Kronos

The Design and Implementation of an Event Ordering Service

Robert Escriva[†]
Ayush Dubey[†] Bernard Wong[‡] Emin Gün Sirer[†]

[†]Department of Computer Science
Cornell University

[‡]School of Computer Science
University of Waterloo

EuroSys '14
April 14, 2014

Distributed Systems

Distributed systems are difficult to build because changes are happening across many computers.

1977: Han shot first

1977: Han shot first
1997: Greedo shot first

We need some way to know who shot first!

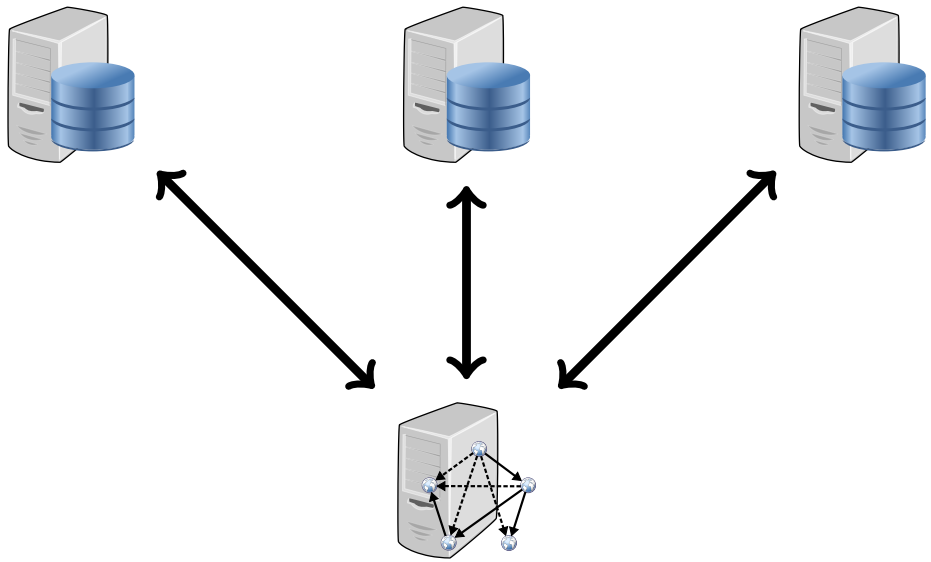
Order in Distributed Systems

- Lamport Timestamps:
  Institute a total order across requests
- Vector Clocks:
  Requires agreement on membership and format
- Consensus Protocols
  Serialized execution which limits concurrency

Kronos: A Time Oracle for Distributed Systems

A time oracle maintains the global timeline for the system:

- Tell the oracle the order in which things happen
- Ask the oracle to recall this information later



