Flexible approaches to replicating shared data consistently

Marc Shapiro

Joint work with Nishith Krishna and Karthikeyan Bhargavan

Microsoft® Research

INRIA ROCQUENCOURT

Sharing information on a global scale

Enterprise collaboration, business information

- Large numbers of users
- Globally distributed
- Concurrent access and update
- Invariants between objects
- Conflicts are rare but do occur
- Variable network bandwidth, high latency

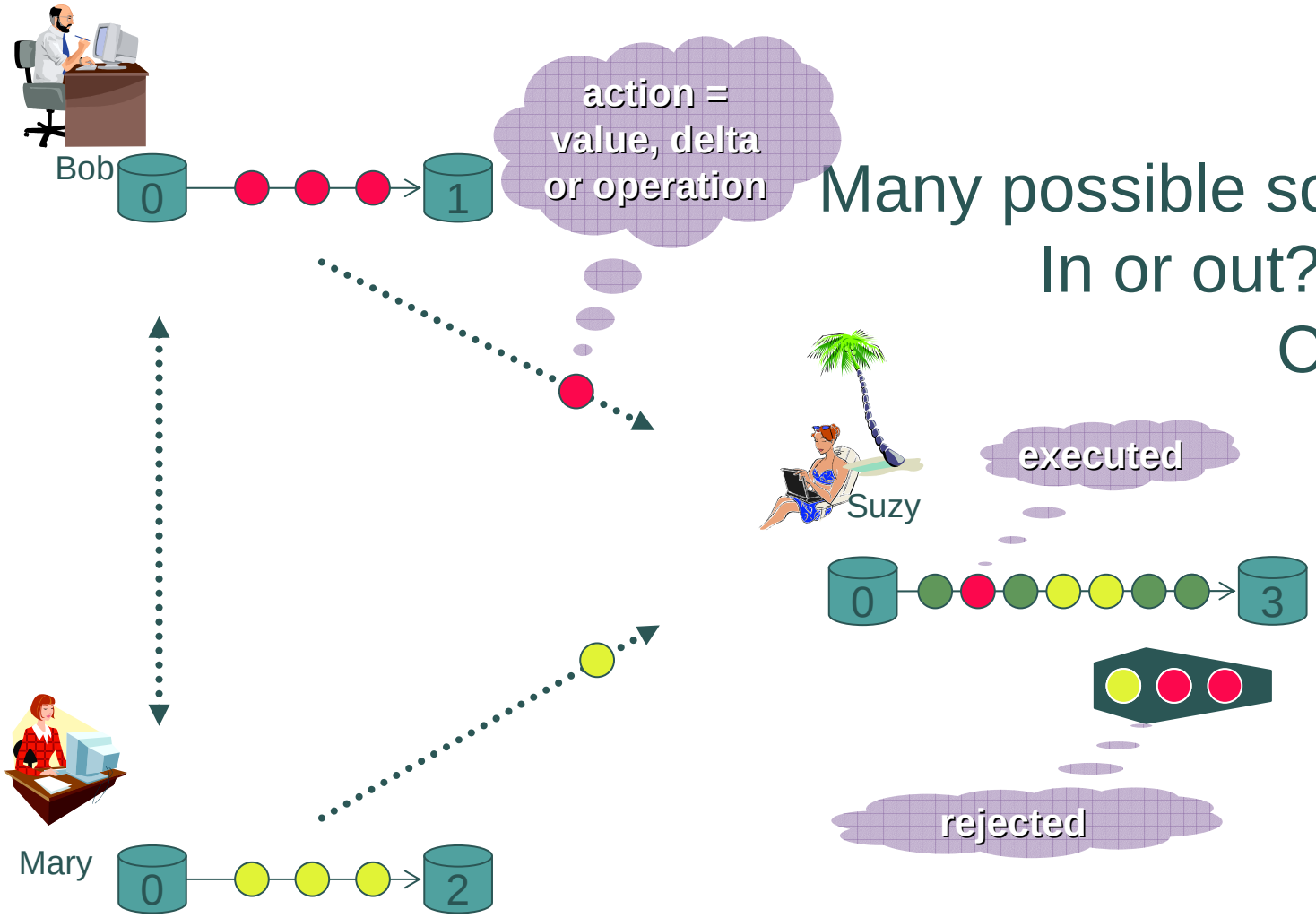
Replicate for fault tolerance, reduced latency, load balancing

Important lesson #1

Replication is beneficial in many information sharing scenarios:

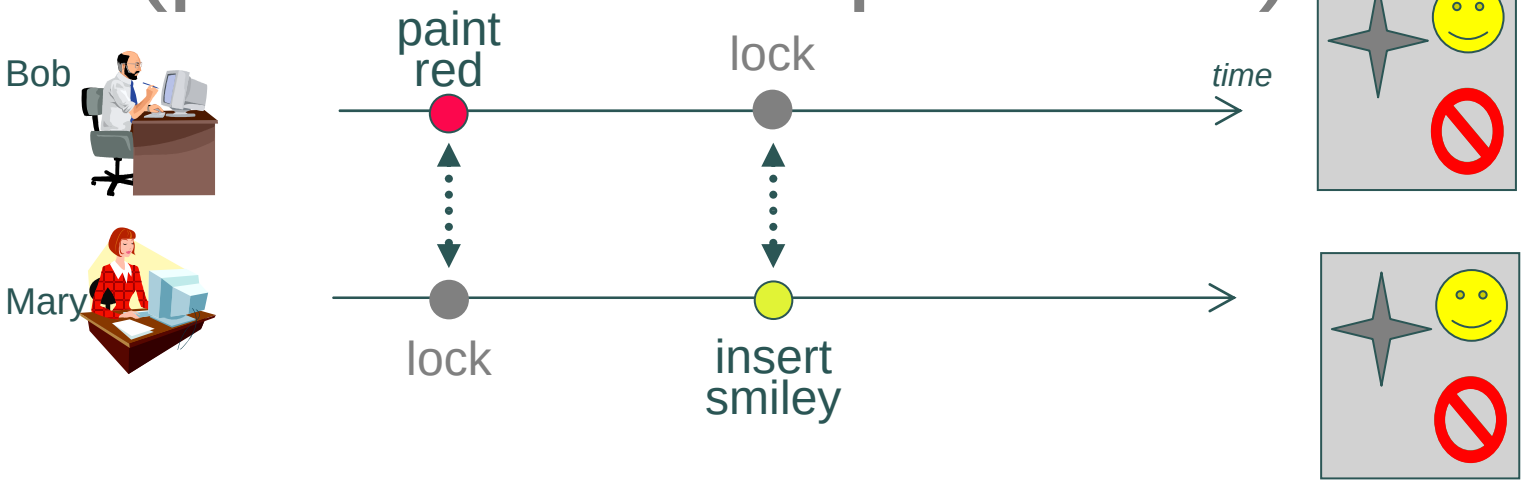
- Preserves autonomy
- Reduces access latency
- Improves fault-tolerance
- Supports disconnected operation

