The H2O Distributed Key-Value Store

Cliff Click

Who Am I?

Cliff Click

Leader, Founder

Cratus, Rocket School, Neurensic,
H2O.ai, Azul, Sun

cliffc@acm.org

PhD Computer Science

1995 Rice University

HotSpot JVM Server Compiler

“showed the world JITing is possible”

45 yrs coding

40 yrs building compilers

35 yrs distributed computation

30 yrs OS, device drivers, HPC, HotSpot

15 yrs Low-latency GC, custom java hardware,
NonBlockingHashMap

10 yrs ML tool building, ML applications

20+ patents, dozens of papers

100s of public talks

What is H2O?

- Machine Learning on Big Data, Big Math
- Parallel, Tightly Coupled In-Memory Computing
- **Fast** Machine Learning
  - e.g. Logistic Regression on 7Tb in 90sec
  - 10x faster than Spark, 100x faster than Hadoop...
- Lots of need for cross-node communication
- Sometimes for **throughput**: Big Data being moved
- Often for **control**: ordering operations
- Need for both **speed** and **exact** consistency

H2O Coding For Big Math

- Data in a (distributed) large 2-D array
- Coding like a single-threaded Java machine:
  - `for( int i=0; i<N; i++ )`
    `A[i] = B[i]*C[i];`
  - "N" can be **trillions**++
  - **Auto-parallel, auto-scale-out**
    - **With many Tb memory and 1000's of cores**
  - Most simple Java "just works"
- Cluster behaves like a large shared-memory
  - Up to the limits of the JMM…

Tightly Coupled Cluster

- Most simple single-threaded Java “just works”!
  - And most math code written by **mathematicians**
  - But lots of edge cases
  - And special algorithms (e.g. distributed sort)
- Very hard to code to true distributed memory
  - Even for **distributed systems engineers**
- Want the Java Memory Model, but Distributed
  - **Exact** (up to the JMM)
  - **Bulk speed**: reliably hit Memory Bandwidth
  - Much faster than the network...

Bulk Speed on Big Data

- Big Data: performance a driving concern
- In-memory compute hits Memory Bandwidth
  - ~50G/sec on modern node, ~30sec for a Tb
- When data not local must use network
  - So want to hit network bandwidth limits
  - And feed the data from NIO buffers direct to CPU
- Actual data movement uses custom serialization
  - gen'd class to write instances in/out of NIO buffers
  - plus aggressive compression
  - Fast as kryo when last tested, without setup costs

The JMM, Distributed

- Volatile read/write well defined in Java
  - Strong ordering requirements
- Non-volatile also well defined…
  - And mostly “anything goes”
- We can find lots of engineers who understand the JMM
  - Not so many to **design** a distributed JMM
- So I go about making “The JMM, Distributed”
  - Long talks with Kevin Normoyle (lead cache designer Azul Systems, Sun)

“The JMM, Distributed”

- Now **two** layers of Memory Model
  - The “normal” one: loads/stores ordered by JMM
  - But only within a single JVM address space
- And a Distributed JMM:
  - loads/stores replaced with get/put
  - In a global (cluster-wide) address space
    - And with bulk operators for Big Data
- This becomes a classic

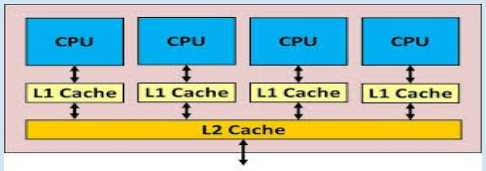
  Distributed **K**ey **V**alue Store
- But with the JMM ordering: *exact* semantics

Distributed Key / Value Store

- "put(key,value)" on one node
- "value = get(key)" on another
- Follows Java Memory Model
- Keys used externally to point to
  - Datasets, ML Models, Results
- Keys used internally to hold almost all state
  - Big Data, temps during model building, ML models, scored data, cached results of all kinds

DKV is Fast

- Hardware-style cache-coherency - MOSI
- All values cache locally
  - Cache-hitting GETs take ~150ns



- PUTs to different Keys can overlap
  - Or not; user choice (i.e. volatile vs non-volatile)
  - Limits of network bandwidth not latency
  - Local PUTs same as cache-hits, ~150ns
- PUTs to same Key must order (network latency)
- Keys pseudo-randomly placed
  - No “hot blocks”; good average behavior

Distributed Keys

- Keys have a **home node**
  - Pseudo-random (except for Big Data)
  - Big Data: round-robin with mini-batching
    - Round-robin: uniform sharing of Big Data across nodes
    - Mini-Batching: most edge cases **do not cross node edges**
- Fully peer-to-peer, no master Key directory
- Home has ultimate “truth” for GETs
- Home breaks ties for racing PUTs
  - Same as JMM on racing stores: “eventually” one wins
    - “Eventually”: speed of network
  - Atomically executes transactions

Values

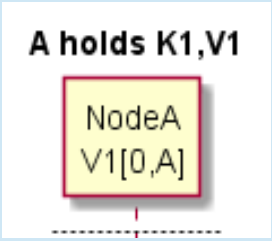
- Values track coherency
- Track which nodes have a replica Value
- Invalidates replicas to preserve JMM
  - Very similar to hardware caching protocols
- Force write ordering from same Node same Key
- Values hold:
  - a reader/writer lock
  - Count of outstanding GETs, or else a PUT
  - Concurrent bitset of caching nodes

Example: Some ML...

- 4-node cluster
- New ML Model stored local to Node A
- Other nodes told to go compute error metrics
  - Nodes B,C,D will need the Model
- Later, Node C produces a New Better Model
  - Writes to Node A
  - Nodes B & D need to be informed
- Here we go...

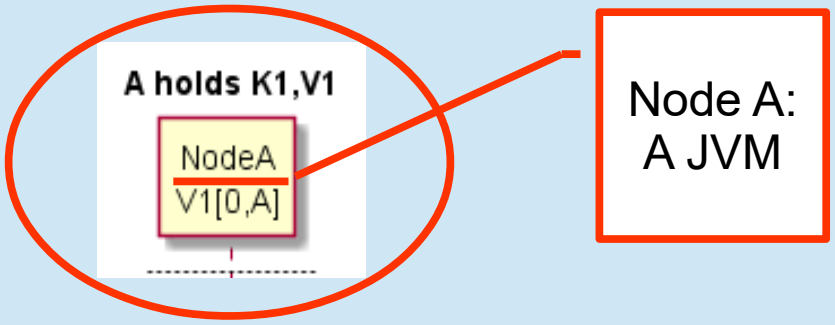
Example: initial conditions

- 4-Node cluster; single Key K1 (the Model name)
- Node A (a single JVM) holds Value V1 (the Model)
- Quiescent: no GETs in flight



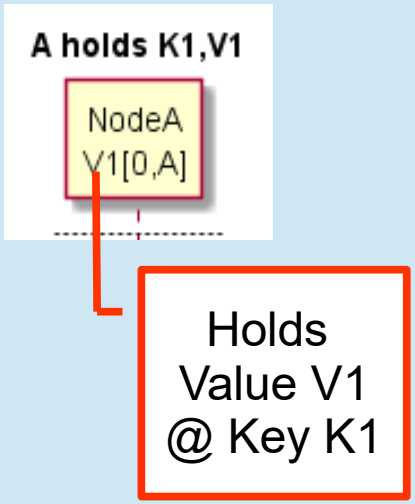
Example

- 4-Node cluster; single Key K1 (the model name)
- Node A (a single JVM) holds Value V1 (the model)
- Quiescent: no GETs in flight



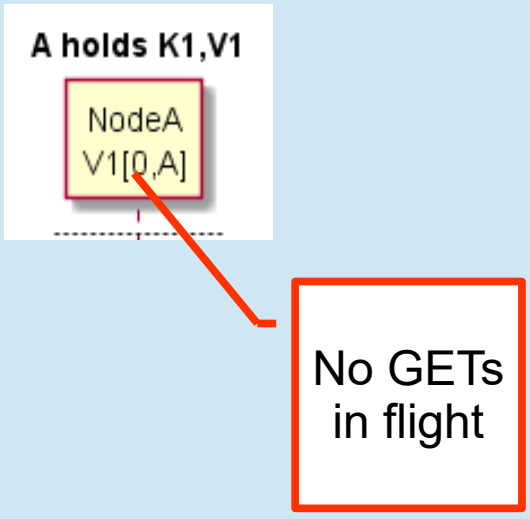
Example

- 4-Node cluster; single Key K1 (the model name)
- Node A (a single JVM) holds Value V1 (the model)
- Quiescent: no GETs in flight



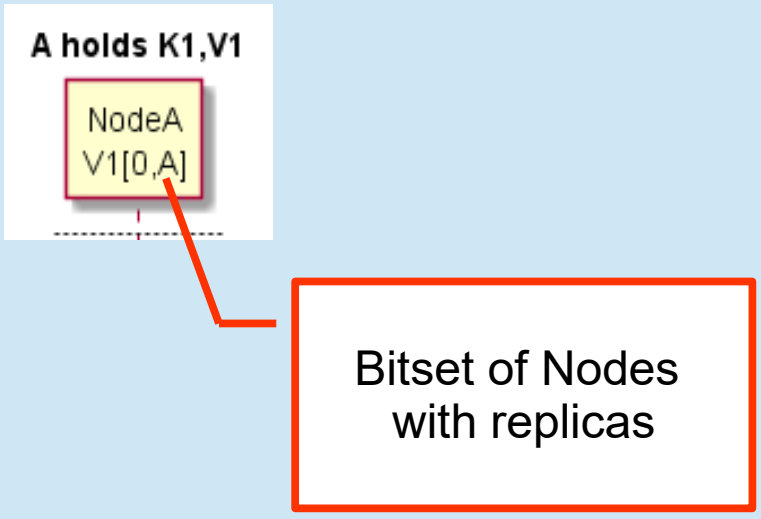
Example

- 4-Node cluster; single Key K1 (the model name)
- Node A (a single JVM) holds Value V1 (the model)
- Quiescent: no GETs in flight



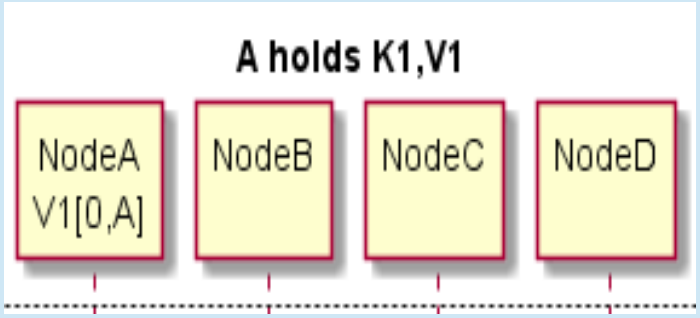
Example

- 4-Node cluster; single Key K1 (the model name)
- Node A (a single JVM) holds Value V1 (the model)
- Quiescent: no GETs in flight



Example

- Actually a 4-node cluster
- Other nodes not caching K1 yet

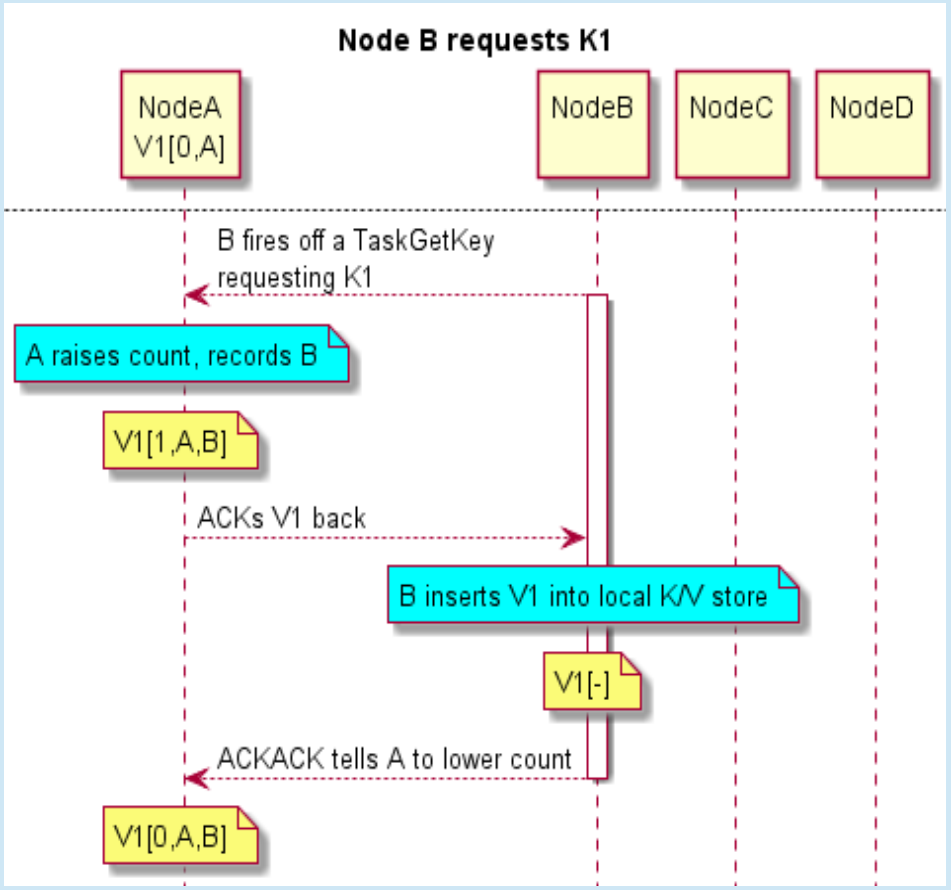


Example: B misses locally

- Node B needs Model, does GET(K1)
  - Misses in B's local K/V store
  - Hashes K1 to get K1’s home: A
- Fetch K1 from A
  - Minimum 2 UDP packets – one send, one receive
  - Use UDP if V1 is small for lowest latency
  - Use TCP if V1 is big for congestion control

Example: B misses locally

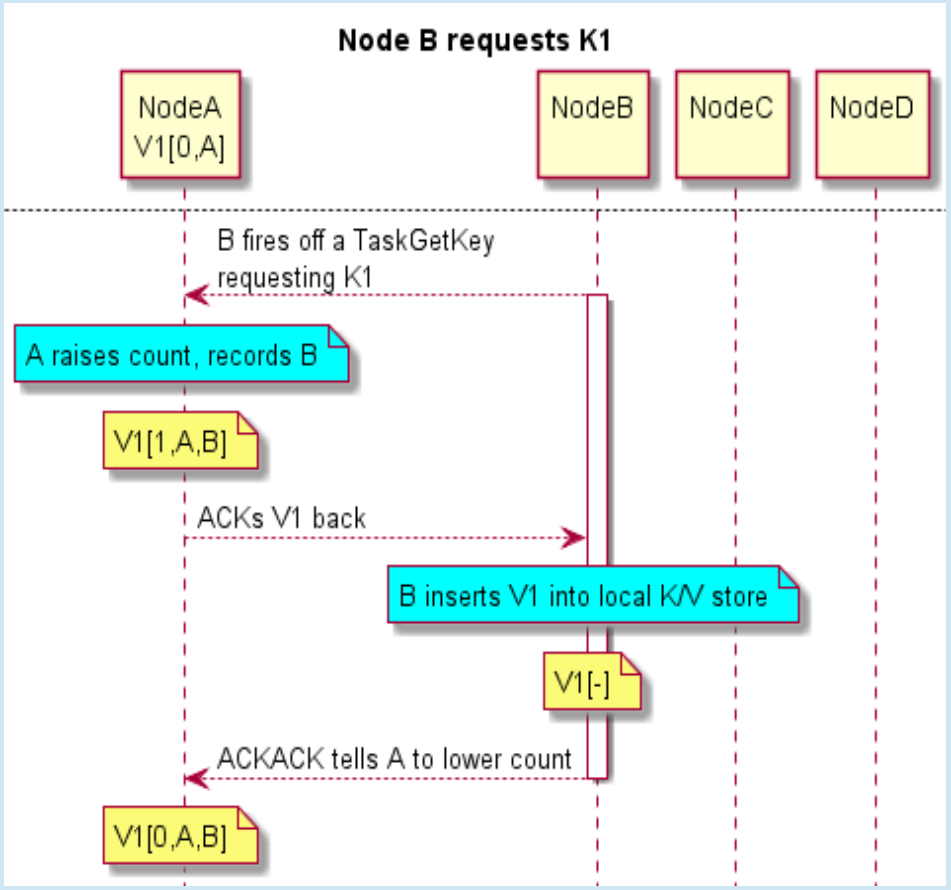
- Node B does GET(K1)
  - Misses in B's local K/V store, fetch from A