Managing Dependencies in a Social Network

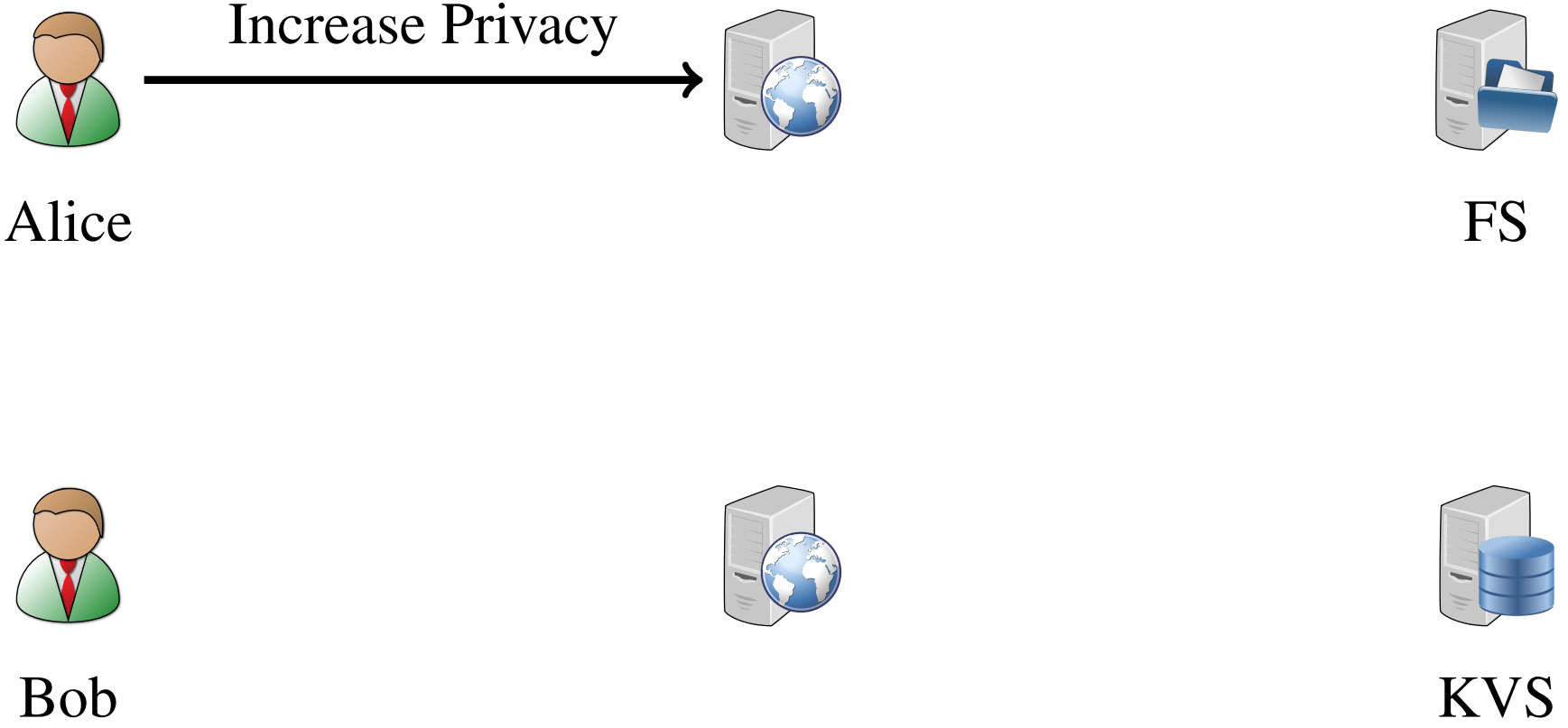
Alice

FS

Bob

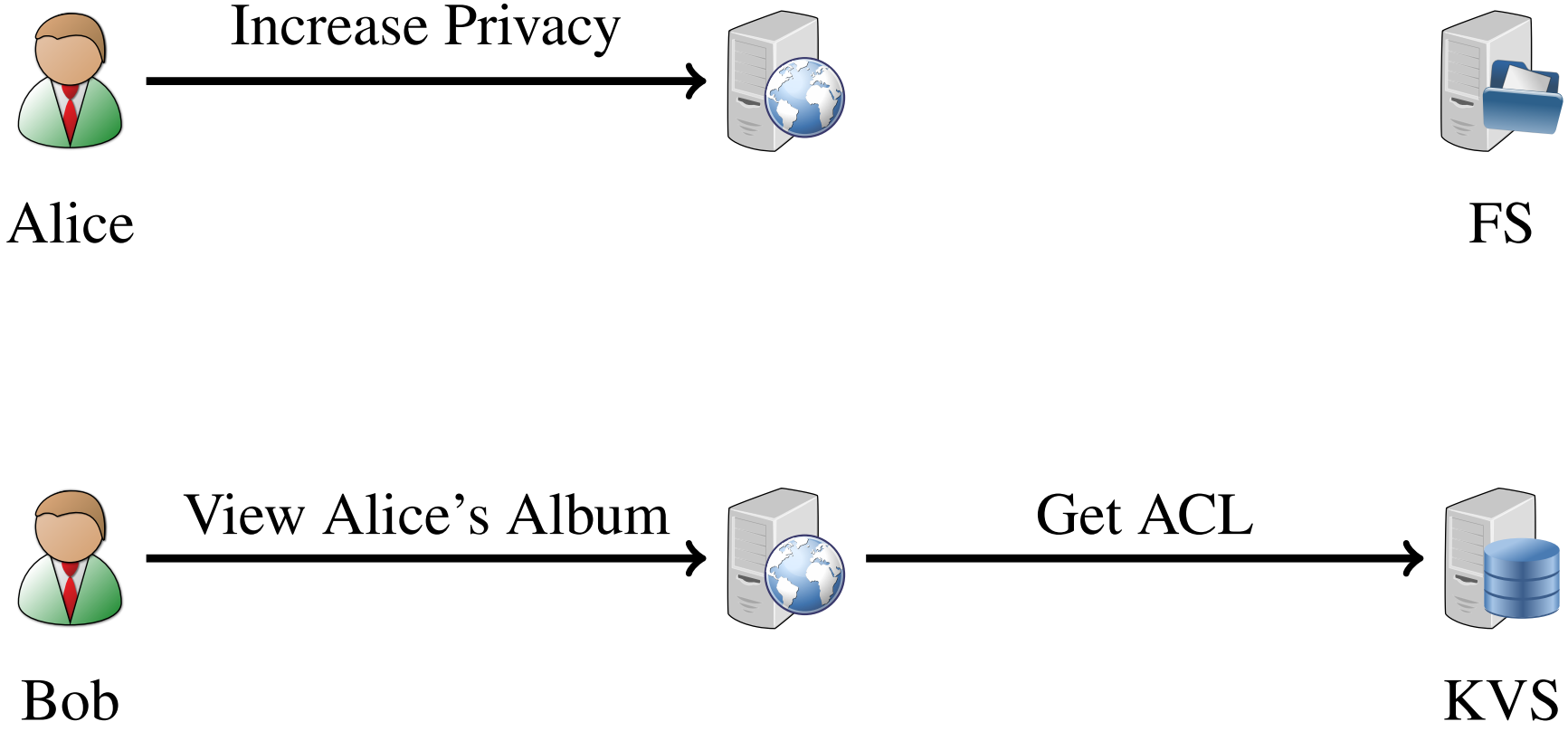
KVS

Managing Dependencies in a Social Network

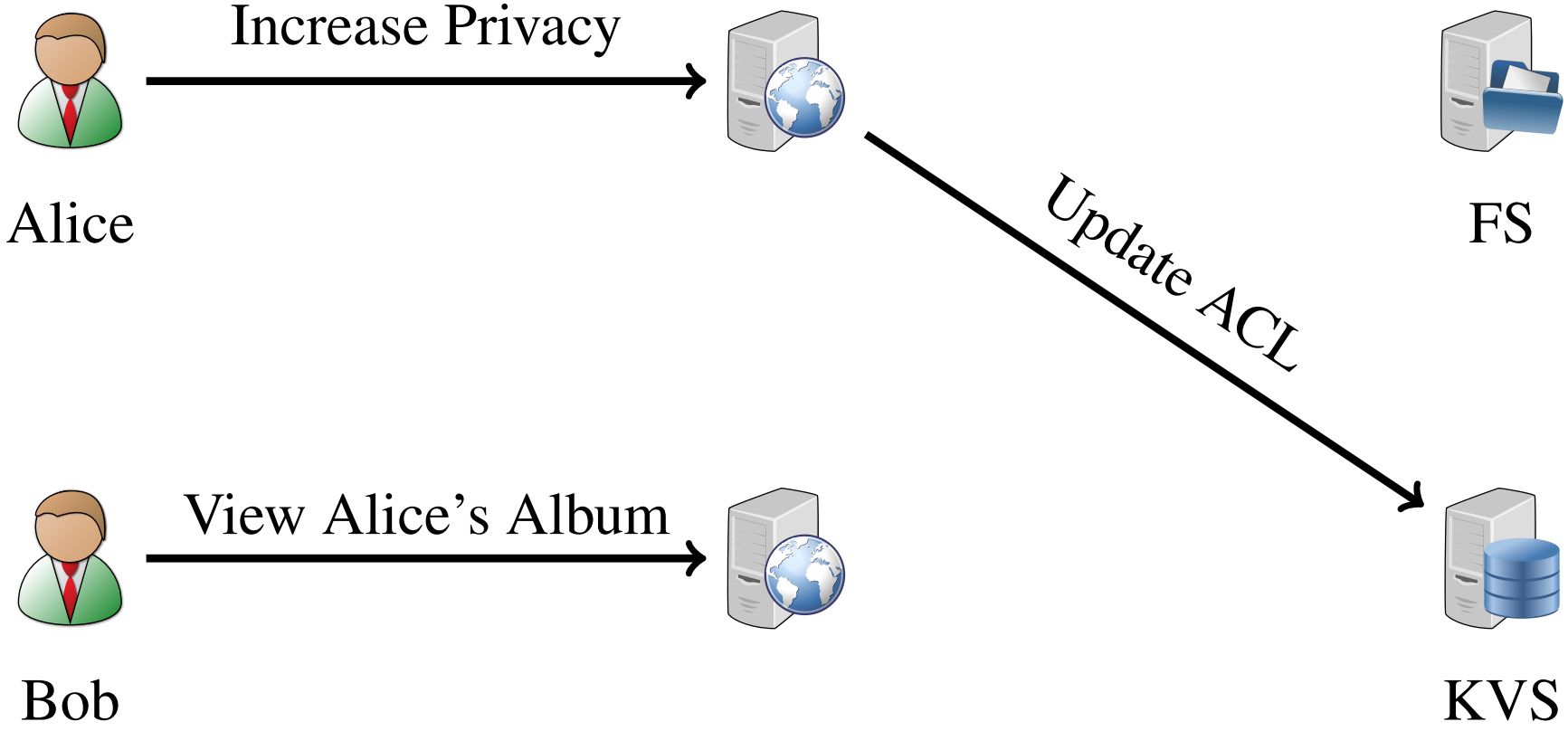


Managing Dependencies in a Social Network

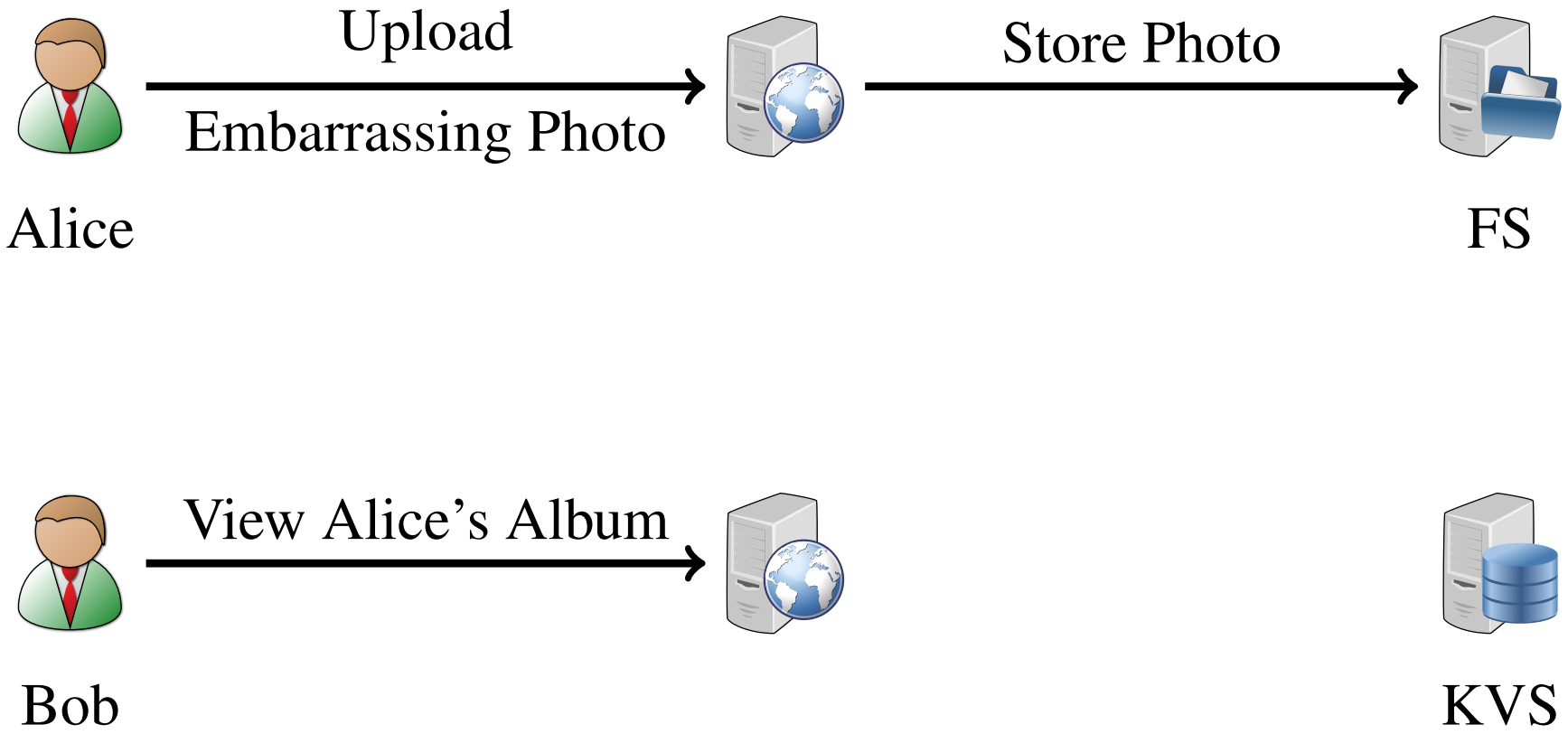
Managing Dependencies in a Social Network