System model



Many possible schedules
In or out? Order?
Converge

Synchronous updates (pessimistic replication)



1SR = 1-Copy Serialisability

Avoid conflicts *a priori* by locking

Sequential access

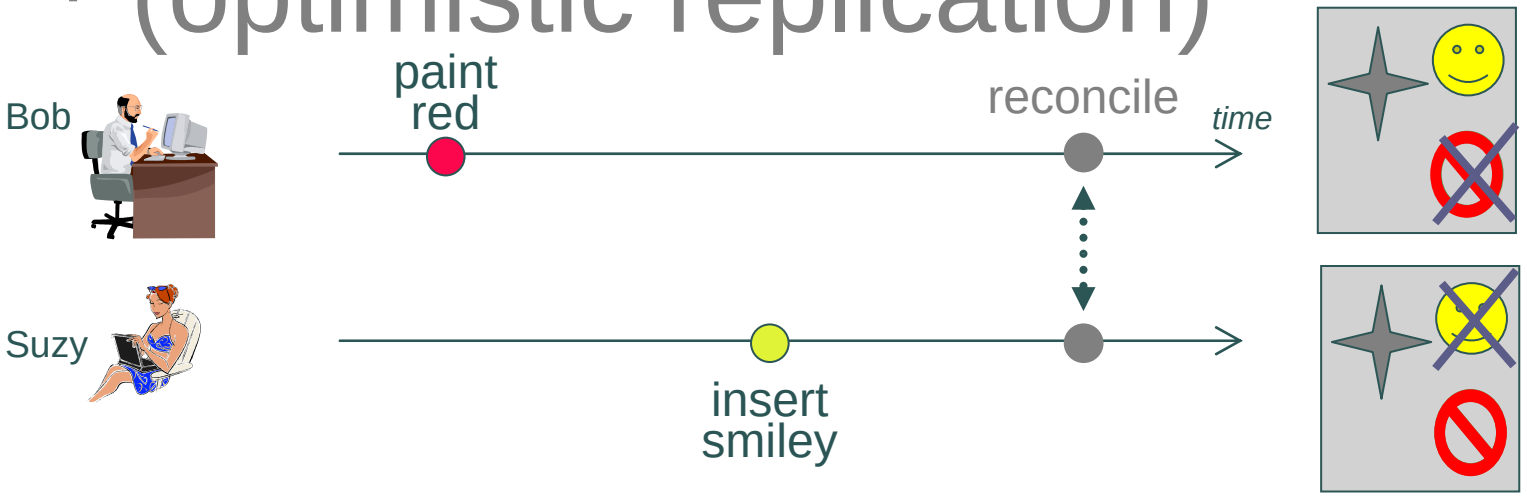
Intuitive, transparent

Vulnerable to:

- latency
- disconnection, faults
- deadlock

Doesn't scale if write contention

Asynchronous updates (optimistic replication)



Resolve conflicts *a posteriori*

- Disconnected, cooperative
- Powerful
- Batch & optimise

Tentative:

- Diverge, rollback
- Different user experience

Doesn't scale if write contention

Important lesson #2

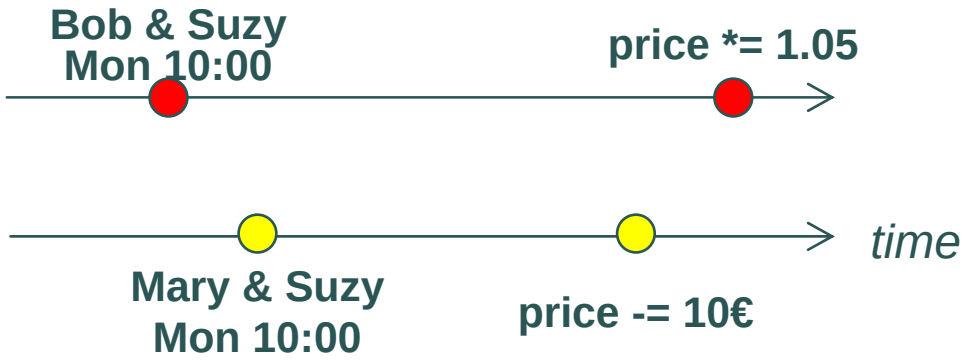
No replication scheme is ideal for all applications.

- Performance, complexity, consistency trade-offs
- No “one-size-fits-all” design
- Pessimistic / optimistic mode visible to users
- Contention / conflicts critical

Conflicts & non-commute

Bob

Suzy



Conflict: concurrent execution would violate application *invariant*

- e.g. calendar: no double booking

Non-commuting operations: decide order

- Commuting: optimisations

Scheduling

- Conflict: Is action in or out?
- Non-commuting: Ordering?

Important lesson #3

Understand your application needs and design with replication in mind.

- Capture invariants.
- Design for commutativity.
- Avoid concurrent non-commuting operations.
- Avoid conflicting operations.
- Otherwise have modest scalability expectations.

Exploring the consistency design space

Understanding replication & consistency

- Semantics
- Asynchronous / optimistic updates
- Partial replication
- Decentralised

