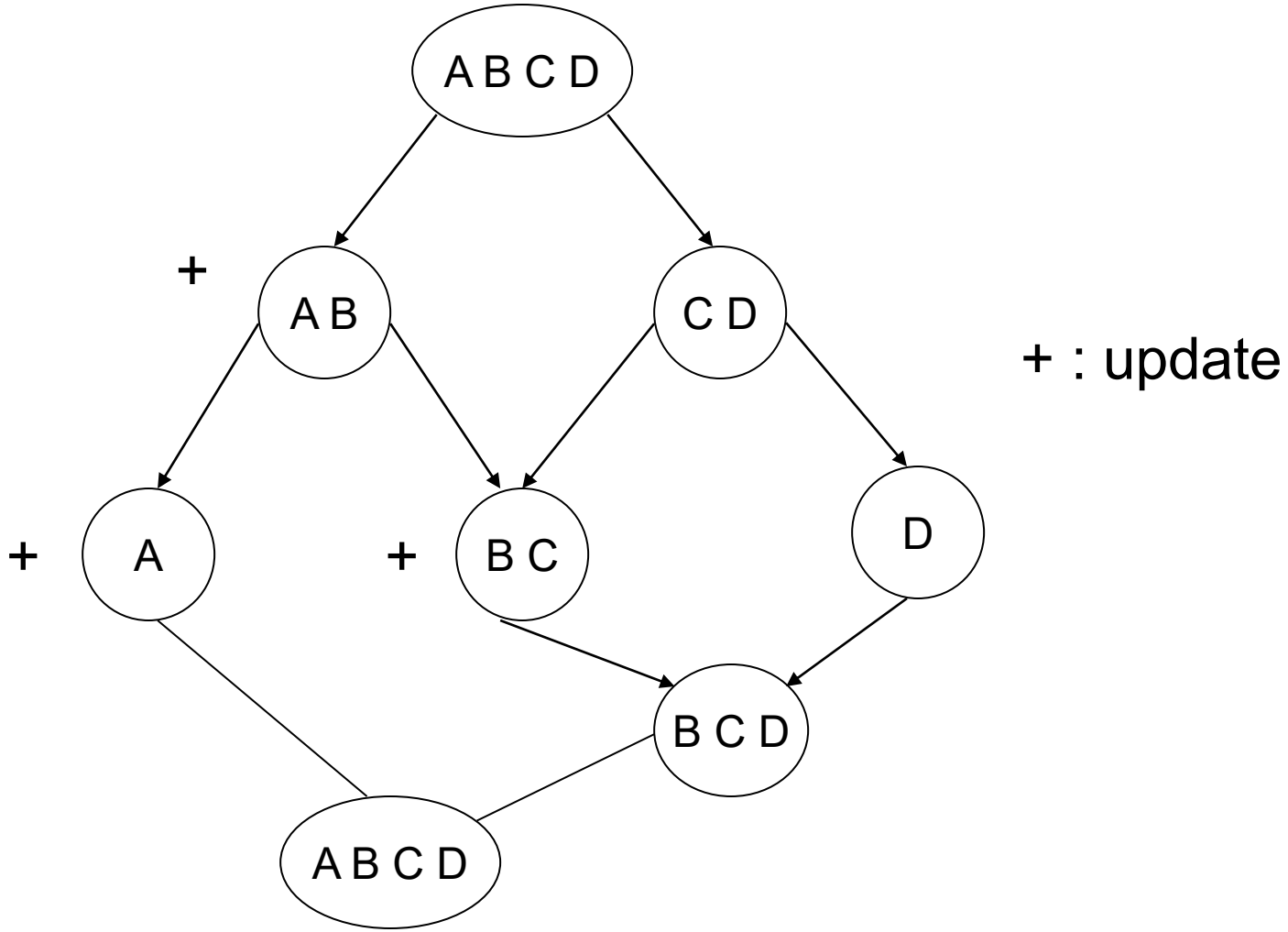
Only works for single file consistency, not transactions…