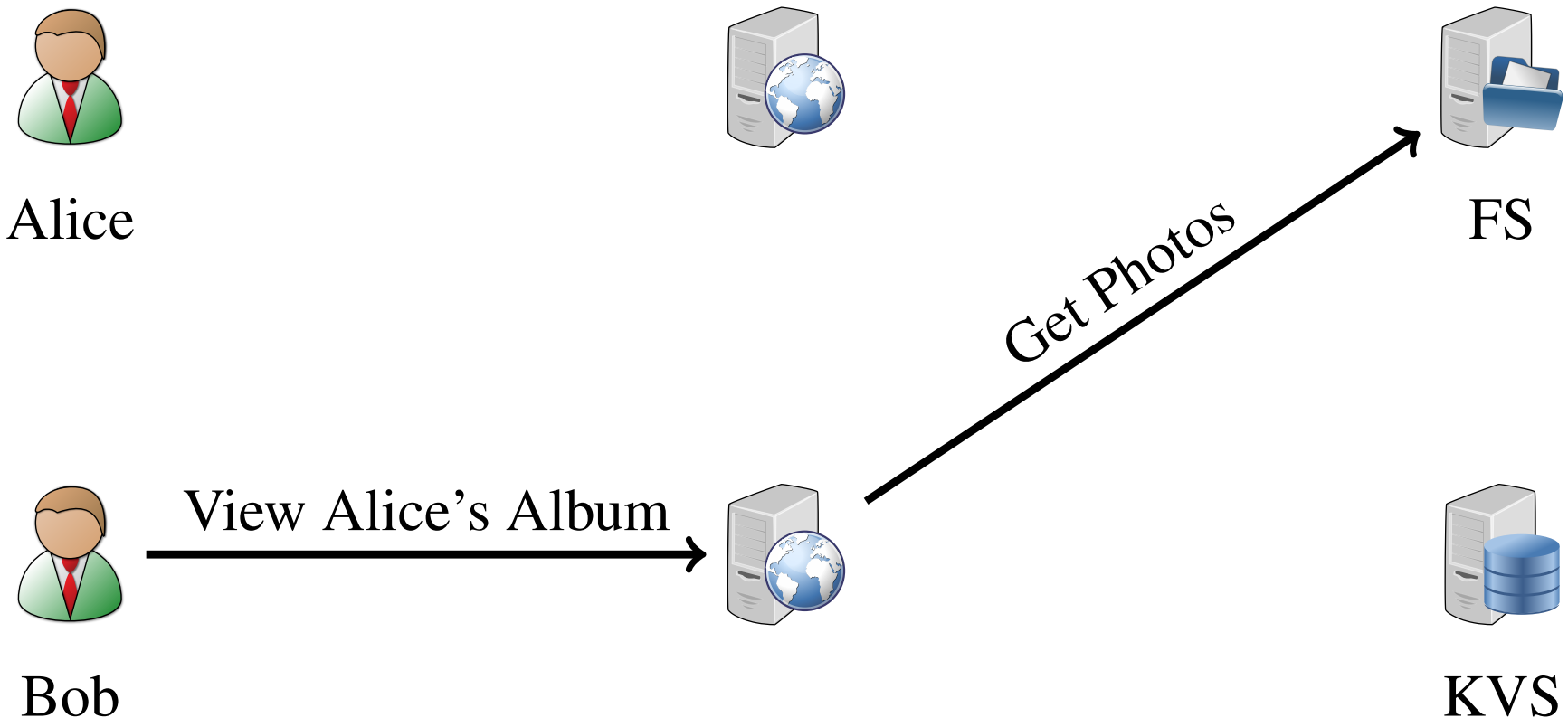
Managing Dependencies in a Social Network



Managing Dependencies in a Social Network



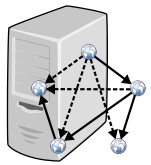
Managing Dependencies in a Social Network



Managing Dependencies in a Social Network

Alice

FS



Kronos

Bob

KVS

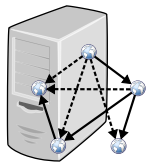
Managing Dependencies in a Social Network

