6.852: Distributed Algorithms
Fall, 2009

Class 20

Today's plan

- Transactional Memory
- Reading:
  - Herlihy-Shavit, Chapter 18
  - Guerraoui, Kapalka, Chapters 1-4
- Next:
  - Asynchronous networks vs asynchronous shared memory
  - Agreement in asynchronous networks
  - Reading:
    - Lynch, Chapter 17
    - Lamport paper: The Part-Time Parliament (the Paxos algorithm)

Techniques for Shared-Memory Concurrent Programming

- Coarse-grained locking
  - Simple, works well for low contention.
  - Liveness: Guarantees progress, lockout-freedom (if lock does).
- Fine-grained locking
  - Allows more concurrency, but introduces deadlock possibilities.
  - Greater time and space overhead (due to more locks).
  - Two-phase policy guarantees atomicity, but doesn't help for list.
  - Hand-over-hand locking: Short-duration locks; pipelines operations, so slow operations delay fast ones.
  - Liveness: Guarantees progress, lockout-freedom (if locks do).

Techniques for Shared-Memory Concurrent Programming

- Optimistic locking
  - Search list without locking; lock just the needed nodes.
  - Validate: Verify that the nodes are still adjacent and in list (list might have changed).
    - Requires traversing the list again.
    - Retry if validation fails.
  - Good if validation typically succeeds.
  - Traversal is wait-free, but
    - Traverses the list twice.
    - Even contains() (most common operation) locks nodes.
  - Liveness: Guarantees progress, but not lockout-freedom.

Techniques for Shared-Memory Concurrent Programming

- Lock-free techniques
  - No locks, but uses CAS.
  - Separate "logical" and "physical" node removal.
  - Operations "help" other operations, to make the algorithm nonblocking.
  - Liveness:
    - Nonblocking: lock-free (even if some processes stop, if some keep taking steps, then some operation finishes), but not wait-free.
    - Guarantees progress, but not lockout-freedom.
    - Can make contains() wait-free.

- Lazy techniques
  - No CAS, but uses short-duration locks.
  - Separate "logical" and "physical" node removal.
  - No helping---each operation does its own physical removal.
  - Local validation.
  - contains() doesn't need to lock/validate.
  - Liveness:
    - contains() is wait-free.
    - add(), remove() are blocking; satisfy progress, but not lockout-freedom.

All these methods

- Are difficult to use!
- Fine-grained concurrent programming is very hard.
- Leads to recent attempts to simplify the task, such as Transactional Memory (TM).

Transactional Memory

- Raise level of abstraction, leads to simpler programs.
- Programmer specifies units of atomicity: transactions.
  - Consist of any number of operations on individual memory objects.
- System guarantees atomicity of entire transaction.
  - Performs all the operations and commits the transaction, if possible.
  - Otherwise, aborts the transaction, rolling back any changes.
    - Calling program can retry on abort.
- System also manages contention (possibly separable functionality).
- Transactions may be nested.

TM history

- Herlihy and Moss (1993) proposed hardware TM
  - Hardware; exploits cache coherence protocol
  - Platform-dependent limits
- Shavit and Touitou (1995) proposed software TM
  - Lock-free, not adaptive, very expensive (not practical)
- Revisited by many in early 2000s
  - DSTM (PODC 2003), OSTM (OOPSLA 2003), then lots more.
  - SLE (2001), TCC (2004), then lots more.
  - Very active area (e.g., several new workshops).

Transactional memory

- Object-based vs word-based
- Hardware vs software (or combination)
- Blocking vs nonblocking
  - “user” blocking vs “system” blocking
  - obstruction-freedom vs lock-freedom
- Contention management
- Encounter-time vs commit-time acquire
- Eager vs lazy conflict detection (“zombie” transactions)
- Undo log vs write set
- Visible vs invisible vs “semivisible” readers
- Feature interaction: i/o, exceptions, conditional waiting, privatization, strong vs weak atomicity

Using Transactional Memory

- Use atomic { code } to delineate transaction.

```
Q.enqueue(x)
  node := new Node(x)
  node.next := null
  atomic{
    oldtail := Q.tail
    Q.tail := node
    if oldtail = null then
      Q.head := node
    else
      oldtail.next := node
  }
```

```
Q.dequeue()
  atomic{
    if Q.head = null then
      return null
    else
      node := Q.head
      Q.head := node.next
      if node.next = null then
        Q.tail := null
      return node.item
  }
```

Transactional Memory Interface

- beginTxn(); beginOk
- inv(op); resp(v)
  - Typically just read or write
- commitTxn(); commitOk
- aborted

TM's Jobs

- Handle request to begin transaction

- Handle requests by transaction to perform operations on objects.
  - Typically just reads and writes.
  - Different handling for reads vs. writes.
  - Manage bookkeeping: Versions, timestamps,…
  - Detect and manage requests from conflicting transactions.
  - Validate while handling operations.

- Handle commit/abort:
  - Decide when it's necessary to abort a transaction.
  - Respond to transaction's req-commit by either committing or aborting.
  - Validate before committing.
  - Maintain roll-back functionality to support abort.

