Verifying Cache Coherence in ACL2

Ben Selfridge
Oracle / UT Austin


30
6

Goals of this talk

- Define cache coherence
- Present a cache coherence protocol I designed
- Present an ACL2 proof that the protocol is "safe" (whatever that means)
- Discuss how we might use ACL2 to verify more complicated protocols
    - Is it worth it? (Why not just use a model checker?)
    - Is it possible? (Inductive invariants are hard…)

Goals of this talk

- Define cache coherence
- Present a cache coherence protocol I designed
- Present an ACL2 proof that the protocol is “safe” (whatever that means)
- Discuss how we might use ACL2 to verify more complicated protocols
    - Is it worth it? (Why not just use a model checker?)
    - Is it possible? (Inductive invariants are hard…)

What are caches?

- Small, quick-access memory on a chip
- Used for repeated accesses to the same locations
- To read/write, the processor must obtain the line from memory, copy it to cache
- When it’s done writing, the cache line is copied back to main memory (at some point)

Cache Coherence

- With 2+ processors, this gets complicated
- We can allow multiple processors to *read* simultaneously
- But write simultaneously? Hmm…

Cache Coherence

Cache Coherence

Cache Coherence

Cache Coherence

Both caches believe they have up-to-date copies of the memory location, but they see different values!

Cache Coherence

- To prevent this from happening, add state to each cache line (invalid, read-only, read-write)
- Two in read-only? **Allowed**
- Two in read-write? **Not Allowed**
- One in read-only, one in read-write? **Not Allowed**
- These guarantees are commonly called **cache coherence**.

Cache Coherence Protocols

- To ensure coherence, **cache coherence protocols** are used to manage the state across caches
- Cores send messages to communicate
  - If I want a cache line, I must **request** it
  - If I hold a cache line, I must **send my data back** at some point
- Network properties vary widely between protocols
- Protocols must be designed VERY CAREFULLY to maintain coherence

Example Protocol: “VI”

- I designed a simple cache coherence protocol called VI.
- Cache lines can be in one of two states:
    - V = “valid” (read/write)
    - I = “invalid”
- There is no read-only state!

Example Protocol: “VI”

- There are n **caches** along with an additional agent, the **directory**, residing in the main memory
    - The directory keeps track of who currently “owns” each cache line (either a cache, or the main memory)
- For the remainder of this talk, assume there is **only one cache line** that can be shared between memory and caches. (This avoids confusion.)

Caveat: Dir "state" is interpreted differently

- The Directory has two states, I and V
- Dir in state I means "Dir has the data, and no one else does"
- Dir in state V means "Dir does not have the data; either someone else has it in state V, or it's currently in transit"
- Remember this, otherwise you'll get confused

VI Transition Tables

Cache Controller

	load/ store	evict	Data	Fwd-Get	Put-Ack
I	Send Get to Dir / IV^D	ILLEGAL	ILLEGAL	ILLEGAL	ILLEGAL
IV^D	stall	stall	Copy to cache / V	stall	
V	perform load/ store	Send Put to Dir / VI^A	ILLEGAL	Send Data to Req / I	ILLEGAL
VI^A	stall	stall	ILLEGAL	Send Data to Req / II^A	-/I
II^A	stall	stall	ILLEGAL	ILLEGAL	-/I

VI Transition Tables

Cache Controller

	load/ store	evict	Data	Fwd-Get	Put-Ack
I	Send Get to Dir / IV^D	ILLEGAL	ILLEGAL	ILLEGAL	ILLEGAL
IV^D	stall	stall	Copy to cache / V	stall	
V	perform load/ store	Send Put to Dir / VI^A	ILLEGAL	Send Data to Req / I	ILLEGAL
VI^A	stall	stall	ILLEGAL	Send Data to Req / II^A	-/I
II^A	stall	stall	ILLEGAL	ILLEGAL	-/I

“stable” states

VI Transition Tables

Cache Controller

	load/ store	evict	Data	Fwd-Get	Put-Ack
I	Send Get to Dir / IV^D	ILLEGAL	ILLEGAL	ILLEGAL	ILLEGAL
IV^D	stall	stall	Copy to cache / V	stall	
V	perform load/ store	Send Put to Dir / VI^A	ILLEGAL	Send Data to Req / I	ILLEGAL
VI^A	stall	stall	ILLEGAL	Send Data to Req / II^A	-/I
II^A	stall	stall	ILLEGAL	ILLEGAL	-/I

“transient” states

VI Transition Tables

Directory Controller

	Get	Put (from owner)	Put (from non-owner)
I (Cache line in main mem only)	Send Data to Req, set Owner to Req / V	ILLEGAL	Send Put-Ack to Req / -
V (some cache has cache line in state V)	Send Fwd-Get to Owner, set Owner to Req / V	Copy data to memory, send Put-Ack to Owner, clear Owner / I	Send Put-Ack to Req / -

Example Protocol: “VI”

Before we go any further, let’s take a look at how this protocol works in practice.

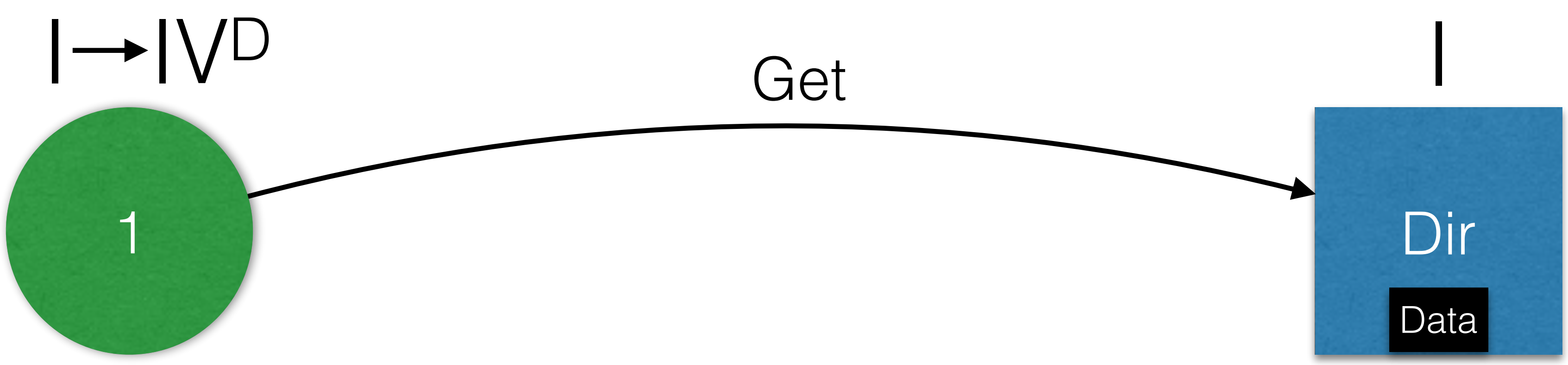
“Get” transaction

"Get" transaction

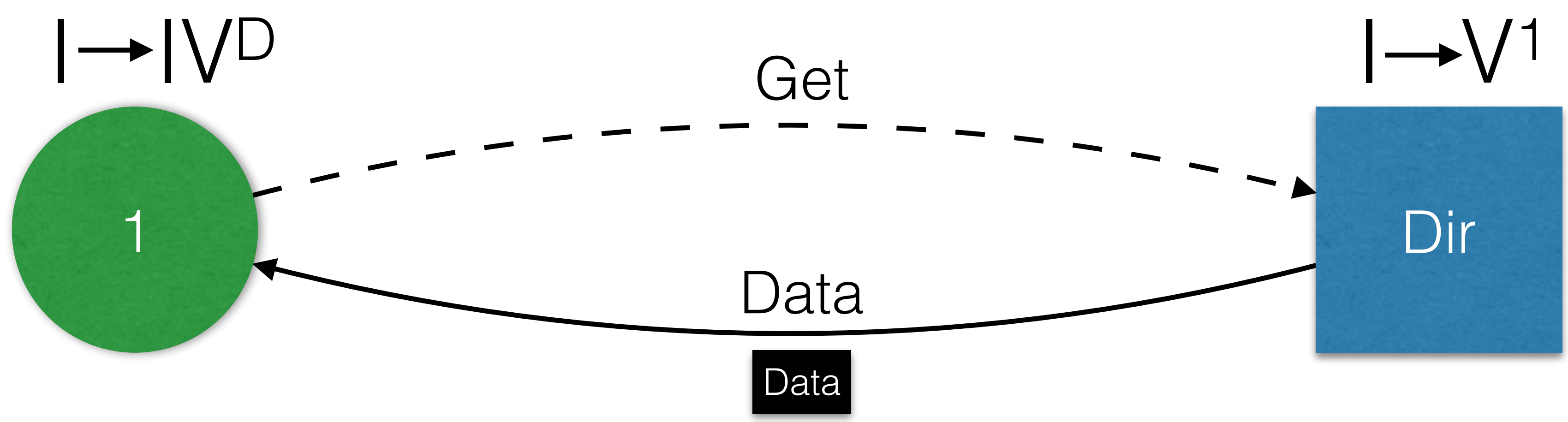
Cache 1 wishes to obtain the cache line. Dir is in state I, indicating no other cache currently has the data.

“Get” transaction

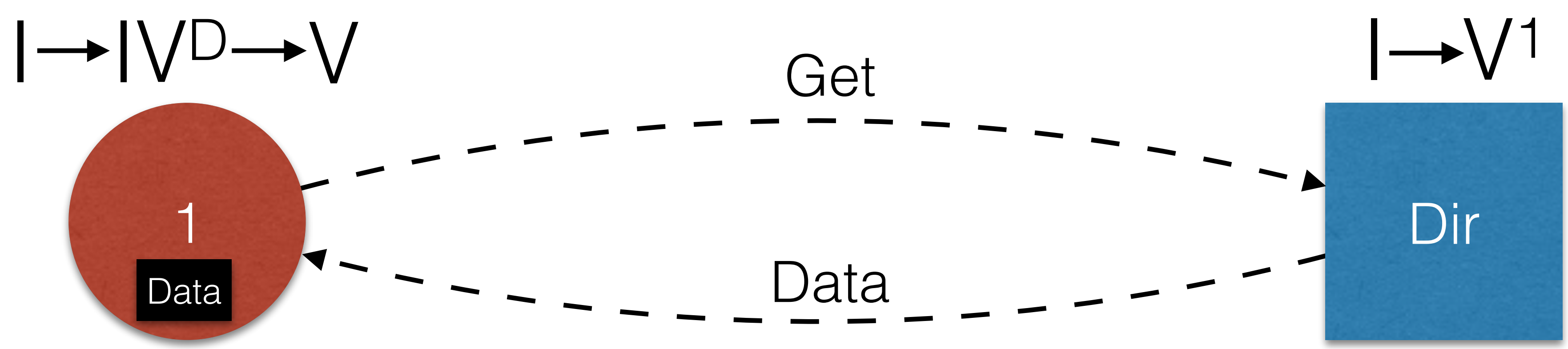
“Get” transaction



“Get” transaction



“Get” transaction



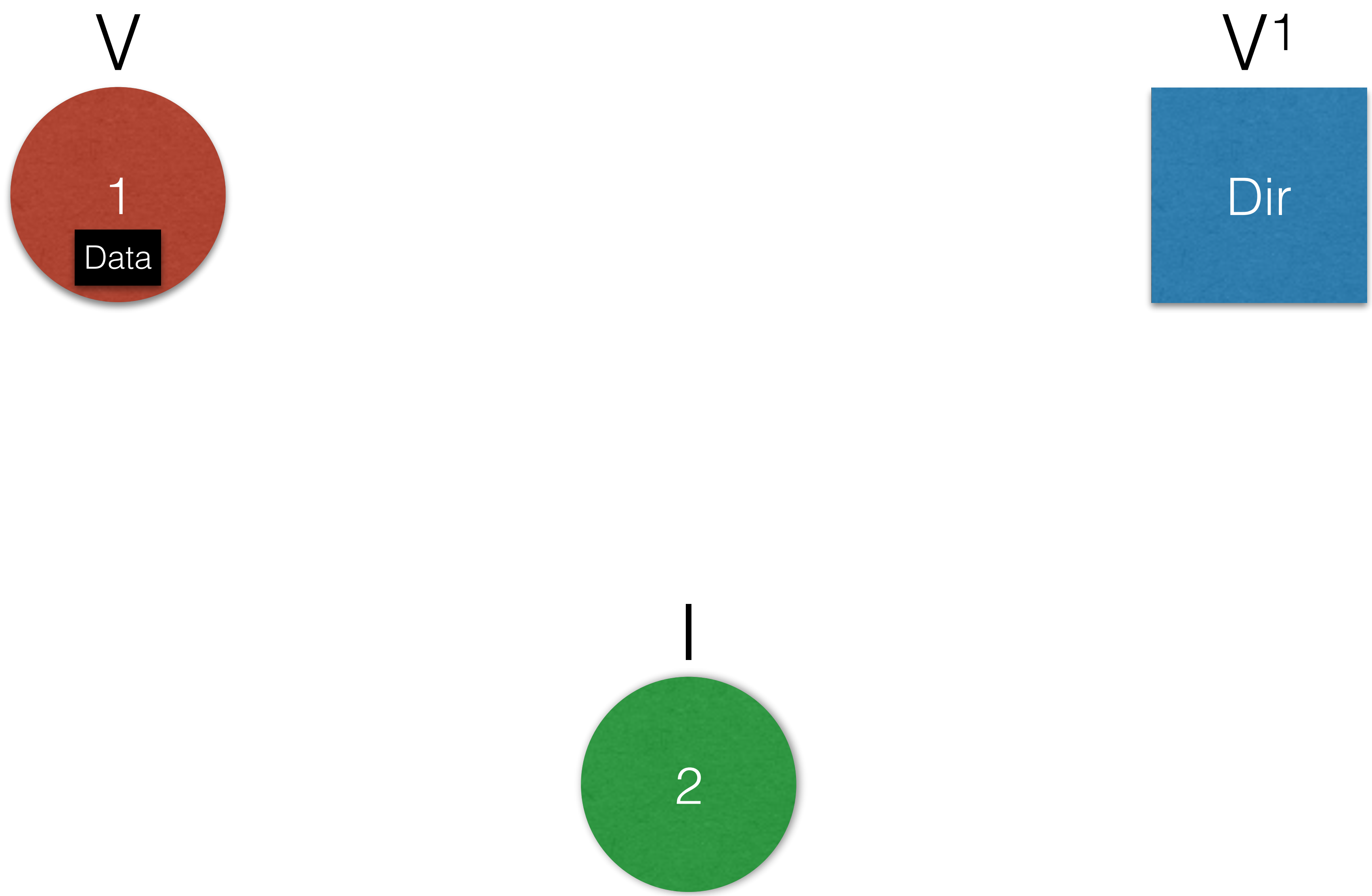
“Get” transaction

"Get" transaction

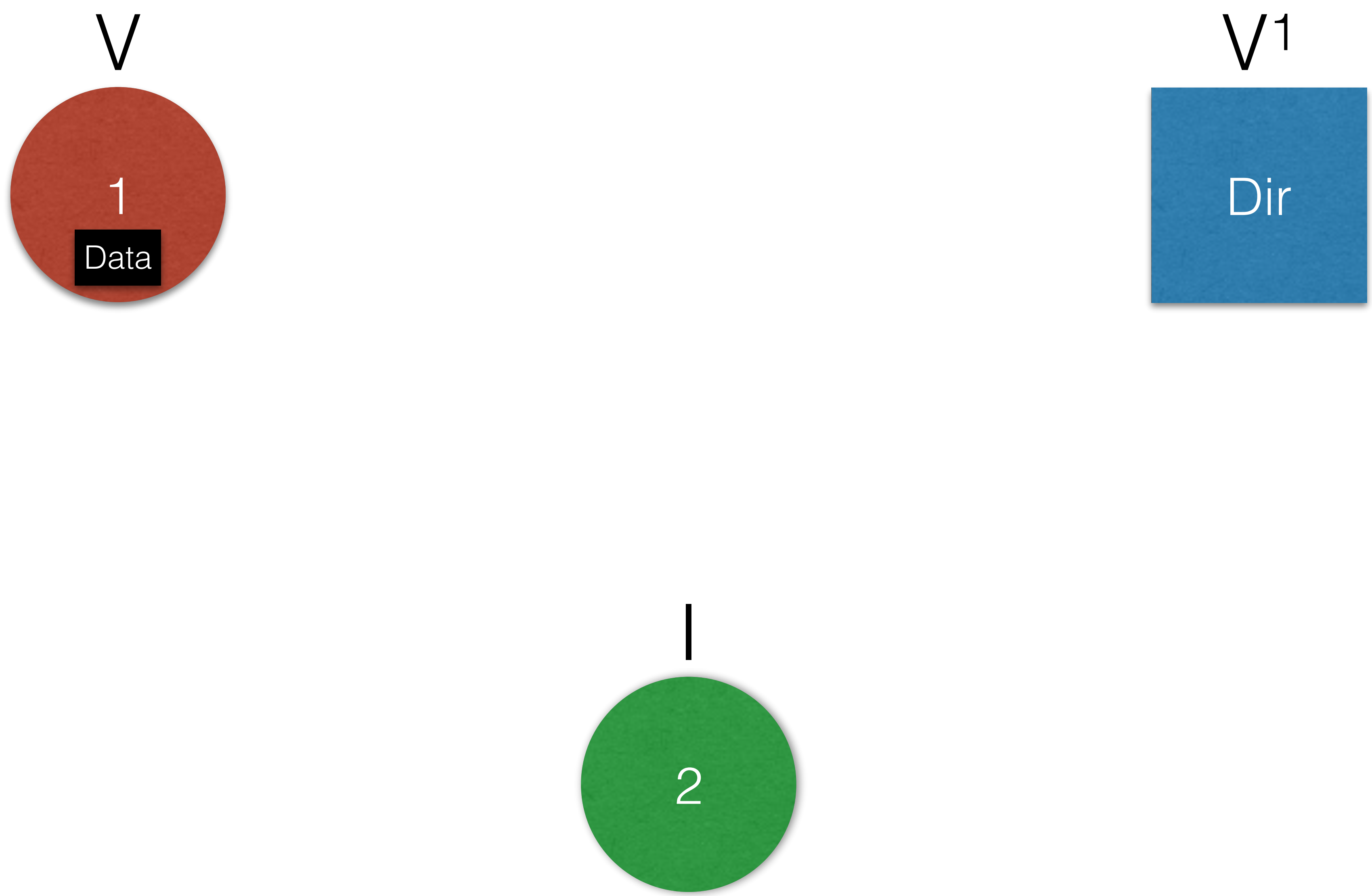
Cache 1 has successfully obtained the cache line.

“Get” transaction

“Get” transaction

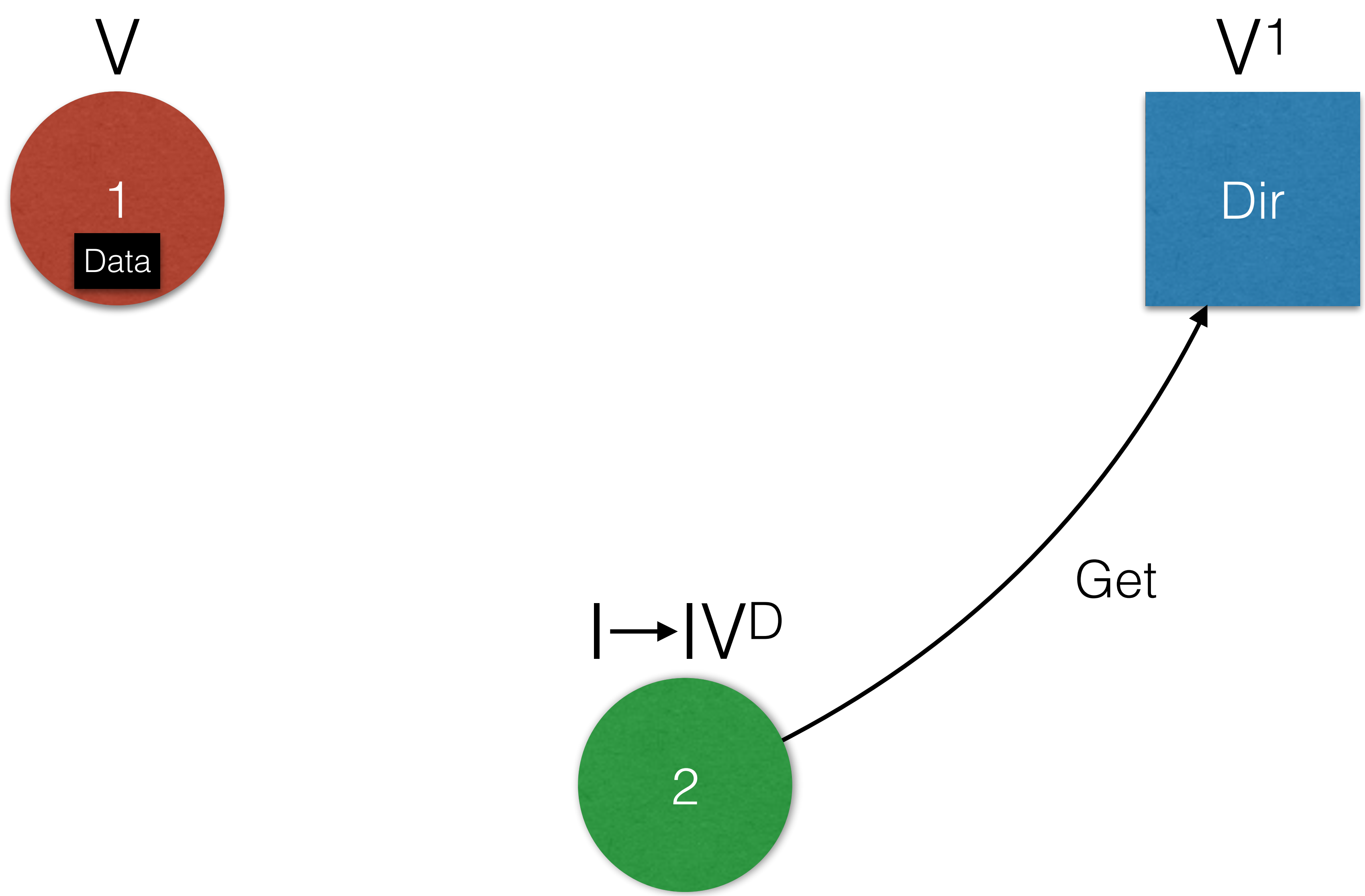


“Get” transaction

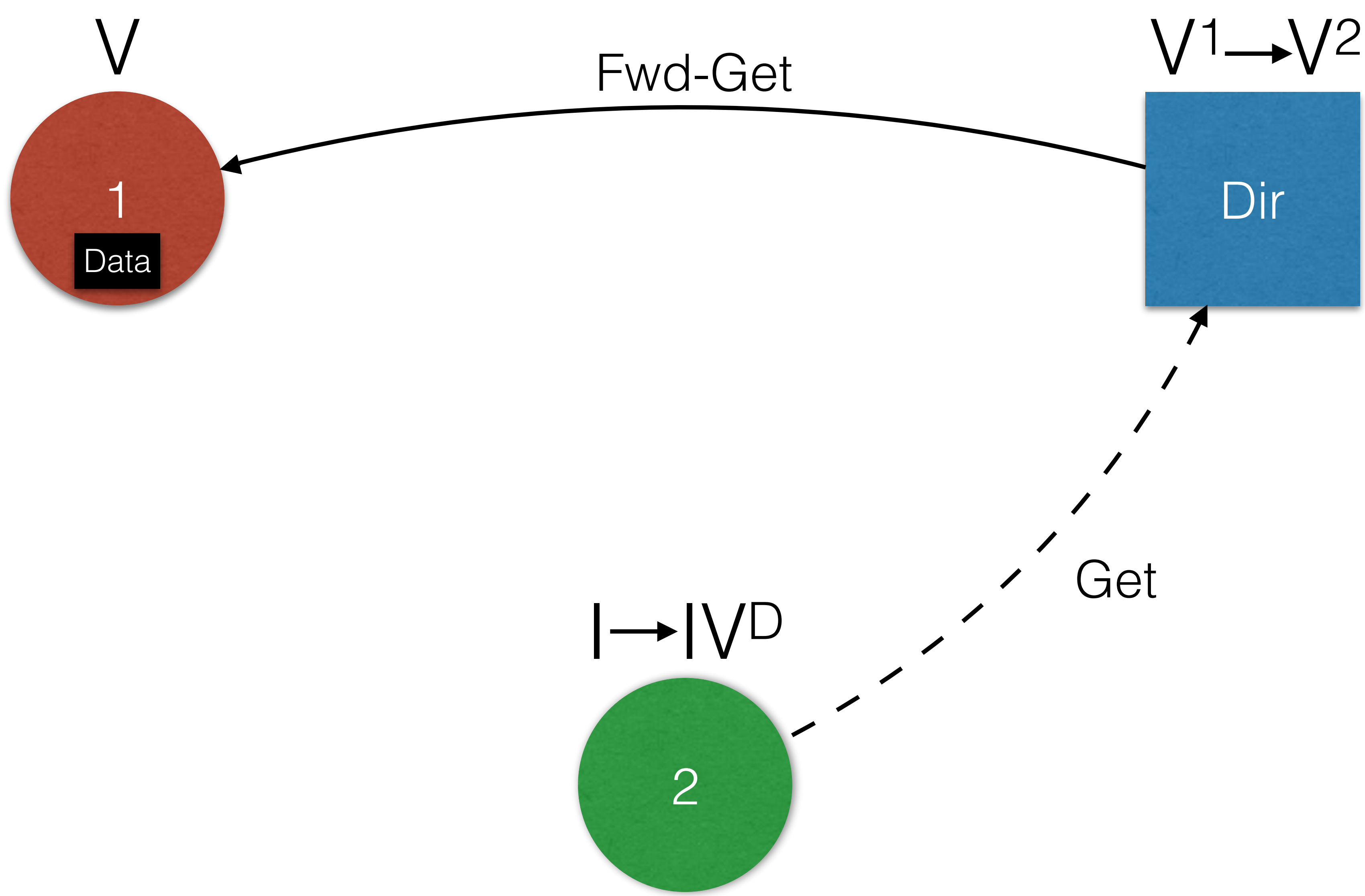


Cache 2 wants to obtain the cache line, and Cache 1 already has it.