Alice

Increase Privacy

Bob

FS



Kronos

KVS

Managing Dependencies in a Social Network

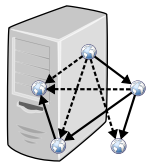
Alice

Increase Privacy

Bob

View Alice's Album

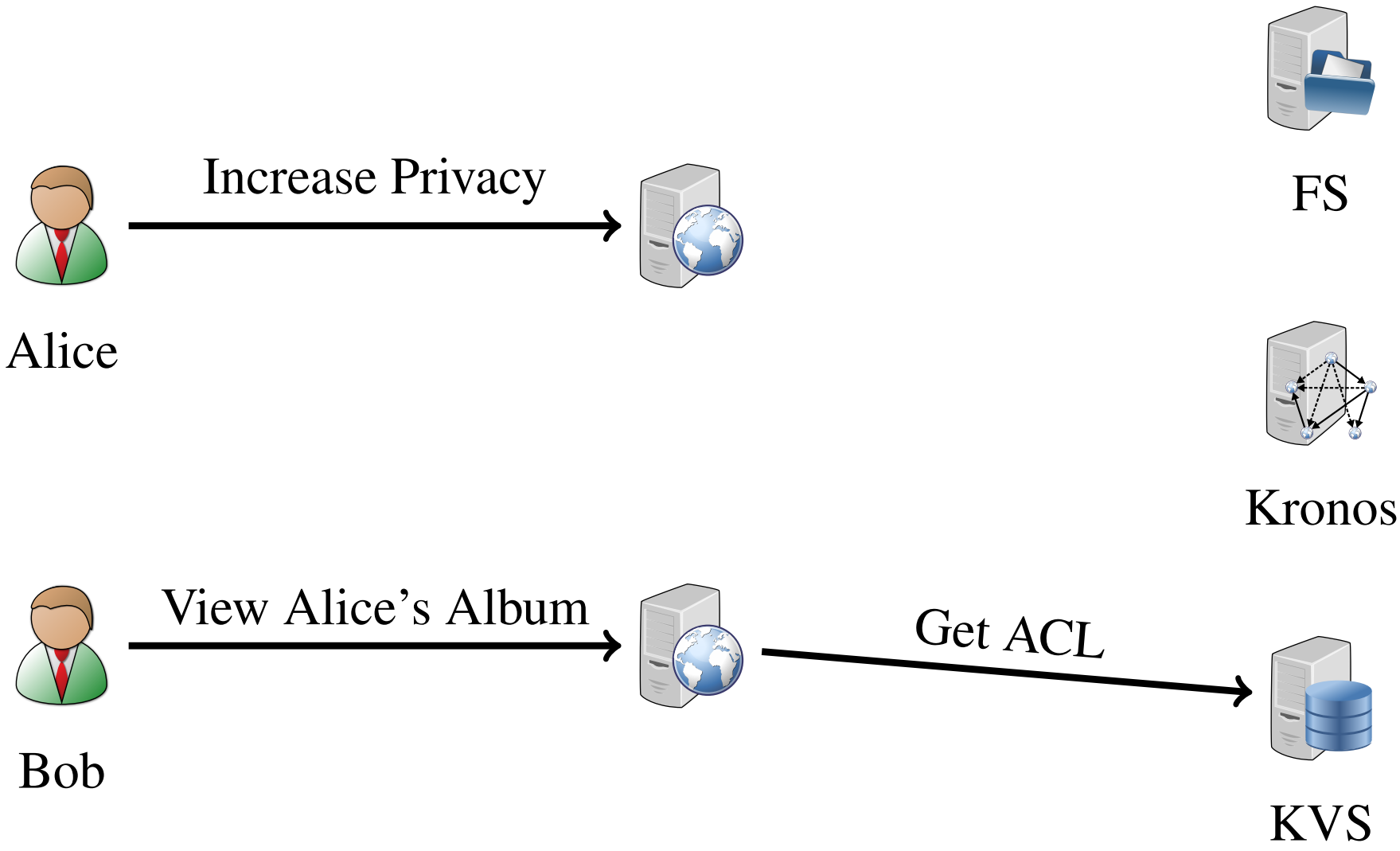
FS