Implementing TM

- Begins transactions.
- Maintains objects, plus bookkeeping data for all objects and transactions.
- Handles requests by transaction to perform read and write operations on objects.
- Handle commit/abort.
- Assume no hardware support (use CAS).
- Three methods:
  - Two-phase locking
  - Obstruction-free STM, based on Dynamic Software Transactional Memory (DSTM) algorithm of [Herlihy, Luchangco, Moir, Scherer].
  - Lock-based STM, based on TL2 algorithm of [Dice, Shalev, Shavit].

Two-phase locking

- Consider just read and write operations.
- Associate shared/exclusive lock with each object.
- Object x has fields:
  - old-version, value of object x before any changes by the current lock-holder
  - new-version, value of object x including the current lock-holder's updates (could be kept locally by updating process)
  - lock (can be exclusive or shared)
- When transaction T tries to read x:
  - If x is locked by T in exclusive mode, then T reads new-version.
  - If x is locked by T in shared mode, then T reads old-version.
  - If x is locked by another transaction in exclusive mode, then T aborts, releases all its locks and discards all its new-versions.
  - If x is locked by another transaction in shared mode, or is unlocked, then T sets a shared lock on x and reads old-version.

Two-phase locking

- Object x has fields:
  - old-version
  - new-version
  - lock (exclusive or shared)
- When T tries to read x: …
- When T tries to write x:
  - If x is locked by T in exclusive mode, then T overwrites new-version.
  - If x is locked by T in shared mode and no one else shares the lock, then T upgrades to an exclusive lock and writes new-version.
  - If x is locked by another transaction in any mode, then T aborts, …
  - If x is unlocked, then T sets an exclusive lock on x and writes new version.
- When T wants to commit:
  - Replace all T's old-versions with new-versions and release locks.

Evaluation: Two-phase locking

- Atomicity
- Deadlock-free

- Can have livelock or starvation
- Blocking: slow process can delay everyone

Improving two-phase locking

- Reduce the number and duration of locks.
- Use optimistic strategy, with validation.
  - Check that objects read haven't changed.
- Reads don't lock at all.
  - Need to keep version numbers.
- Objects that are written are locked only during commit.