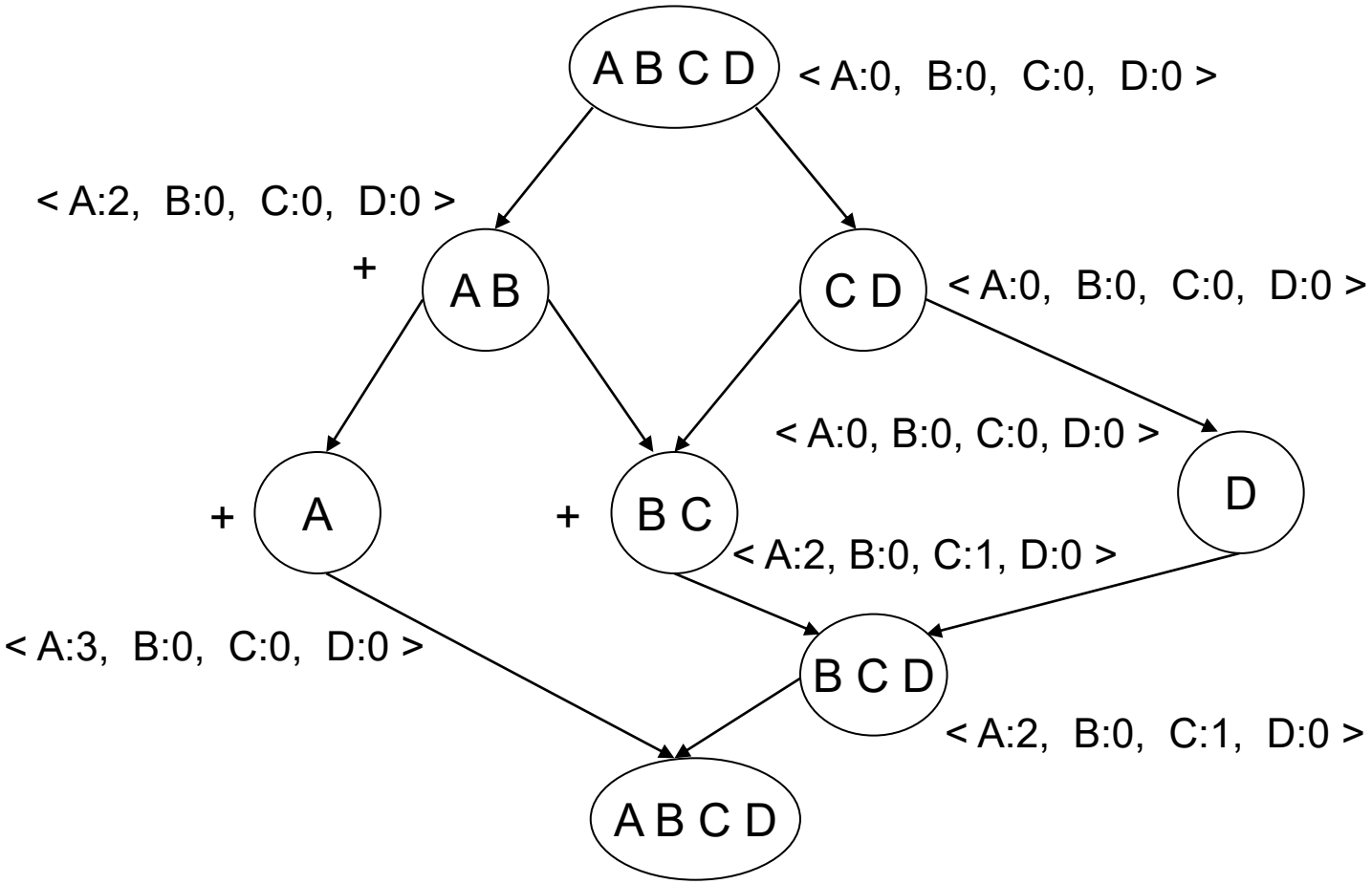


Version vector

$VV_i = (S_i ; v_i)$

v_i update to file f at site S_i


A B C D
+
A B
C D
+ : update
+
A
+
B C
D
B C D
A B C D



CONFLICT!

After reconcilation at site B

< A:3, B:1, C:1, D:0 >

- Resolution of Conflicts

  Semantics-Dependent

- Automatic resolution desirable, where possible:

  Mailbox, Directory (R/W)

  Only two types of object update (R/W)

  Simple union/intersection works

- Other Scenarios:
  - Credits and Debits

    $x + \delta_i(x)$ in each partition

    $x + \Sigma\, \delta_i(x)$ after merge

    May have to constrain $\delta_i(x)$ to prevent overdraft
  - Airline Reservations

- General resolution rules not possible.
- External (irrevocable) actions prevent reconciliation, rollback, etc.
- Resolution should be inexpensive.
- System must address:
  - detection of conflicts (when, how)
  - meaning of a conflict (accesses)
  - resolution of conflicts
    - automatic
    - user-assisted

Conclusions

- Effective detection procedure
  - providing access without mutual
  - exclusion (consent).
- Robust during partitions (no loss).
- Occasional inconsistency tolerated for the sake of availability.
- Reconciliation semantics…
- Recognize dependence upon semantics.