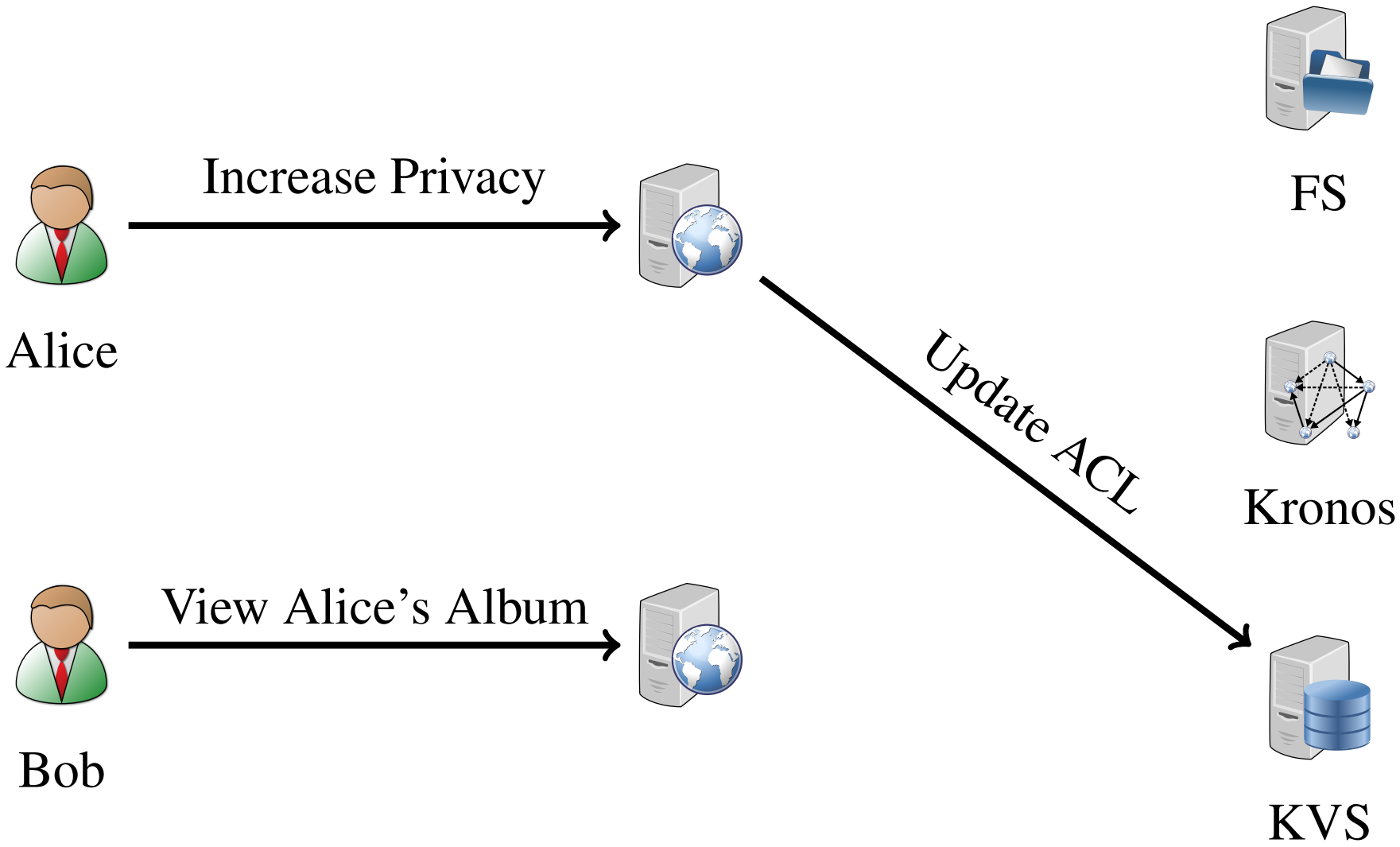
Kronos

KVS

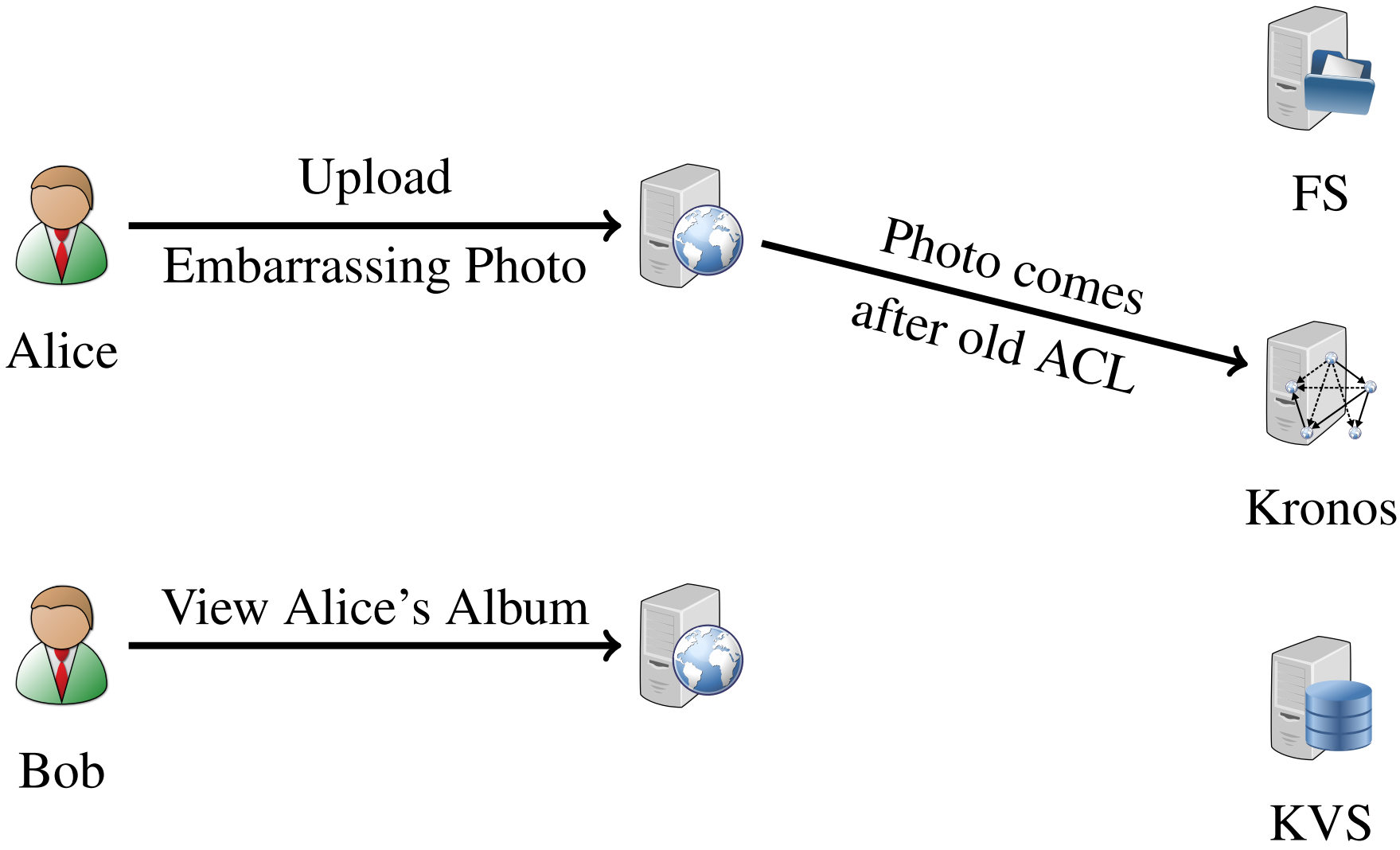
Managing Dependencies in a Social Network



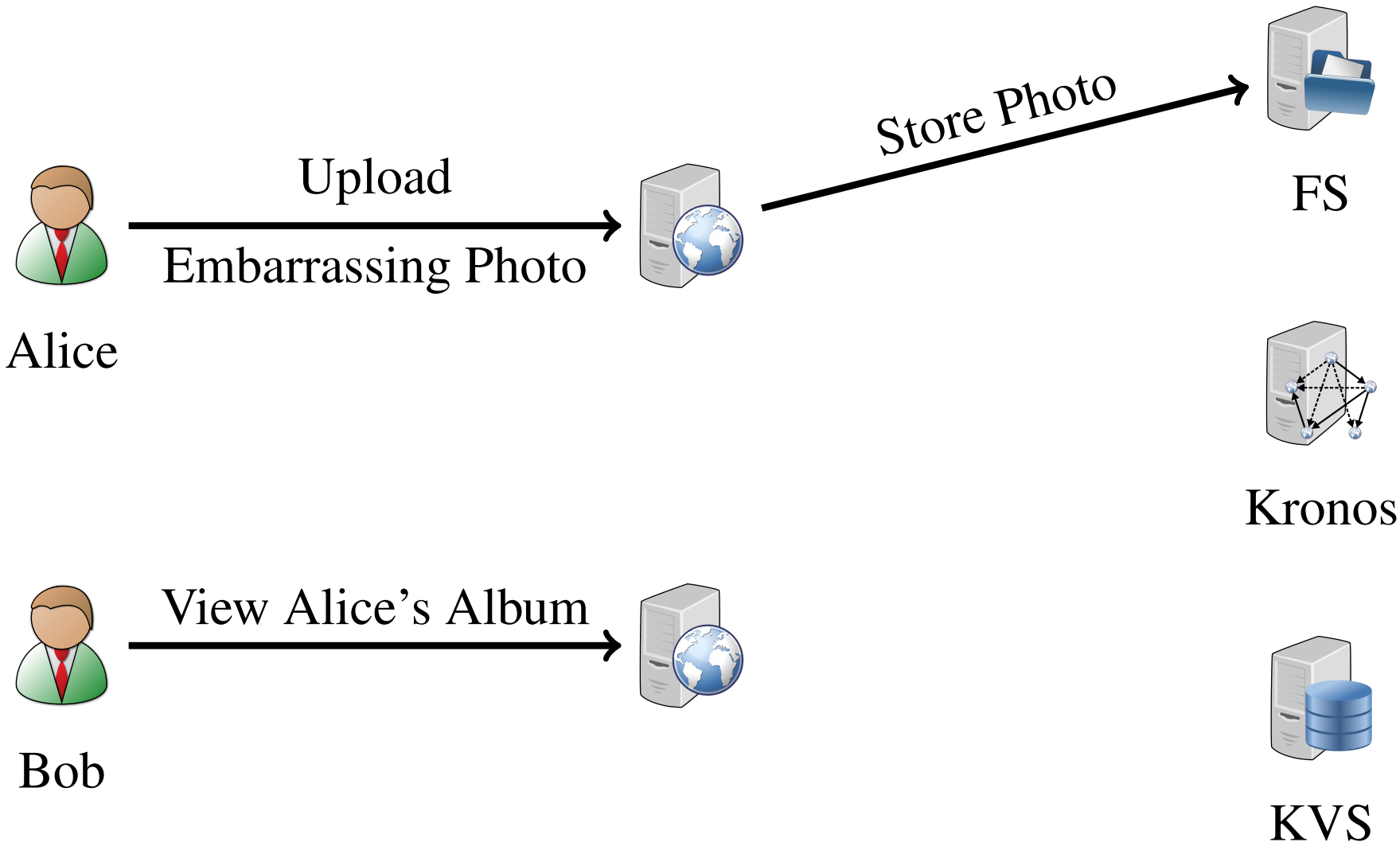
Managing Dependencies in a Social Network