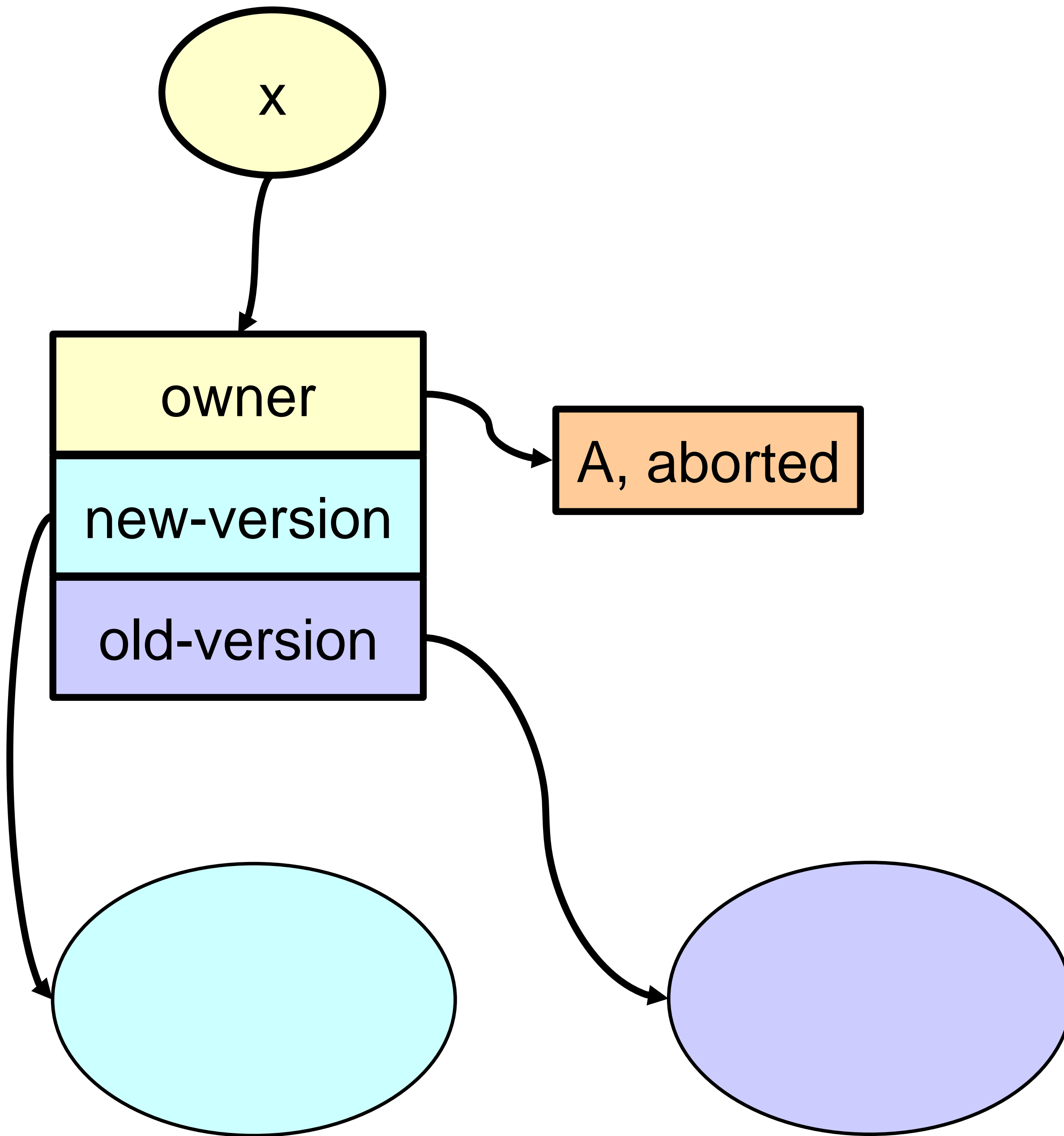
- Will still be blocking

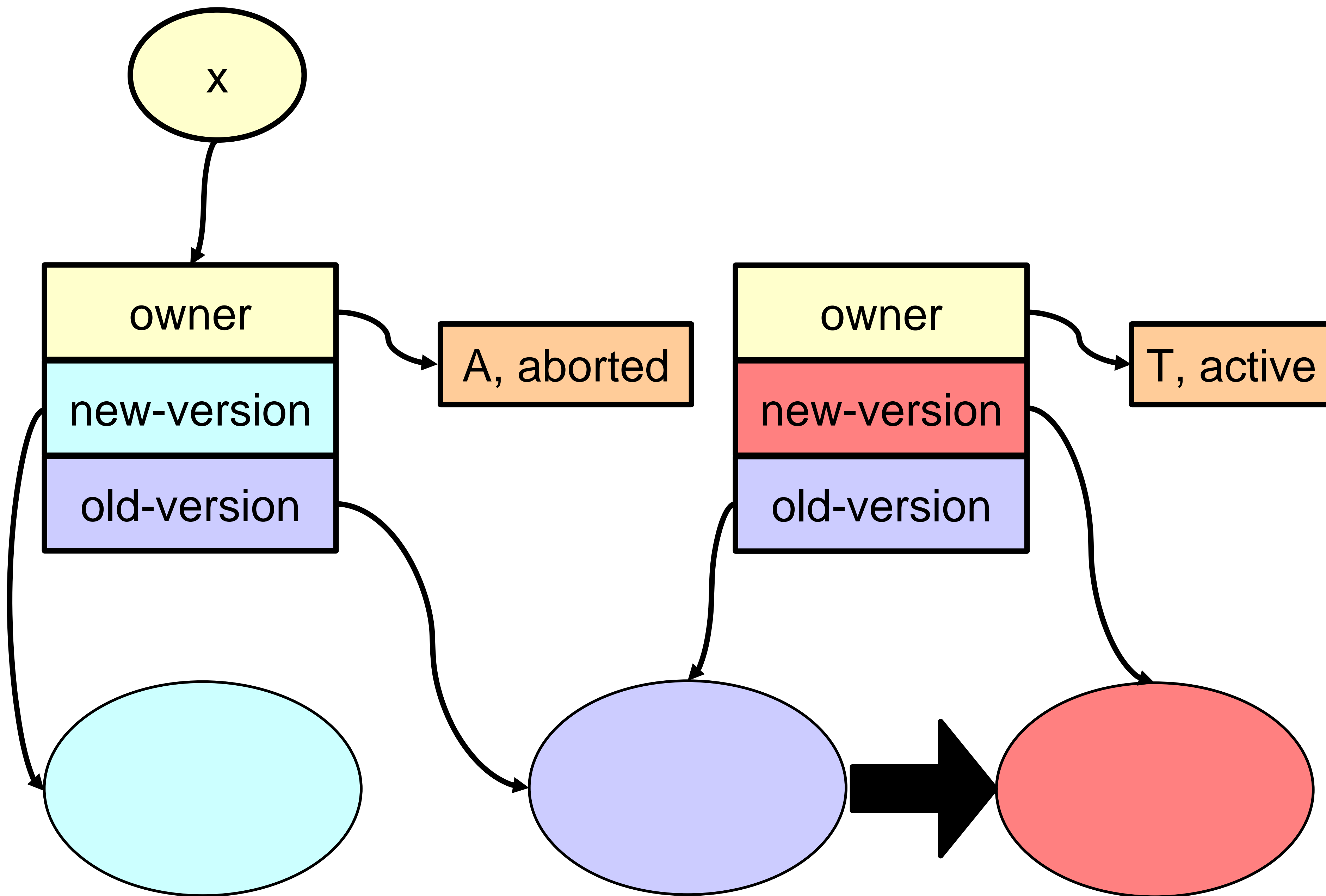
Obstruction-free algorithm

- From Dynamic Software Transactional Memory (DSTM) algorithm.
- No locks.
- Obstruction-free: from any point, if a transaction runs by itself, it eventually commits.
  - This is nonblocking: can’t use locks.
- Separable contention management.