Example: B misses locally

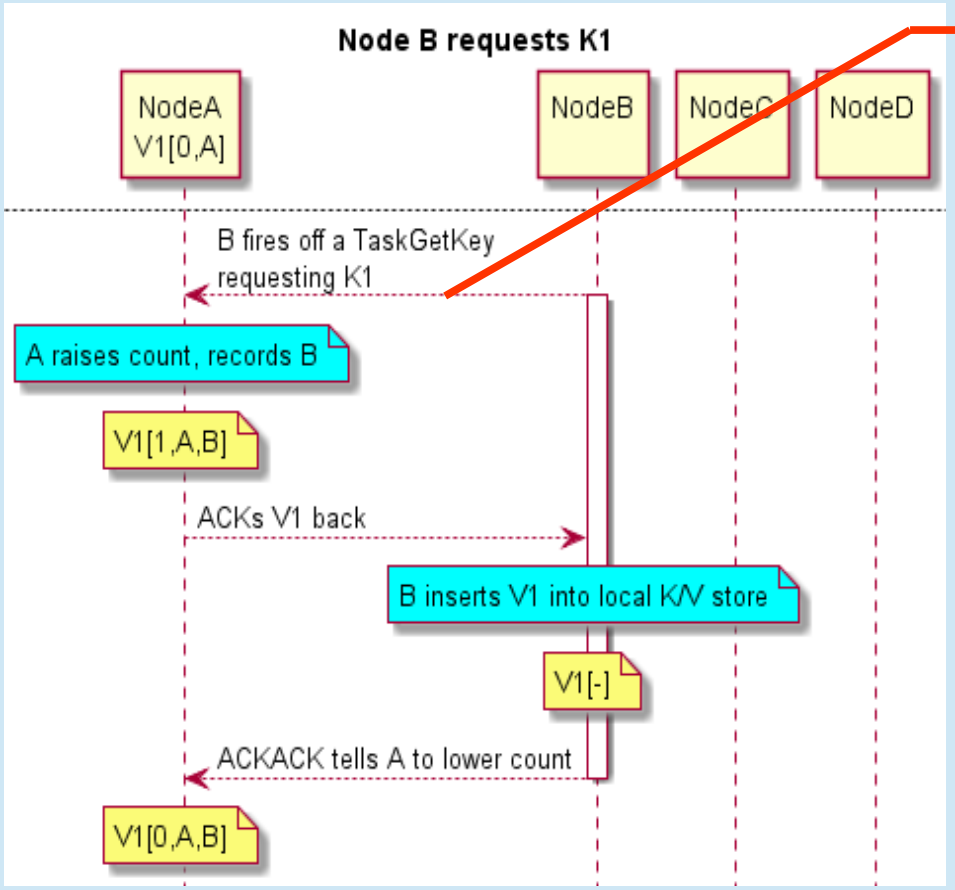
- Node B does GET(K1)
  - Misses in B's local K/V store, fetch fr

GET(K1)
...which misses locally.
Must wait at least a
send & recv UDP



Example: B misses locally

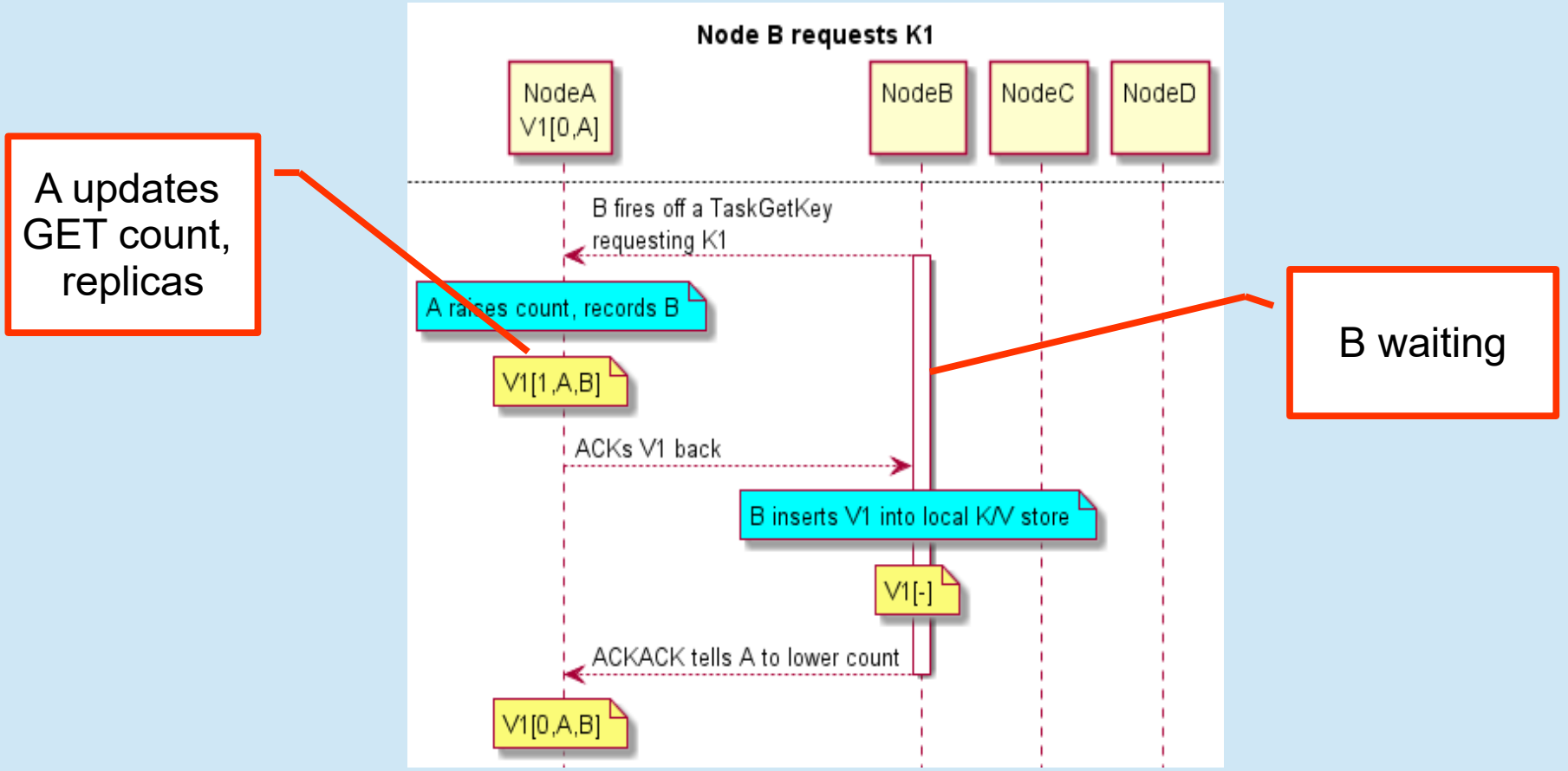
- Node B does GET(K1)
  - Misses in B's local K/V store, fetch from A