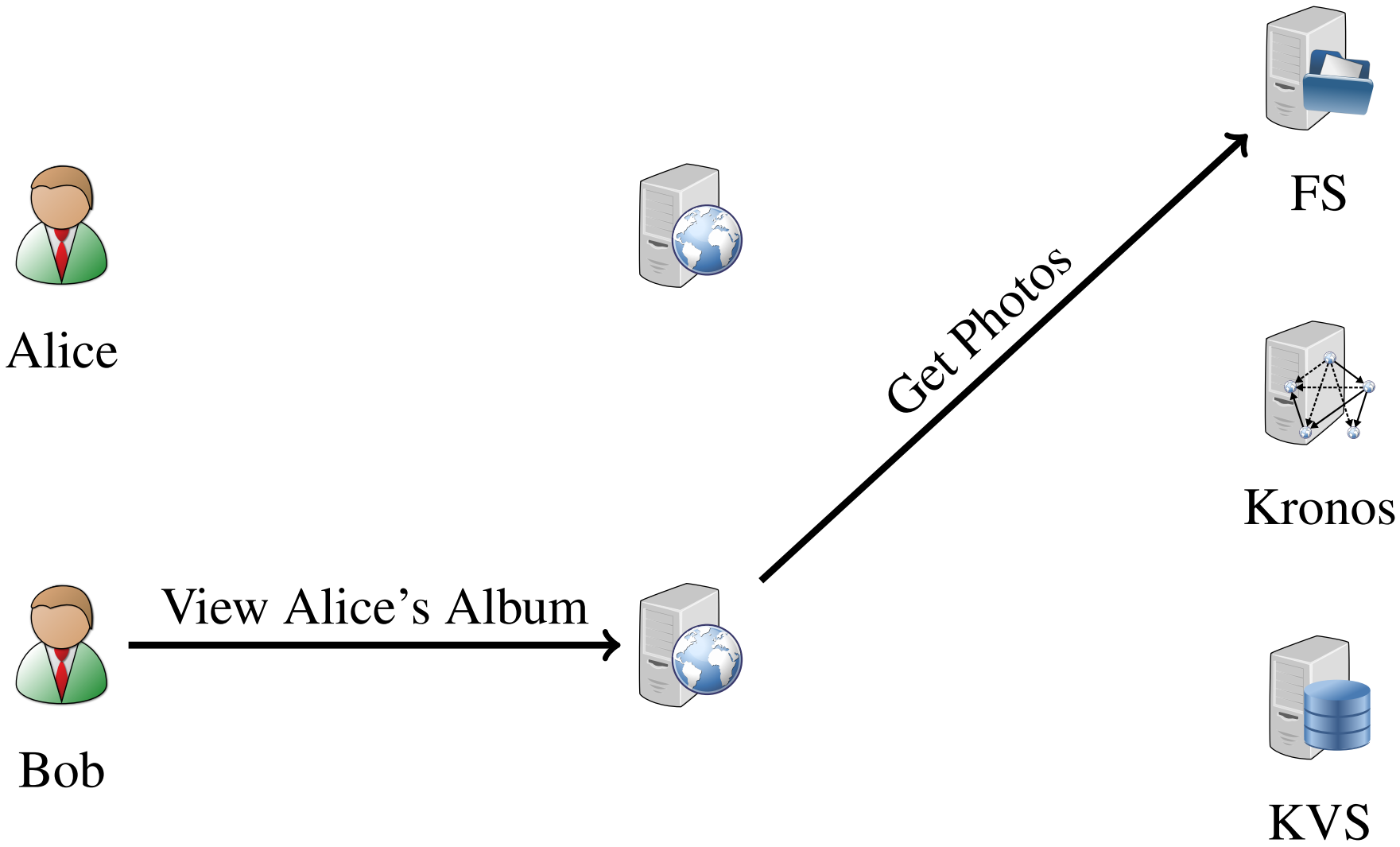
Managing Dependencies in a Social Network



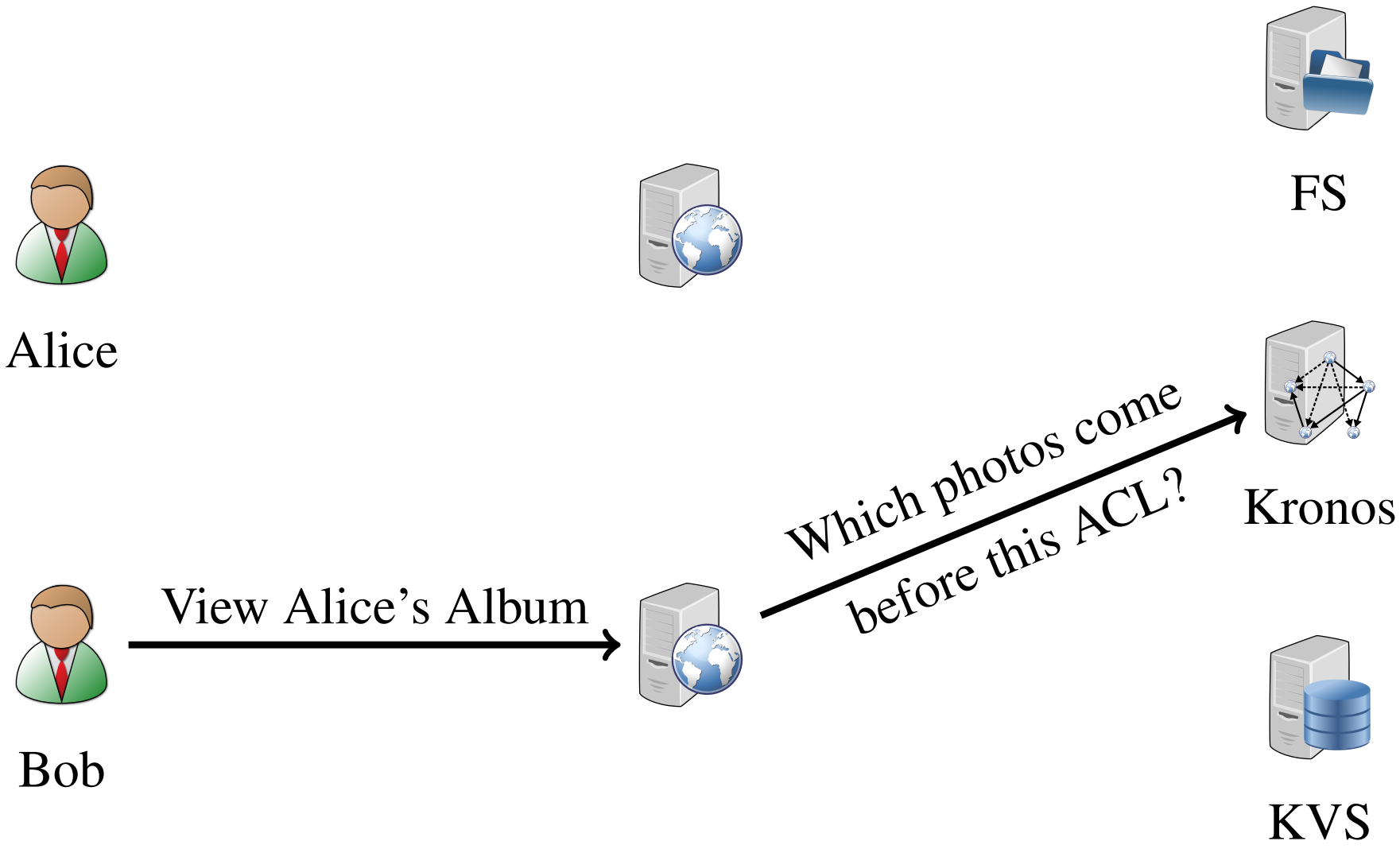
Managing Dependencies in a Social Network



Managing Dependencies in a Social Network



Managing Dependencies in a Social Network

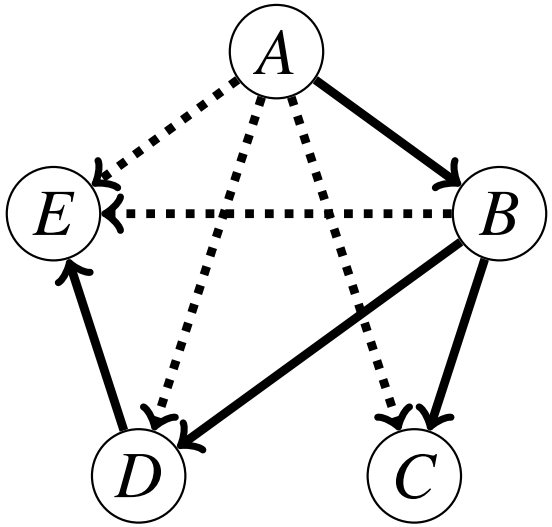


What is an Event?