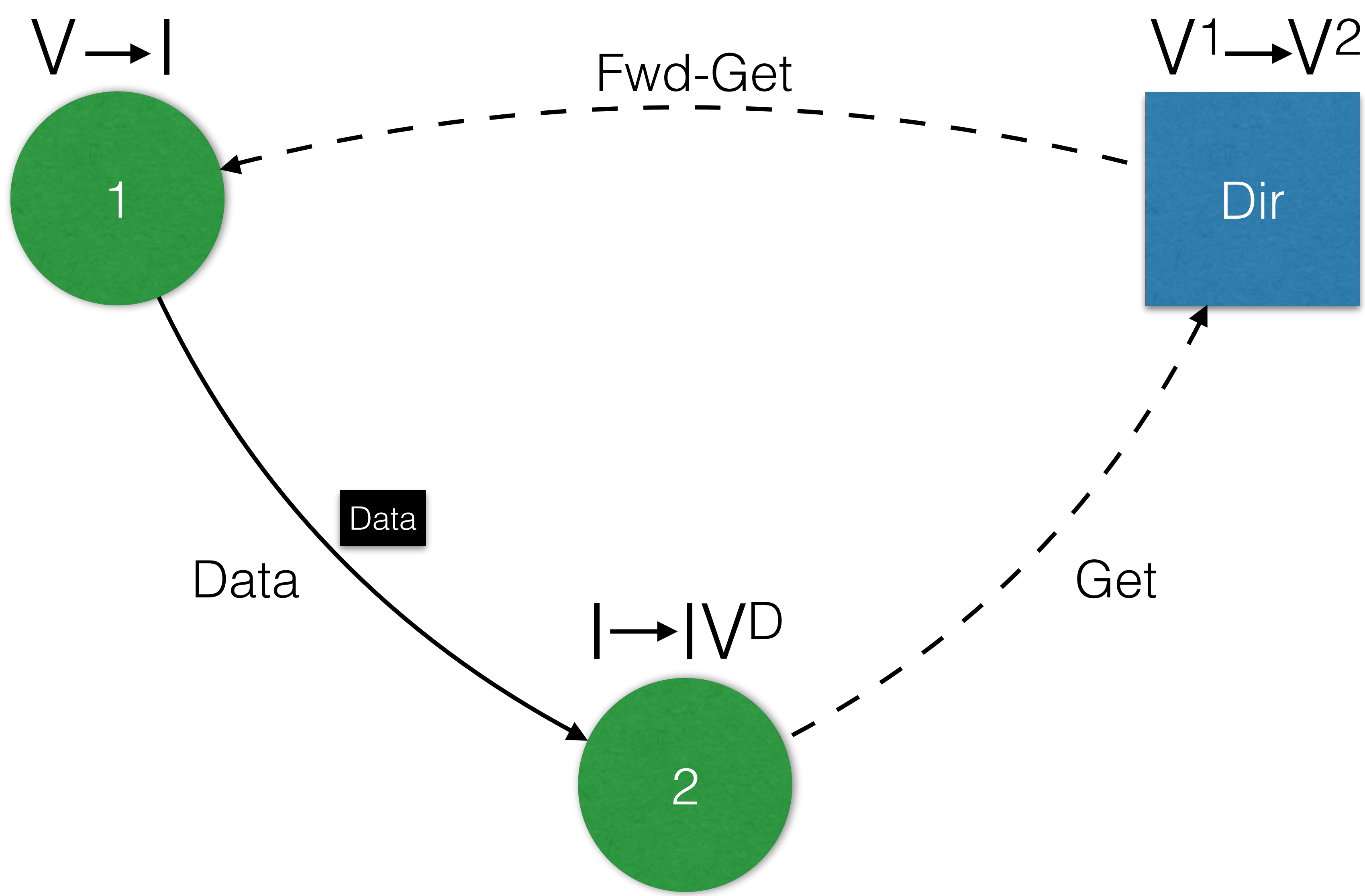
“Get” transaction



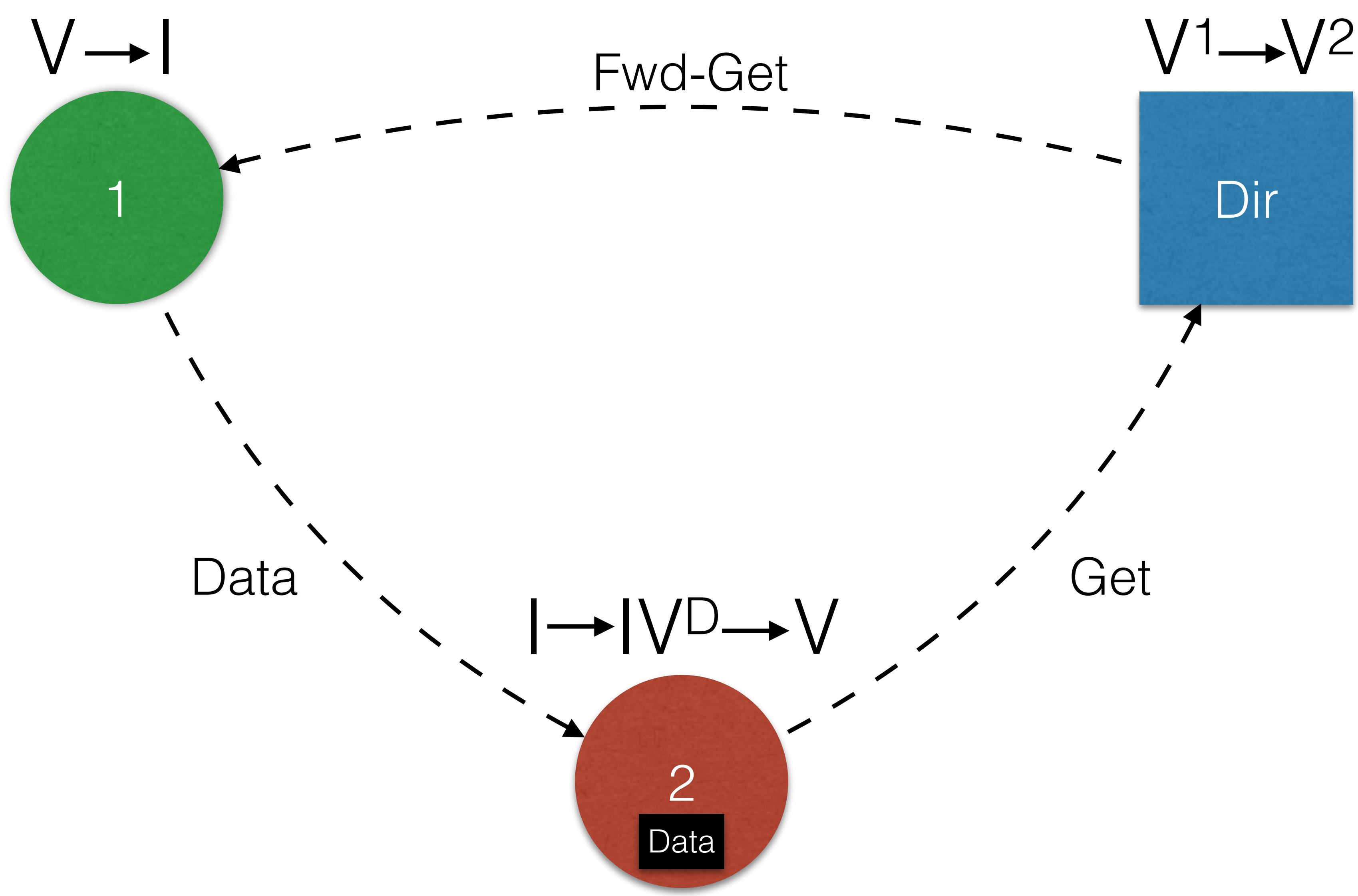
“Get” transaction



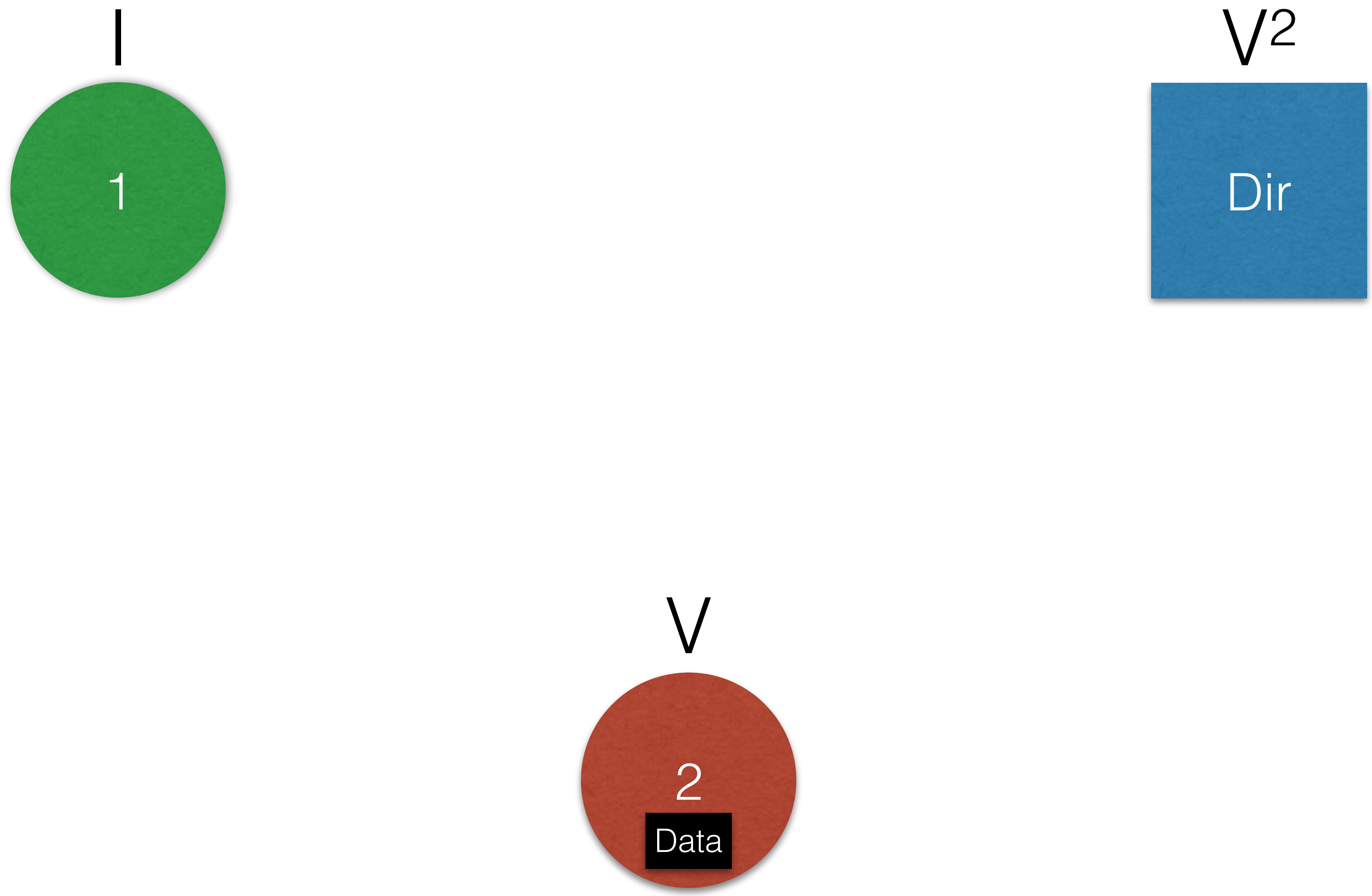
“Get” transaction



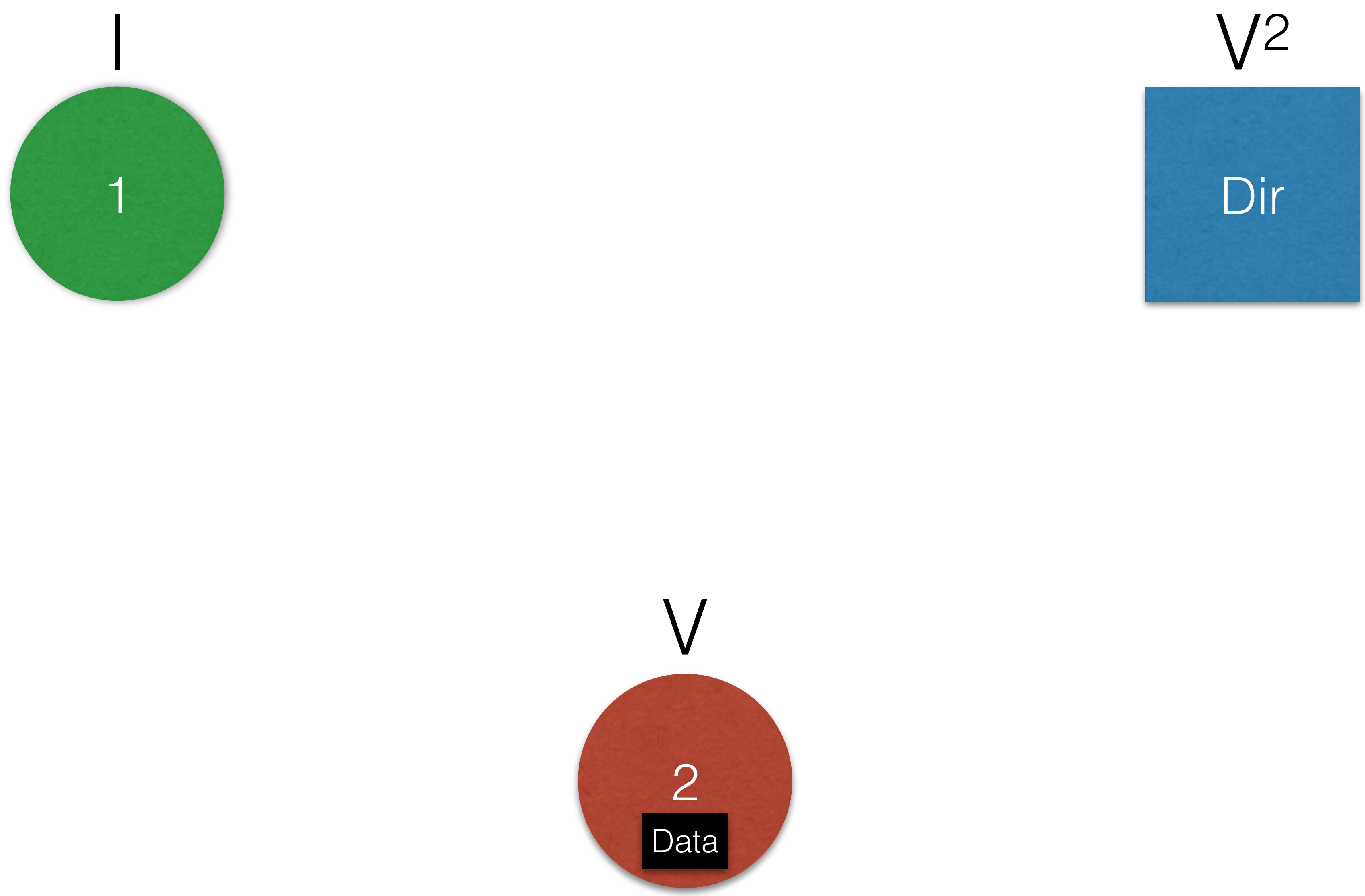
“Get” transaction



“Get” transaction



“Get” transaction



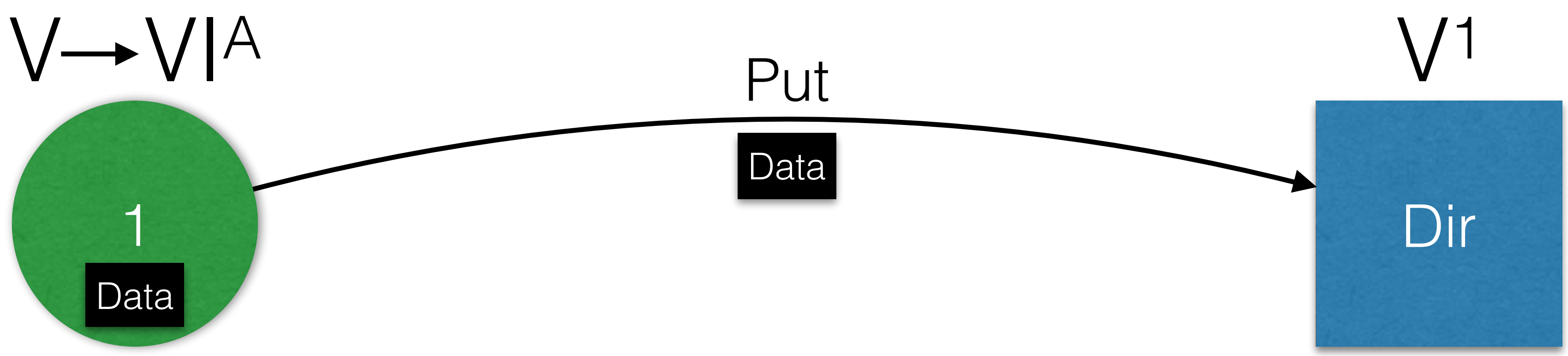
Cache 2 has successfully obtained the cache line.

“Put” transaction

“Put” transaction

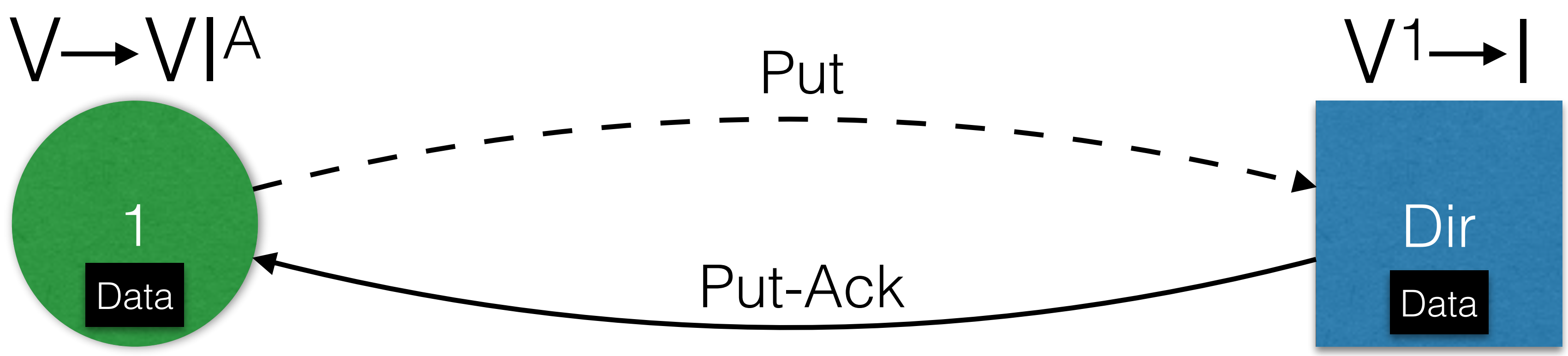
Cache 1 has the cache line, and wishes to evict (transition to I).