B knows K1's home is A
… so sends packet to A

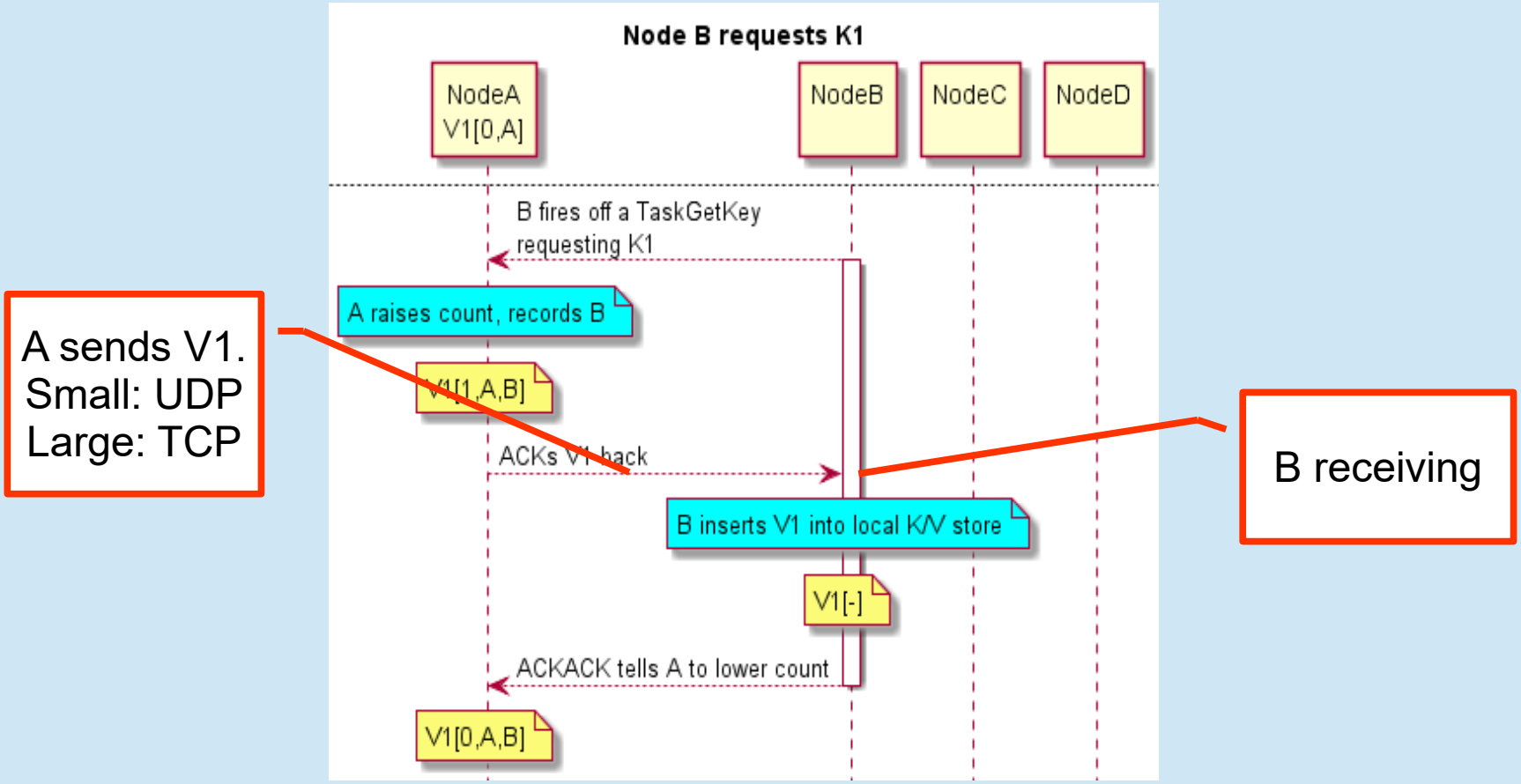
Example: B misses locally

- Node B does GET(K1)
  - Misses in B's local K/V store, fetch from A



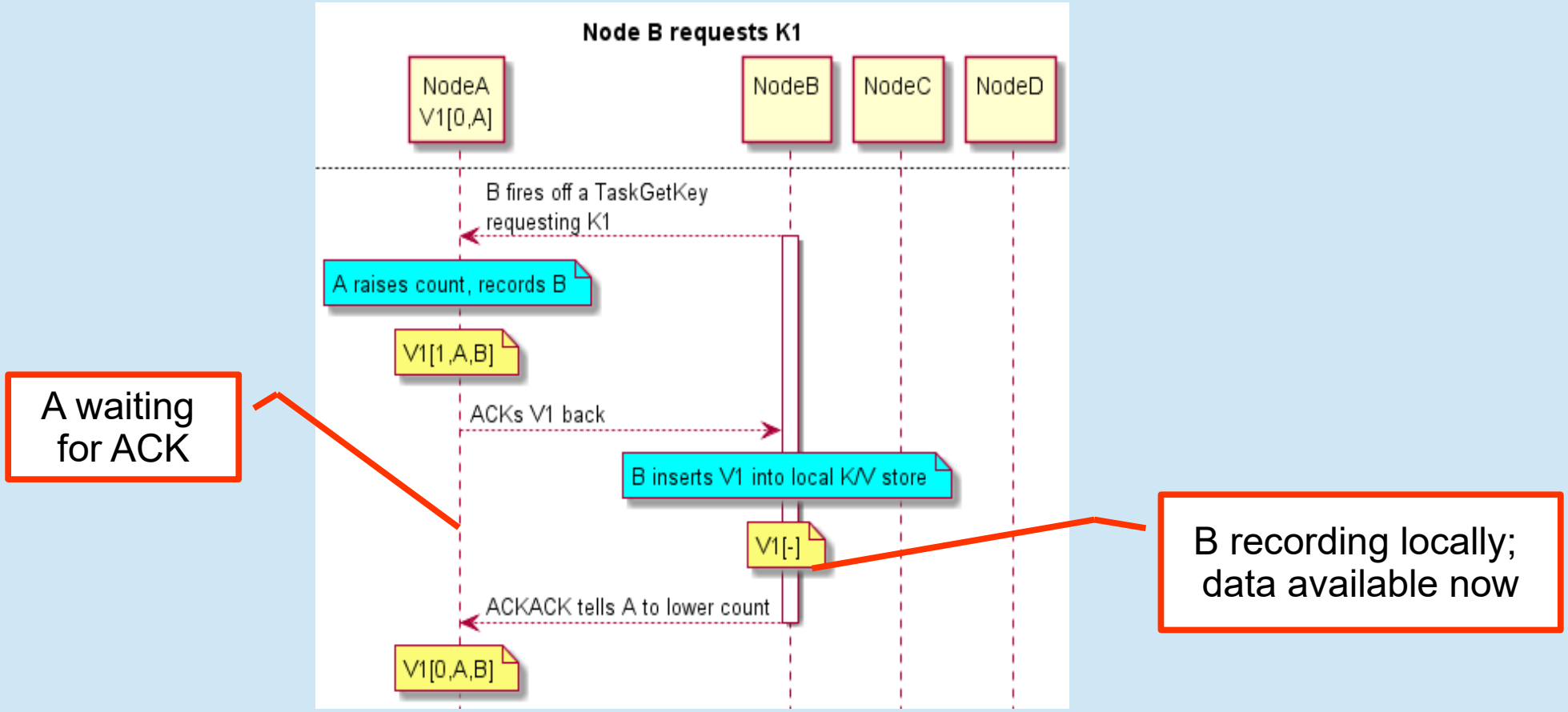
Example: B misses locally

- Node B does GET(K1)
  - Misses in B's local K/V store, fetch from A



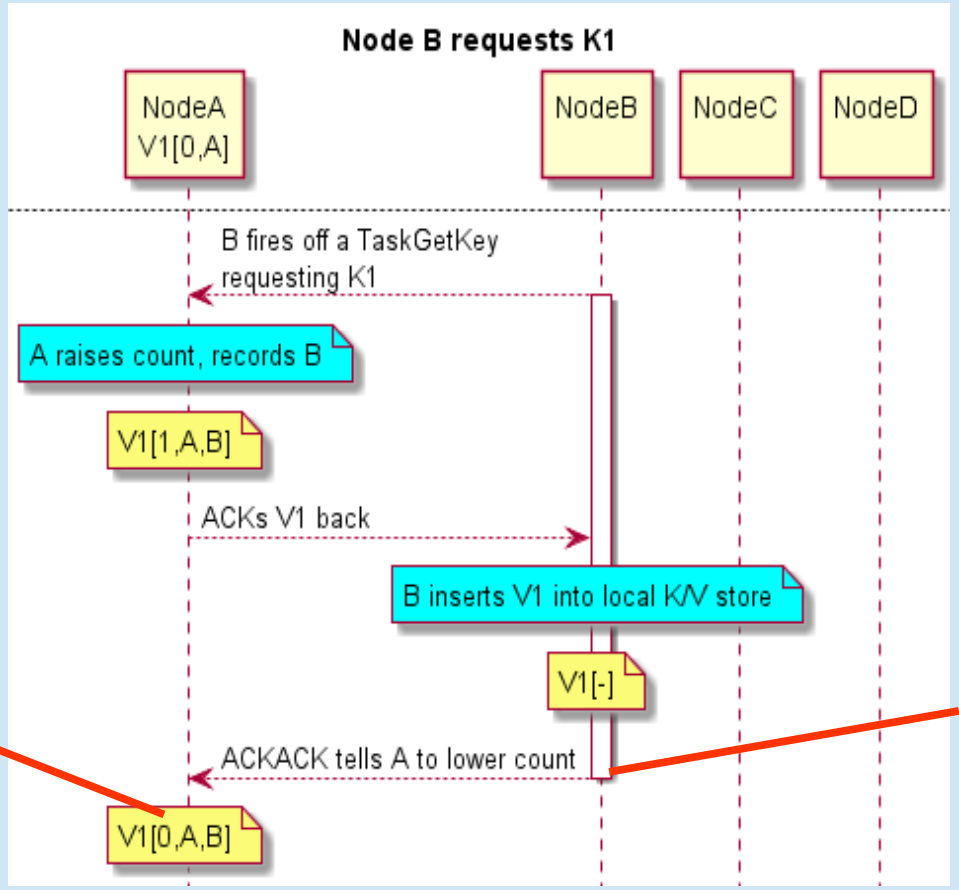
Example: B misses locally

- Node B does GET(K1)
  - Misses in B's local K/V store, fetch from A



Example: B misses locally

- Node B does GET(K1)
  - Misses in B's local K/V store, fetch from A



A updates GET count

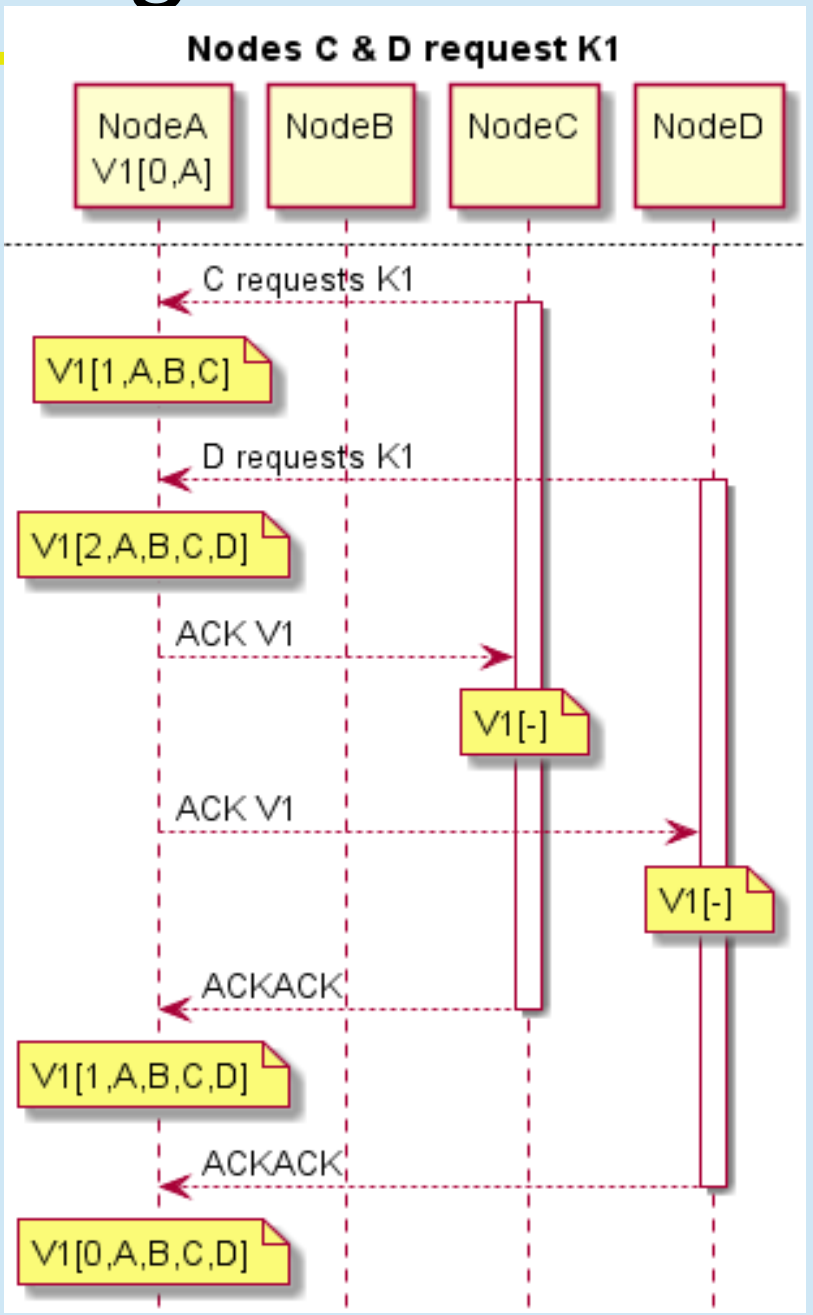
B sends ACKACK in parallel with using V1

Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping
- Data available on ACK
  - (no stalling for ACKACK)
- Reader-lock unlocked on last ACKACK
- K1 / V1 fully cached
- All future GETs 150ns

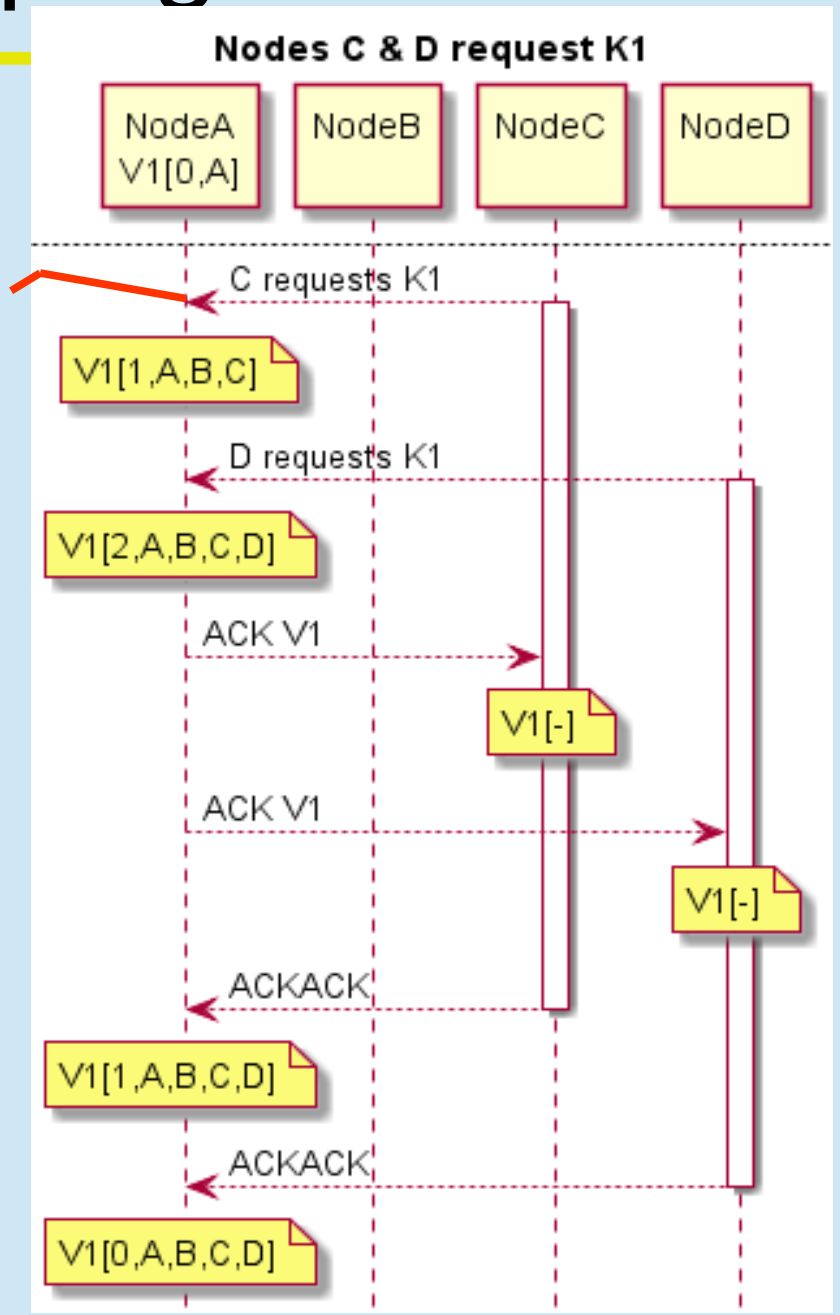
Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping
- Data available on ACK
  - (no stalling for ACKACK)
- Reader-lock unlocked on last ACKACK
- K1 / V1 fully cached
- All future GETs 150ns



Example: Overlapping GETs

- Node C & D do GET(K1)

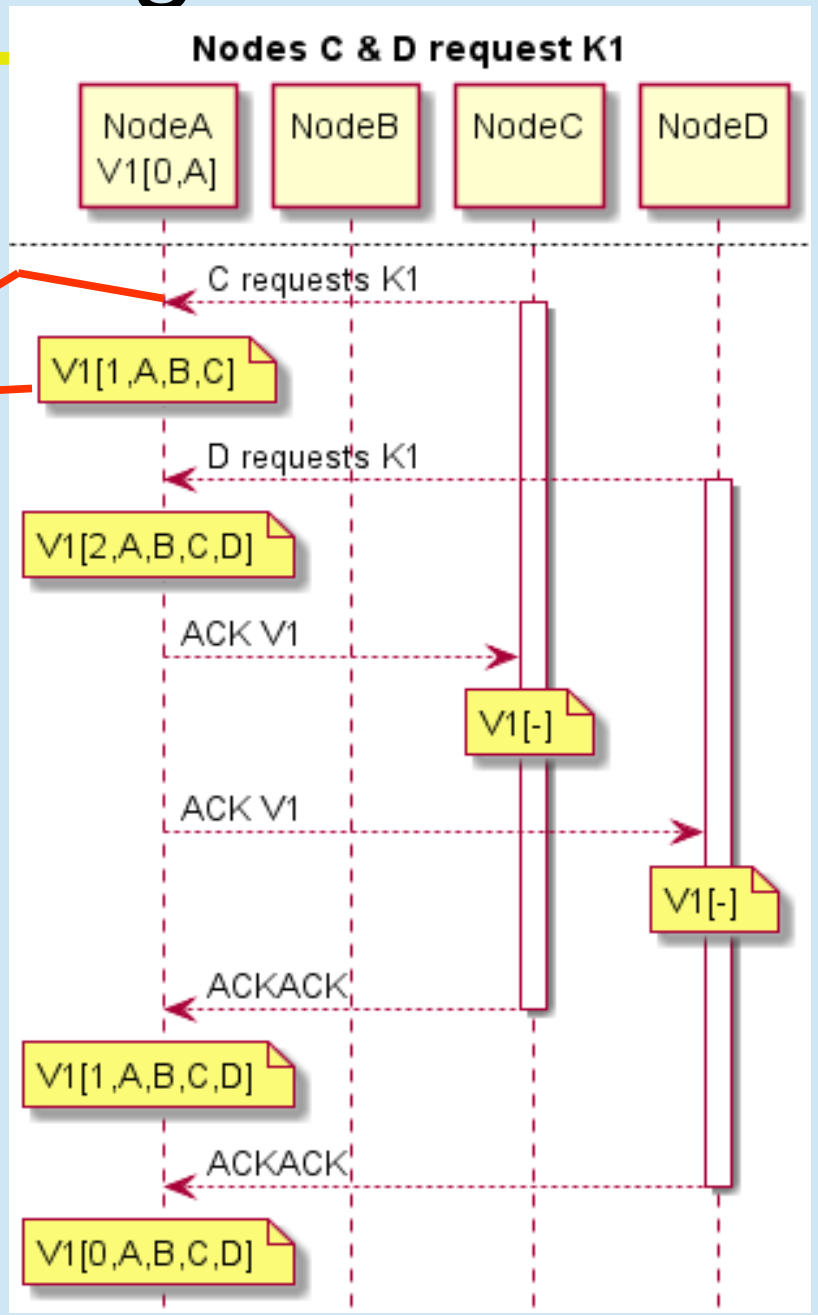


Example: Overlapping GETs

- Node C & D do GET(K1)