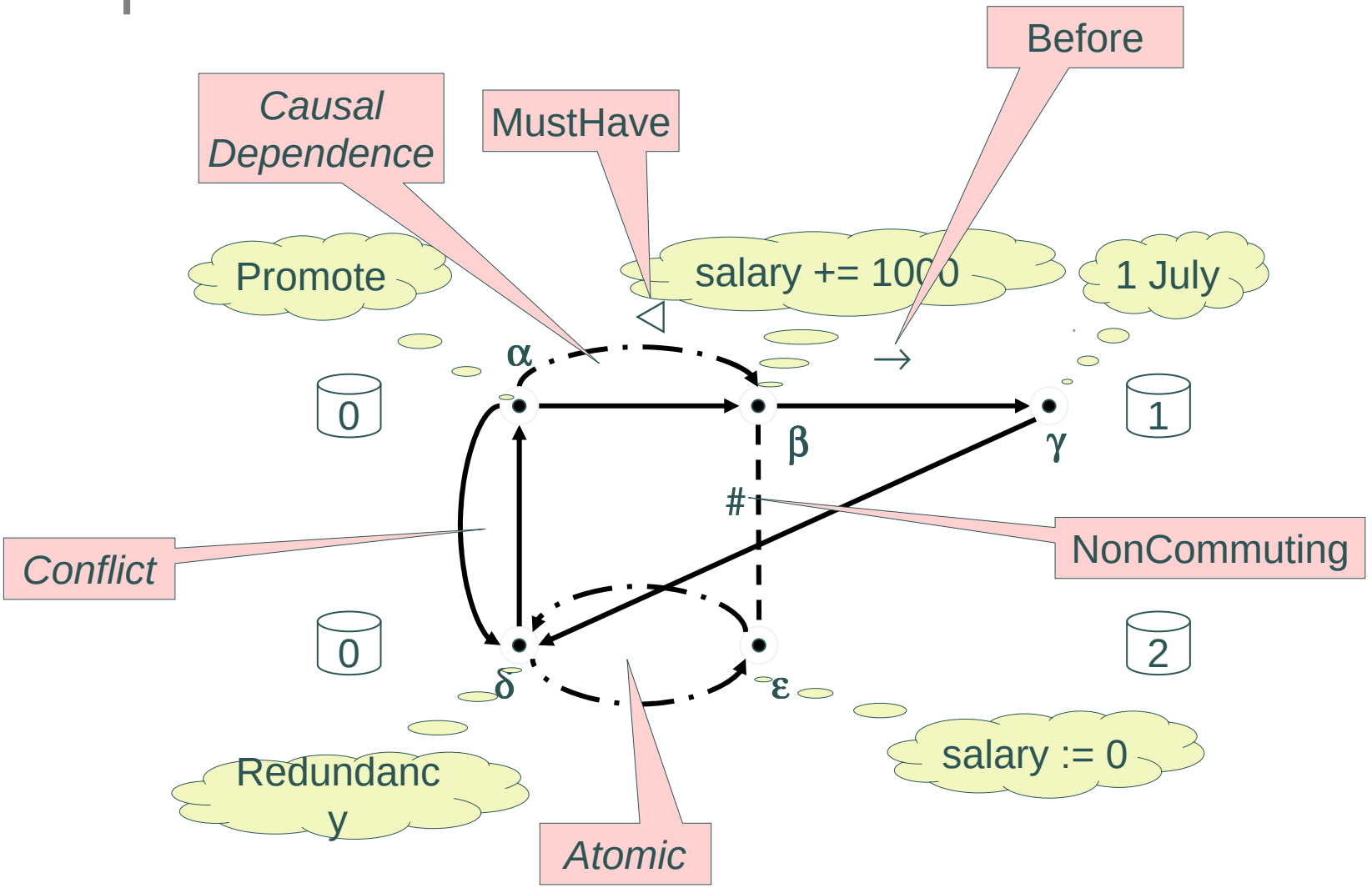
Constraint-graph representation

- Break into simpler sub-problems
- Composable sub-algorithms
- Spectrum of solutions

New serialisation algorithm

- No unnecessary aborts

Scenario


Before
Causal Dependence
MustHave
Promote
salary += 1000
1 July
α
β
γ
δ
ε
0
1
0
2
#
Conflict
NonCommuting
Redundanc y
Atomic
salary := 0

Multilog & schedule

$M = (K, \rightarrow, \lhd, \#)$ Local view per site:

- Known actions
- Known constraints
- Grows over time

Sound schedule: $S = \text{init}\ \alpha\ \gamma\ \ldots \in \Sigma(M)$:

- known actions, zero or once
- $\alpha \rightarrow \beta \wedge \alpha, \beta \in S \Rightarrow \alpha <_S \beta$
- $\alpha \lhd \beta \wedge \beta \in S \Rightarrow \alpha \in S$

M sound $\Leftrightarrow \Sigma(M) \neq \emptyset$

Protocol primitives

$$\text{Guar}(M) = \{\alpha \mid \alpha \in \text{every sound schedule}\}$$

$$\text{Dead}(M) = \{\alpha \mid \alpha \notin \text{every sound schedule}\}$$

$$\text{Serialised}(M) = \{\alpha \mid \alpha \,\#\, \beta \Rightarrow \alpha \rightarrow \beta \vee \beta \rightarrow \alpha \vee \beta \in \text{Dead}(M)\}$$

$$\text{Decided}(M) = \text{Dead}(M) \cup (\text{Guar}(M) \cap \text{Serialised}(M))$$

Monotonic in t

M sound $\Leftrightarrow \text{Guar}(M) \cap \text{Dead}(M) = \varnothing$

Consistency: a formal definition

Omniscient observer:
$(\cup \text{Dead}) \cap (\cup \text{Guar}) = \varnothing$

Mergeability: Any combination of multilogs remains sound:

$\forall\ i, i', i'', \ldots, t, t', t'' \ldots$

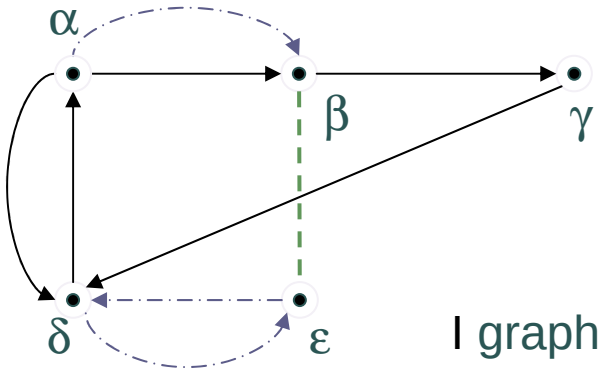
$M_i(t) \cup M_{i'}(t') \cup M_{i''}(t'') \ldots$ sound

Eventual Decision: Every action eventually decided everywhere:

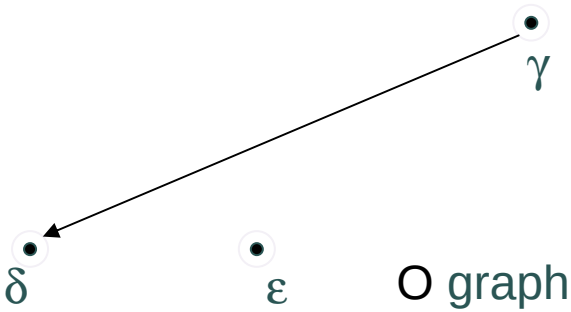
$\forall \alpha, i, j, t$:

$\alpha \in K_i(t) \Rightarrow \exists t',\ \alpha \in \text{Decided}\ (M_j(t'))$

Abstract consistency algorithm



I graph



O graph

Input: any application semantics
(K, →, ◁, #)

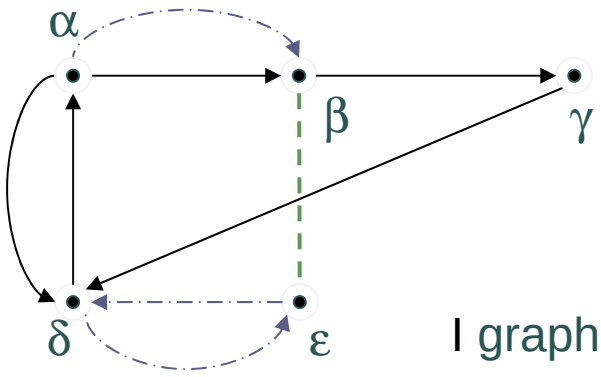
Decompose into very simple sub-problems

Graphs:

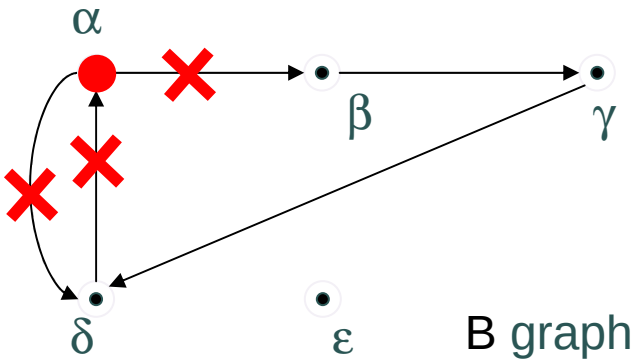
- I = input
- B = Before
- M = MustHave
- S = Serialisation
- O = output

Output: scheduling partial order

Conflict breaking



I graph



B graph

Make dead at least one action per → cycle

B: Before edges from I

- Redden a node
- Delete red node and its edges
- Terminate when acyclic

Concurrent, asynchronous

Numerous variants

Conflict-breaking spectrum

B-Null: B assumed acyclic; do nothing [file systems, Usenet, ESDS]

B-TotalOrder, B-LocalMin: UIDs [DB]

B-Conservative: Redden every node $\in$ cycle [Holliday]

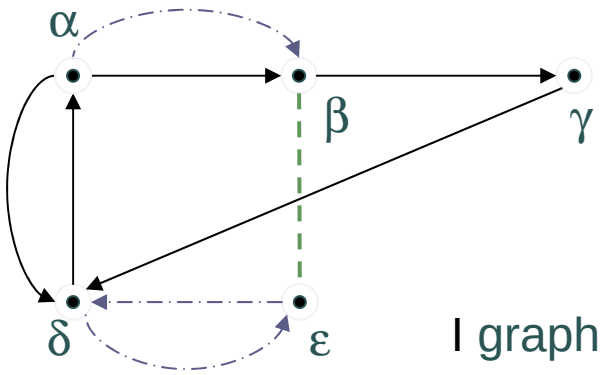
B-HighDegree: Redden highest-degree node [Hamadi]

Sub-algorithms: Not optimal

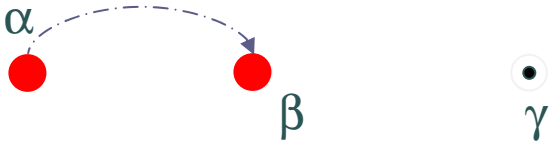
B-IceCube: *Globally* minimise red nodes

B-Arbitrary: application/user

Agreement



I graph



M graph

If $\alpha \in$ Dead $\wedge$ $\alpha \triangleleft \beta$ *then* $\beta \in$ Dead

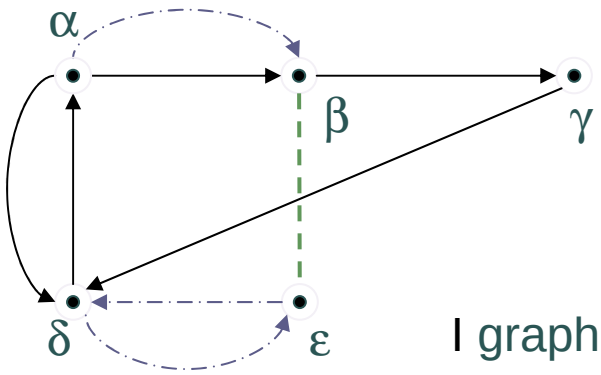
M: MustHave edges from I

Colour shared across graphs

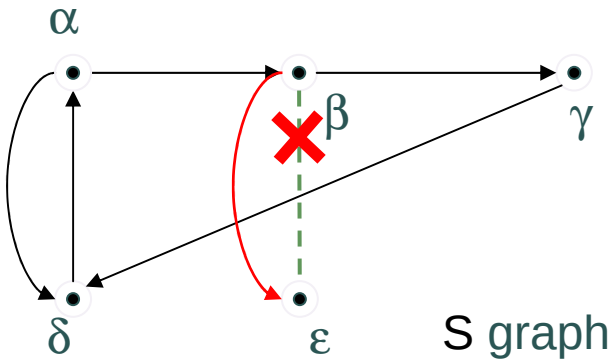
- Propagate colour along edges

Concurrent, asynchronous

Serialisation



I graph



S graph

Serialise non-dead # edges

S: →, # edges from I

- Delete red node & edges
- Insert → along # in S, B, O
- Delete # when →
- Terminate when no # edges

Concurrent

May create new cycles in B

Many variants

Serialisation spectrum

S-Null: Assume no unordered #; do nothing [Usenet, C-ESDS]

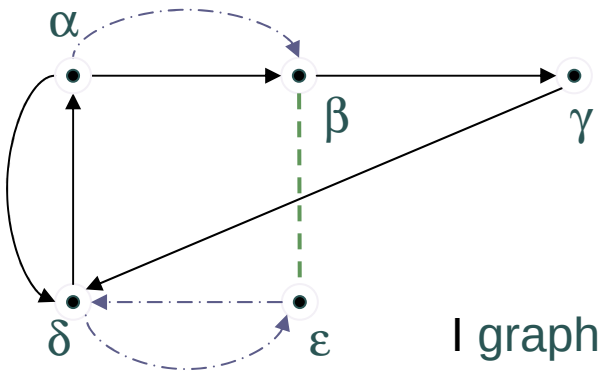
S-Random: baseline

S-Conservative: convert to conflict [DB]

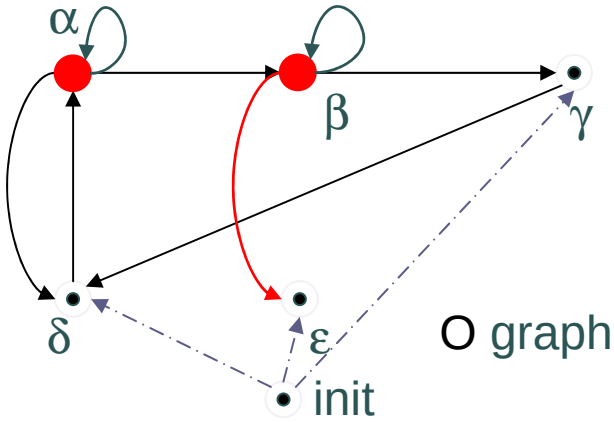
S-TotalOrder: UIDs [NC-ESDS]

S-HappensBefore: follow Happens-Before [state-machine replication]

Output



I graph



O graph

O: edges, → from I

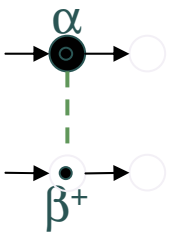
Colours from Conflict Breaking, Agreement

→ edges from Serialisation

- When 3 sub-algorithms have all terminated
- Make red nodes dead, others guaranteed

Scheduling partial order

Cycle-avoiding serialisation algorithm



Idea: given some node α

- Consider all 2^4 possible neighbourhoods
- Serialise in direction that cannot create a cycle, if exists
- Otherwise deterministic global order