"Put" transaction



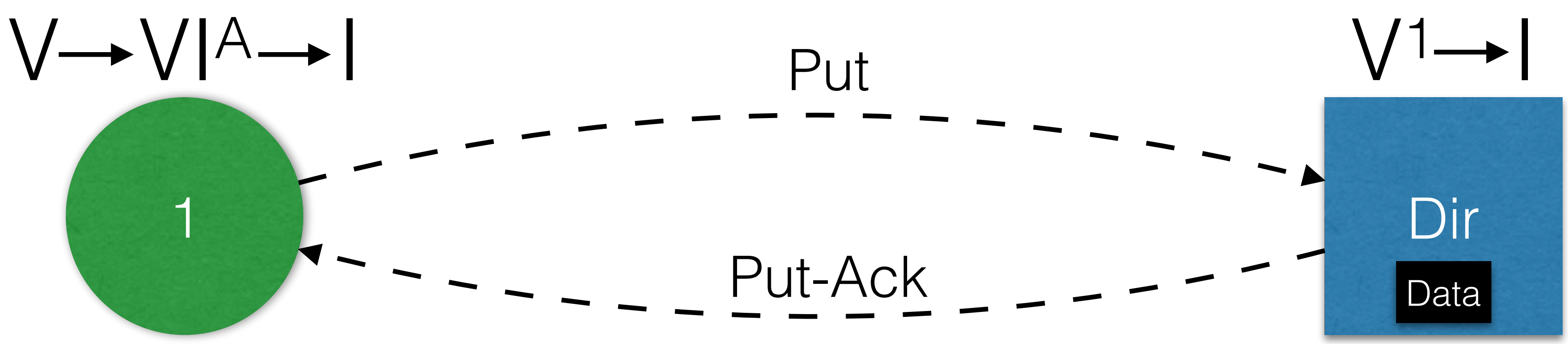
Cache 1 sends a Put to Dir, but does not evict the cache line yet.

"Put" transaction



When Cache 1 receives Put-Ack from Dir, it is safe to evict the cache line.

"Put" transaction

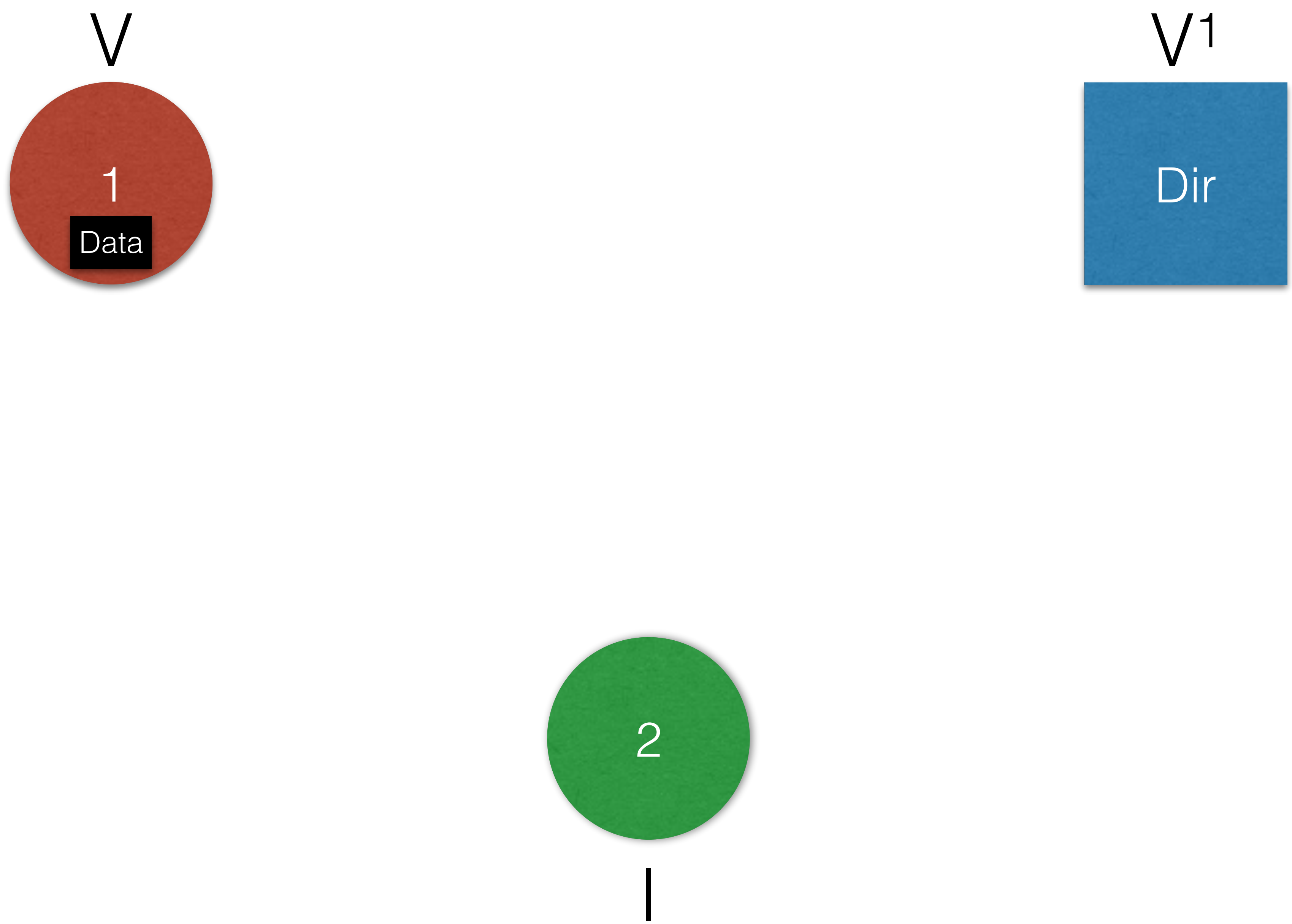


When Cache 1 receives Put-Ack from Dir,
it is safe to evict the cache line.

“Put” transaction

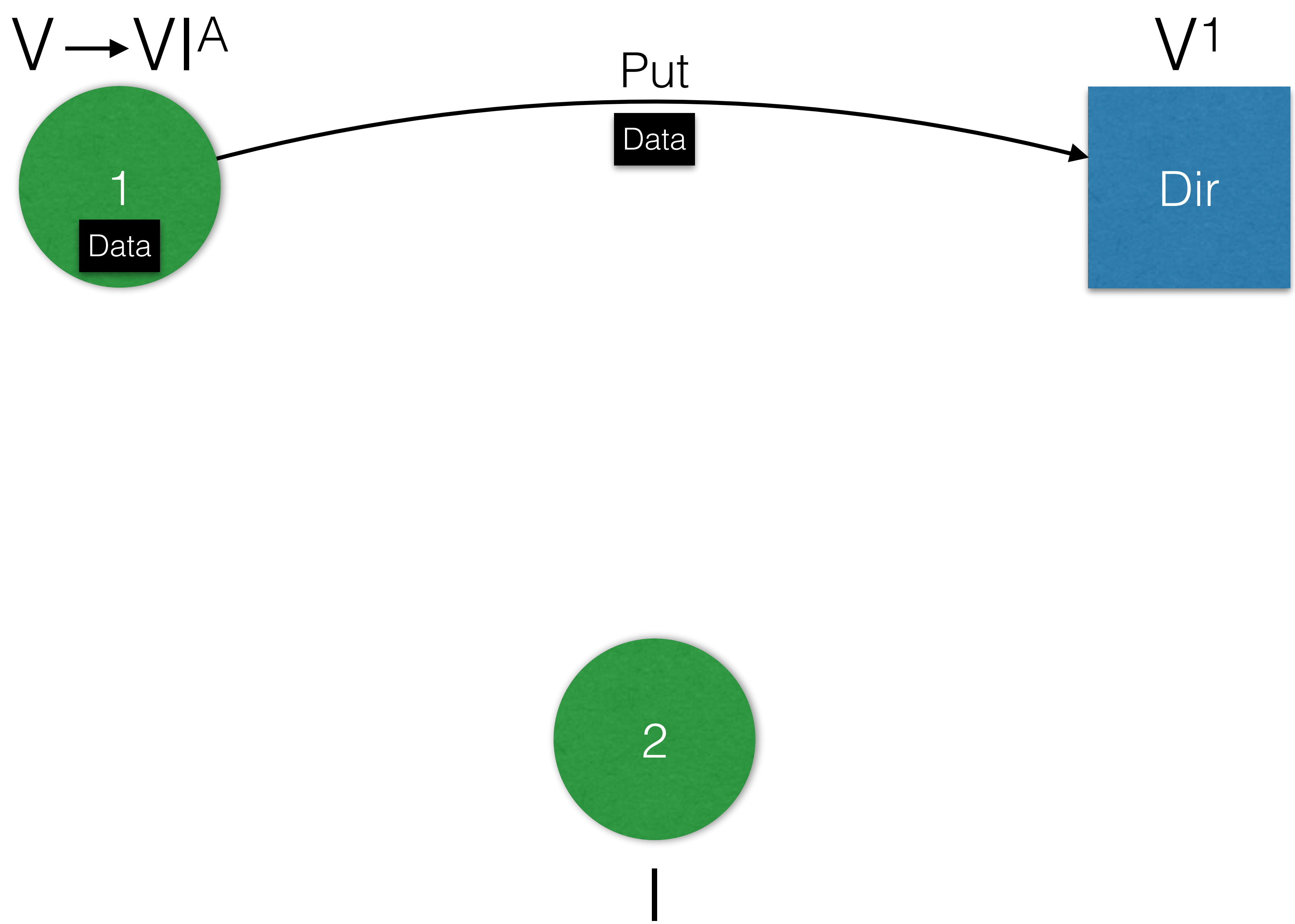
Cache 1 has successfully evicted,
and Dir now “owns” the data.

Put/Get race

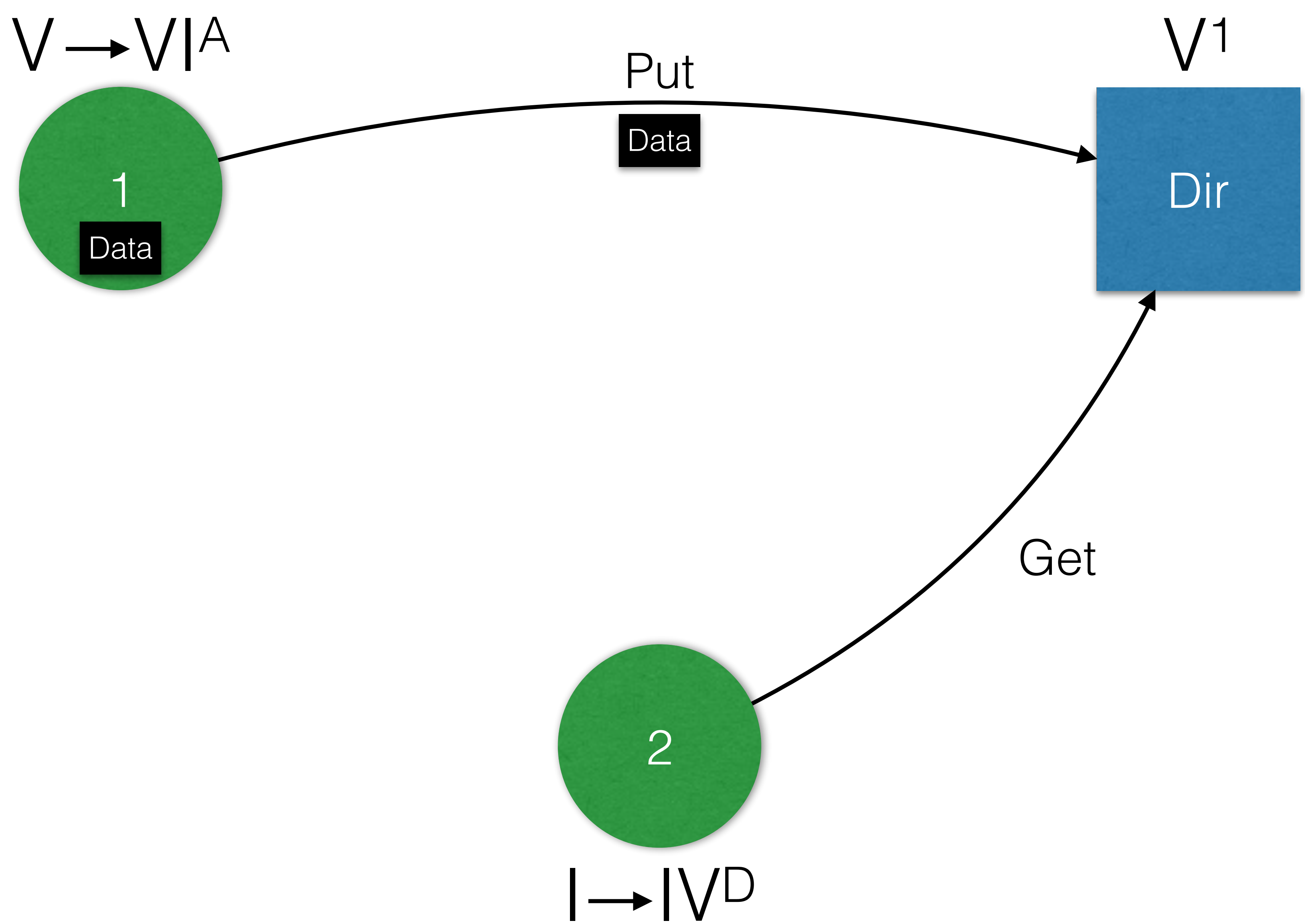


What if Cache 1 Puts, and Cache 2 Gets at the same time?

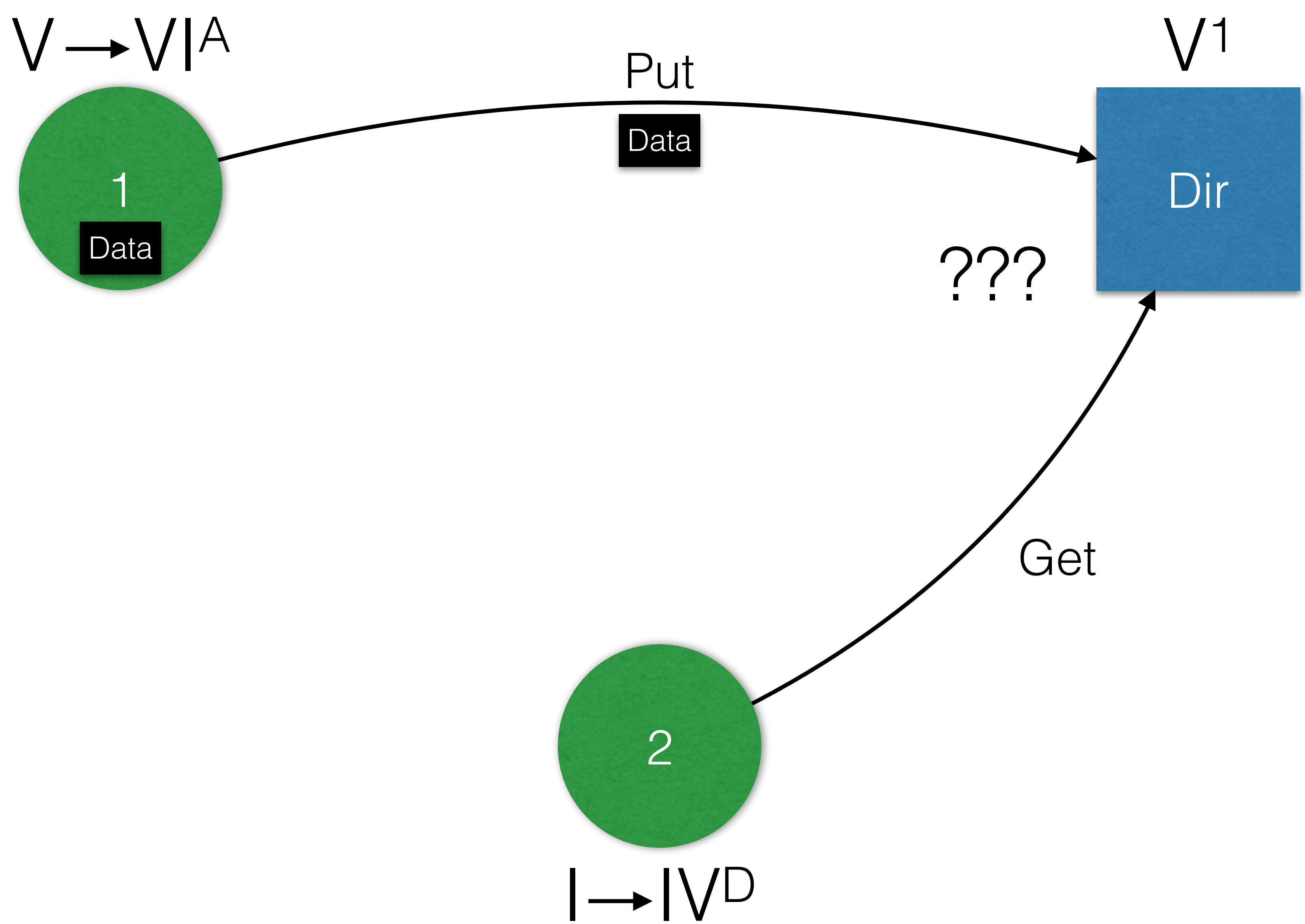
Put/Get race



Put/Get race

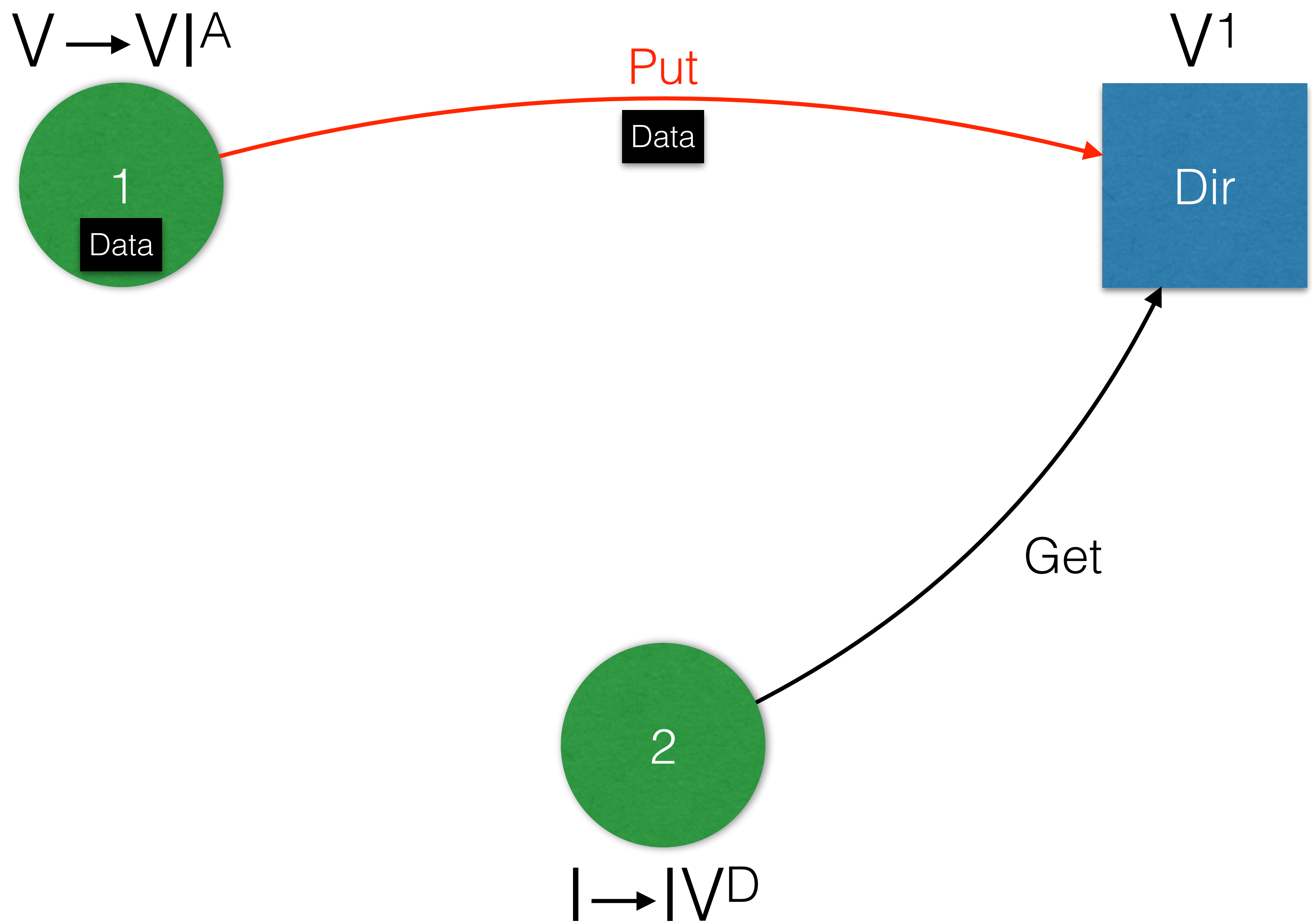


Put/Get race



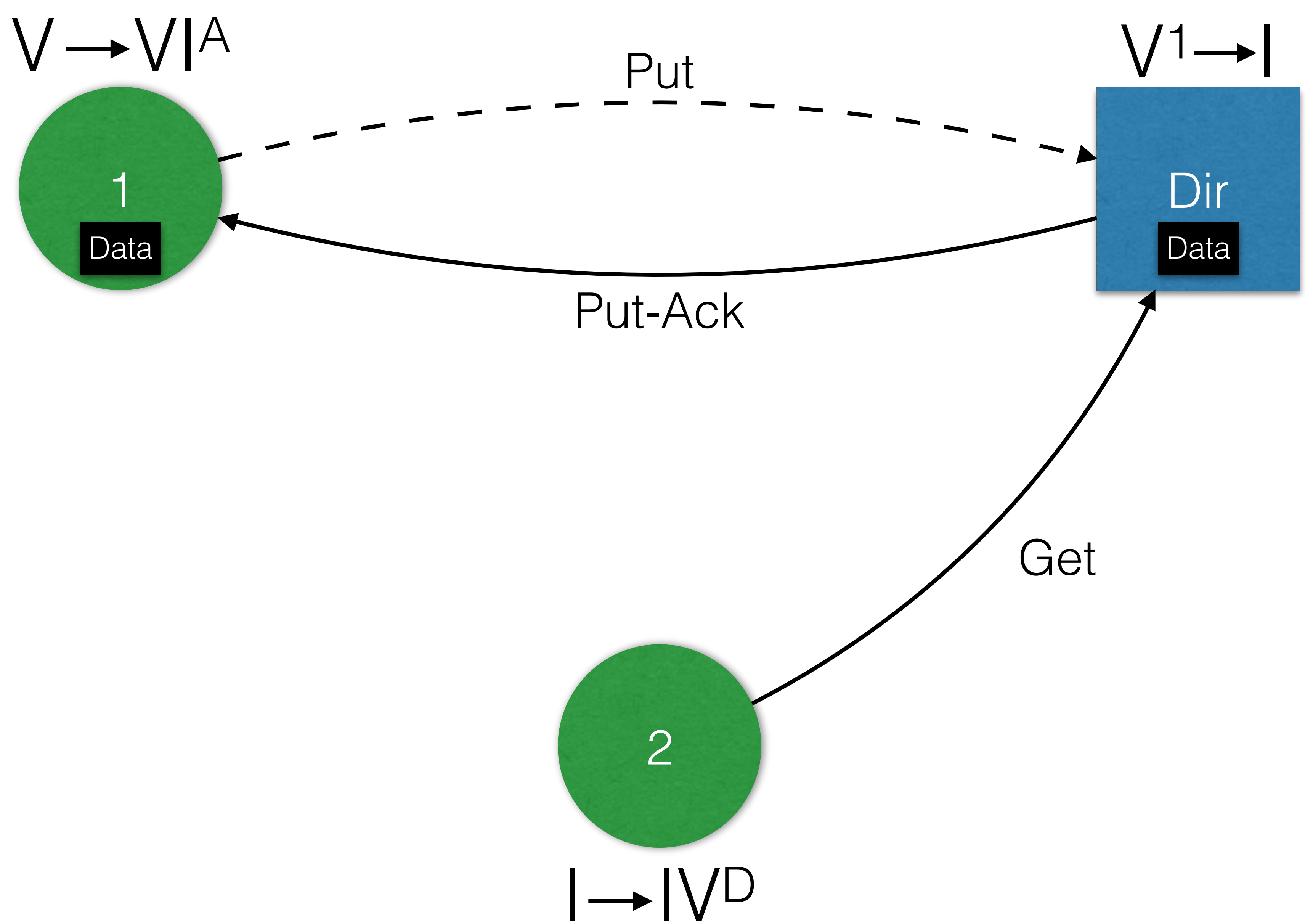
Which message arrives first?