C sends packet to A

A gets Cs packet
A processes

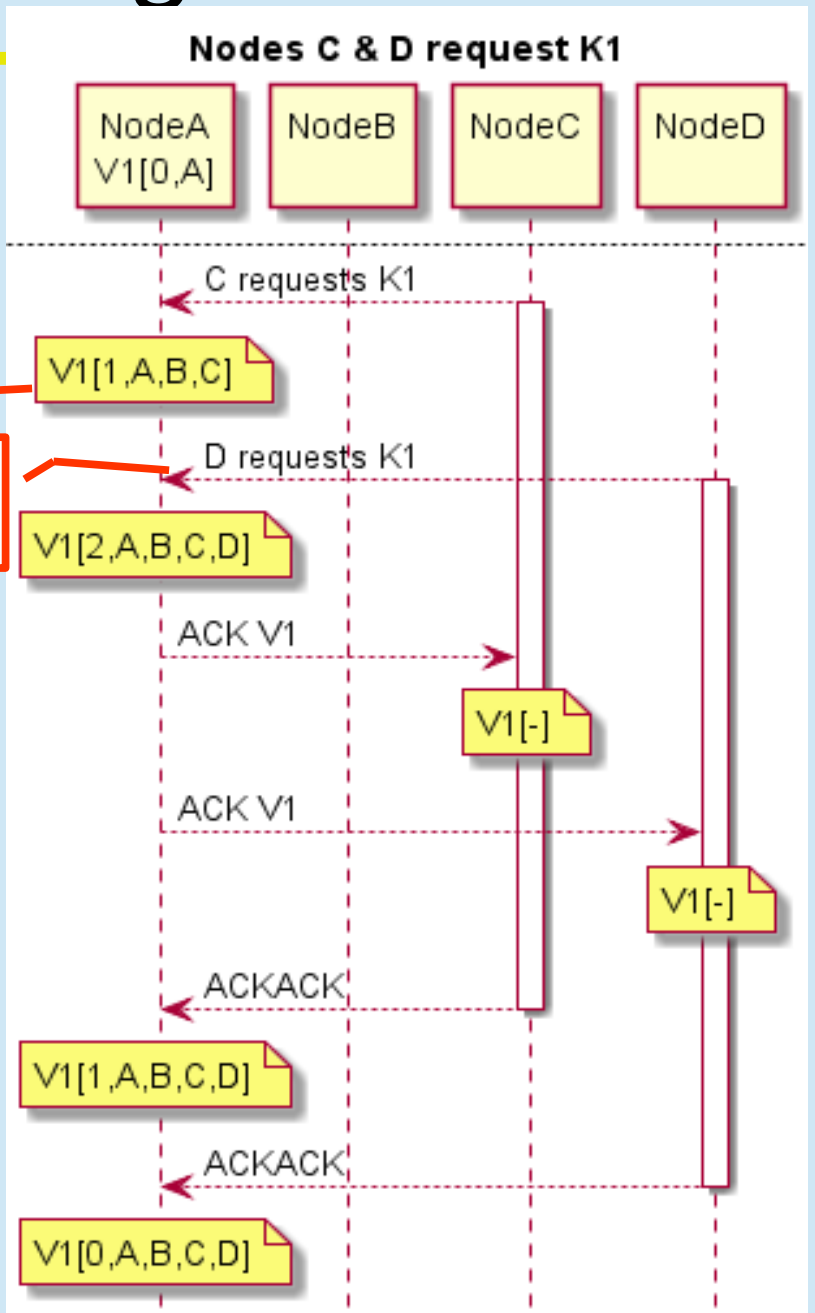


Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping

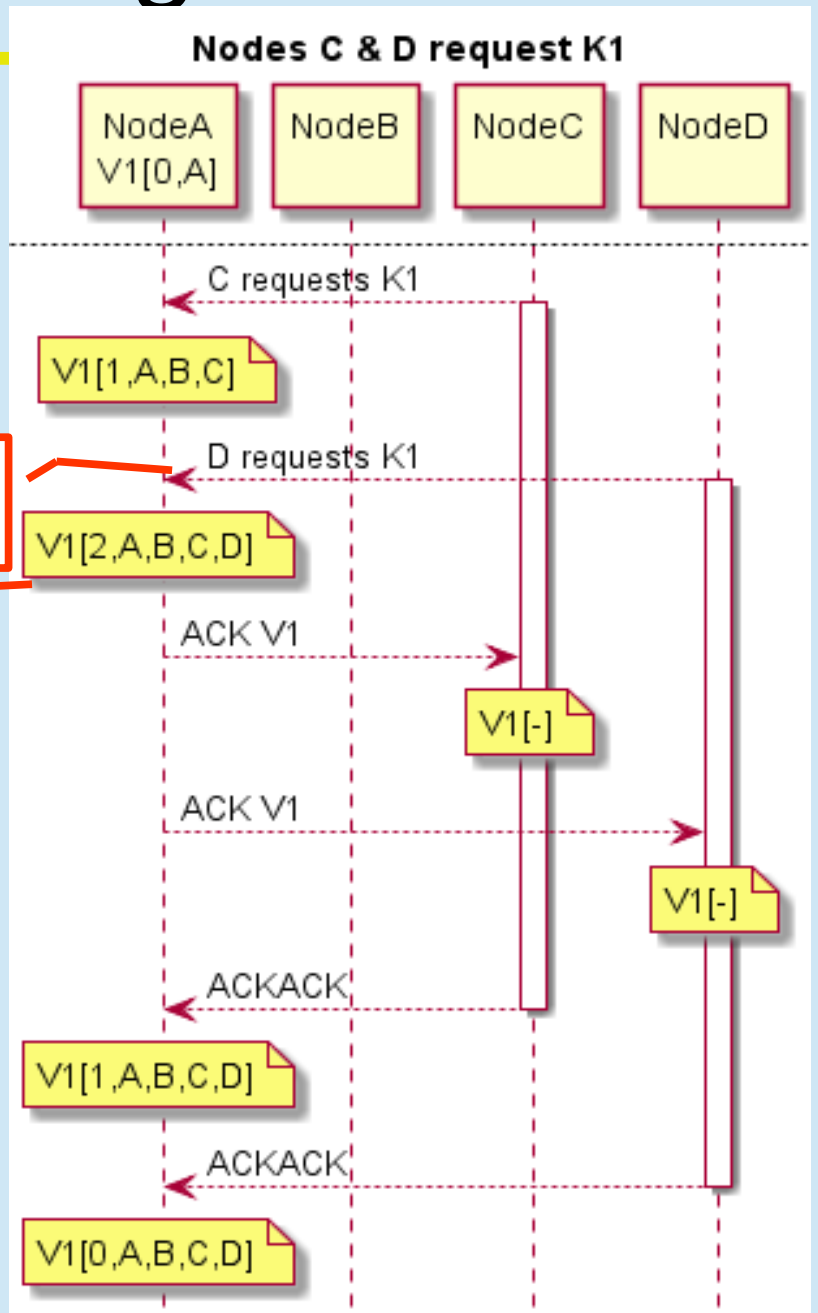
A gets Cs packet
A processes

D sends
packet to A



Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping

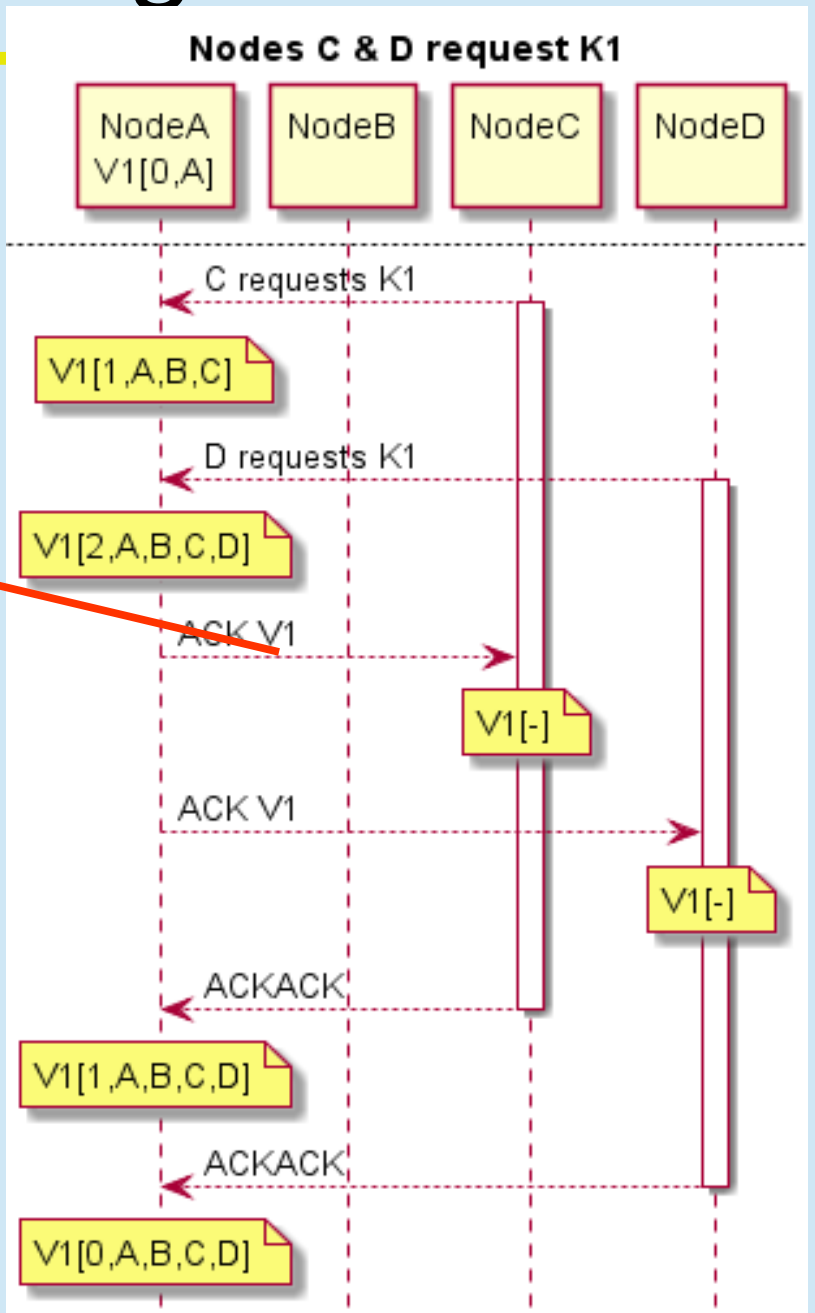


D sends packet to A

A gets Ds packet
A processes

Example: Overlapping GETs

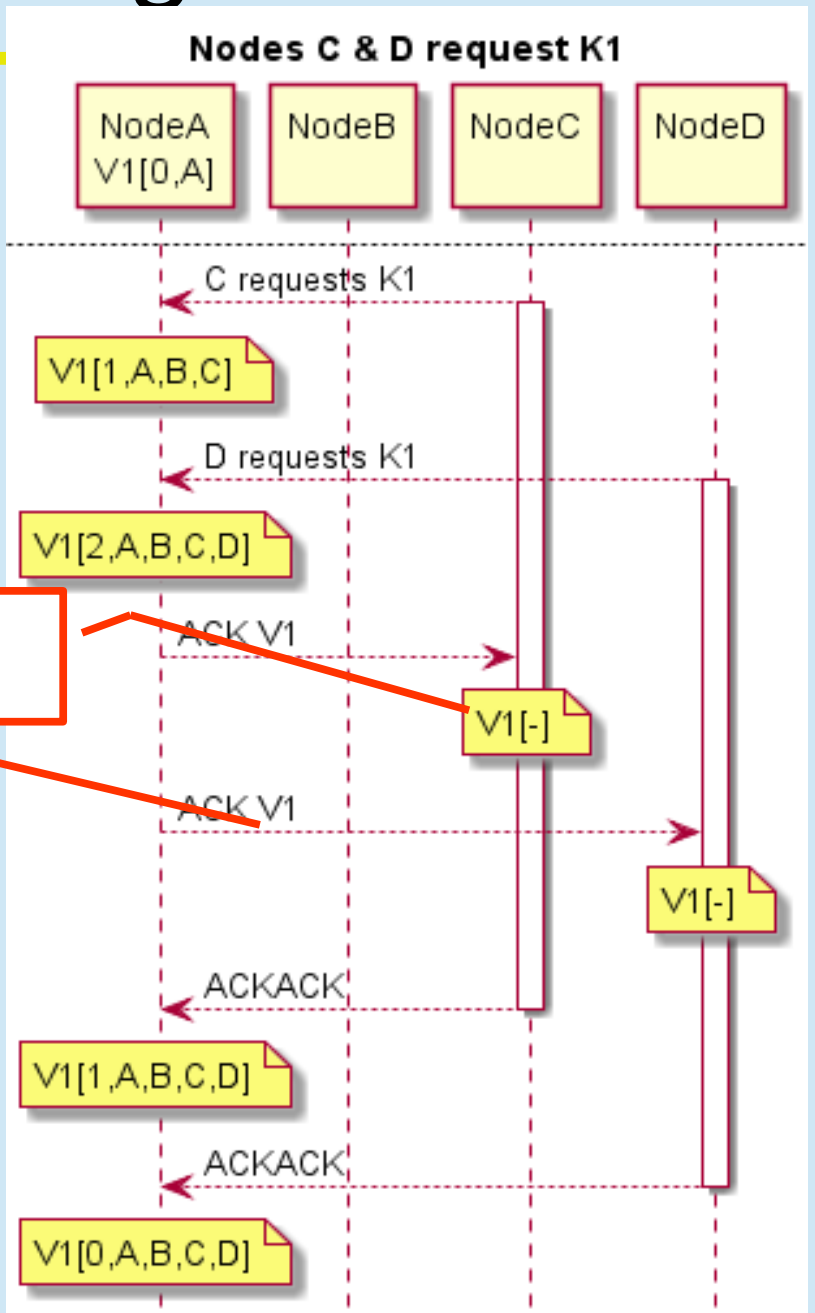
- Node C & D do GET(K1)
  - Overlapping



A sends V1 to C

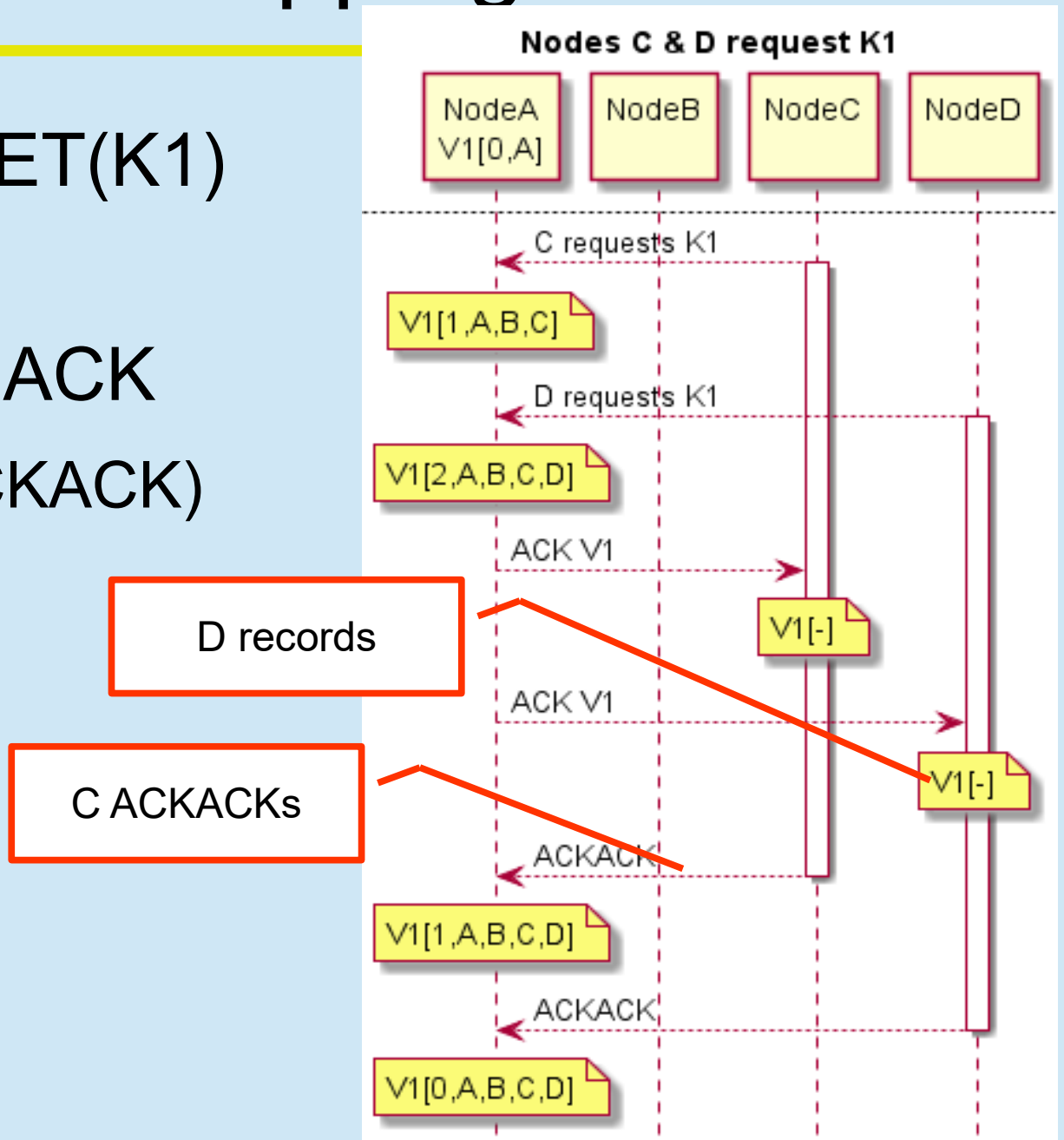
Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping
- Data available on ACK
  - (no stalling for ACKACK)