Cycle-free serialisation algorithm

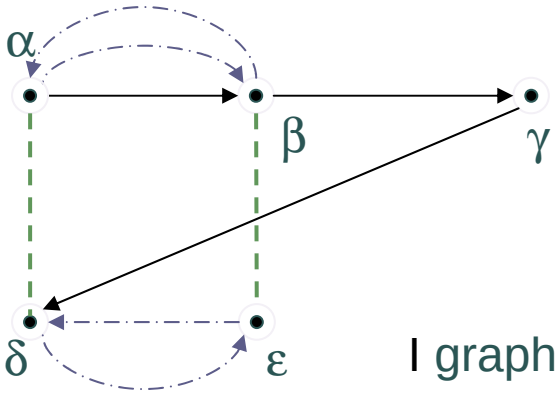
Start when B acyclic. In S:

- Choose two nodes α, β
- Lock α, β
- Atomically perform the cycle-avoiding serialisation move:
  - Insert $\rightarrow$ in S, O
  - Delete #
- Unlock

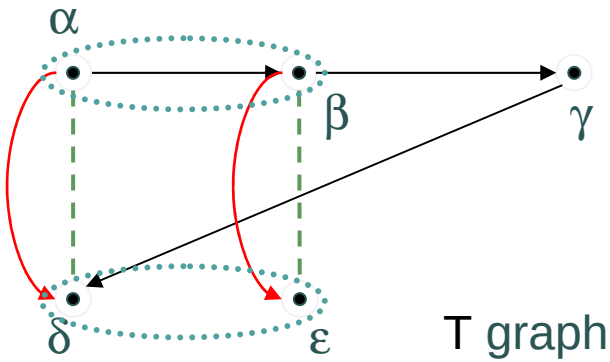
Never causes aborts

Pairwise agreement

Isolation



I graph



T graph

Transaction isolation

T: initially same as S + transactions

- If $\exists \rightarrow$ with # between T1, T2
- Then $\forall \rightarrow$ with # between T1, T2
- Terminate when done