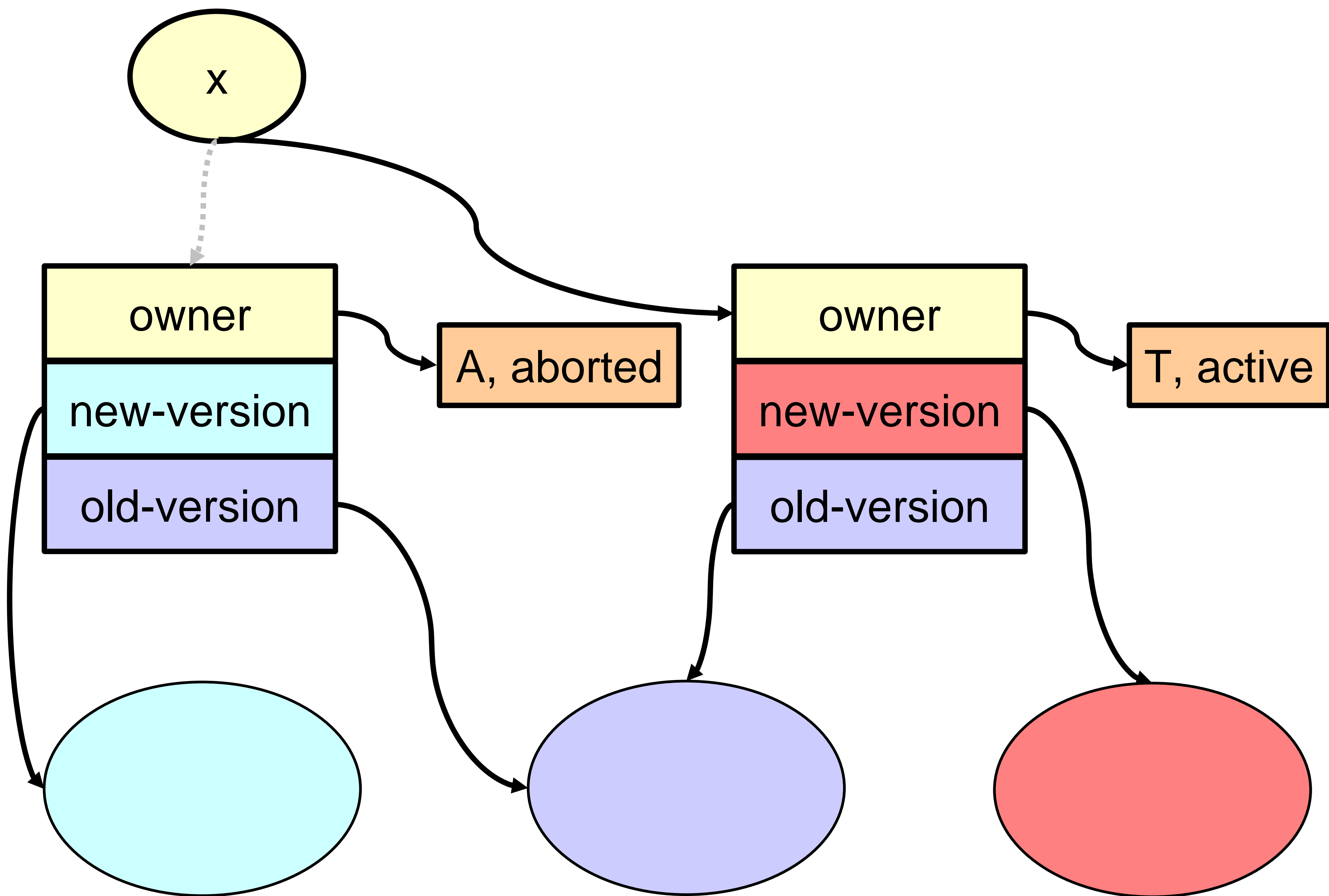
Obstruction-free algorithm

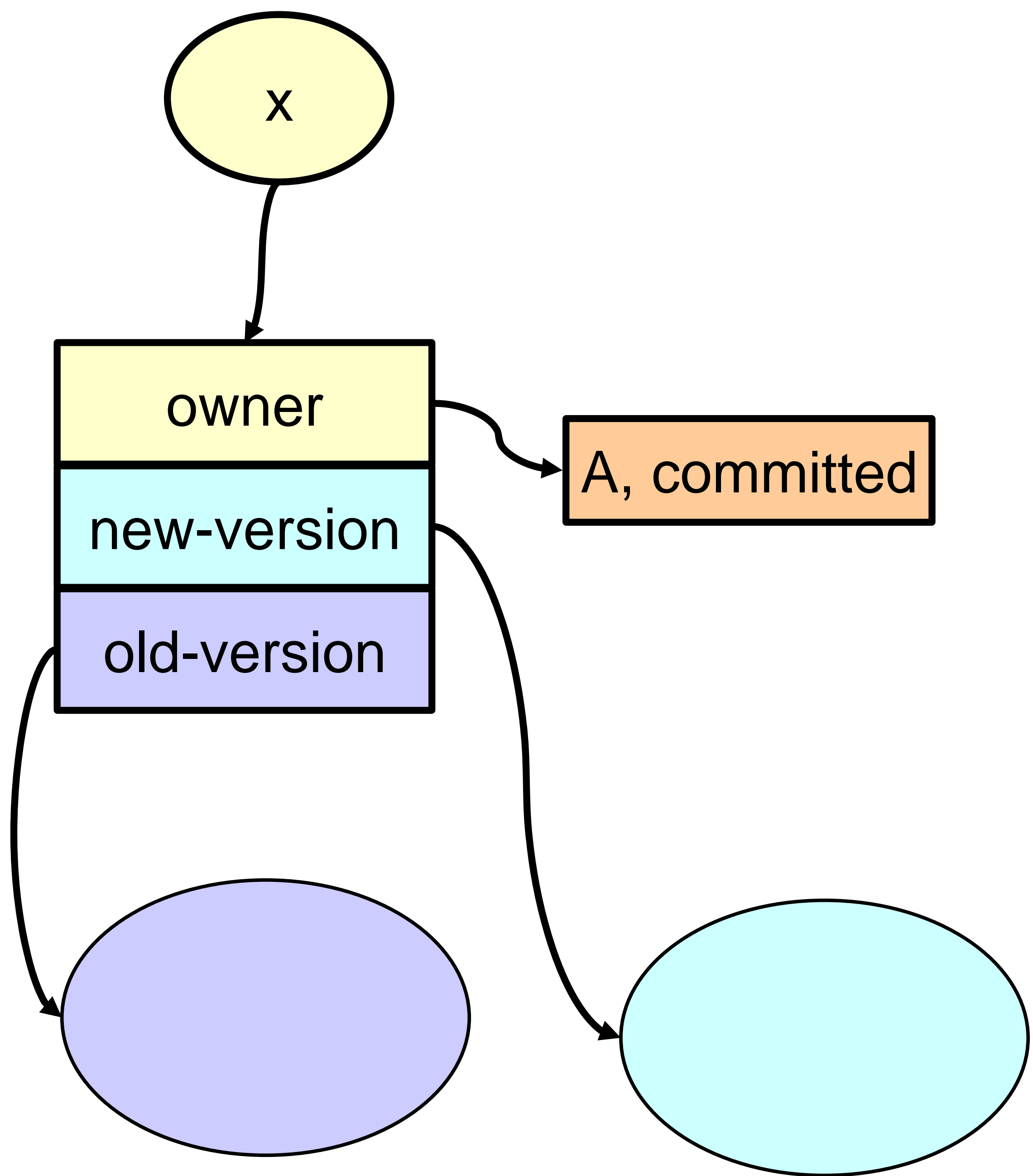
- Object x has fields:
  - owner, the last transaction to access object x,
  - old-version, value of object x before the owner transaction arrived,
  - new-version, value of object x including the owner's updates.
- Transaction T has field:
  - status, in {committed, aborted, active}.
  - active when T begins, later changed to committed or aborted.
- If owner of x is:
  - committed, then new-version is current,
  - aborted, then old-version is current,
  - active, then there is no current version (or old-version is current).
- When transaction T accesses object x, with owner A:
  - If A is committed, T becomes the new owner, sets old-version and new-version to previous new-version, performs operation on new new-version.
  - If A is aborted, T becomes the owner, sets old-version and new-version to previous old-version, performs operation on new new-version.
  - If A is active, T calls contention manager to abort A (perhaps waiting for some time to give it a chance to finish), using a CAS.
- When transaction T wants to commit:
  - Use CAS to set T.status to committed.