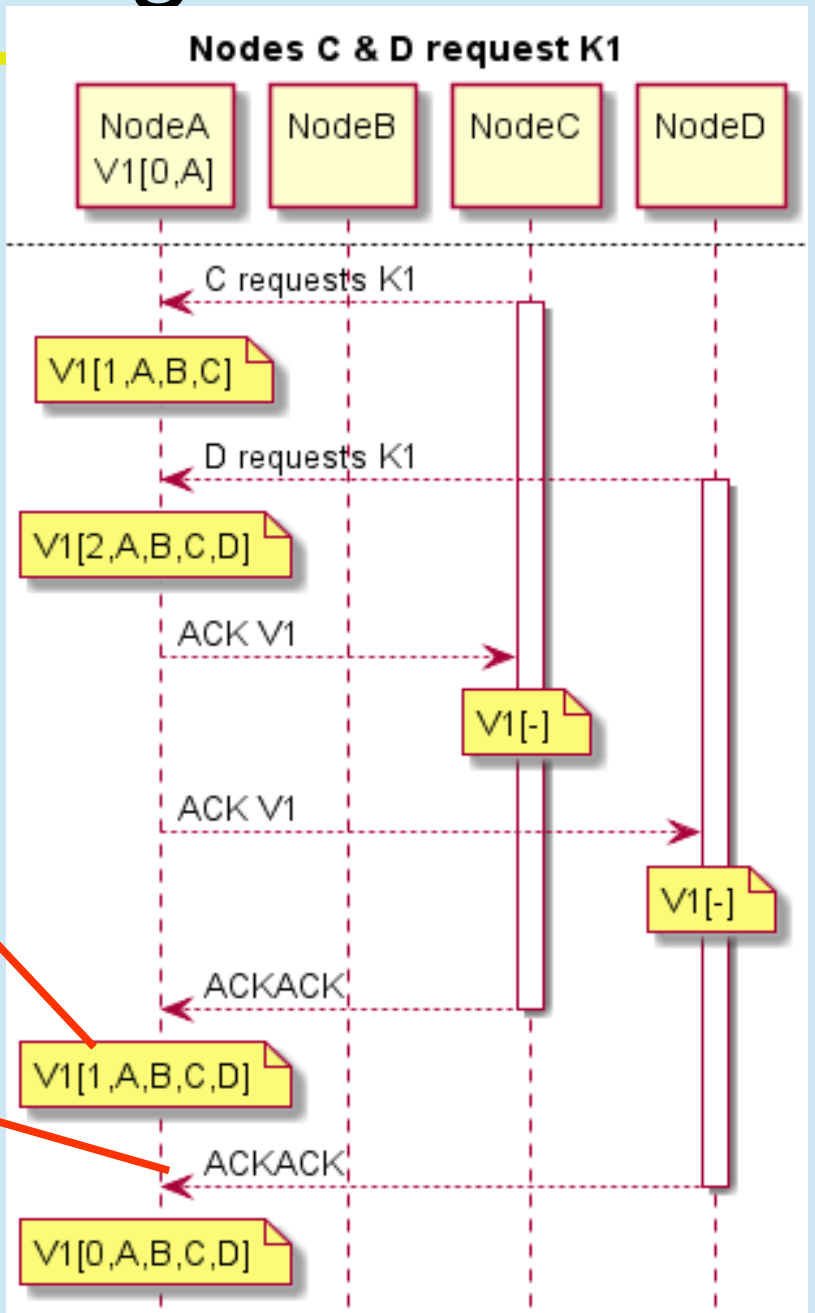
Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping
- Data available on ACK
  - (no stalling for ACKACK)



Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping
- Data available on ACK
  - (no stalling for ACKACK)
- Reader-lock unlocked on last ACKACK

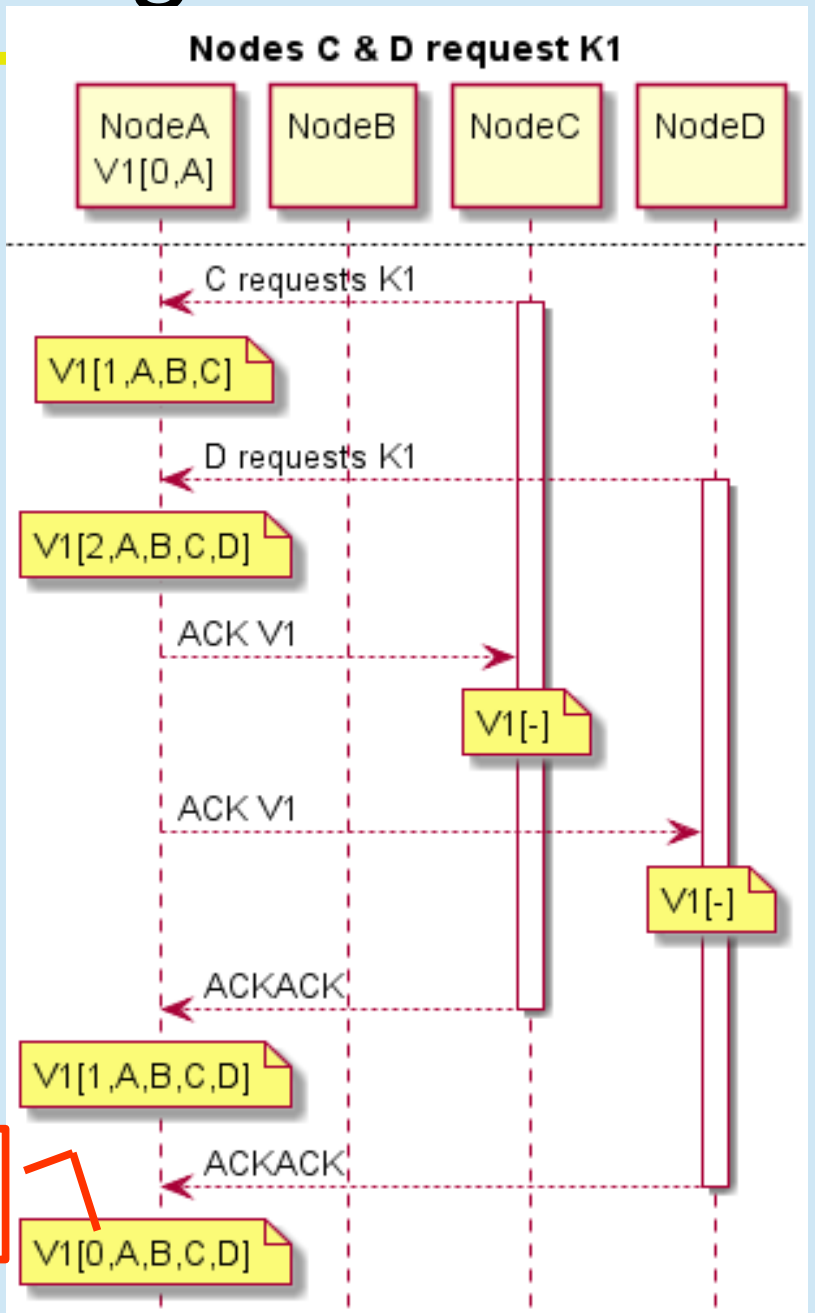


A lowers GET count

D ACKACKs

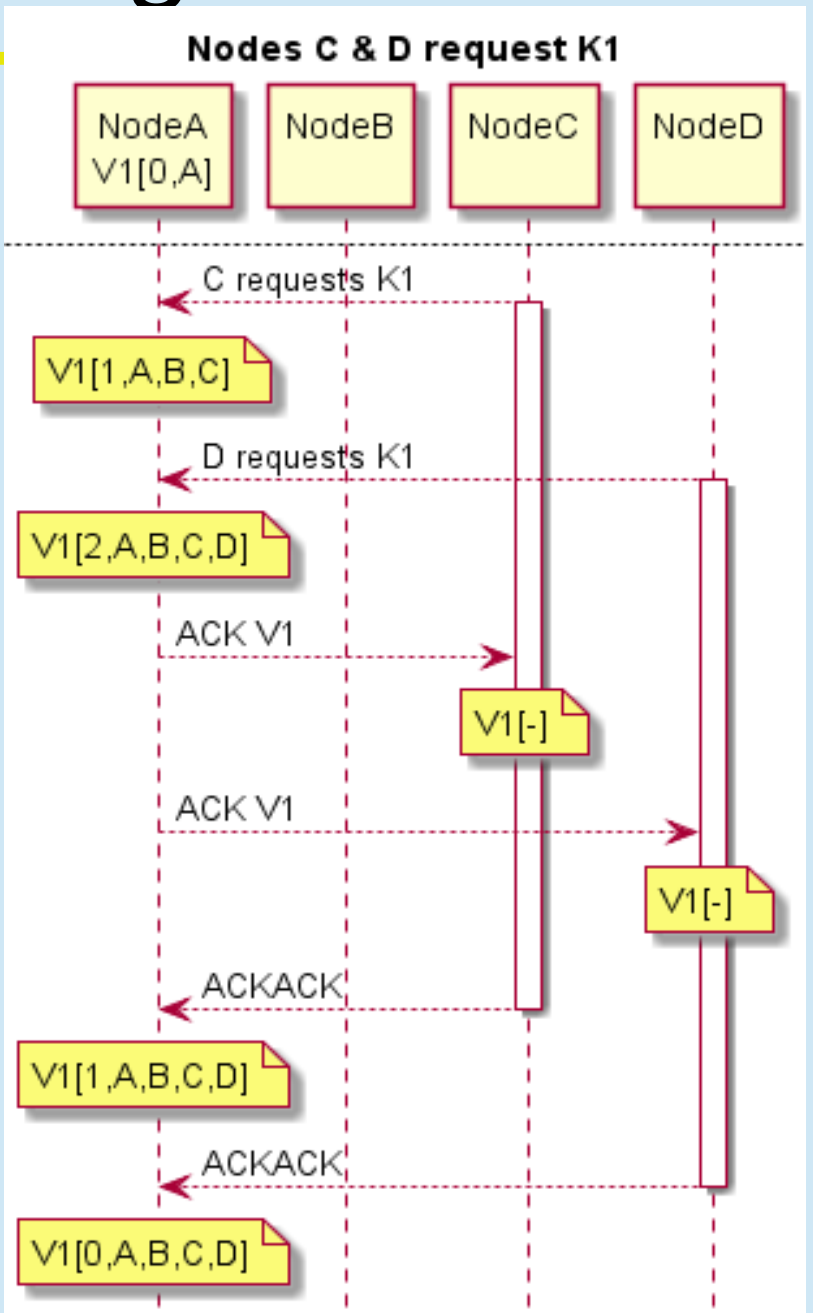
Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping
- Data available on ACK
  - (no stalling for ACKACK)
- Reader-lock unlocked on last ACKACK



Example: Overlapping GETs

- Node C & D do GET(K1)
  - Overlapping
- Data available on ACK
  - (no stalling for ACKACK)
- Reader-lock unlocked on last ACKACK
- K1 / V1 fully cached
- All future GETs 150ns



Example: PUT new Model

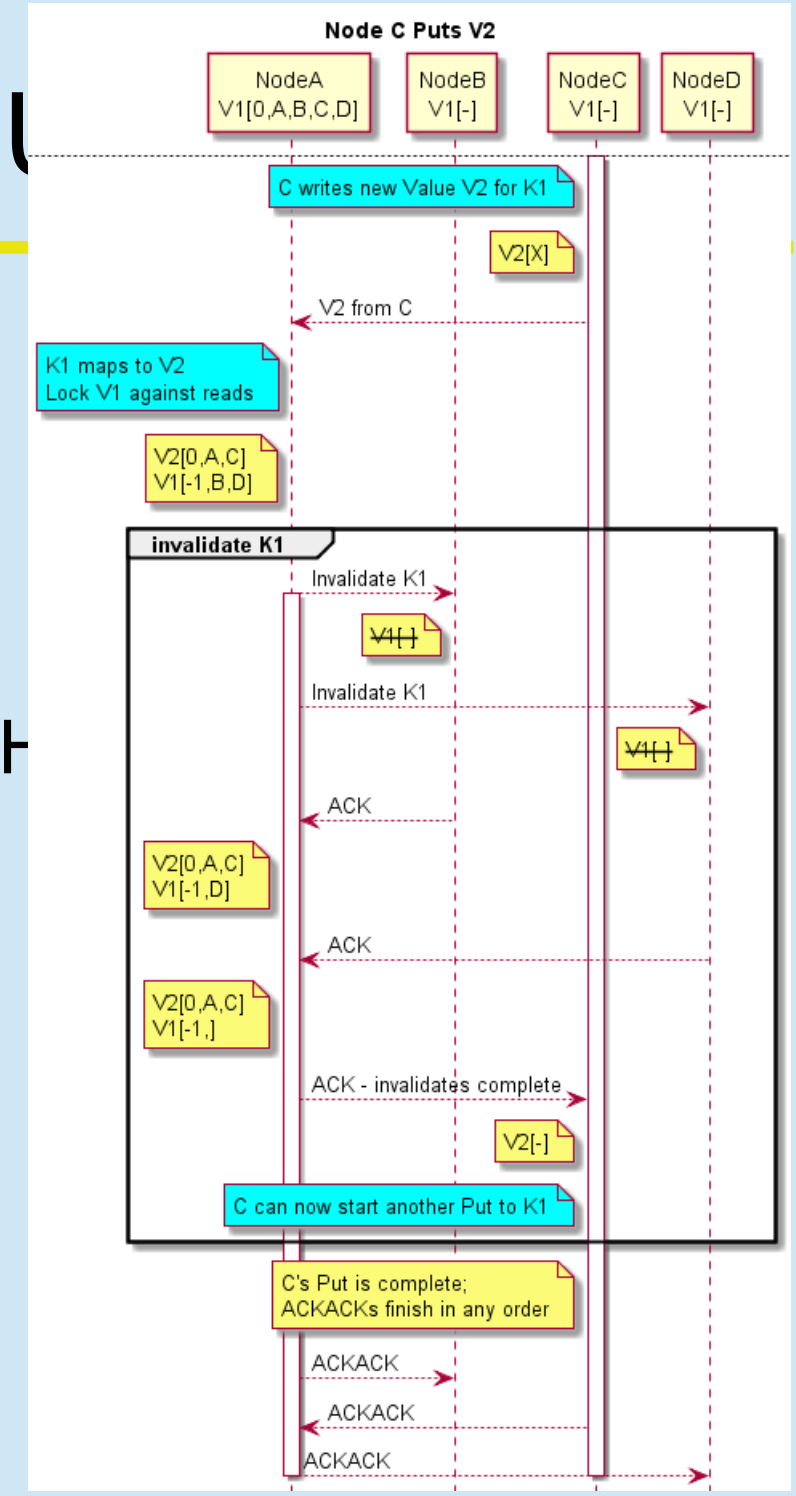
- Node C does PUT(K1, **V2**)
  - Cache local, write to A
- Caching local: GET after PUT is fast
- Write to A: Cluster “eventually” hears about PUT
- If non-volatile then non-blocking on C
  - No need for C to hear back from A
- If volatile then C will block until A responds
  - And A needs to do a cluster-wide invalidate