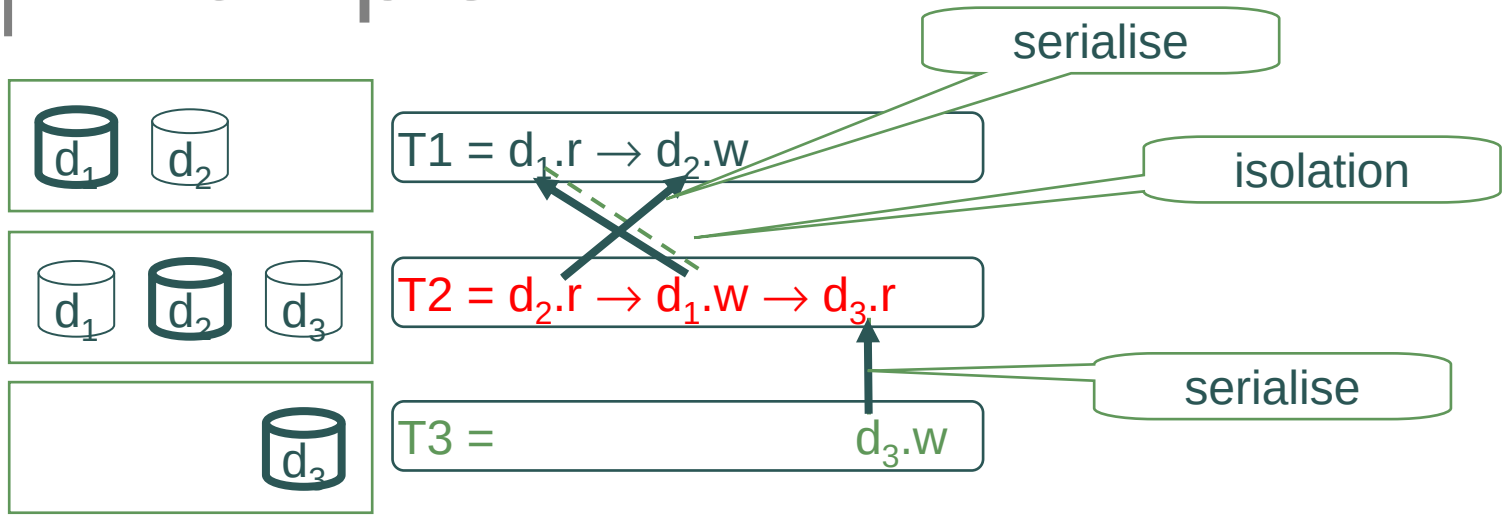
Concurrent, asynchronous

May create new cycles in B

- C-B does not terminate before isolation

Many variants

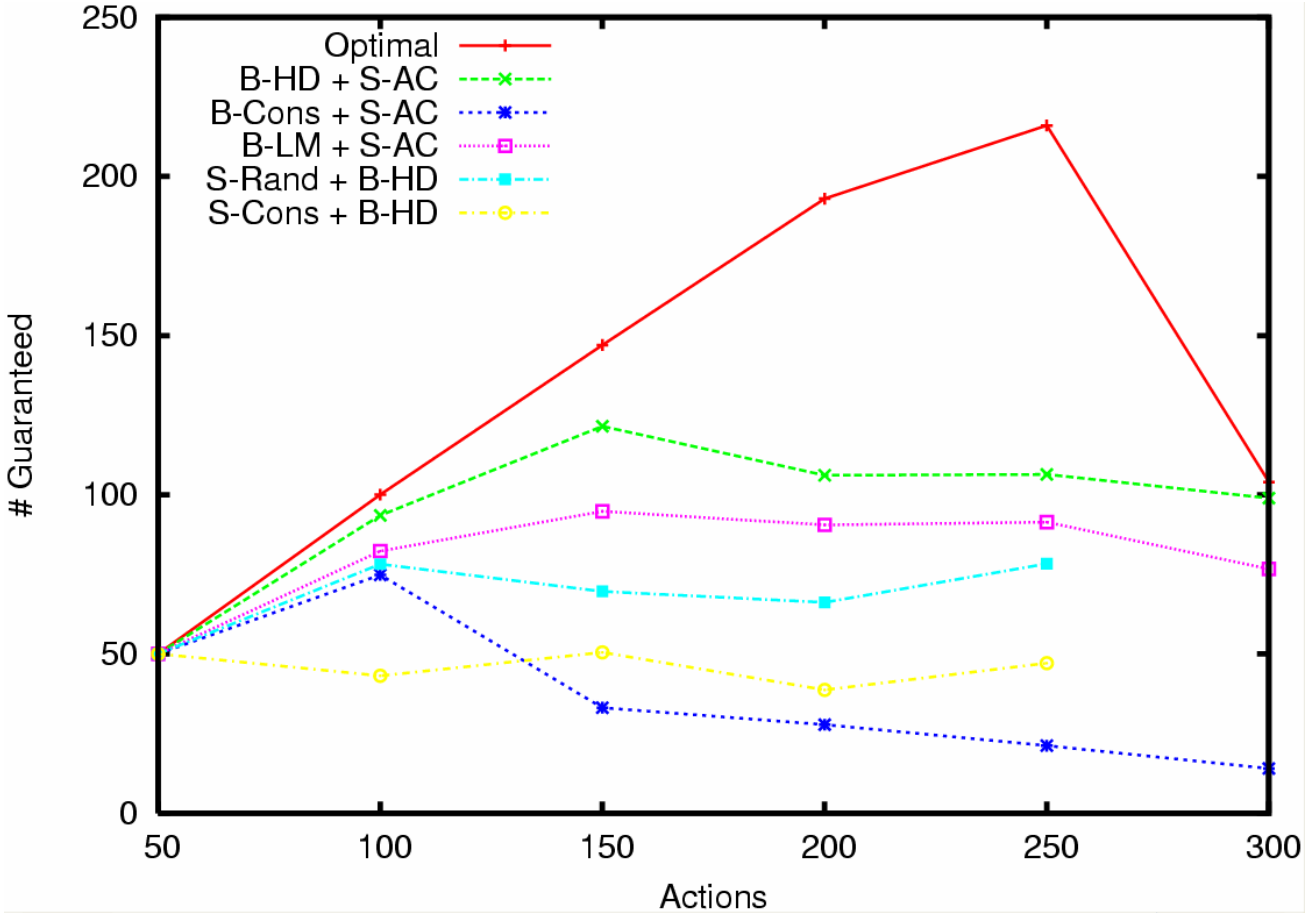
Example



schedule = $d_2.r$ $d_1.w$ $d_3.w$ $d_3.r$ $d_1.r$ $d_2.w$

No two-phase commit

Simulations: Pseudo-realistic



B-HD = high-degree
B-Cons = conservative
B-LM = local minimum

S-AC = avoid-cycles
S-Rand = random
S-Cons = conservative

Joyce

Document = multilog

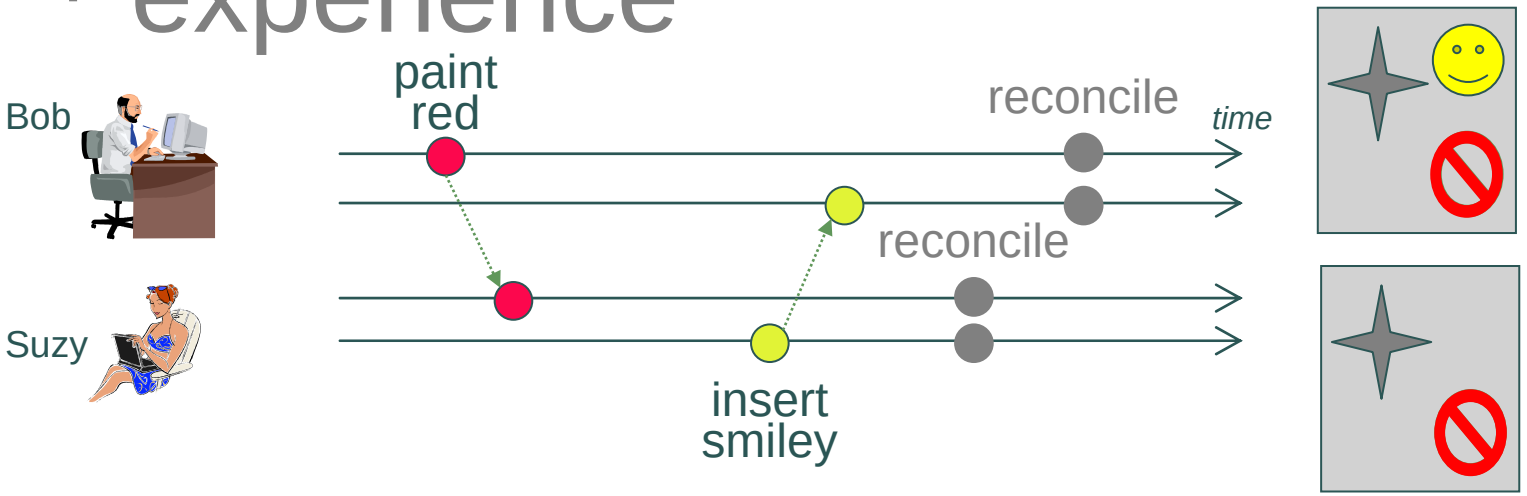
- 1 operation log / user
- Operations
- Constraints = logical invariants

Local views

- Write to my log: updates are local
- View: collect multilog, break conflicts, replay
- Consistent: resolution & replay satisfies constraints

Convergence: authoritative log

Joyce collaboration experience



Local views

- Reconcile
- Unlimited, selective undo

Convergence

- Commit log

Conclusion

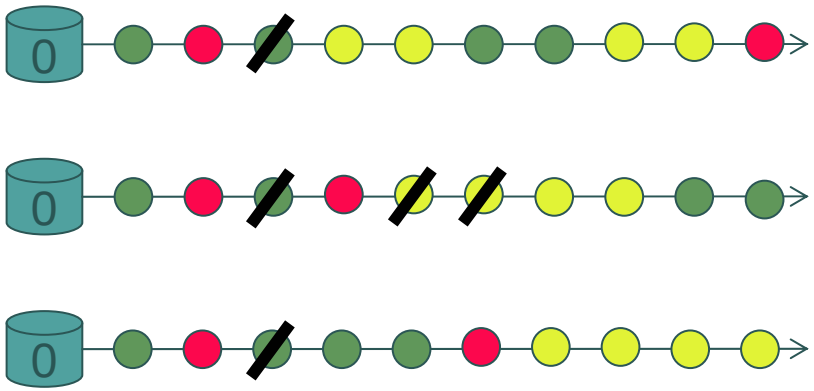
Actions & constraints: simple, formal model
- Encode application semantics
- Express consistency

Basic components of consistency
- Decide
- Mergeability

Universal consistency protocol
- Sub-algorithms
- Mix & Match
- Cycle-avoiding serialisation

Partial replication

Site schedule



$S_i(t) \in \Sigma(M_i(t))$

- Choose any sound schedule
- $S_i(t+1)$ / $S_i(t)$ / $S_{i'}(t)$ may differ greatly