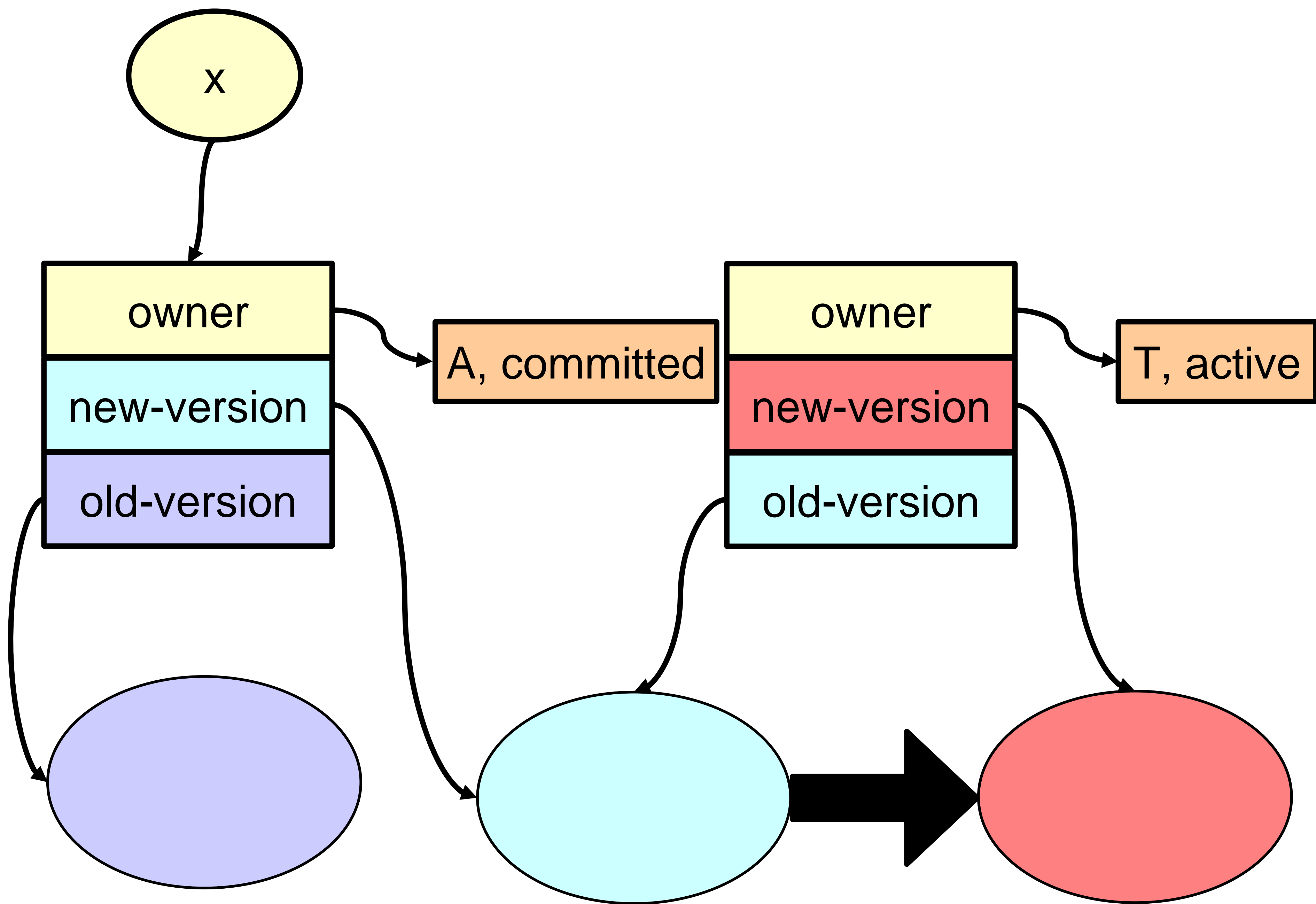

x
owner
A, aborted
new-version
old-version

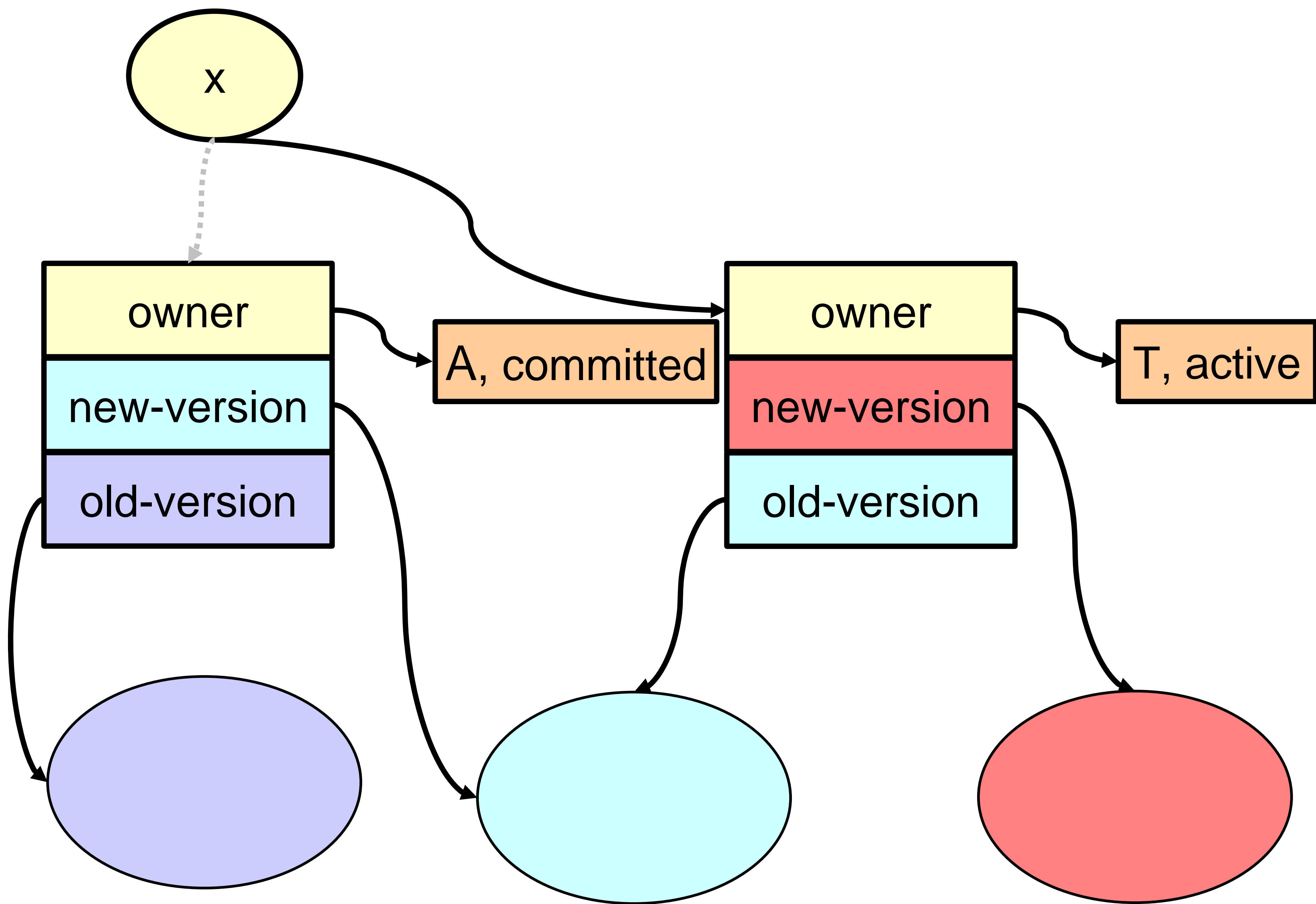


x
owner
new-version
old-version
A, aborted
owner
new-version
old-version
T, active

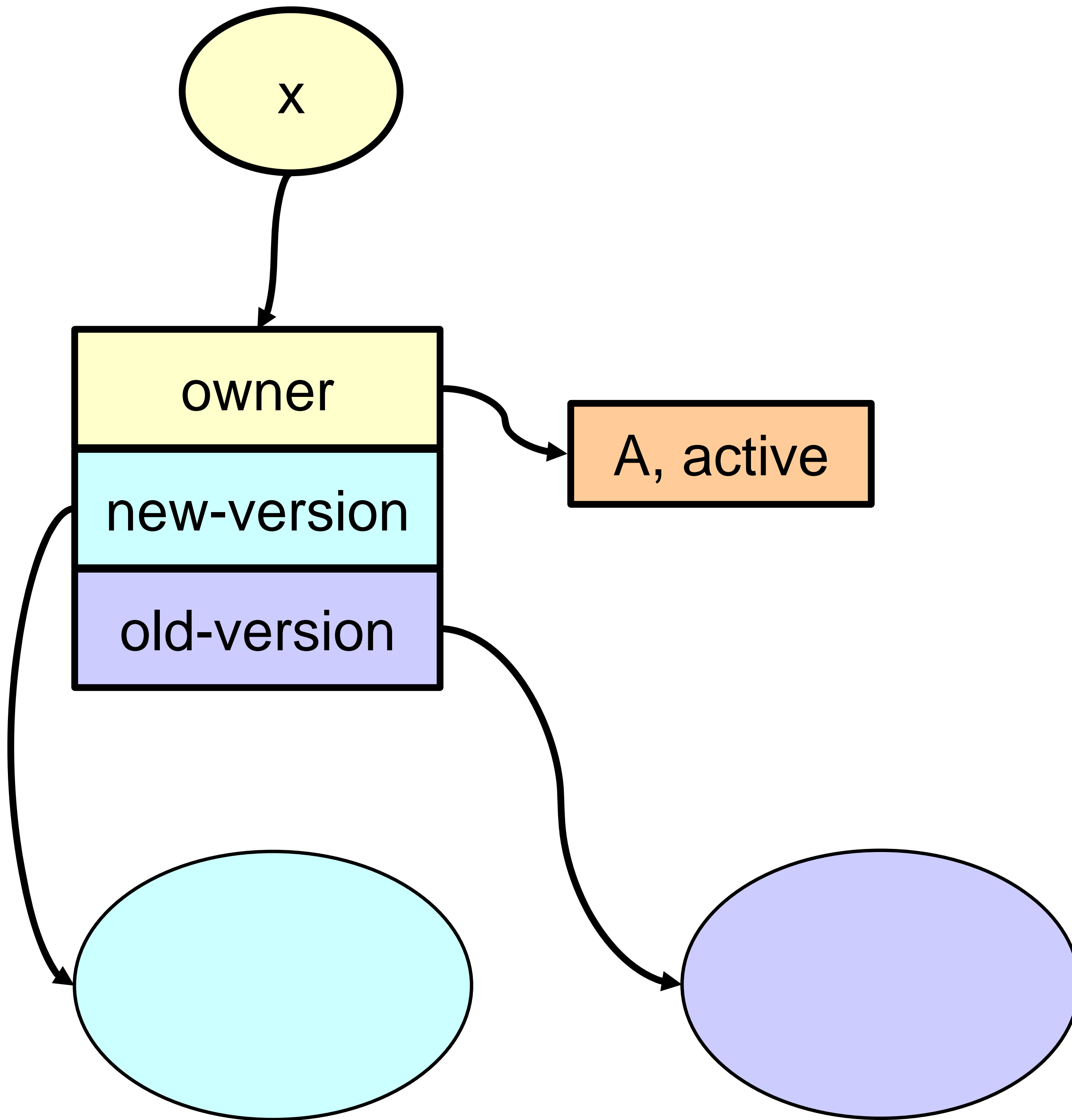

x
owner
new-version
old-version
A, aborted
owner
new-version
old-version
T, active


x
owner
new-version
old-version
A, committed


x
owner
new-version
old-version
A, committed
owner
new-version
old-version
T, active



x
owner
new-version
old-version
A, committed
owner
new-version
old-version
T, active


x
owner
A, active
new-version
old-version

Reading and Validation

- What about reads?
  - If object is owned (for writing), read new-version.
  - If not, could acquire ownership, as for writes, but this inhibits concurrency.
  - Instead, remember current version (aborting active owner if necessary), and validate.
- Validation: check that all object read but not written have same version as when read.
  - For consistency, need to do this after every read!