Put/Get race

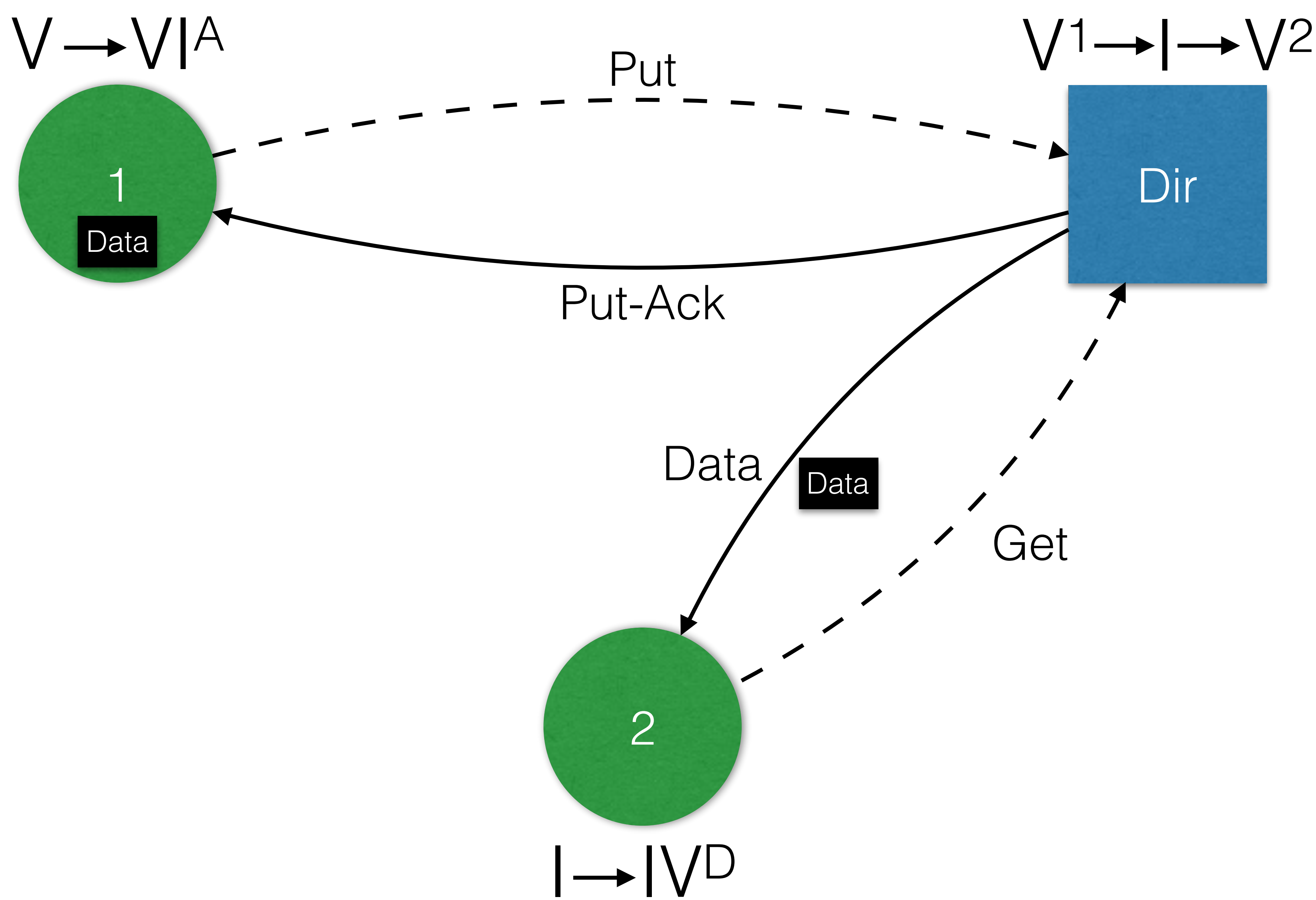


Suppose the Put from Cache 1 arrives first.
(We will explore the other case in a moment.)

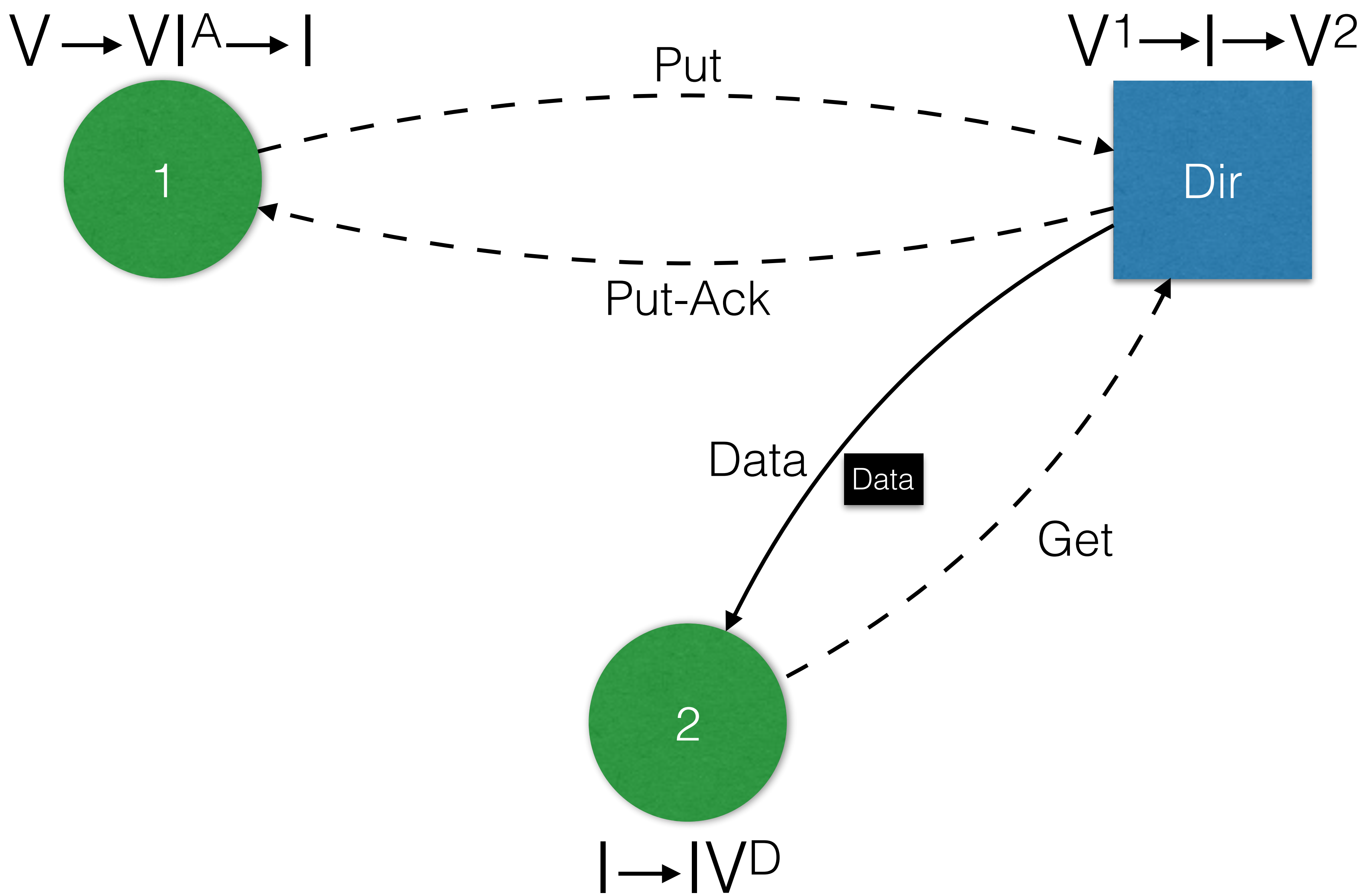
Put/Get race



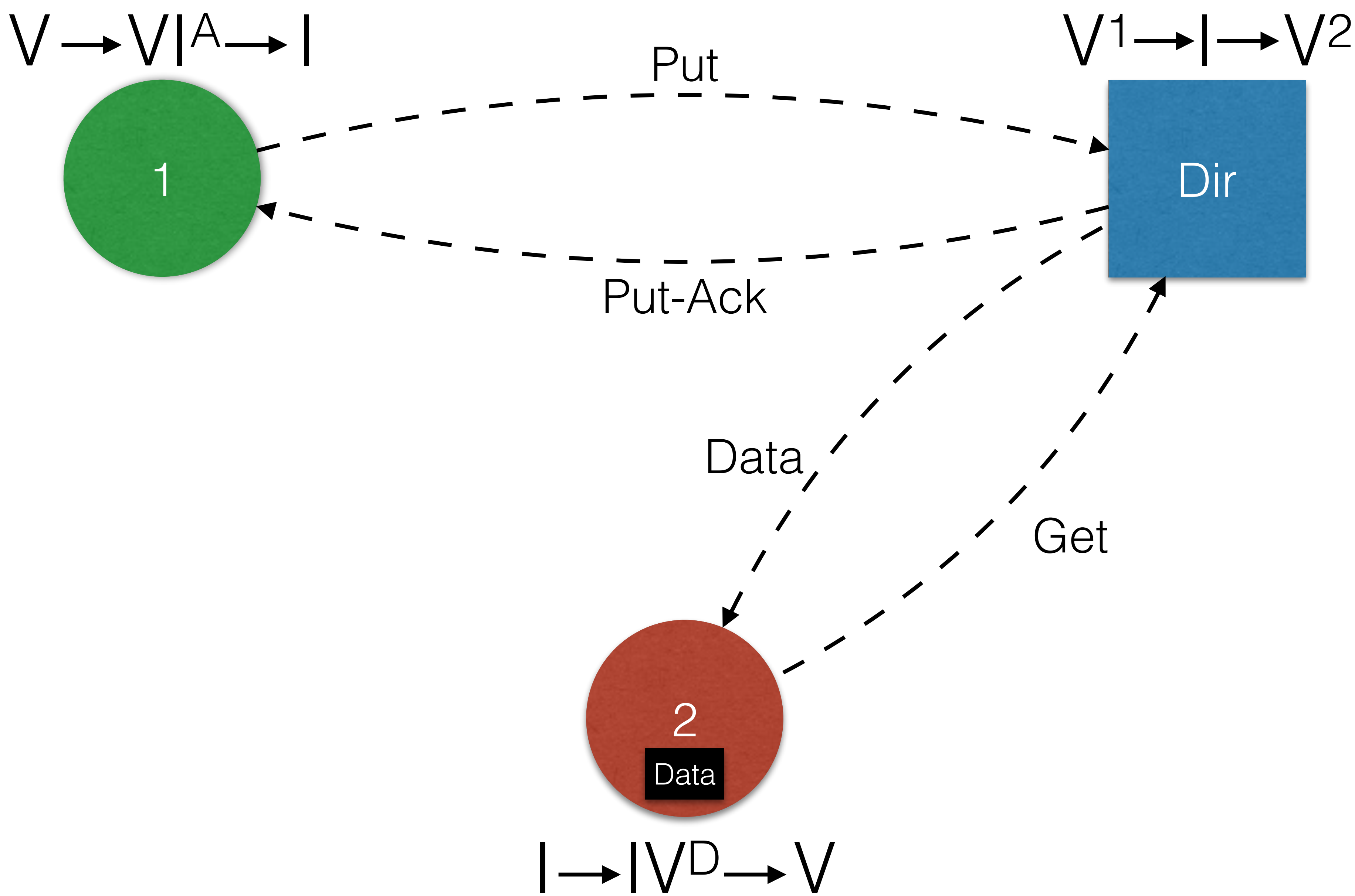
Put/Get race



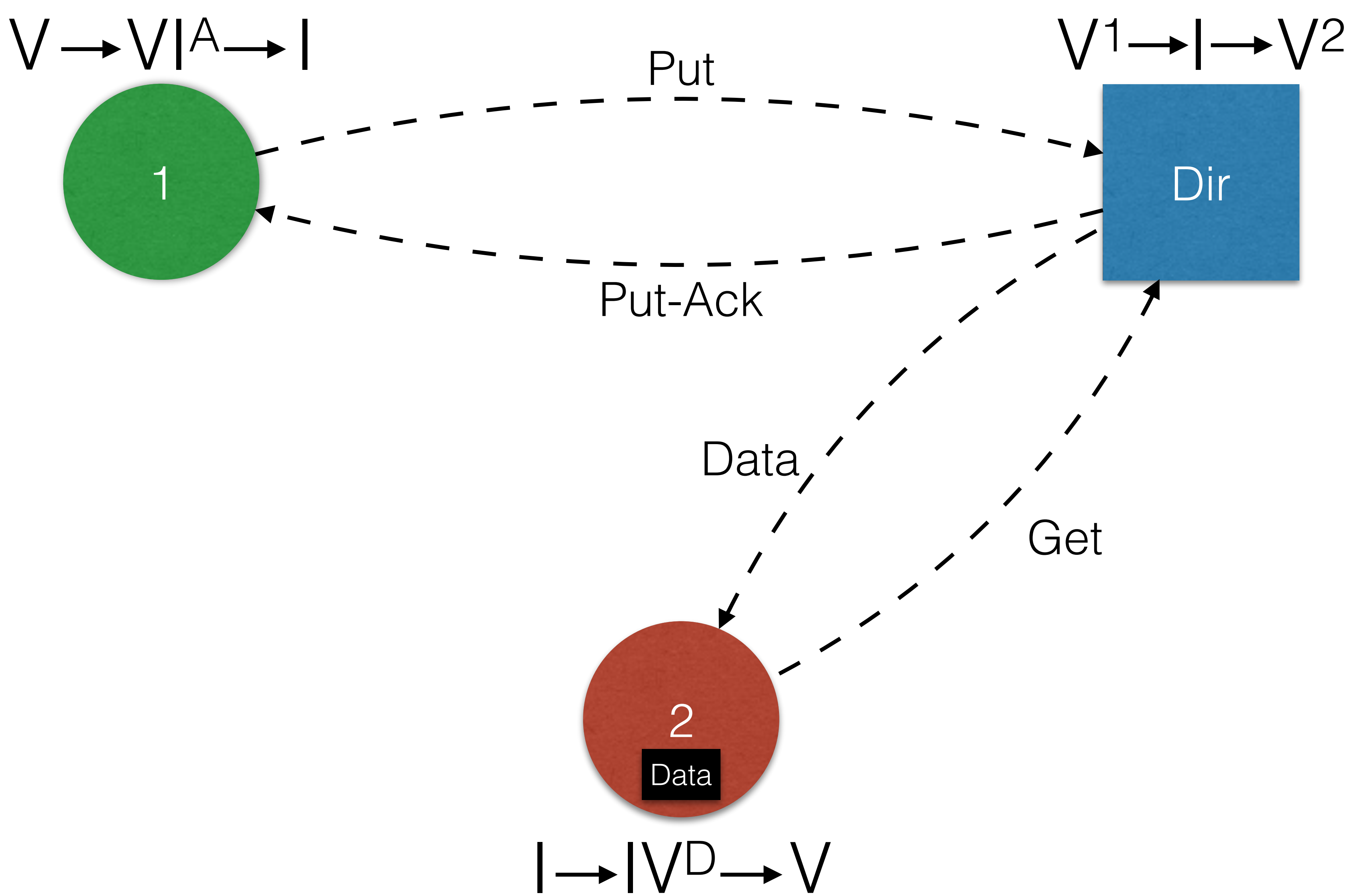
Put/Get race



Put/Get race

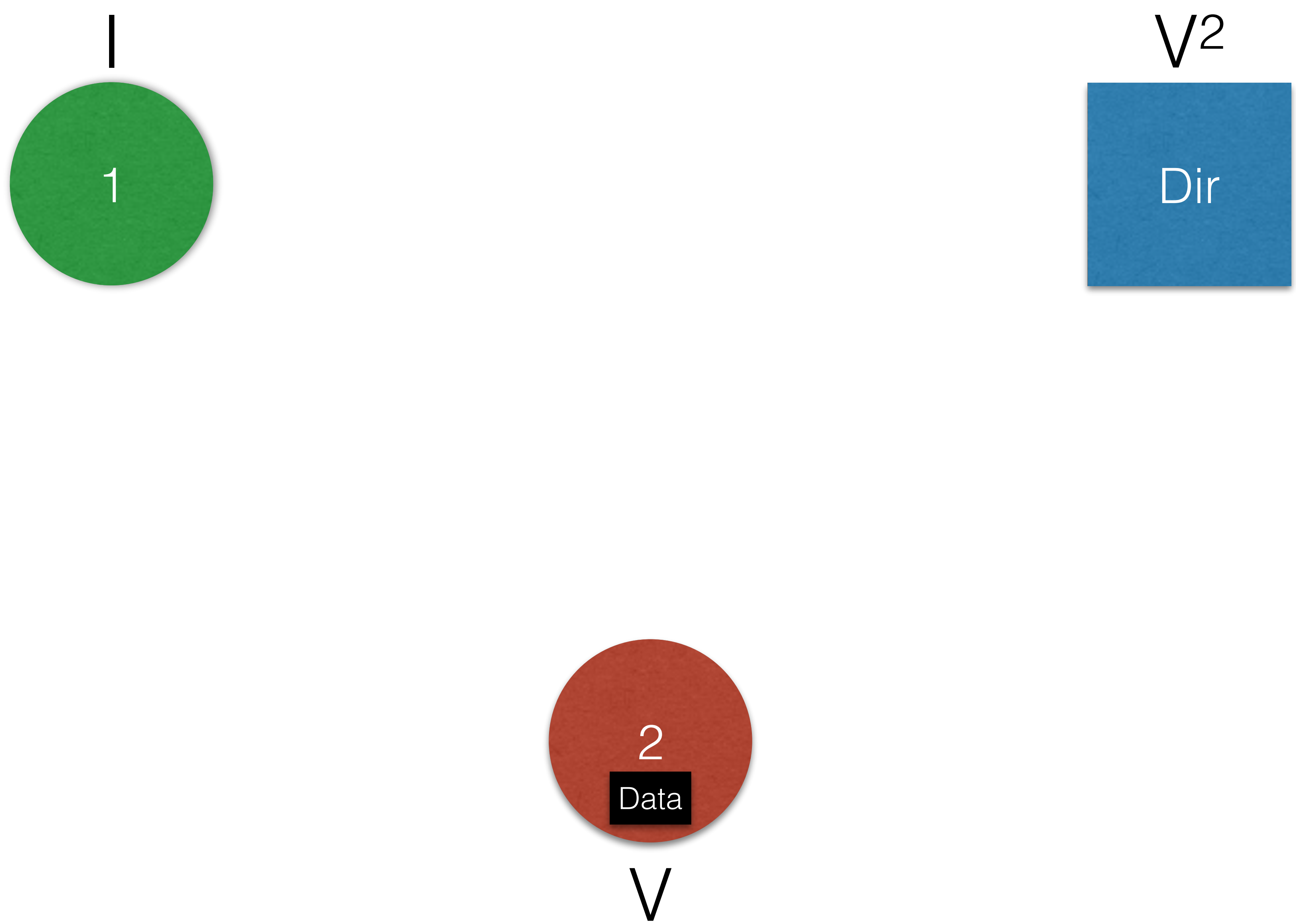


Put/Get race

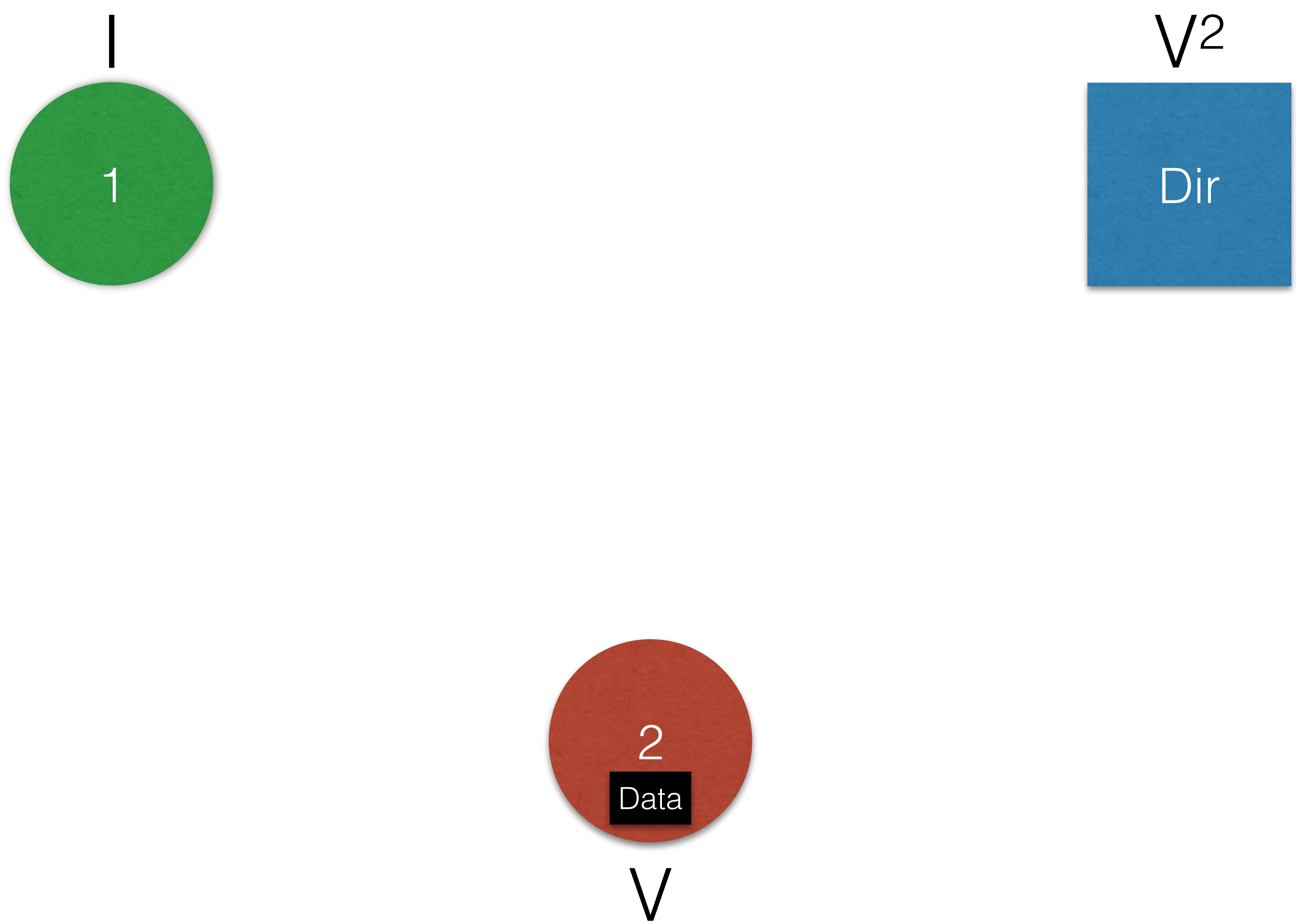


Since Dir received the Put first, there is no need for the two Caches to communicate directly.