More actions $\Rightarrow$ more non-determinism

More constraints $\Rightarrow$ less non-determinism

Enough to ensure consistency

Example

more actions
⇒
more schedules

more constraints
⇒
fewer schedules

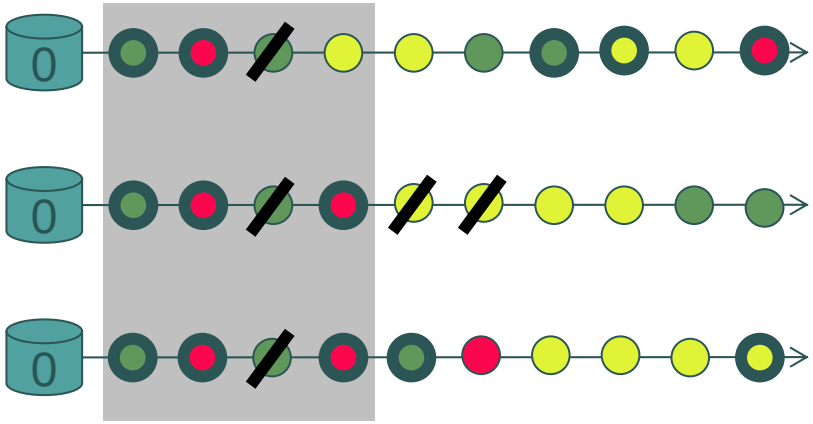
t	0
$K_i(t)$	$\emptyset$
$\rightarrow_{i,(t)}$	$\emptyset$
$\lhd_{i,(t)}$	$\emptyset$
$\Sigma(M_i(t))$	INIT
$K_j(t)$	$\emptyset$
$\rightarrow_{j,(t)}$	$\emptyset$
$\lhd_{j,(t)}$	$\emptyset$
$\Sigma(M_j(t))$	INIT

Eventual Consistency

From the literature: EC =

- If all clients stop submitting new updates,
- Then eventually all replicas converge to the same value
- (Eventually decide)

Common monotonic prefix property



There exists prefix $\pi(i,t)$:

- Monotonic: $t < t' \Rightarrow \pi(i, t) << \pi(i, t')$
- Equivalence: $\pi(i, t) \equiv \pi(i', t)$
- Eventually inclusive: $\alpha \in K_i(t) \Rightarrow \alpha \in \pi(i, t')$

CMP: goals to achieve

Composing sub-algorithms

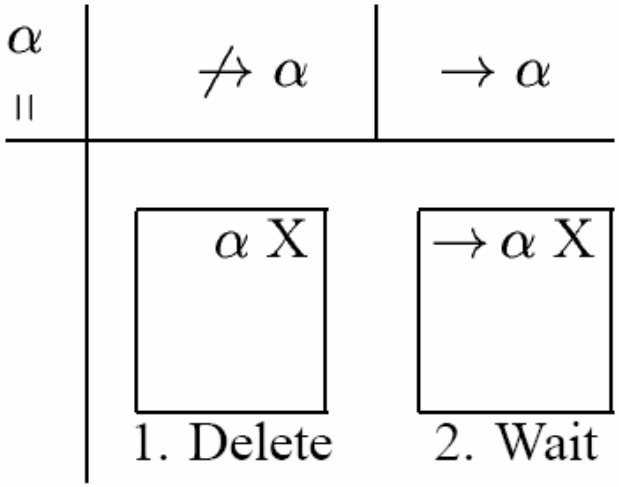
Parallel composition:

- Any conflict-breaking algorithm
- Any serialisation algorithm

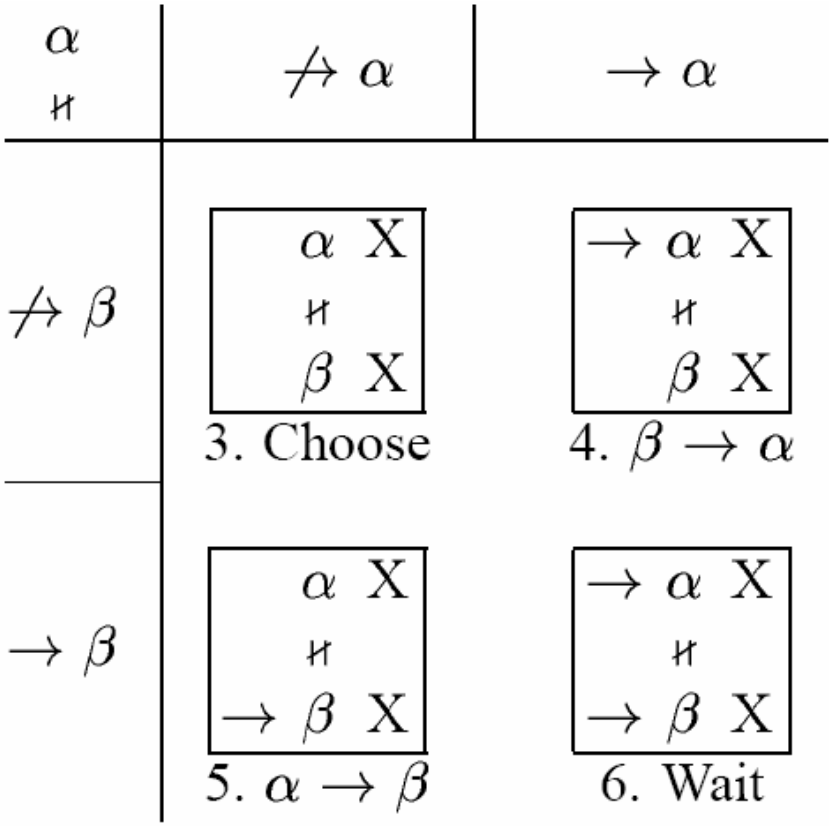
Subtle termination conditions:

- Parallel composition. Terminate: (1) Serialisation, (2) Conflict-breaking, (3) Agreement
- Fast agreement minimises red nodes
- Sequential composition: conflict breaking + agreement $\Rightarrow$ S acyclic. Then S-NoCycles + synchronisation