Evaluation

- Liveness: Obstruction-freedom
  - From any point, if a transaction T runs by itself, it eventually commits (assuming that, if it succeeds in accessing every object, it eventually commits).
  - T acting alone proceeds in a wait-free manner except when it encounters an active transaction A.
  - But then it (eventually) aborts A.
- Serializability:
  - Each committed transaction T is serializable at some point in the interval between its creation and commit.
  - Serialization point can't be the commit point (CAS).
  - If the CAS succeeds, T has not been aborted by any other transaction and so is still the owner of all the objects it accessed.
- Cost: O(1) for reading/writing; O(n) commit.
  - But O(n) validation: if we validate after every read, $O(n^2)$ total cost.

Transactional Locking 2 (TL2)

- Global integer-valued version clock, used to generate a readstamp for each transaction, and a writestamp for each transaction that tries to commit.
- Transaction T keeps all modifications local, only installs them during commit (optimistic).
- Keeps track of:
  - Readstamp, writestamp
  - read set, objects it has read,
  - write set, objects it has written, with their tentative new versions.
- When transaction T starts:
  - Record current clock as its readstamp.
- Object x has fields:
  - stamp, writestamp of the last transaction to write to x,
  - version,
  - lock (exclusive).

TL2

- When T tries to read x:
  - If x is in T.writeset, then read T's tentative new version.
  - Validate: If x is locked or x.stamp > T.readstamp, then T aborts.
  - Otherwise, read x.version and add x to T.readset.
- When T tries to write x:
  - If x is in T.writeset, then overwrite T's tentative new version.
  - Otherwise, add x to T.writeset with the tentative new version.
- When T tries to commit:
  - T locks all the objects in its write set.
  - Reads and increments the global clock (fetch&increment), stores new value as T's writestamp.
  - For every x in T.readset:
    - Validate: If x is locked by another transaction or x.stamp > T.readstamp then T aborts.
  - Installs the new versions in all the objects, with T.writestamp.
  - Releases all the locks.

TL2

Animation ideas:

1. T arrives, gets readstamp 0, does one read and one write, commits successfully with writestamp 1.
2. T1 and T2 both get readstamp 0. Operate concurrently, doing reads of the same object x and then writes of different objects y1 and y2 (respectively). No interference, neither aborts, both commit. T1 gets writestamp 1 and then T2 gets writestamp 2.
3. T1 and T2 both get readstamp 0. Now T1 proceeds first, writes an object T2 reads, and commits. Then T2 does the read, sees the conflict, and aborts because it reads a new value during its read step.
4. T1 and T2 both get readstamp 0. Proceed concurrently, T1 writes an object T2 reads. But now T1 commits first, installing the new value of the object. T2 aborts upon final validation.

Correctness

- Serializability:
  - Each committed transaction T can be serialized where it determines T.writestamp (fetch&Increment).
  - Thus, transactions are serialized in order of their writestamps.
  - Why this works (reads get the last preceding versions):
    - Every version T read must have been written by some other transaction with writestamp ≤ T.readstamp.
    - For each object x, T must read the version of x with the largest writestamp ≤ T.readstamp, since they are already written when T starts.
    - Final validation ensures that no transaction A with T.readstamp < A.writestamp < T.writestamp wrote any object in T.readset.
- Liveness: Guarantees progress, but allows starvation.

Current Status of TM Research

- Many algorithms have been developed.
  - Not yet efficient enough for most practical use.
- Libraries are available for Java, etc.
- Compiler support beginning to be available.
  - Intel, Sun
- Draft spec of transactional constructs in C++.
- Nascent theory
  - Specification, verification, lower bounds
- Other: tools, interaction with other synchronization mechanisms

Combining hardware and software

- Hardware-assisted transactional memory
  - new required hardware
  - hardware support can accelerate implementation
- Hybrid Transactional Memory (HyTM)
  - STM that can work with HTM
    - hardware transactions and software transactions must “play nicely”
    - can be used now (with no hardware support)
    - can exploit HTM support with little change
  - Phased Transactional Memory (PhTM)
    - can switch dynamically between using STM and HTM

Next time

- Asynchronous networks vs asynchronous shared memory
- Agreement in asynchronous networks
- Reading:
  - Lynch, Chapter 17
  - Lamport paper: The Part-Time Parliament (the Paxos paper)

- Wait-free consensus hierarchy
- Universality of consensus

MIT OpenCourseWare
http://ocw.mit.edu

6.852J / 18.437J Distributed Algorithms
Fall 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.