Example: PUT new Model

- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2 (up from V1)
  - atomic update in NonBlockingHashMap
  - Locks V1 against GETs
  - Sends invalidates to B & D
  - Awaits invalidate ACKs
  - ACKs back to C
- C's PUT completes
  - further ACKACKs are lazy

Example: PU

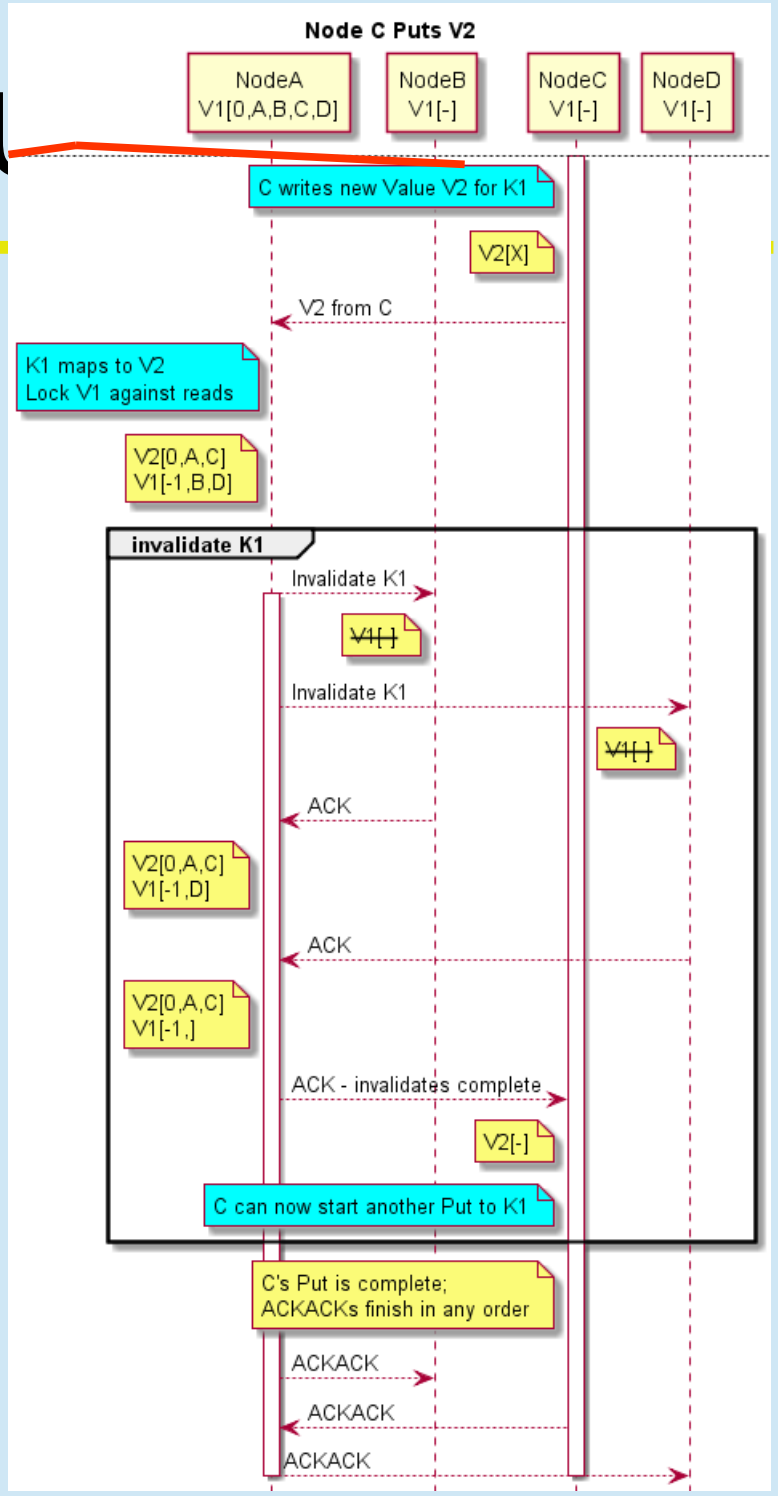
- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH
  - Locks V1 against GETs
  - Sends invalidates to B & D
  - Awaits invalidate ACKs
  - ACKs back to C
- C's PUT completes
  - further ACKACKs are lazy



E

C writes new Model
C Puts(K1,**V2**) local
V2 marked as pending

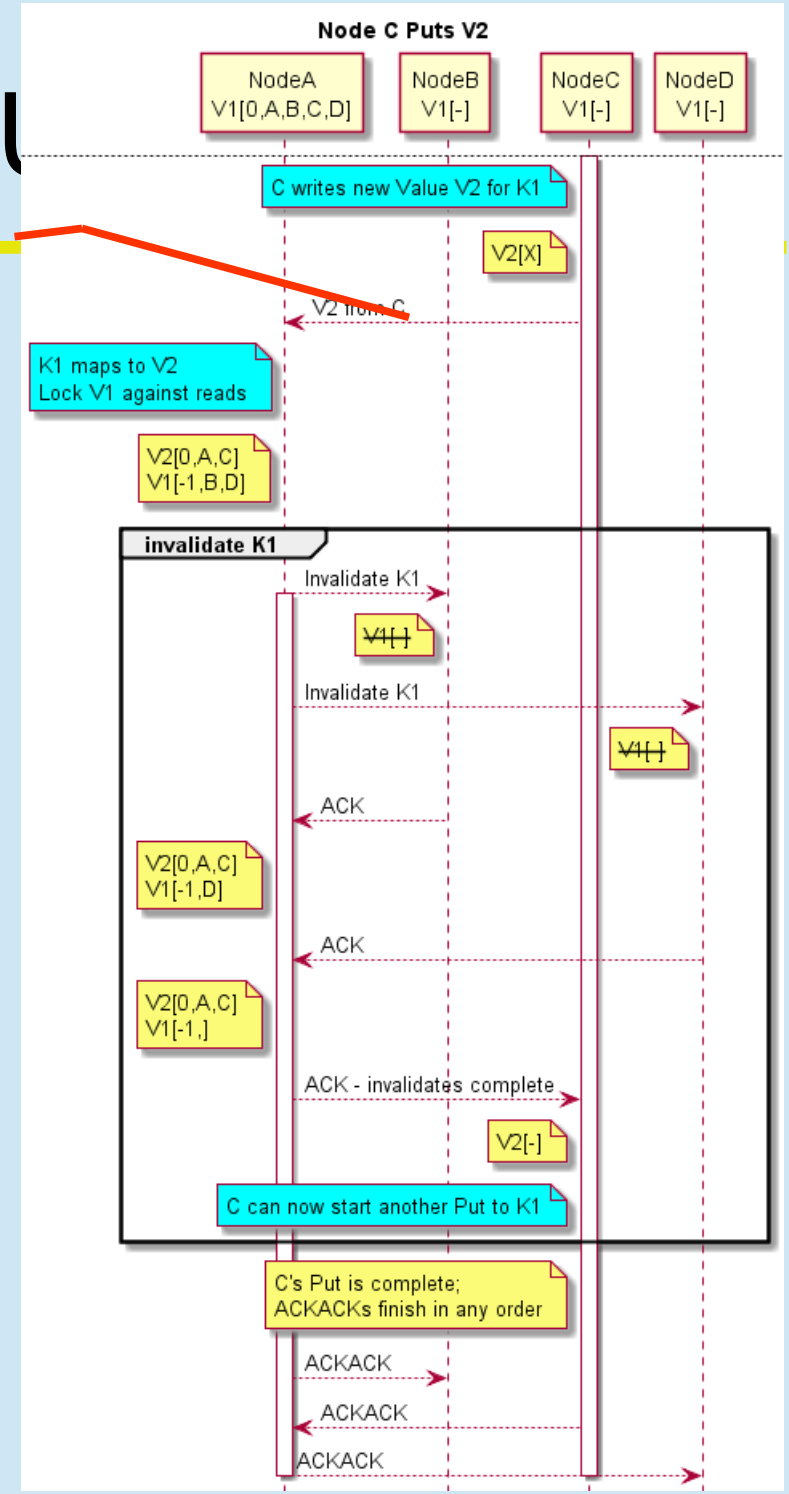
- Node C does PUT(K1, V2)