S-AvoidCycles

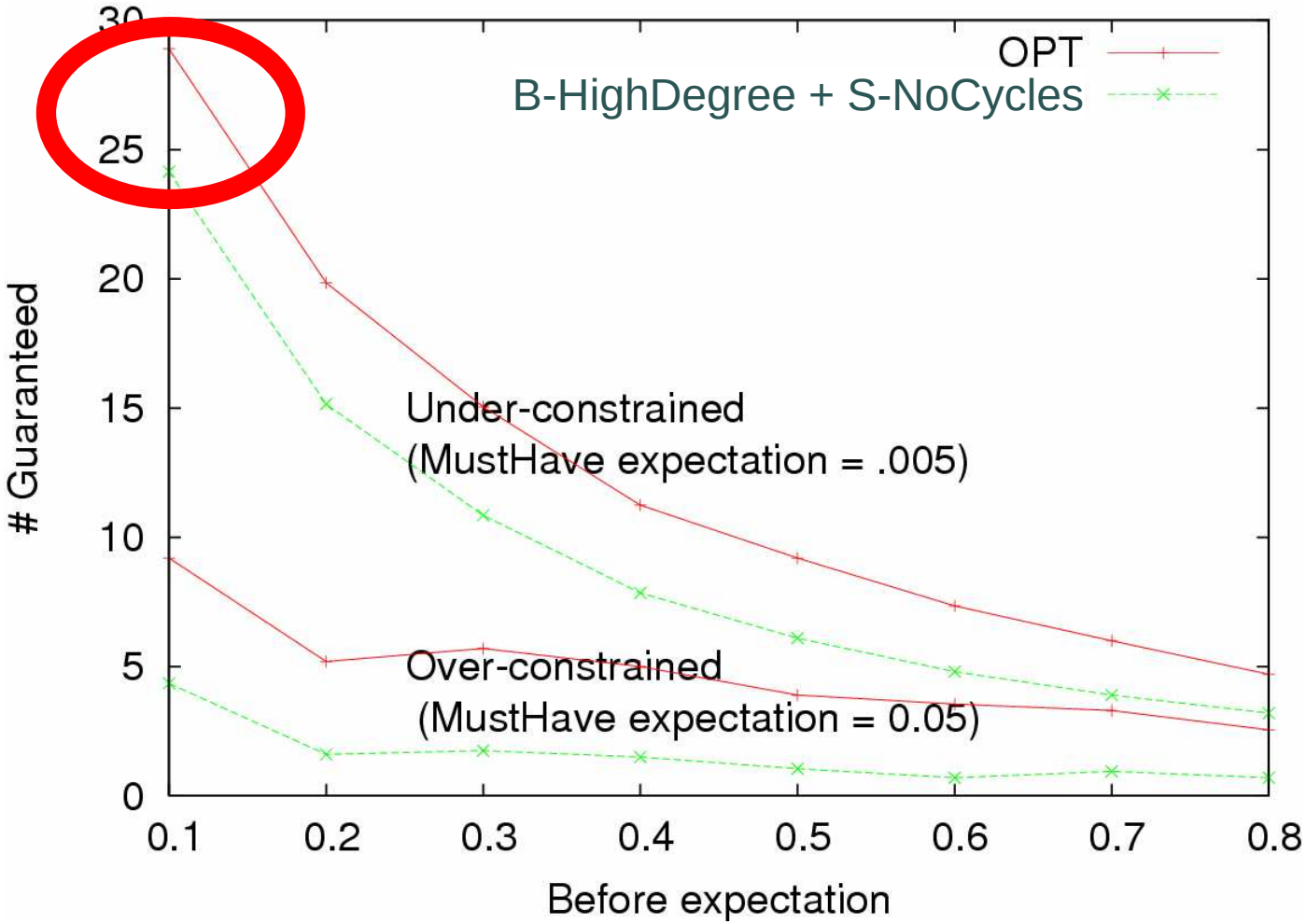


(a) No undirected edge

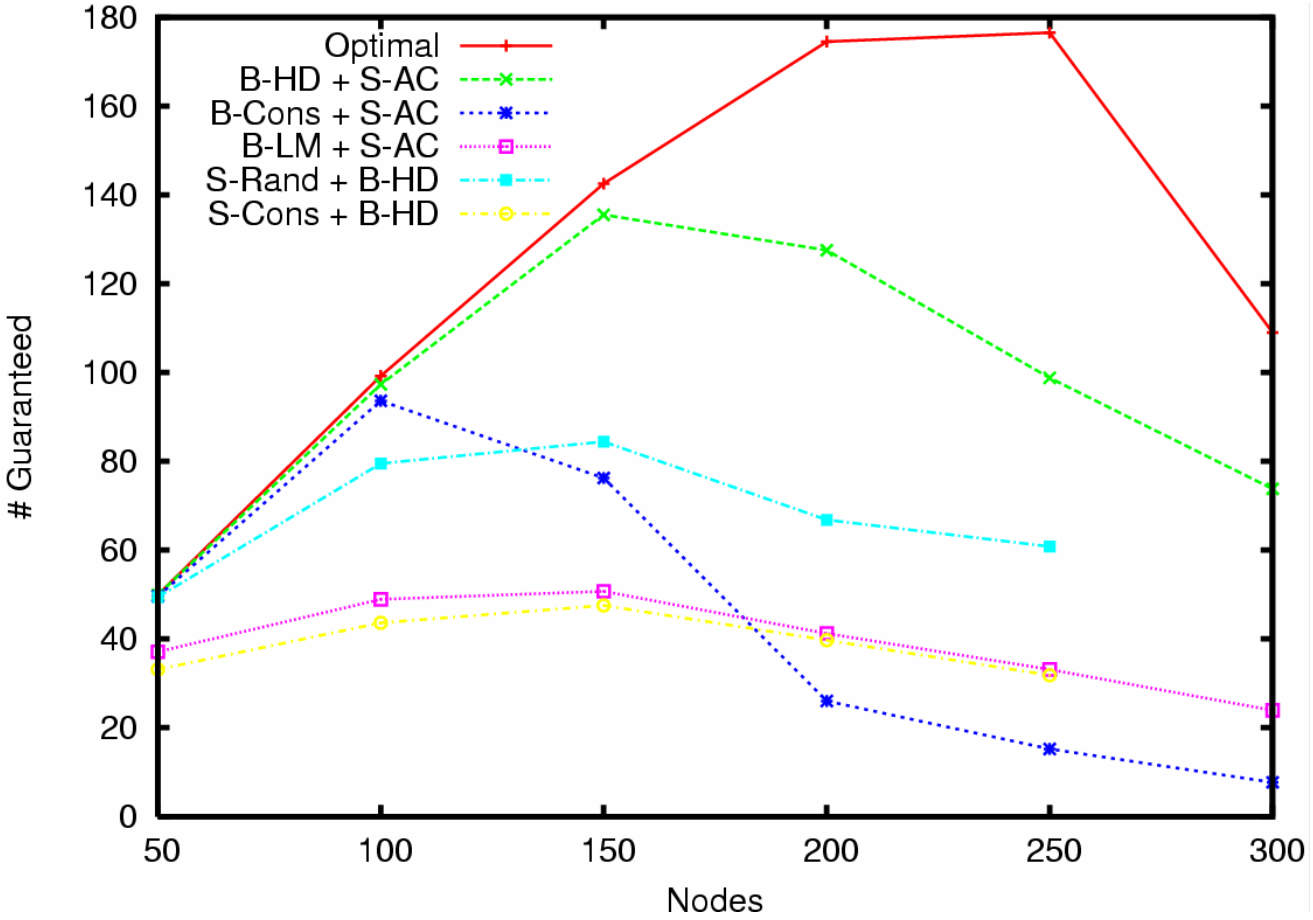


(b) One or more undirected edges

Simulations: range



Simulations: Random



B-HD = high-degree
B-Cons = conservative
B-LM = local minimum

S-AC = avoid-cycles
S-Rand = random
S-Cons = conservative

Incremental algorithm

Cannot decide α until all its constraints known

Iteratively dectect quiescent subgraph: timestamp matrix

Output from interation *n* = input to iteration *n*+1

- Verify inclusion property

Partial replication

A site replicates any number of disjoint databases

Receives actions, constraints relative to its replicas only

Consistency:

- Mergeability
- Eventual decision w.r.t. database

No need for global consensus

Omniscient observer = full replication site

Partial replication + Cycle-free serialisation

Partitioned database + partial replication

Operations commute across partition

A small number (often 1) of *primary* nodes decide partition

In-partition NonCommute: primary decides

Cross-partition: pairwise agreement

Total order unnecessary (≠ state-machine replication)