An event is an application-determined set of state changes that take place atomically, associated with a unique identifier, e.g.:

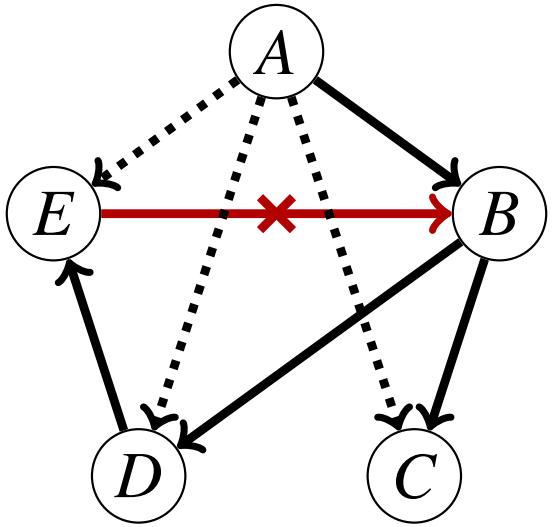
- Reads or writes in a distributed filesystem
- Transactions in a key value store
- Queries on a graph store

The Event Dependency Graph



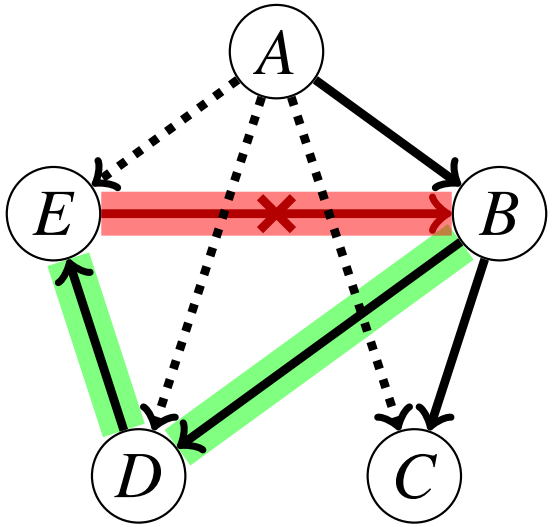
The event dependency graph captures happens-before relationships and enables queries over the timeline.

The Coherency Invariant



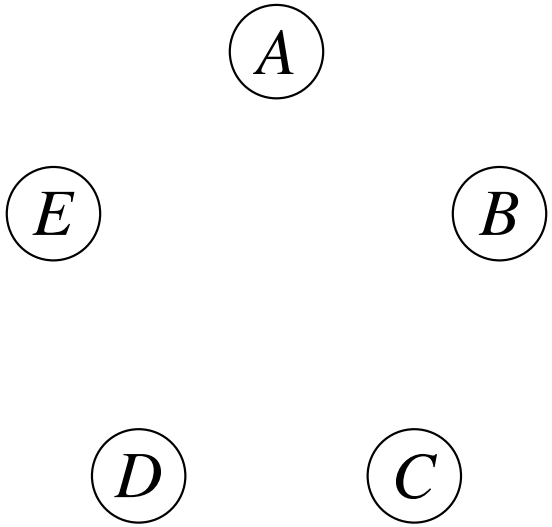
Ensures the timeline makes logical sense by preventing cycles.

The Coherency Invariant



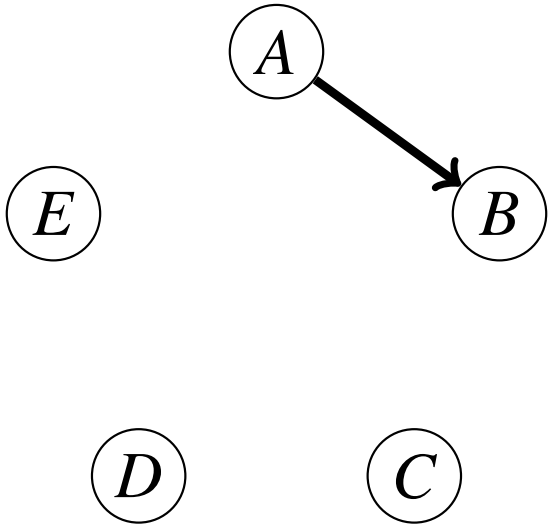
Ensures the timeline makes logical sense by preventing cycles.

The Monotonicity Invariant



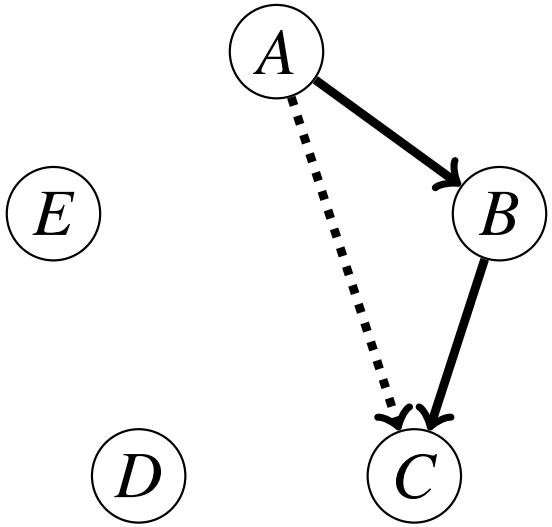
Relationships, once established by Kronos, are incontrovertible.

The Monotonicity Invariant



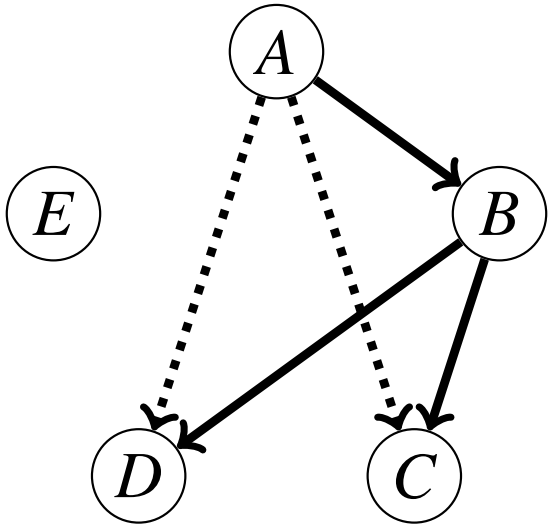
Relationships, once established by Kronos, are incontrovertible.

The Monotonicity Invariant



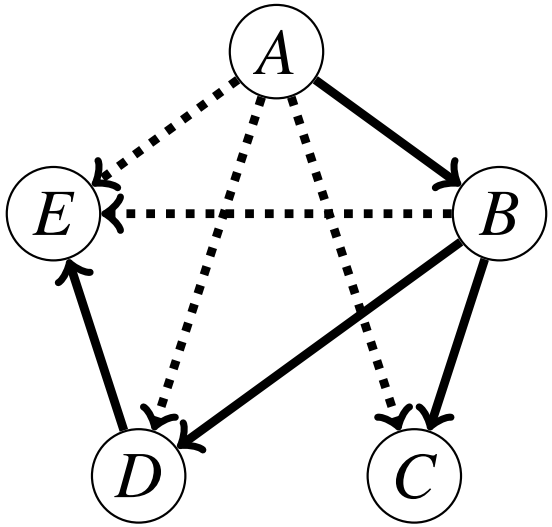
Relationships, once established by Kronos, are incontrovertible.

The Monotonicity Invariant



Relationships, once established by Kronos, are incontrovertible.

The Monotonicity Invariant



Relationships, once established by Kronos, are incontrovertible.

API: Event Creation

- `create_event()`
  $\Rightarrow A$

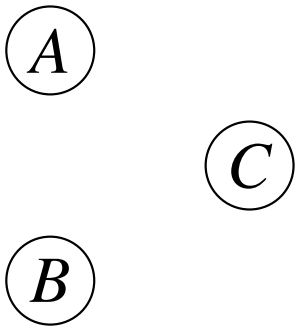
`create_event()`
Create a new event and return a unique identifier e.

API: Event Creation

- `create_event()` $\Rightarrow A$
- `create_event()` $\Rightarrow B$

`create_event()`
Create a new event and return a unique identifier e.