Put/Get race

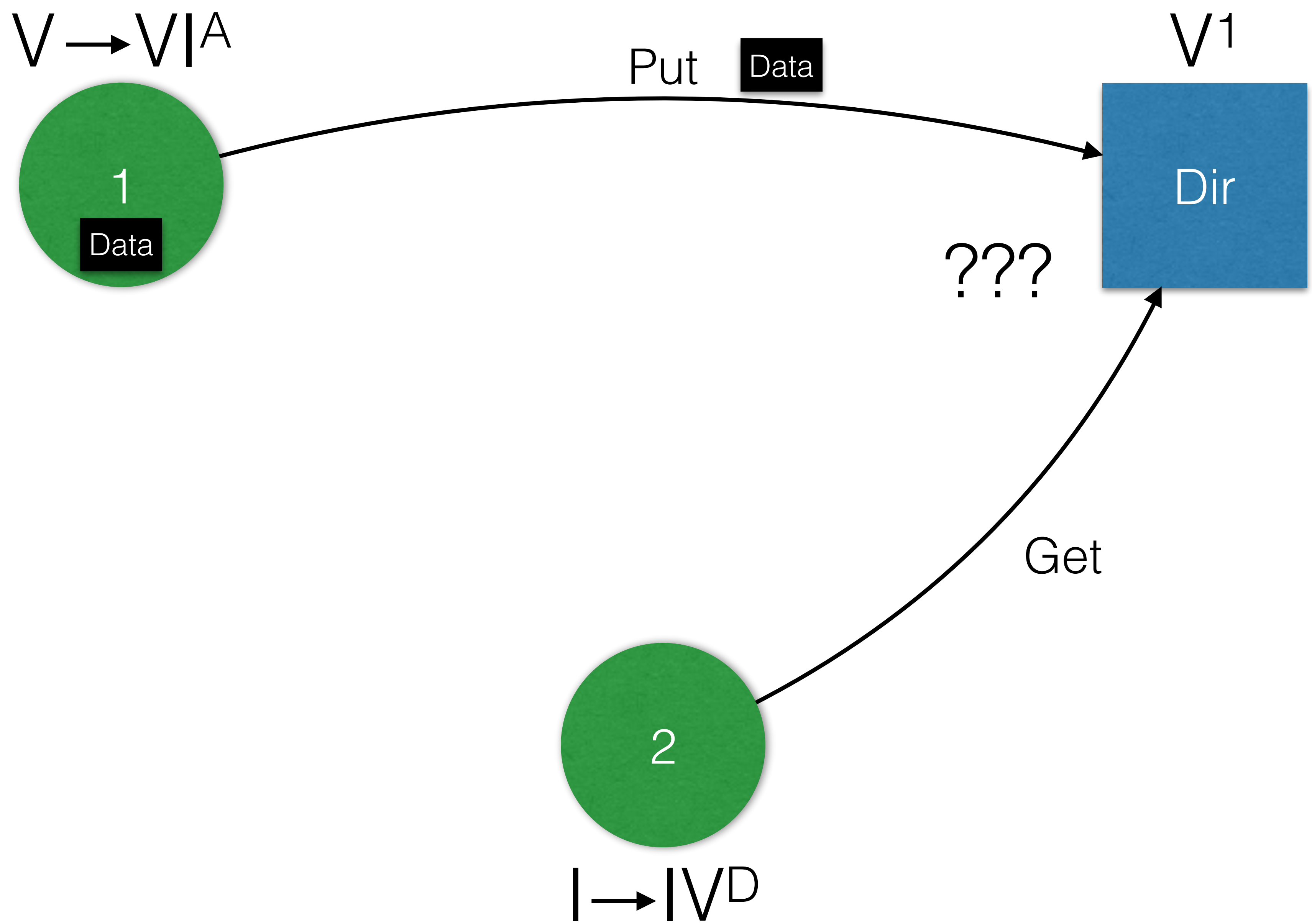


Put/Get race

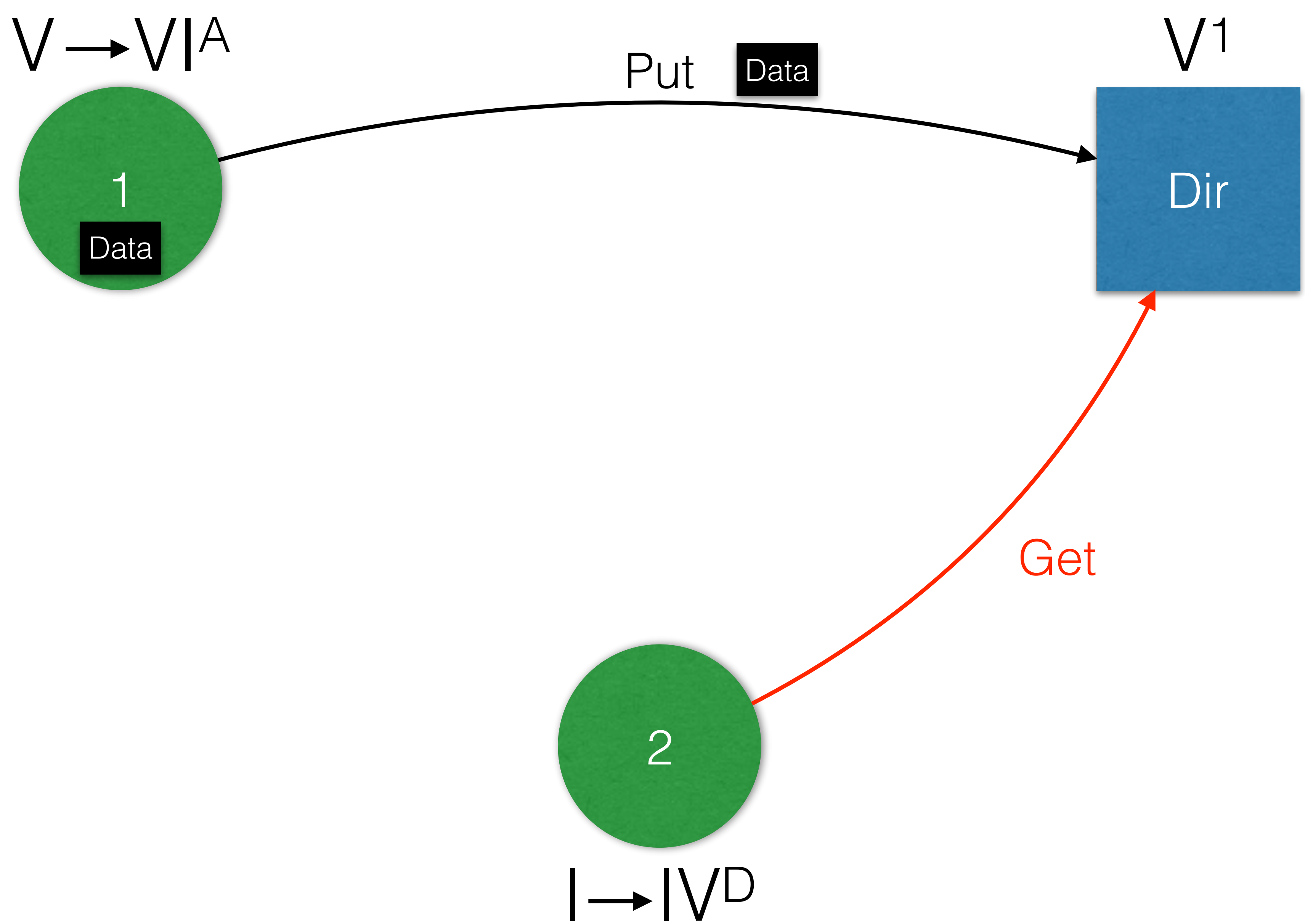


Cache 1 has evicted successfully, and Cache 2 has obtained the cache line successfully.

Put/Get race

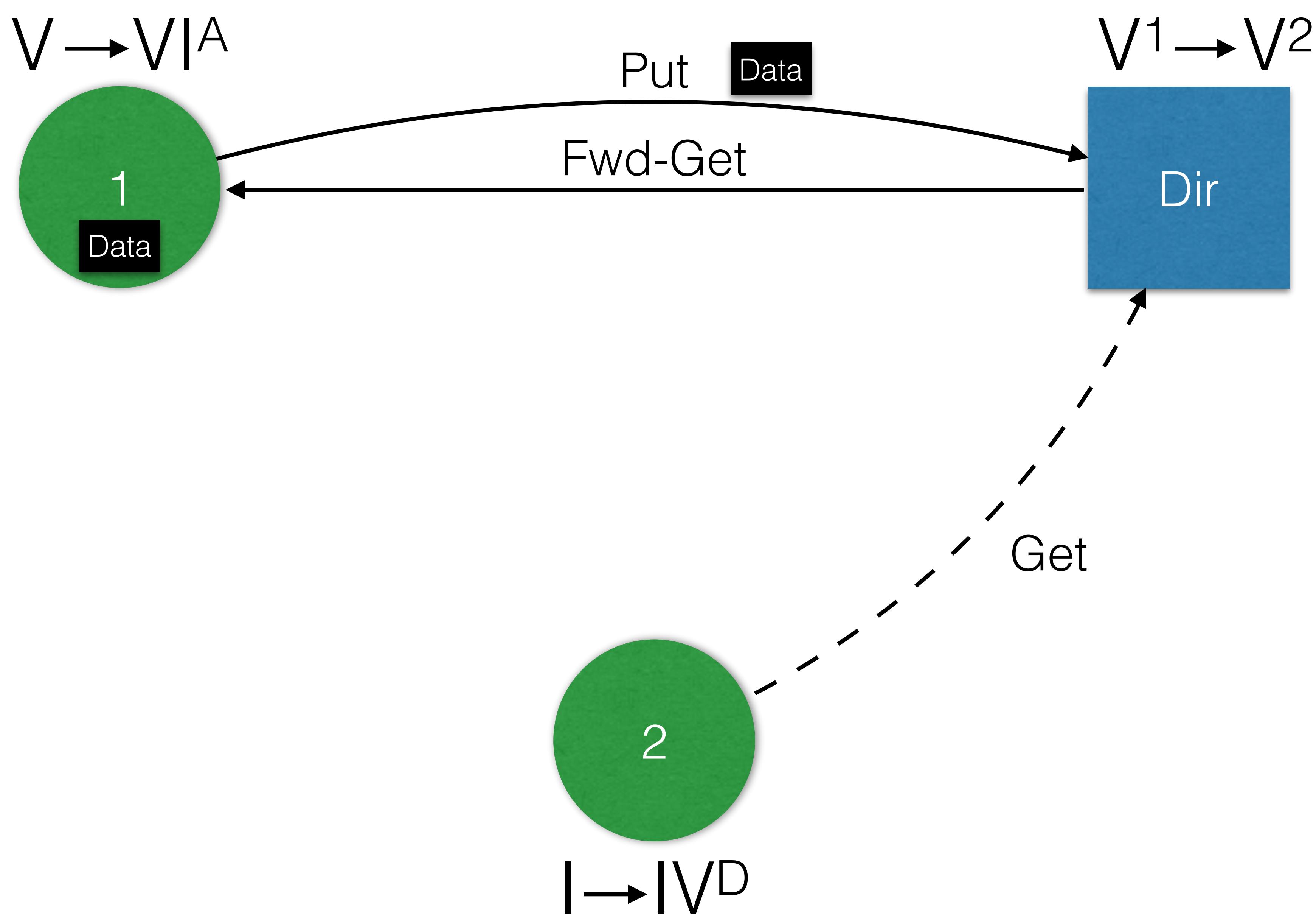


Put/Get race



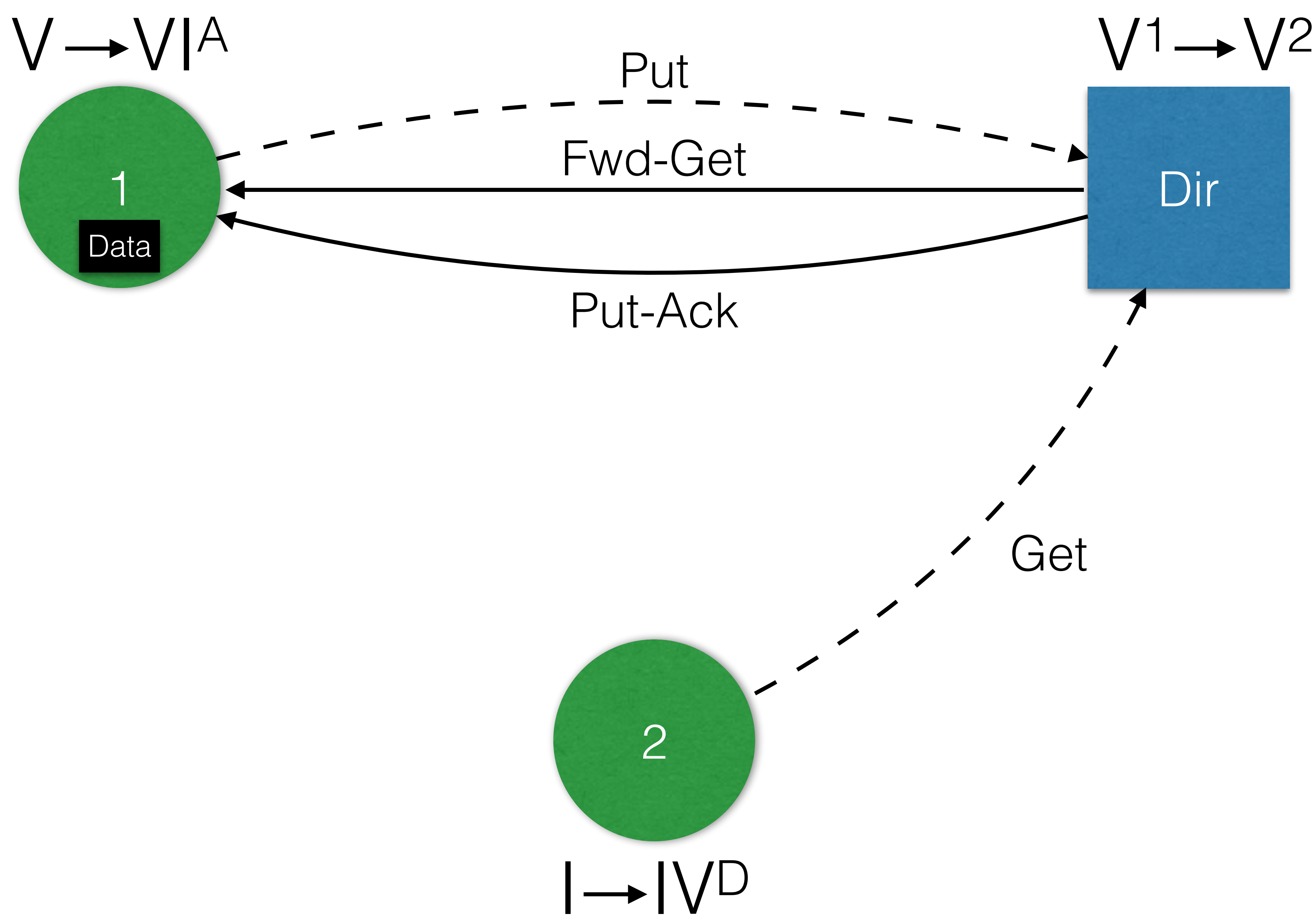
Now, suppose Dir received the Get first.

Put/Get race



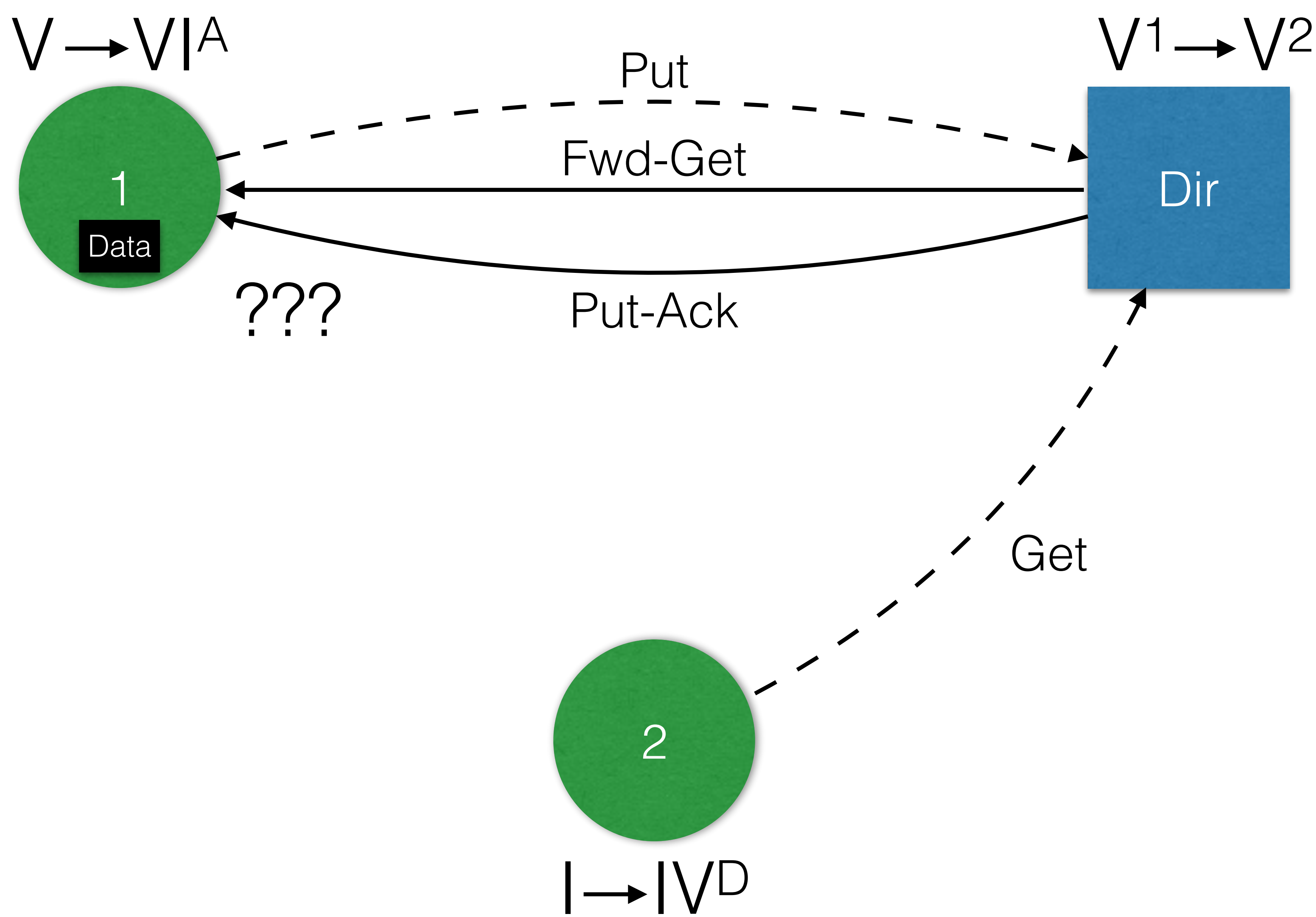
First, Dir forwards the Get request to Cache 1, since Cache 1 is still the owner.

Put/Get race



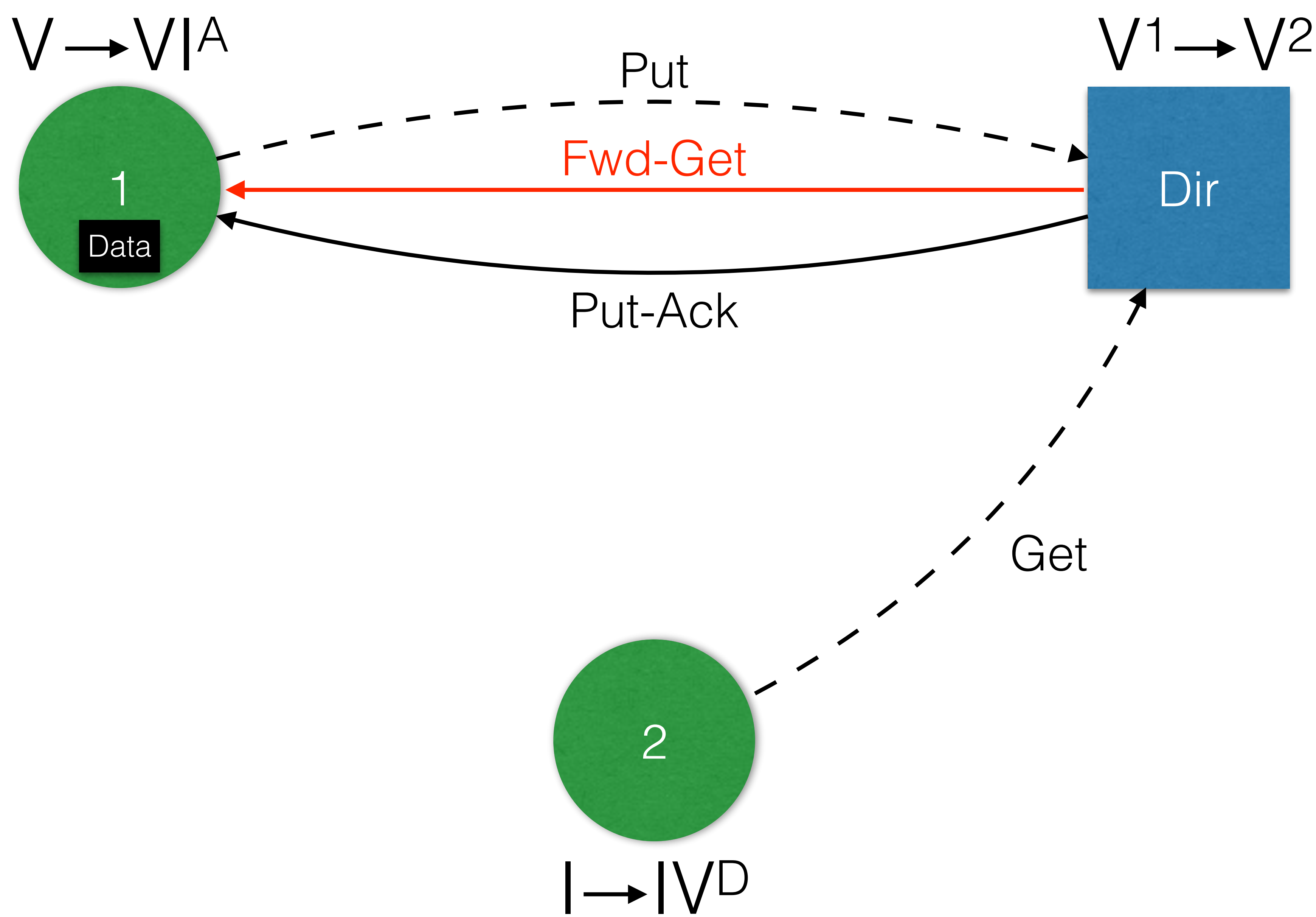
Then, Dir receives Cache 1's Put. Since Cache 1 is no longer the owner, Dir simply responds with a Put-Ack, and throws out the incoming Data.