Example: PU

- Node C does PUT(K1, V2)
  - Cache local, write to A

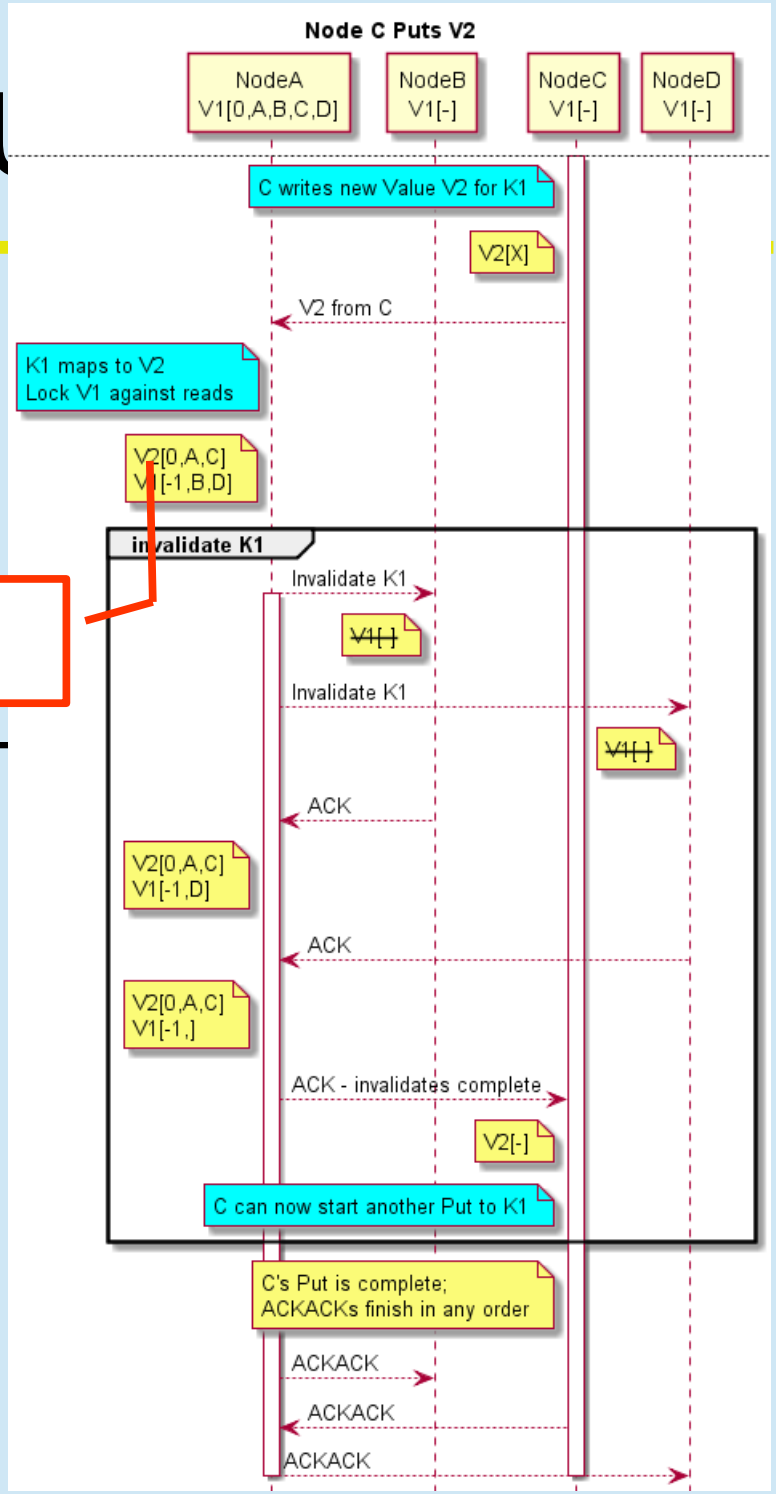
C sends V2 to A
Small: UDP
Large: TCP



Example: PU

- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH

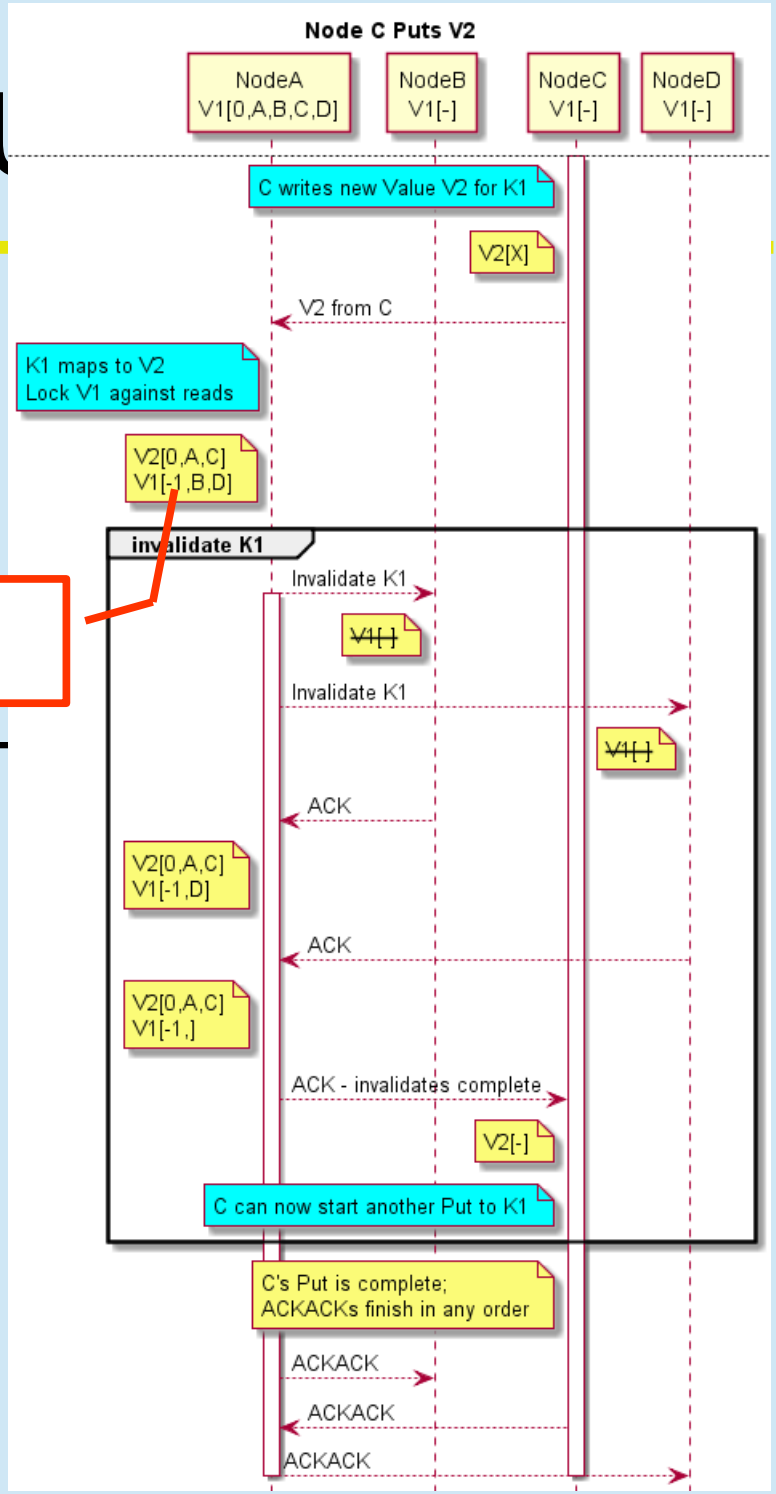
A installs V2 local



Example: PU

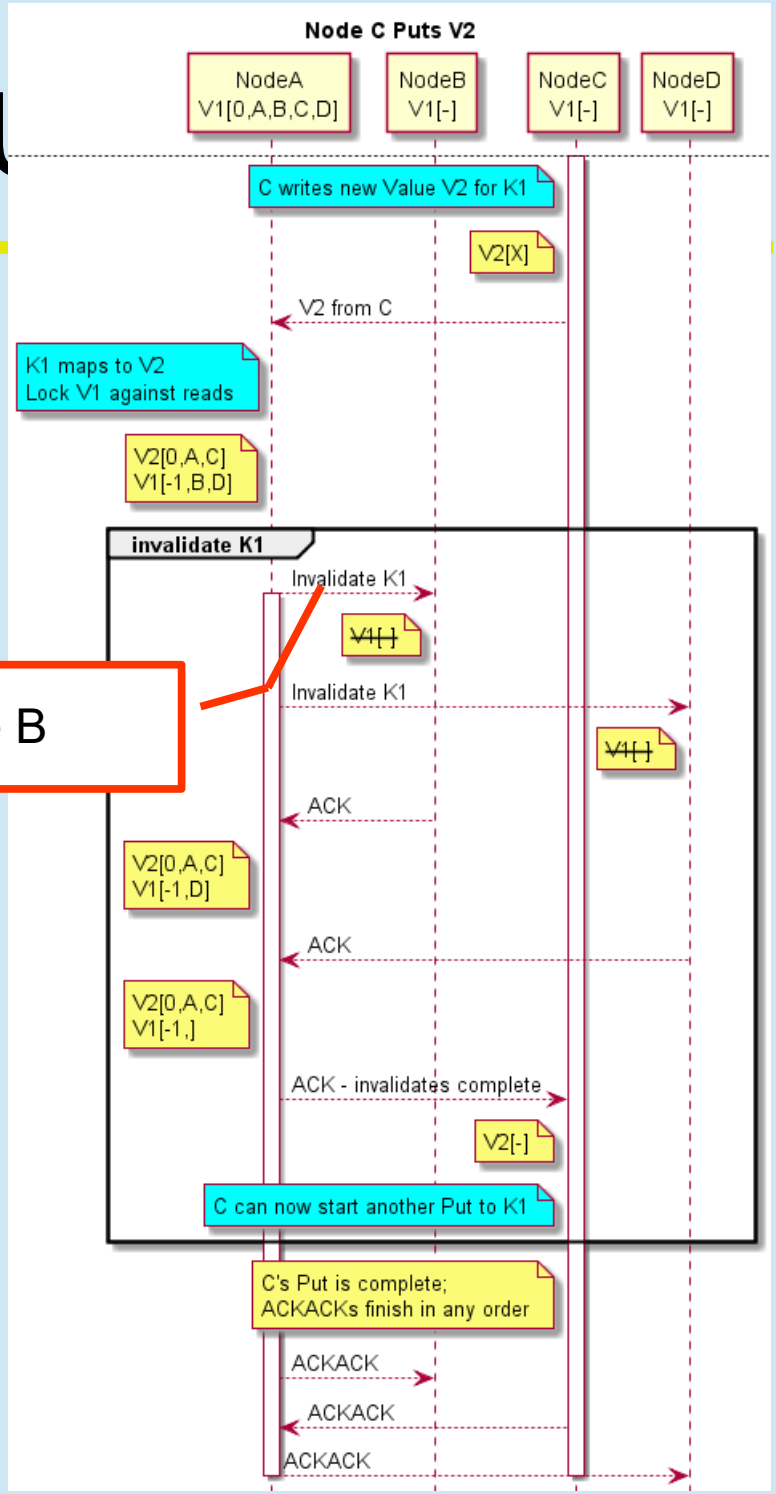
- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH
  - Locks V1 against GETs

Lock V1



Example: PU

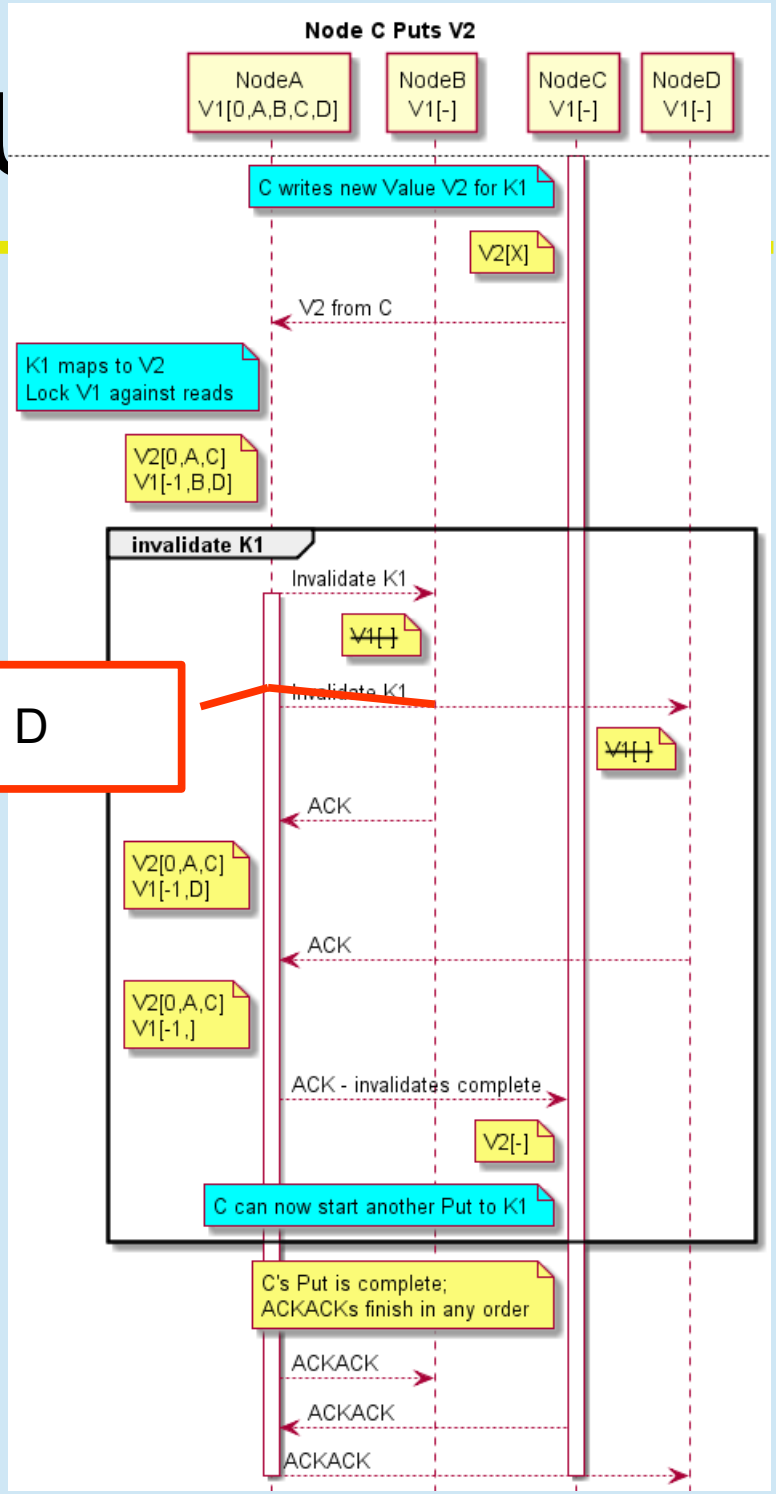
- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in No
  - Locks V1 against GETs
  - Sends invalidates to B & D



Invalidate B

Example: PU

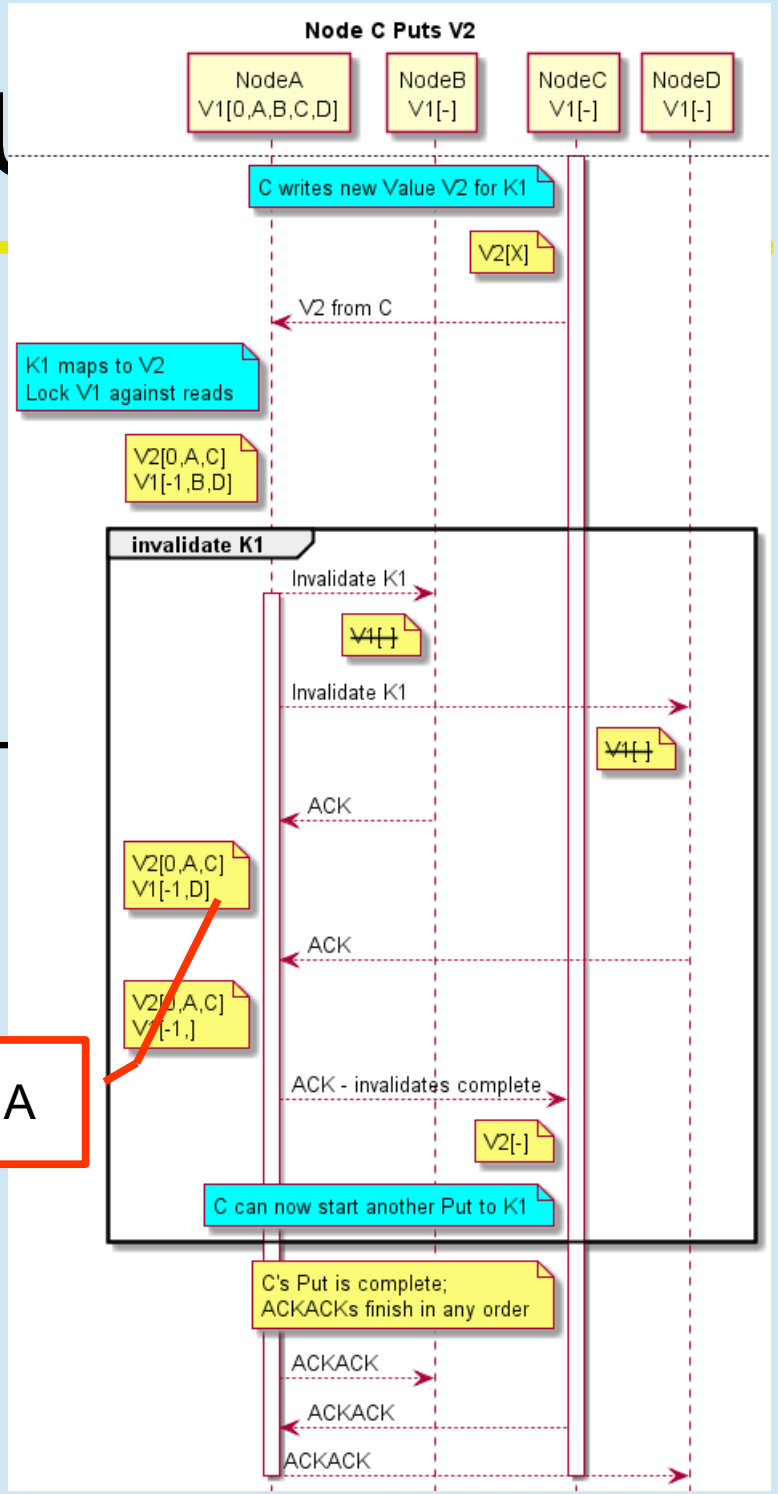
- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in No
  - Locks V1 against GETs
  - Sends invalidates to B & D



Invalidate D

Example: PU

- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH
  - Locks V1 against GETs
  - Sends invalidates to B & D
  - Awaits invalidate ACKs