Put/Get race



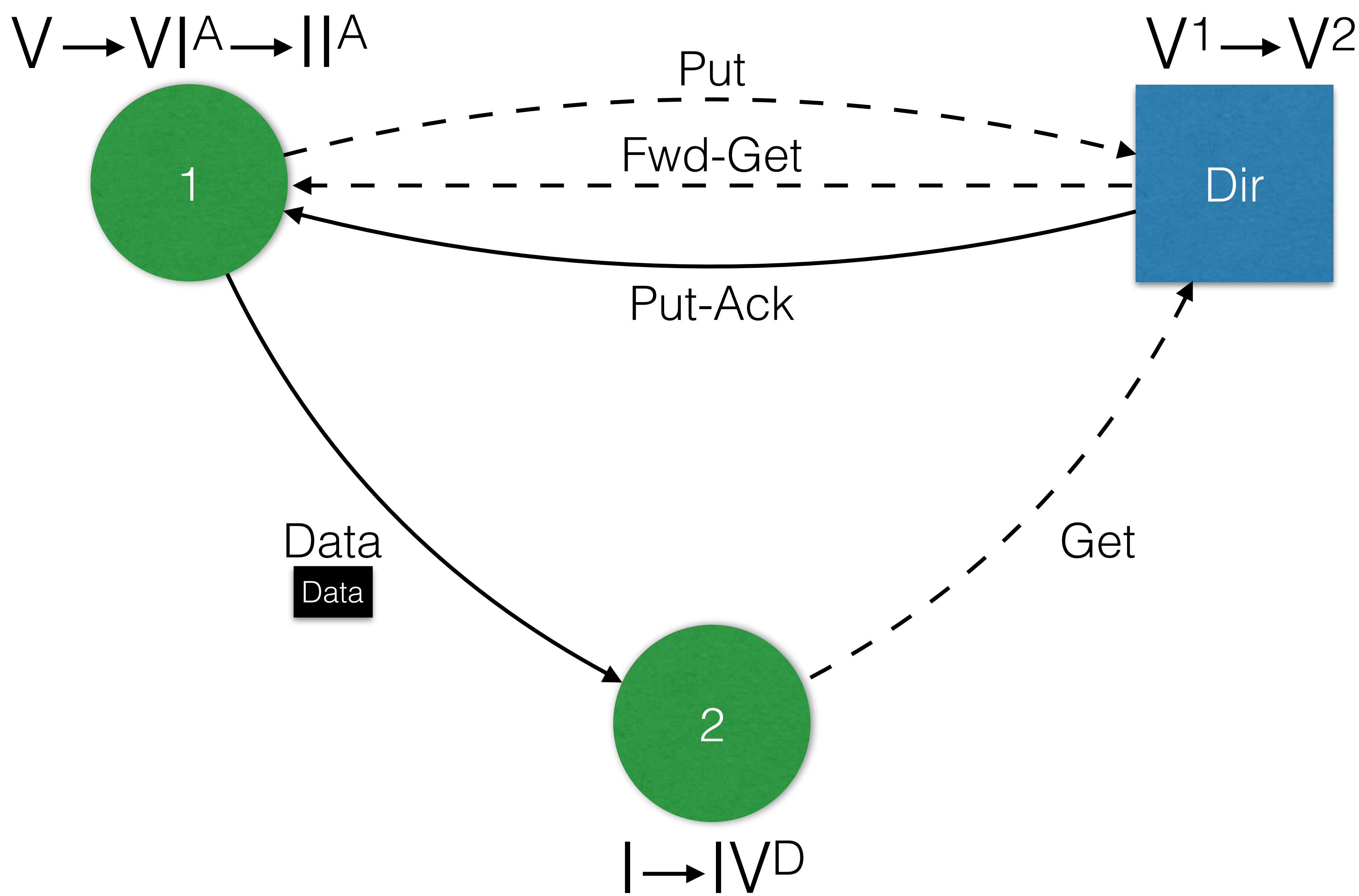
Which message arrives first?

Put/Get race



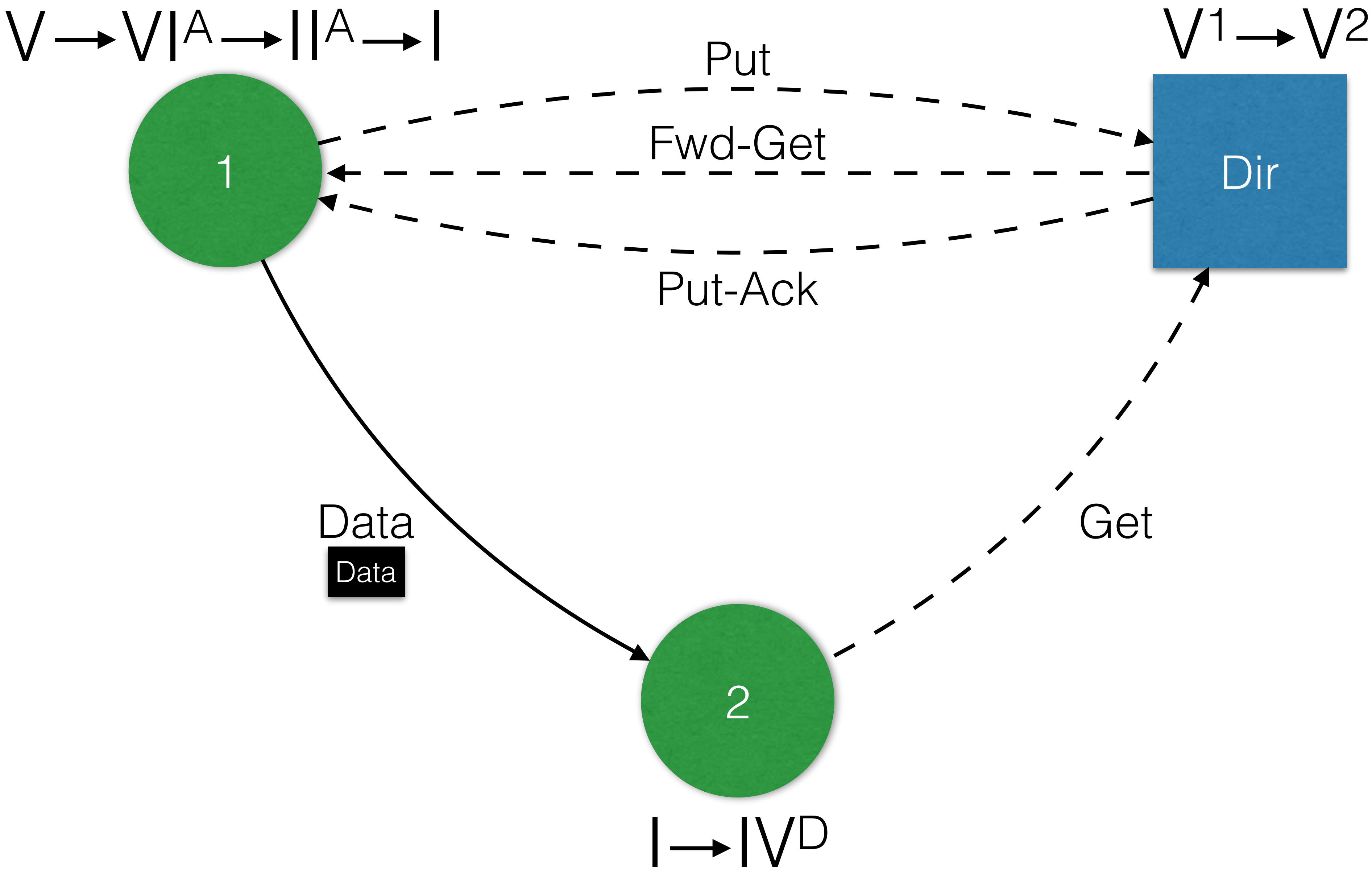
Suppose the Fwd-Get arrives first.

Put/Get race



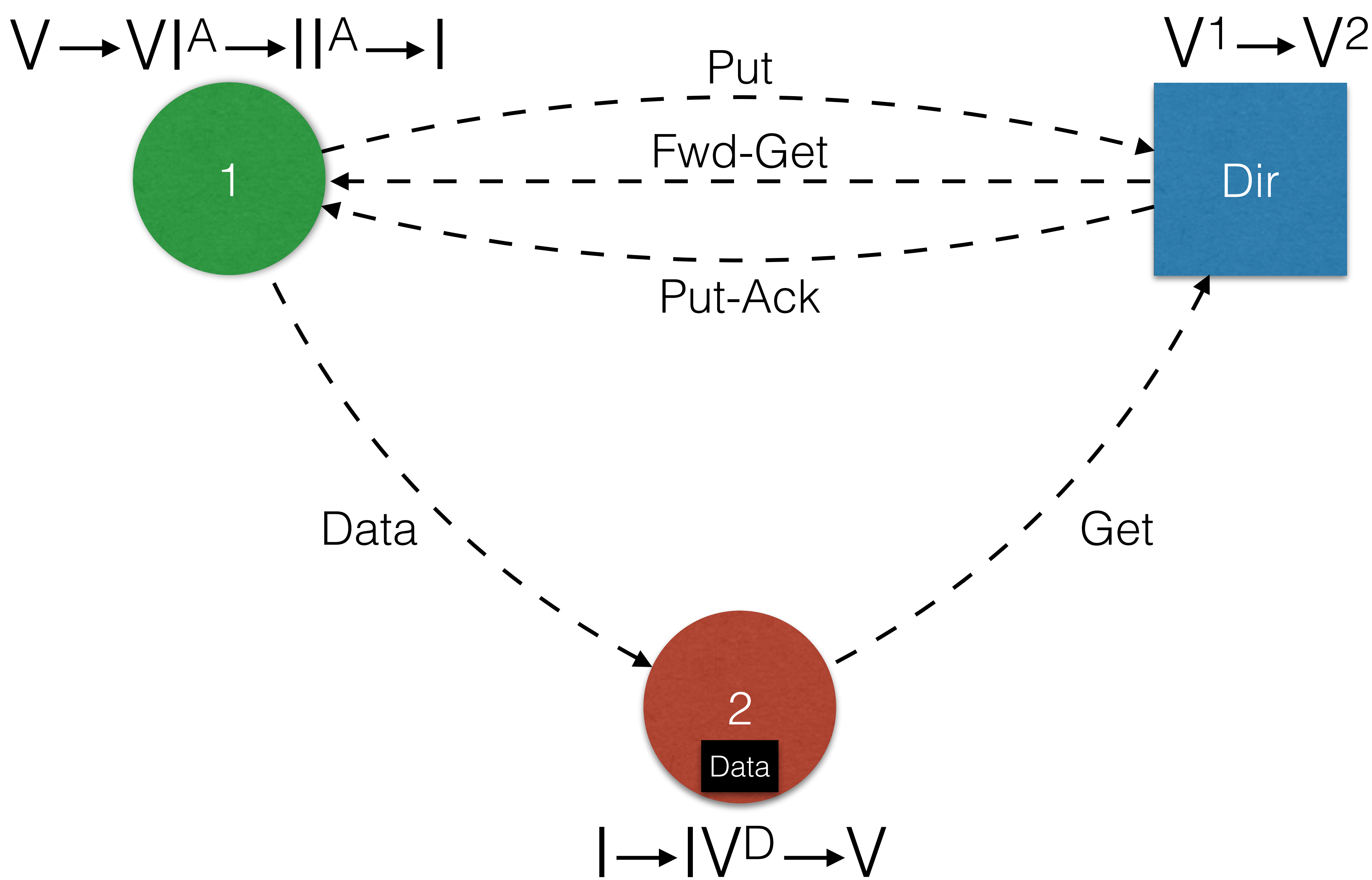
Because Cache 1 hasn't evicted yet, he still has the data. He sends it along to Cache 2 and evicts (although he still awaits a Put-Ack from Dir).

Put/Get race



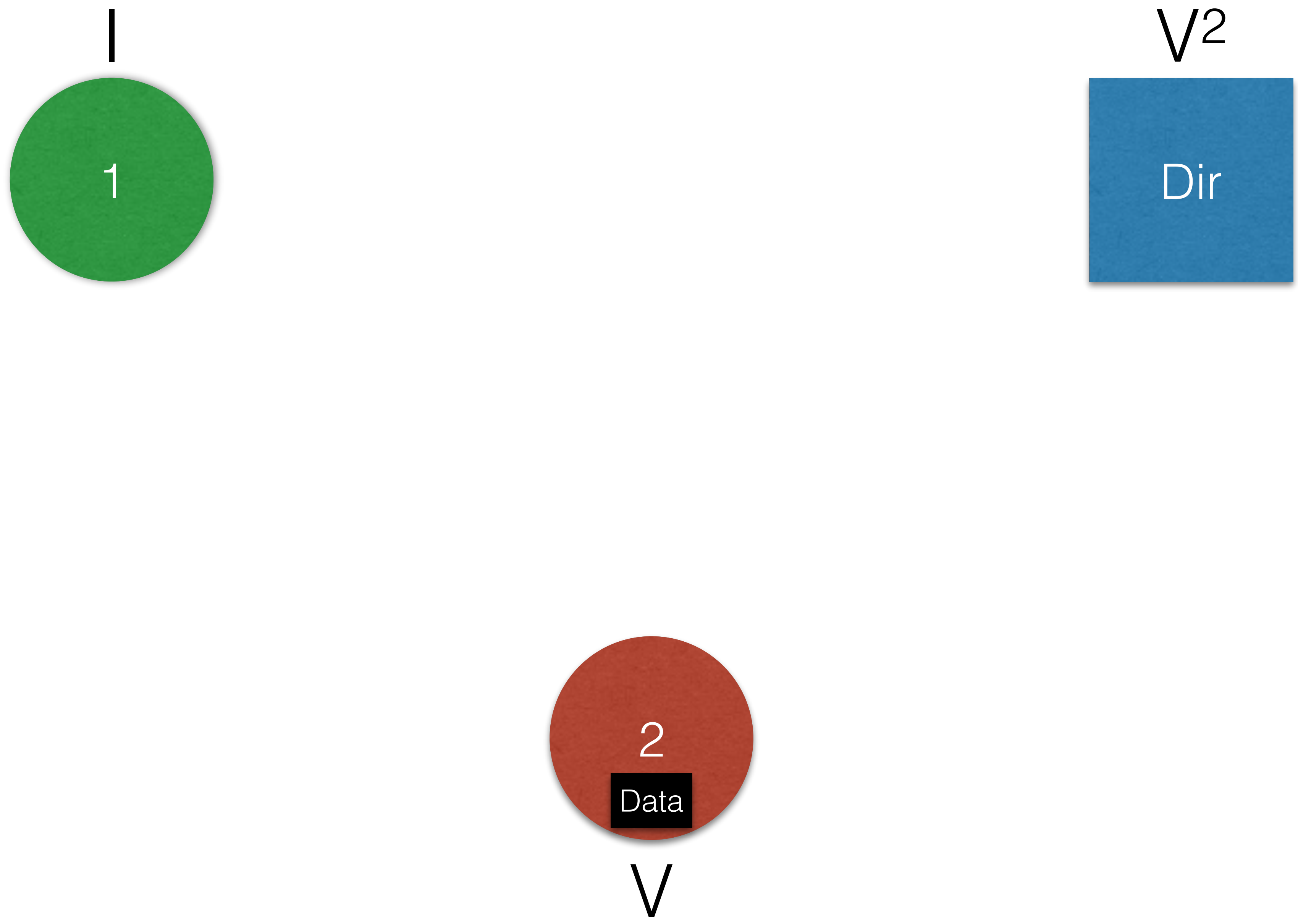
Cache 1 receives the Put-Ack, and transitions to I.

Put/Get race

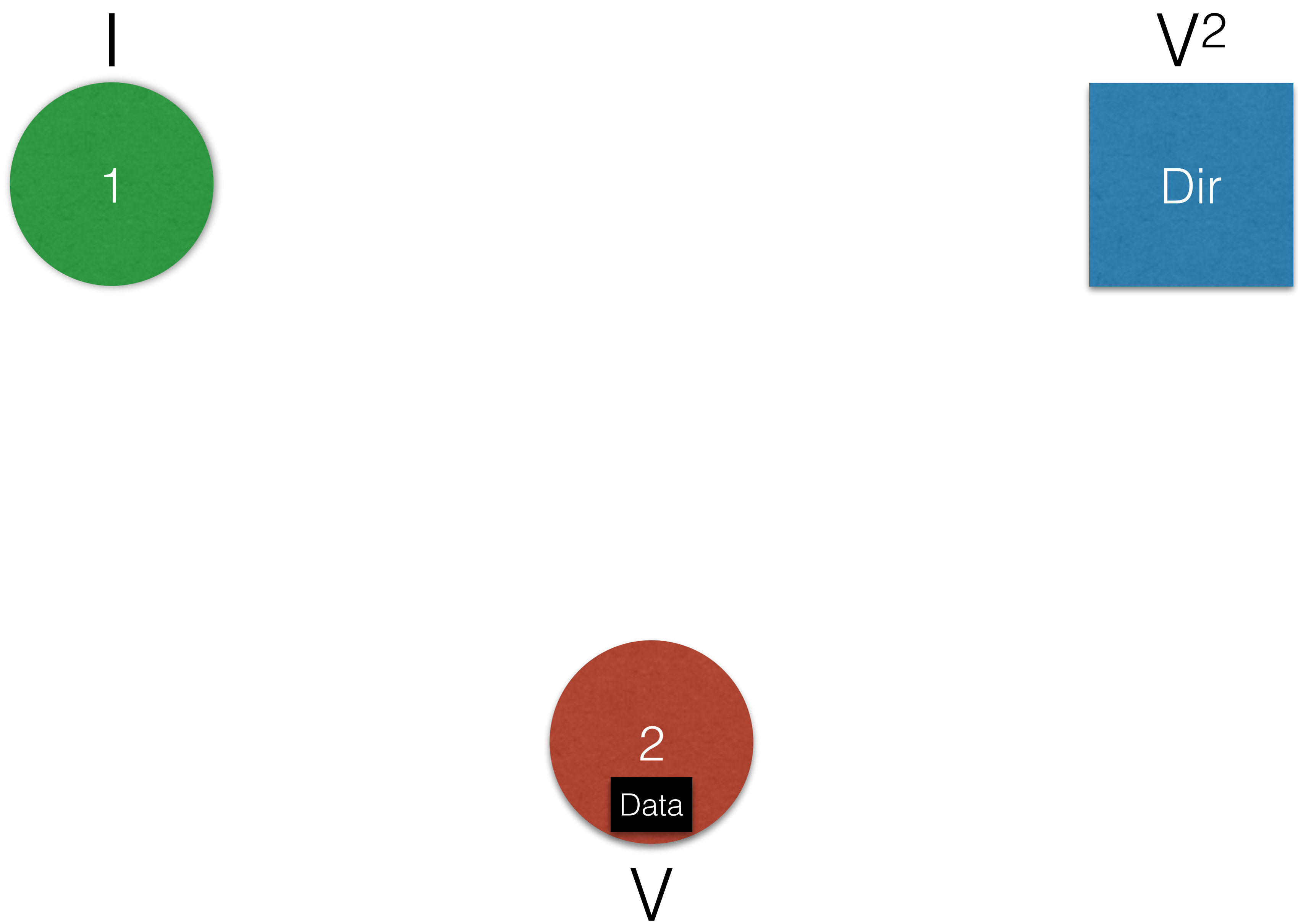


Cache 2 receives the Data, and transitions to V.

Put/Get race

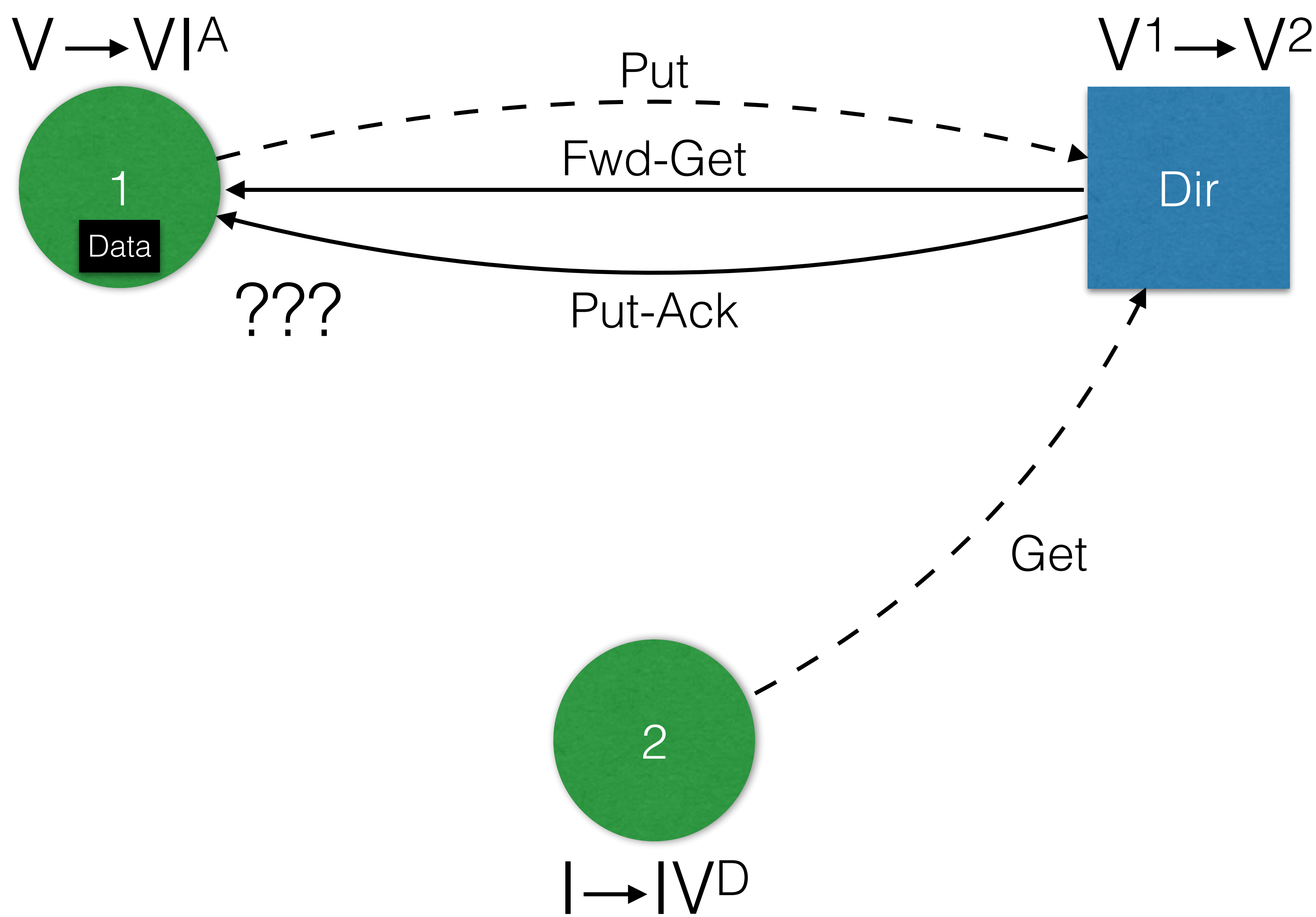


Put/Get race

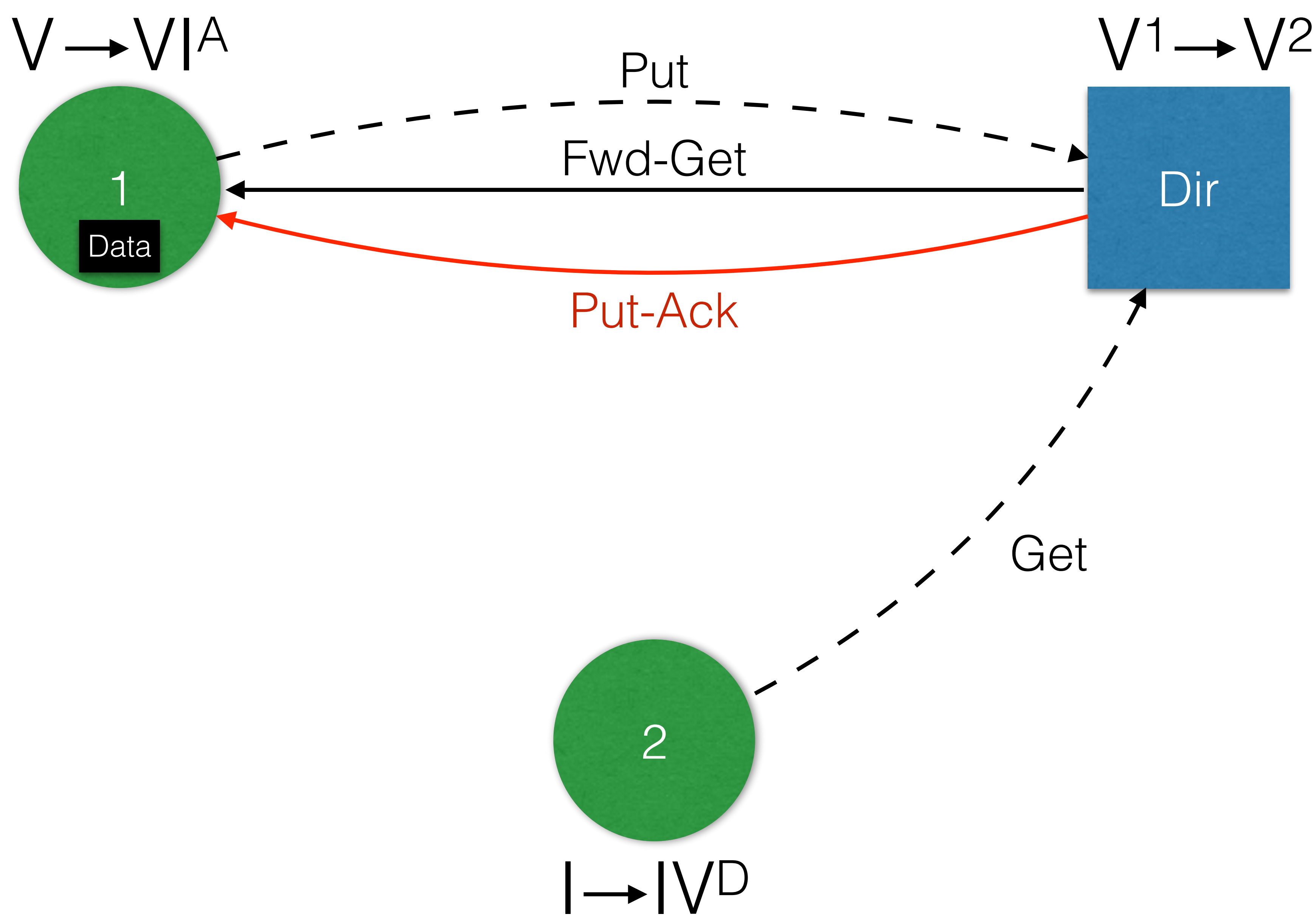


Cache 2 has upgraded successfully. Cache 1's attempt to evict was effectively "aborted" since the Get request was serviced before the Put request arrived.

Put/Get race

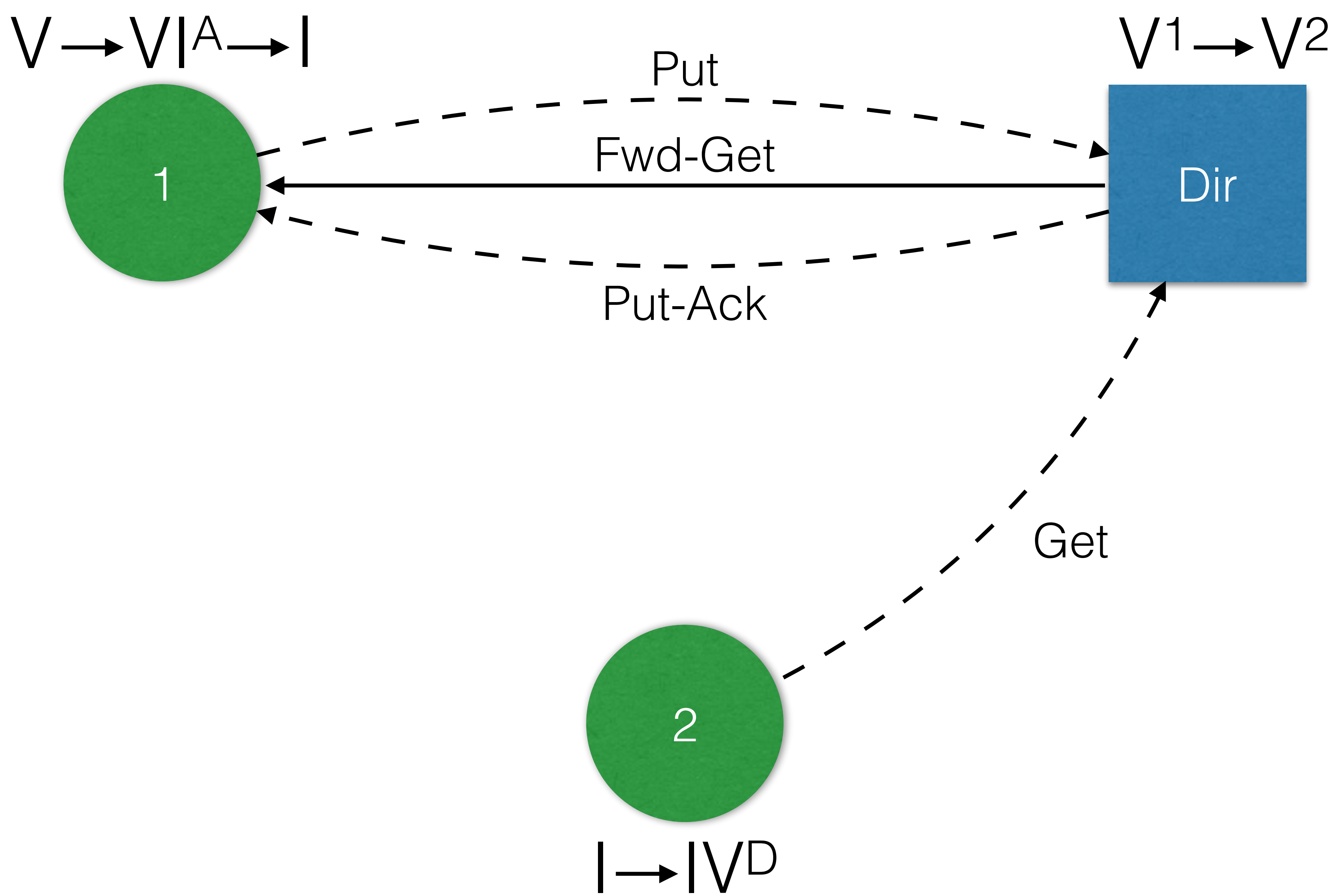


Put/Get race



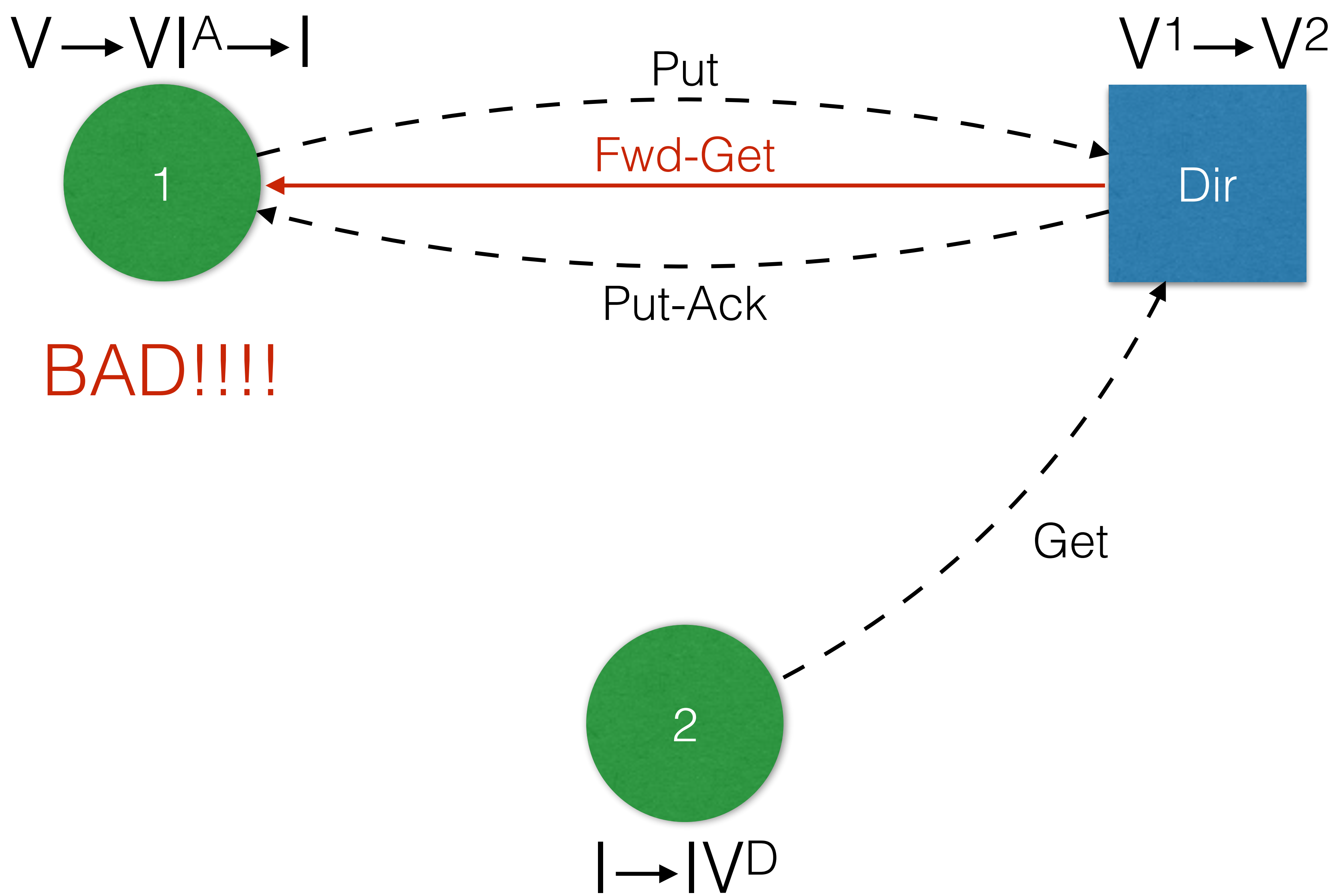
Now, suppose Cache 1 receives the Put-Ack first.

Put/Get race



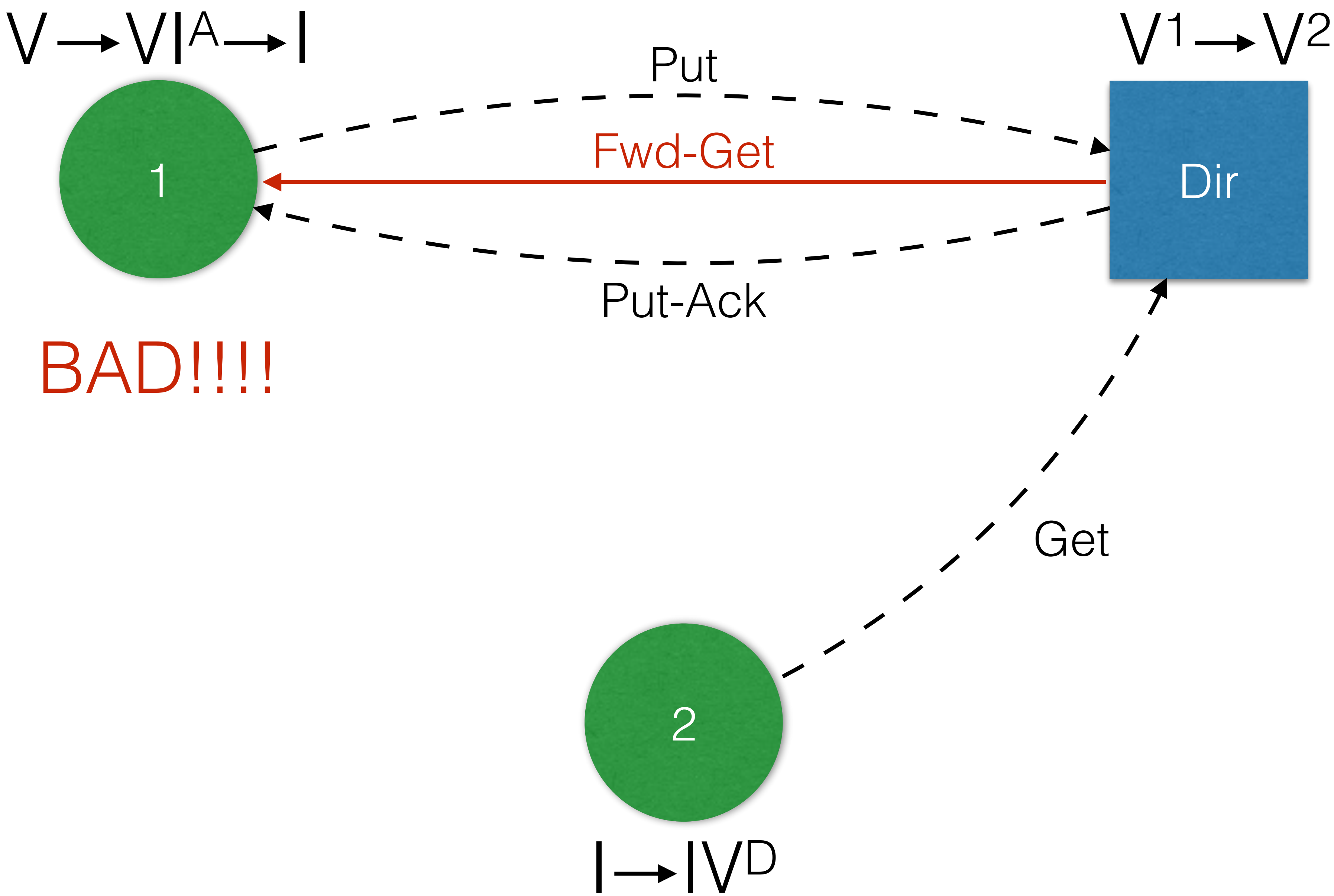
Cache 1, having received Put-Ack, evicts the cache line.
(Um…. where's the data?)

Put/Get race



Then, Cache 1 receives a Fwd-Get. He can't forward the data, because he already evicted!

Put/Get race



Solution: use the same "channel" for Fwd-Get and Put-Ack, and require point-to-point ordering.

Put/Get race

Lesson: A cache coherence protocol may seem relatively simple, but concurrency and data races can lead to some odd behavior.

We need to be VERY careful when designing these protocols in order to ensure bad things don't happen.

Correctness of VI

- We wish to demonstrate that the VI cache coherence protocol is correct.
- For our protocol, this means that no two caches can have the cache line in state V simultaneously.
- We believe we have designed our protocol well, but it's actually deceptively complicated
- We used ACL2 to construct an invariant-style proof of this property

Proof strategy

1. Let P_1 = property we want to prove is an invariant
2. Let Props = $\{P_1\}$
3. for each P_i in Props:
   A. Try to prove: Props(m) -> P_i(step m)
   B. For each failed subgoal, create new P_j that lets us prove that subgoal, and add it to Props until A is proved by ACL2
   C. Repeat until we have proved everything in Props is preserved by step
4. We have shown Props(m) -> Props(step m). Since P_1 is in Props, we have shown that if we start from a state where Props(m) is true, then we can run the protocol as long as we want, and P_1 will always be true.

Next time

- I’ll present the correctness proof in some detail.
- I’ll talk about some ideas I’ve had for using ACL2 both to design AND verify complex, scary cache protocols.

Proving correctness

- In industry, model checkers are usually used to verify coherence for these protocols.

- For complicated protocols, model checkers can fail to terminate fast enough.

- Even if you manage to get model checker to finish the proof, you may need to make so many simplifications in the encoding that the "proof" won't actually convince too many people.

- I'm interested how a theorem prover like ACL2 can be used to aid in these verification efforts.

Other stuff

Correctness of VI: Initial Attempt

- At first, we started the proof by specifying correctness as “no two caches are in state V”
- We then asked ACL2 to prove that this property was preserved by all transitions
- Each failed subgoal suggested a new property we needed to assume
- The hope: at some point, these invariants will become “closed”

```
(define i-v-j-v-unq
  ((m vi-p)
   (i (cache-index-p m i))  ; forall
   (j (cache-index-p m j))) ; forall
  :returns (i-v-j-v? booleanp)
  (b* (((when (= i j)) t)
       ((vi m) m)
       ((cache cachei) (nth i m.caches))
       ((cache cachej) (nth j m.caches)))
    (cache-state-case cachei.cache-state
     (:v
      (cache-state-case cachej.cache-state
       (:v nil)
       (& t)))
     (& t))))

(defun-sk i-v-j-v (m)
  (forall (i j)
          (implies (and (cache-index-p m i)
                        (cache-index-p m j))
                   (i-v-j-v-unq m i j))))
```

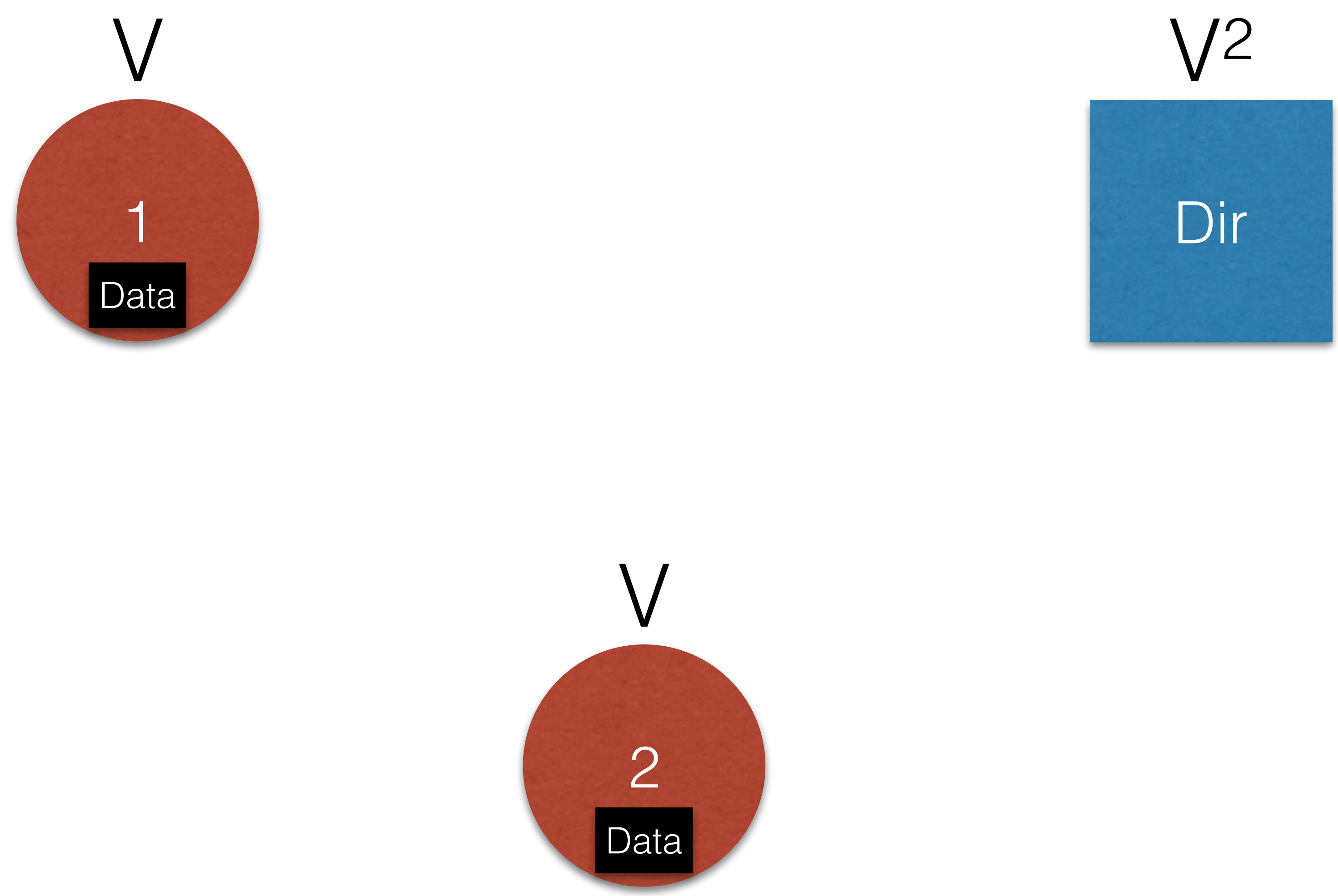
Correctness of VI: Initial Attempt

- I discovered after a lengthy, time-consuming proof attempt, that I had ended needing to assume an invariant that I couldn't prove

- I couldn't prove it because the property "blew up" - in order to prove it, I needed to assume something more complicated, and then to prove the more complicated property, I needed to assume something even MORE complicated, etc.