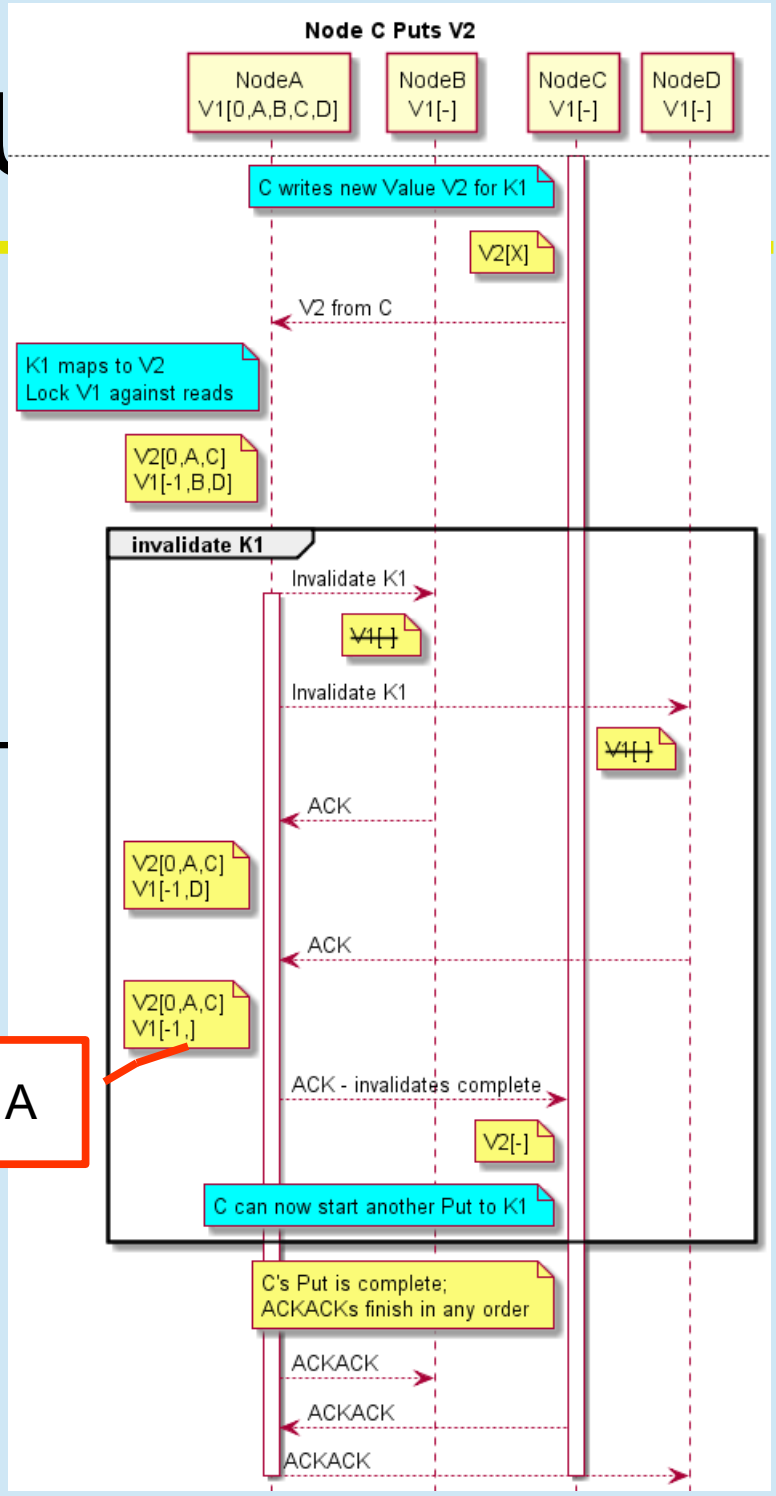


B ACKs A

Example: PU

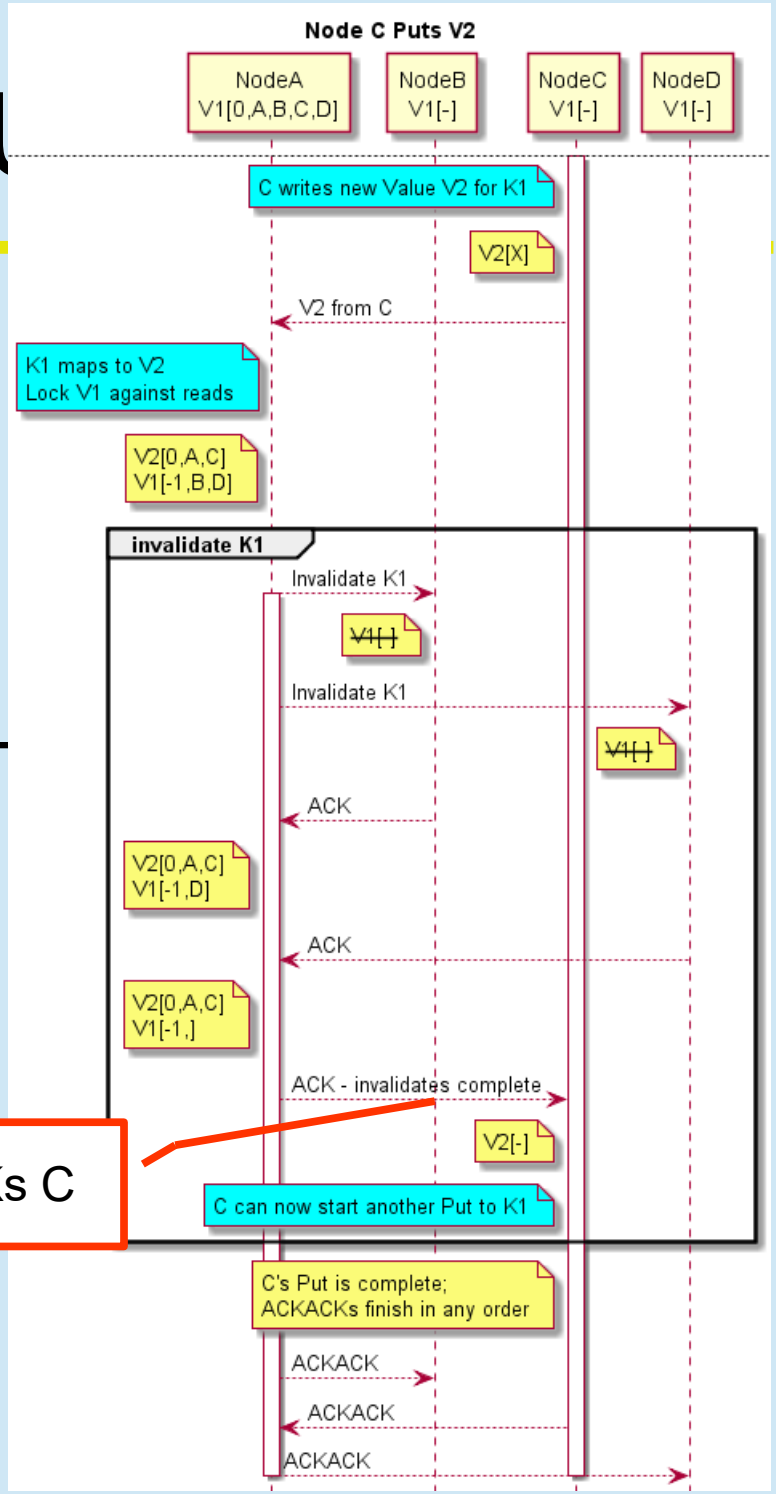
- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH
  - Locks V1 against GETs
  - Sends invalidates to B & D
  - Awaits invalidate ACKs

D ACKs A



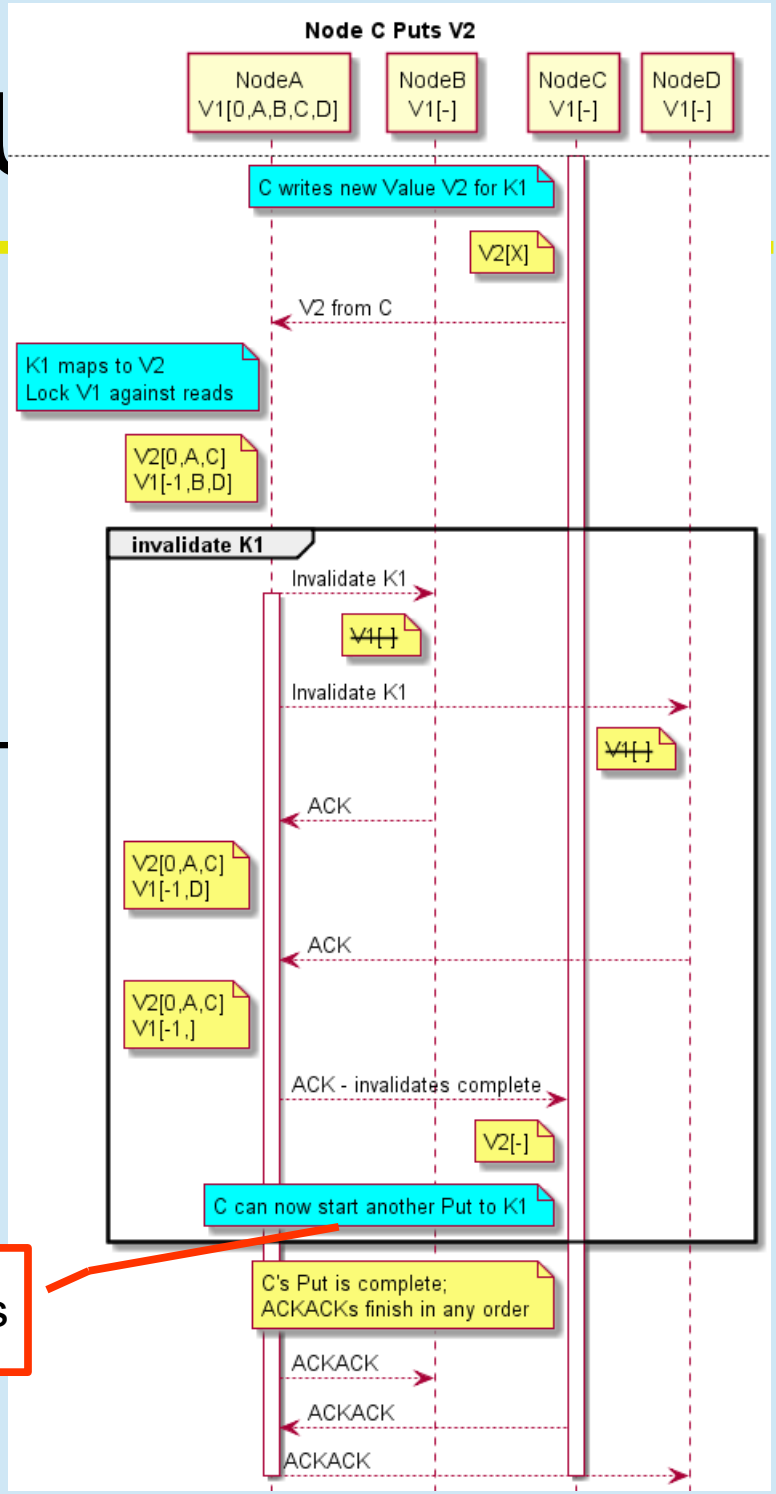
Example: PU

- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH
  - Locks V1 against GETs
  - Sends invalidates to B & D
  - Awaits invalidate ACKs
  - ACKs back to C



Example: PU

- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH
  - Locks V1 against GETs
  - Sends invalidates to B & D
  - Awaits invalidate ACKs
  - ACKs back to C
- C's PUT completes

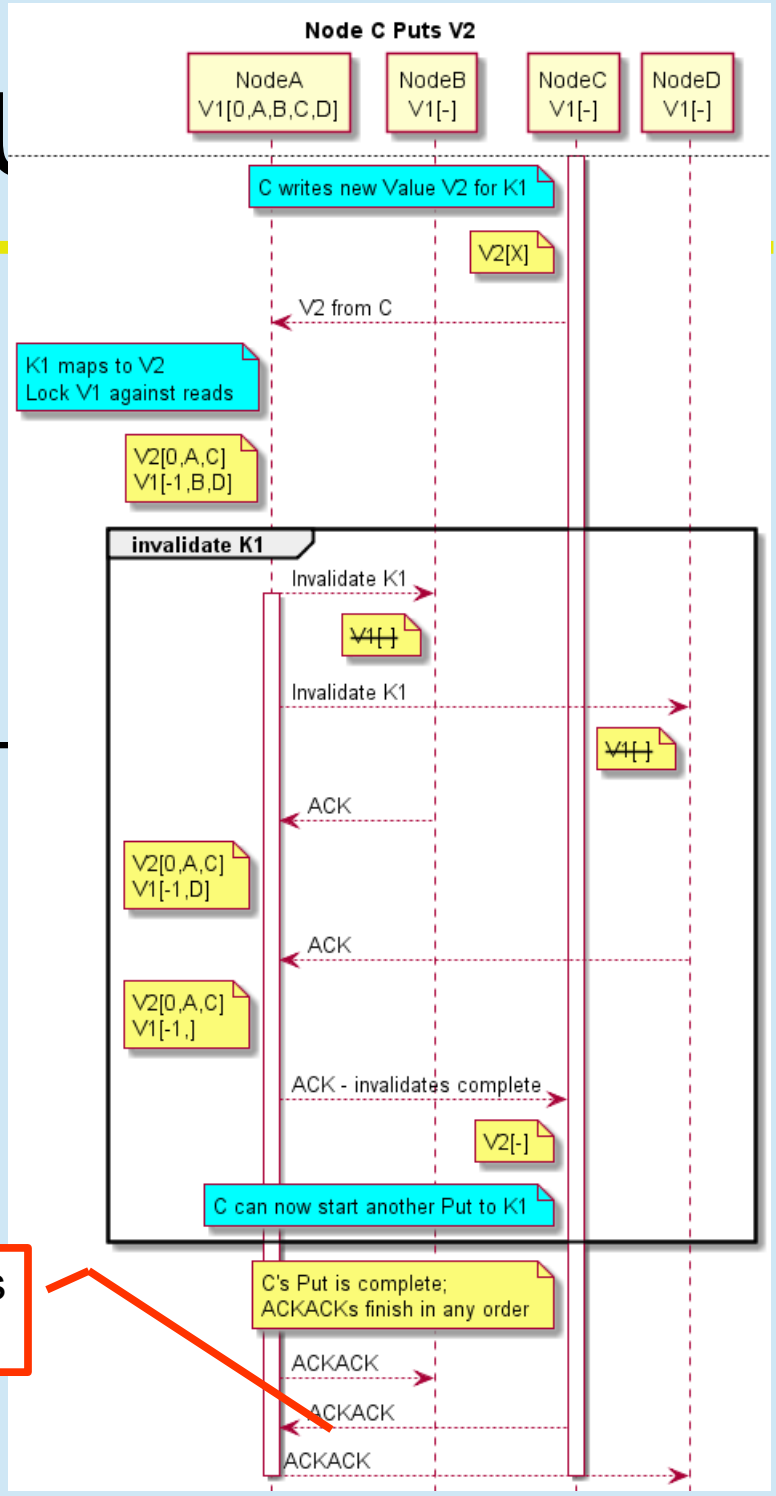


PUT completes

Example: PU

- Node C does PUT(K1, V2)
  - Cache local, write to A
- A maps K1 to V2
  - atomic update in NonBlockingH
  - Locks V1 against GETs
  - Sends invalidates to B & D
  - Awaits invalidate ACKs
  - ACKs back to C
- C's PUT completes
  - further ACKACKs are lazy

Every ACK gets an ACKACK



Key (yes pun) Takeaways

- All Keys can cache locally:
  - **150ns for repeated access**
  - Cache invalidated on a Write
  - Forever good & correct until invalidated
- Bulk cluster-wide reads of a single Key
  - Happen in any order
  - Do not block each other
  - Limits of network bandwidth, not latency
  - No blocking once data is received
    - ACKACKs happen in parallel

Key (yes pun) Takeaways

- PUTs do not stall writer unless volatile
- Writer can always read just-PUT key
- Bulk PUTs to unrelated Keys all in parallel
  - **Fast to bulk-transfer Big Data**
  - Limits of network bandwidth, not latency
- Repeated writes same key are **ordered**
  - Home key breaks ties, determines order
- **Never inconsistent**
  - Racing writes locally see their own write for awhile
  - But this is not inconsistent, since racing writes

Key (yes pun) Takeaways

- All missed GETs wait for round-trip to Home
- Volatile PUTs wait for round-trip to Home
  - Further wait for outstanding GETs to settle
    - Worst case:
      Send from writer to Home,
      Home finishes large GET to 3rd party
      Home sends invalidates to all caching nodes
      Home waits on invalidate-ACKs
      Home sends ACK to writer.
  - Price paid only when rapidly both reading & writing same Key

Key (yes pun) Takeaways

- Cost Model:
  - Repeated reads same Key cache: 150ns
  - Bulk reads: network bandwidth not latency
  - Bulk writes: network bandwidth not latency
  - Racing reader/writer: same as Bulk + Caching
  - Volatile reader: same as normal reader
  - Volatile writer: round-trip + cluster invalidate
- Data always available upon receipt
  - But maybe ACKs and ACKACKs run in background

Single-Key Atomic Transactions

- Home can execute Atomic Transactions
  - Code shipped to Home
  - Transaction executes on Home
  - Atomic update-or-fail on single Key
    - (really putIfMatch on NonBlockingHashMap)
  - Local retry on fail
  - Same ordering as volatile GET/PUT

H2O is -

- Clustered In-Memory Computing
- Clustered Data – Columnar Compressed
- Fine-grained data parallelism via Map / Reduce
  - Single-threaded code running parallel and distributed
- High Speed **Exact** Consistency Java Memory Model
- Tight integration into R, Python, Scala, Java, Web
- Best-of-Breed Math – done distributed
- A Systems' Platform Builder's Delight!

Q & A

Cliff Click