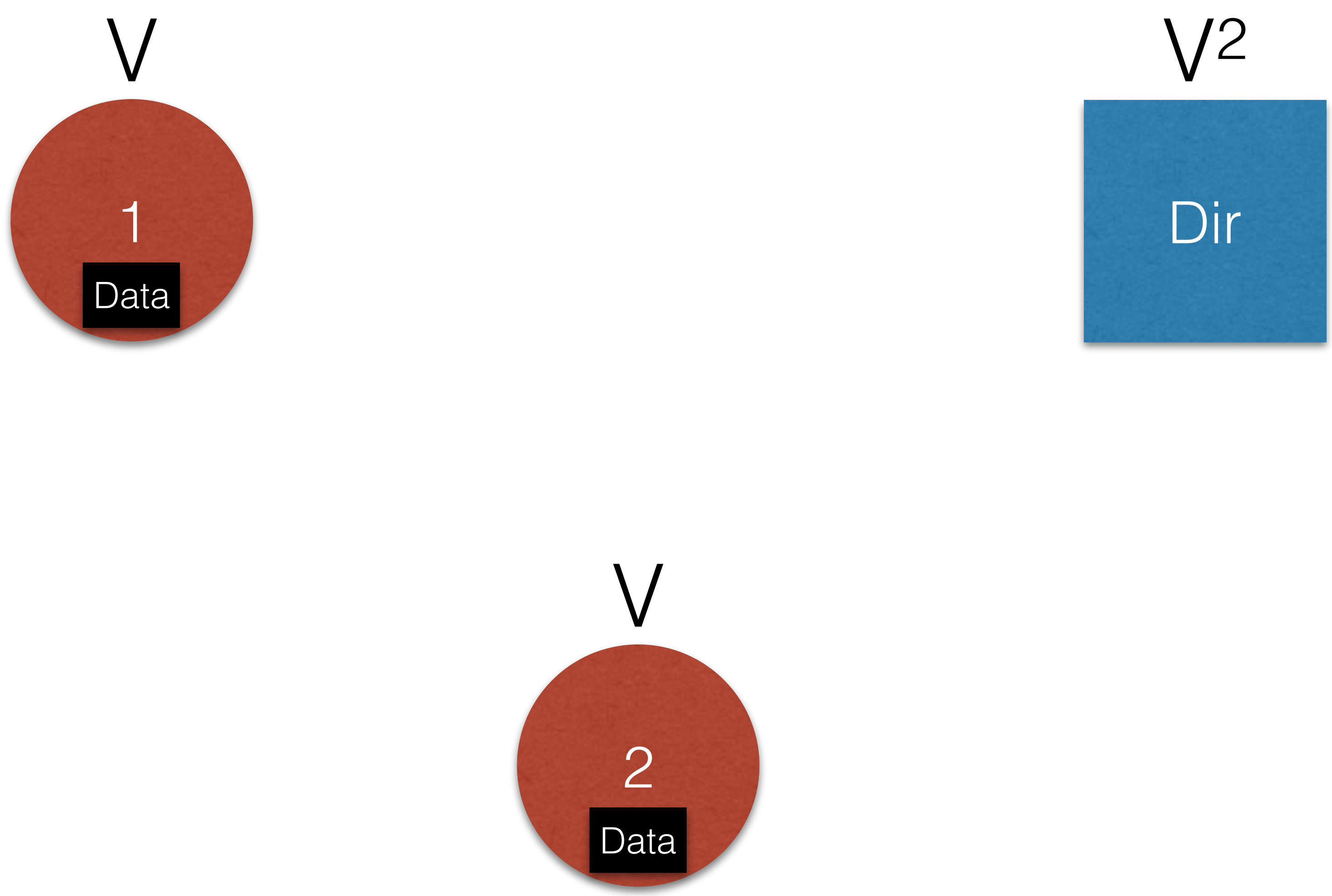
- I still can't figure out exactly where I went wrong; if anyone has any intuition, or is interested enough to discuss it, let me know

I needed to prove that this could never happen:



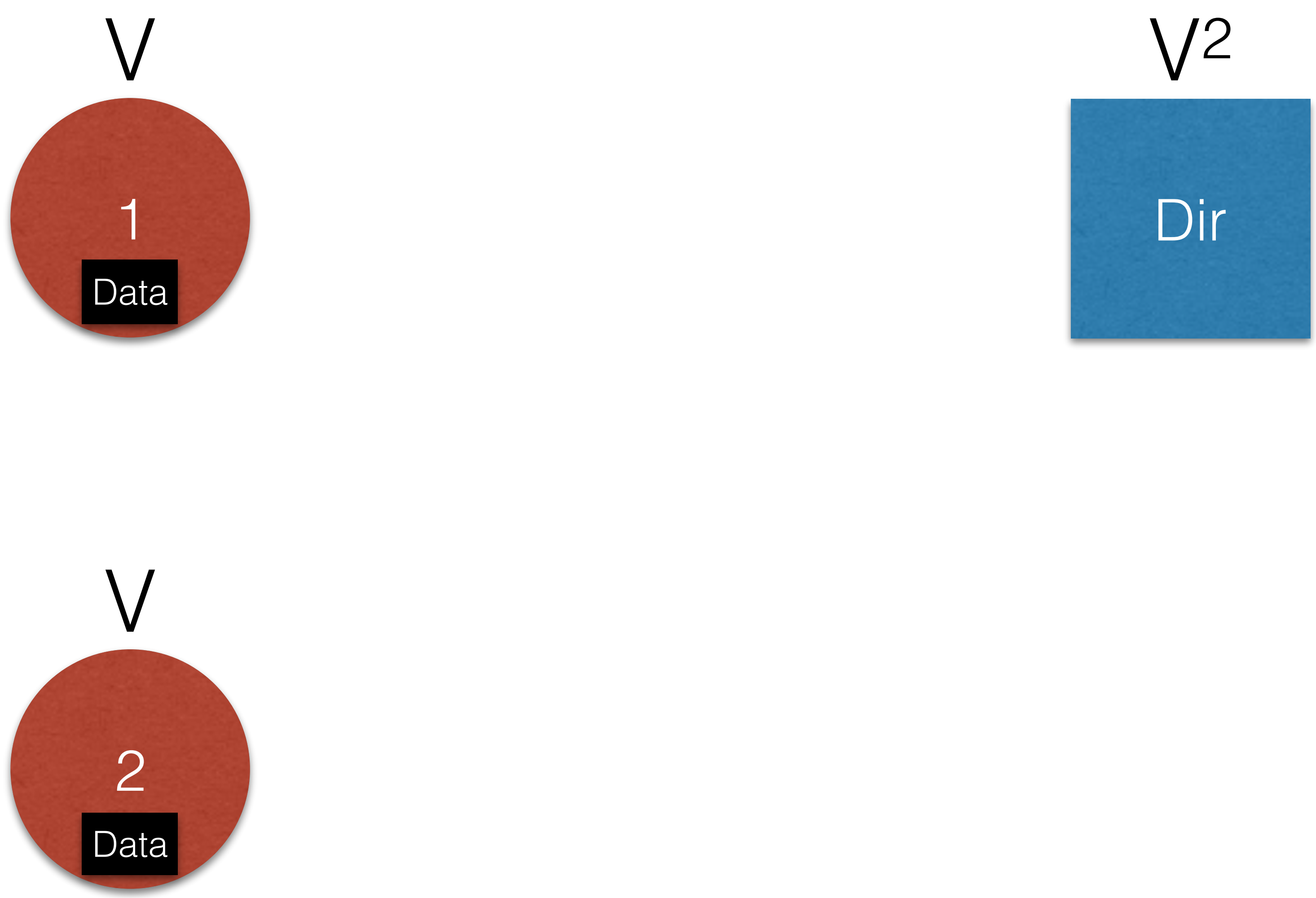
It’s clear why - both Cache 1 and 2 are in V.
The Dir thinks 2 is the owner.

I needed to prove that this could never happen:



Let's backtrack and see how this could have happened…

I needed to prove that this could never happen:

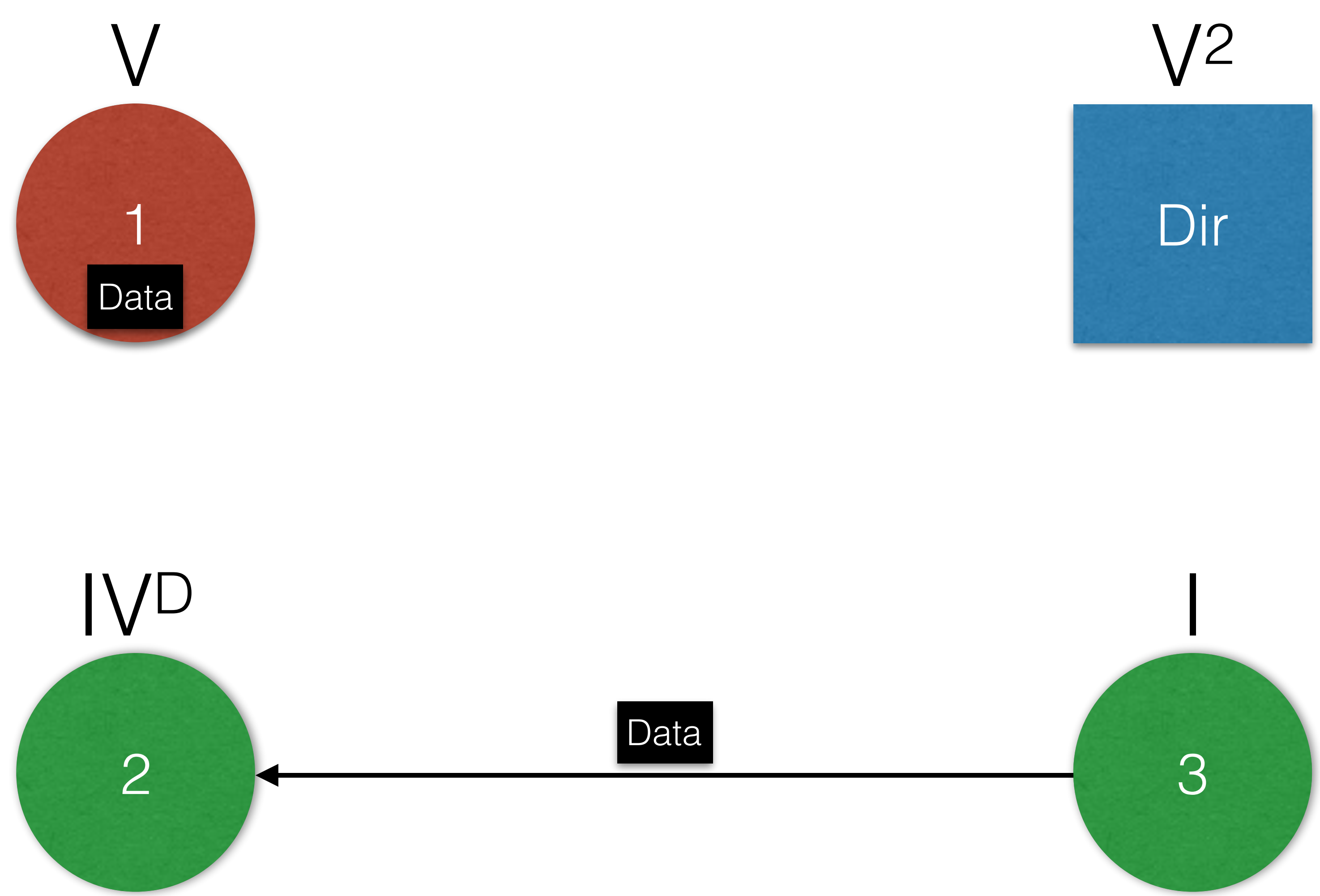


Let's backtrack and see how this could have happened…

I needed to prove that this could never happen:

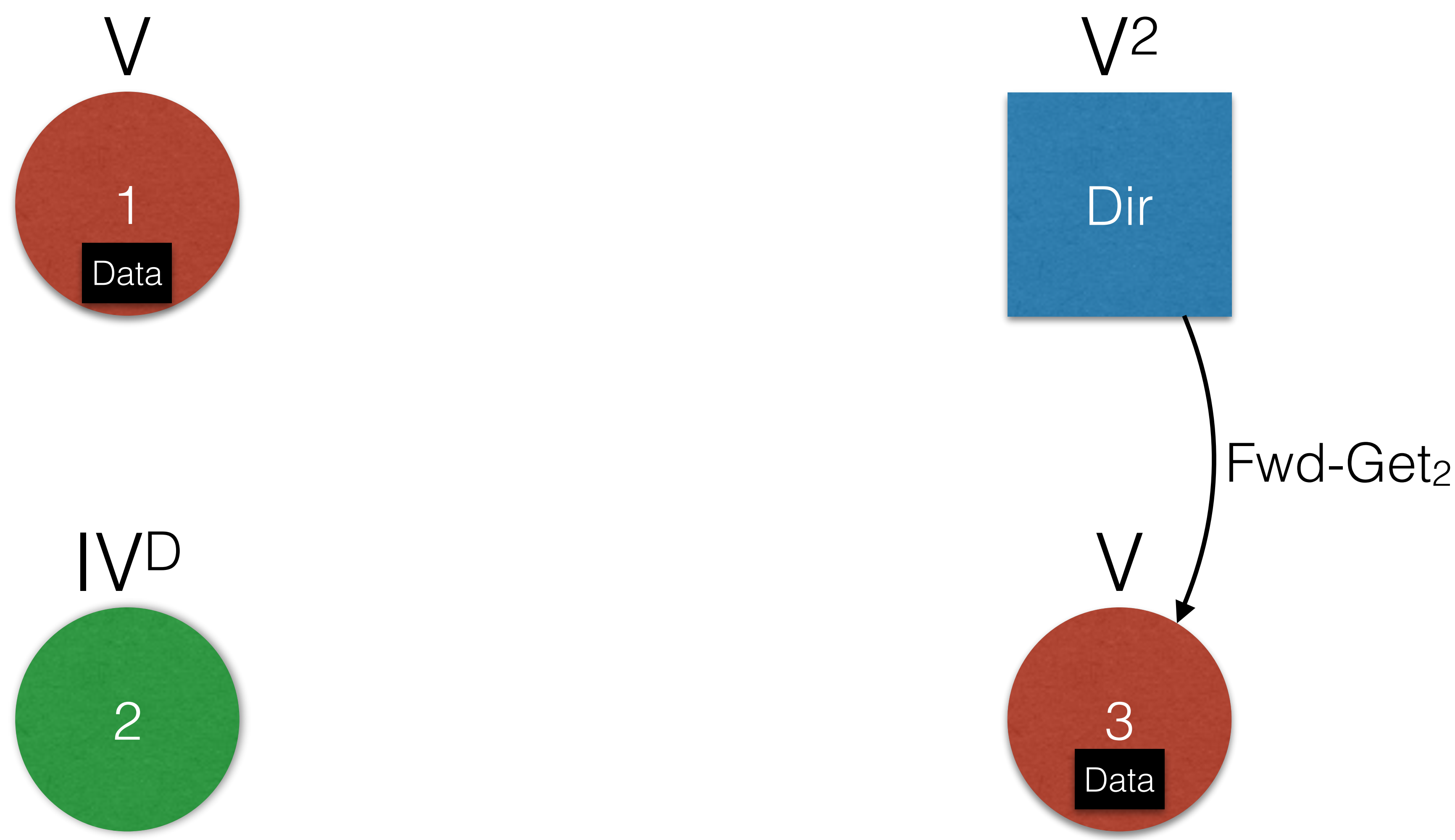
Let’s backtrack and see how this could have happened…

I needed to prove that this could never happen:



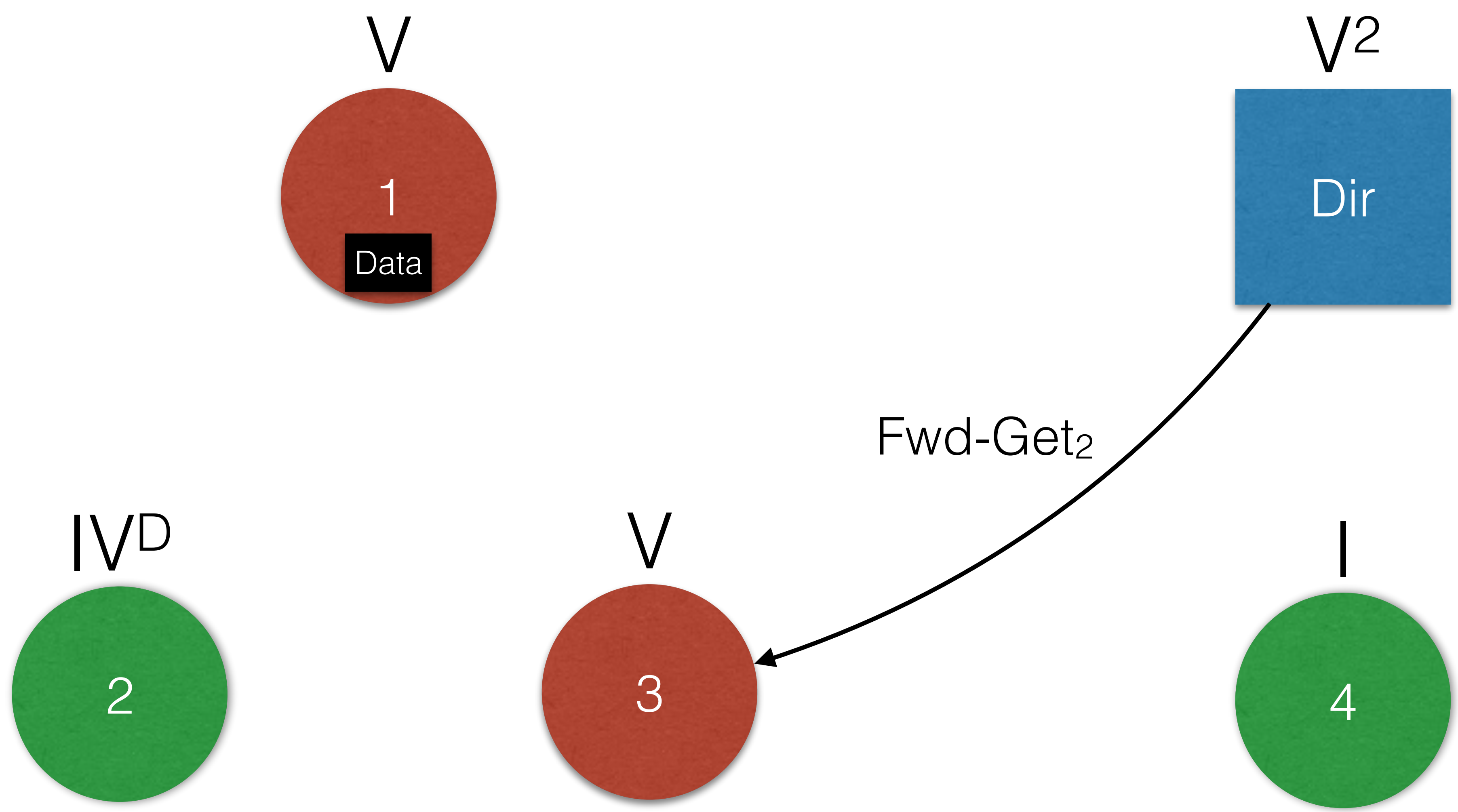
Let’s backtrack and see how this could have happened…

I needed to prove that this could never happen:



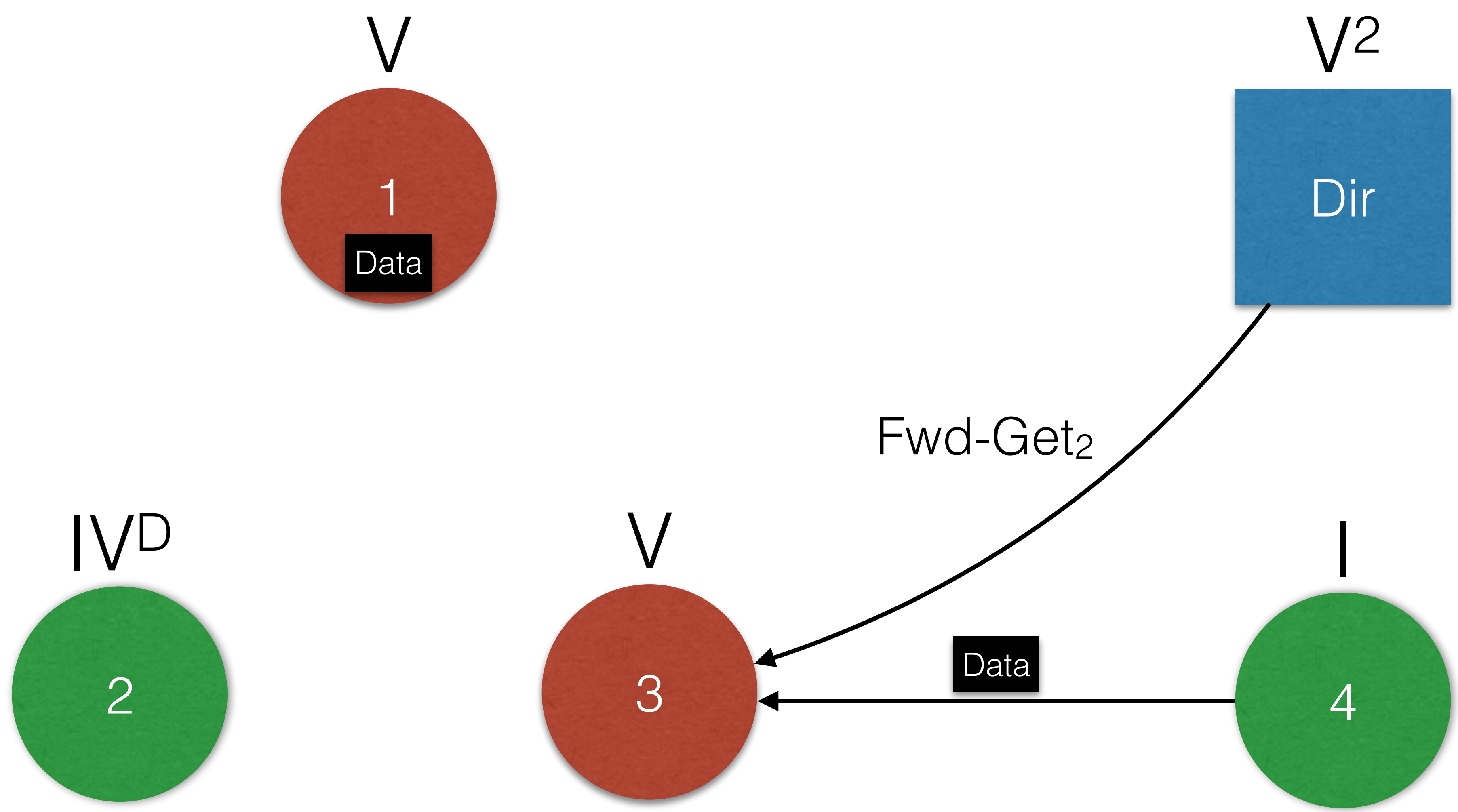
Let’s backtrack and see how this could have happened…

I needed to prove that this could never happen:



Let's backtrack and see how this could have happened…

I needed to prove that this could never happen:



Let's backtrack and see how this could have happened…