API: Event Creation

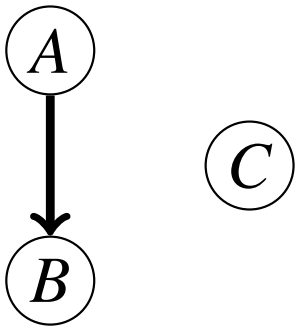


- `create_event()`
  $\Rightarrow A$
- `create_event()`
  $\Rightarrow B$
- `create_event()`
  $\Rightarrow C$

`create_event()`
Create a new event and return a unique identifier e.

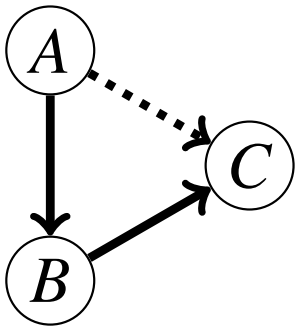
API: Dependency Creation

- `assign_order`(A, B)
  $\Rightarrow$ *True*



`assign_order`(e_i, e_j)
Create the relationship $e_i \rightsquigarrow e_j$ if possible.

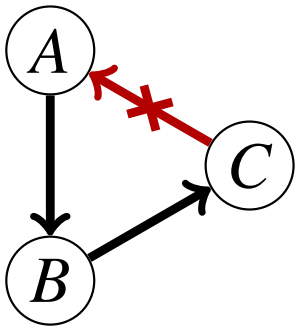
API: Dependency Creation



- `assign_order`(A, B)
  $\Rightarrow$ *True*
- `assign_order`(B, C)
  $\Rightarrow$ *True*

`assign_order`(e_i, e_j)
Create the relationship $e_i \rightsquigarrow e_j$ if possible.

API: Dependency Creation



- `assign_order`(A, B)
  $\Rightarrow$ *True*
- `assign_order`(B, C)
  $\Rightarrow$ *True*
- `assign_order`(C, A)
  $\Rightarrow$ *False*

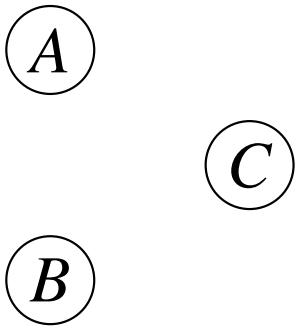
`assign_order`(e_i, e_j)
Create the relationship $e_i \rightsquigarrow e_j$ if possible.

API: Atomic Batch Operations

Batch operations execute atomically. Either all operations complete, or none have any effect.

API: Atomic Batch Operations

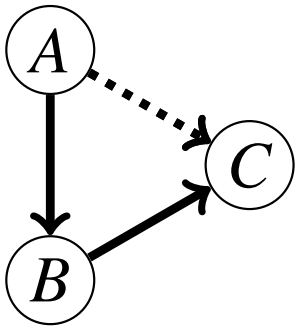
- `create_event(3)`
  $\Rightarrow [A, B, C]$



Batch operations execute atomically. Either all operations complete, or none have any effect.

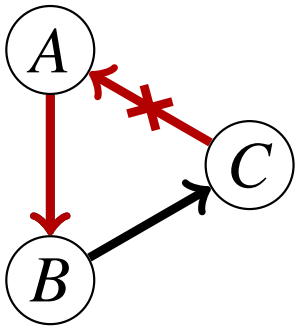
API: Atomic Batch Operations

- `create_event(3)`
  $\Rightarrow [A, B, C]$
- `assign_order([(A,B),(B,C)])`
  $\Rightarrow \mathit{True}$



Batch operations execute atomically. Either all operations complete, or none have any effect.

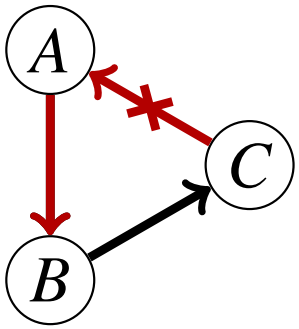
API: Atomic Batch Operations



- `create_event(3)`
  $\Rightarrow [A, B, C]$
- `assign_order([(A, B), (B, C)])`
  $\Rightarrow$ *True*
- `assign_order([(A, B), (C, A)])`
  $\Rightarrow$ *False*

Batch operations execute atomically. Either all operations complete, or none have any effect.

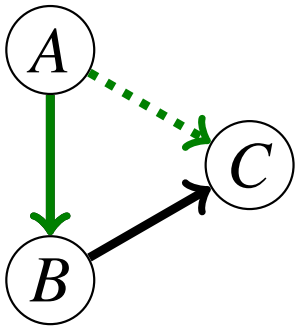
API: Atomic Batch Operations



- `create_event(3)`
  $\Rightarrow [A, B, C]$
- `assign_order([(A, B), (B, C)])`
  $\Rightarrow$ *True*
- `assign_order([(A, B), (C, A)])`
  $\Rightarrow$ *False*

By default, all dependencies in a batch **must** be consistent with the timeline, or the operation and batch will fail.

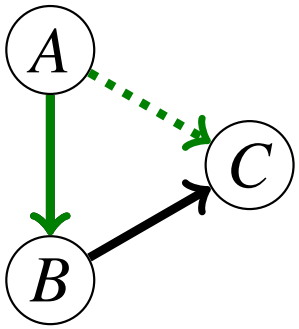
API: Atomic Batch Operations



- `create_event(3)`
  $\Rightarrow [A, B, C]$
- `assign_order([(`A`,`B`),(`B`,`C`)])`
  $\Rightarrow$ *True*
- `assign_order([(`A`,`B`),(`C`,`A`)])`
  $\Rightarrow$ *False*
- `assign_order([(`A`,`B`),`
  `(`C`,`A`,prefer)])`
  $\Rightarrow$ *True*

Applications may specify individual dependencies with a **preferred** ordering that Kronos may reverse if necessary.

API: Atomic Batch Operations

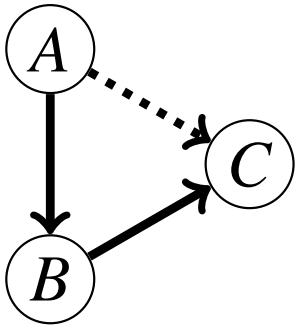


- `create_event(3)`
  $\Rightarrow [A, B, C]$
- `assign_order([(`A`,`B`),(`B`,`C`)])`
  $\Rightarrow$ *True*
- `assign_order([(`A`,`B`),(`C`,`A`)])`
  $\Rightarrow$ *False*
- `assign_order([(`A`,`B`),`
  `(`C`,`A`,prefer)])`
  $\Rightarrow$ *True*

A dependency with a **preferred** ordering will never cause the batch to fail, as Kronos may always align the dependency with the graph.

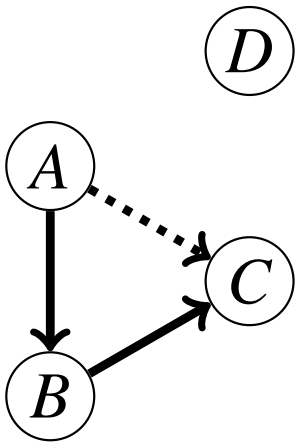
API: Query Operations

- `query_order(A,B);`
  $A \rightsquigarrow B$



Queries discover happens-before relationships within the graph by performing a standard breadth-first search (BFS).

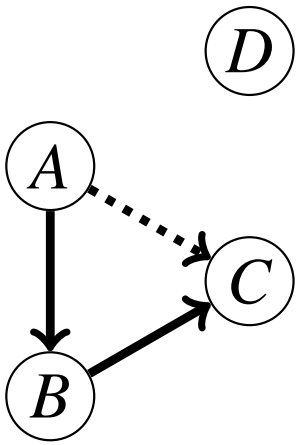
API: Query Operations



- `query_order(A,B);`
  $A \rightsquigarrow B$
- `query_order(A,D);`
  $A \; ? \; D$ // concurrent

If there is no happens-before relationship between two events, Kronos will return that the events are concurrent.

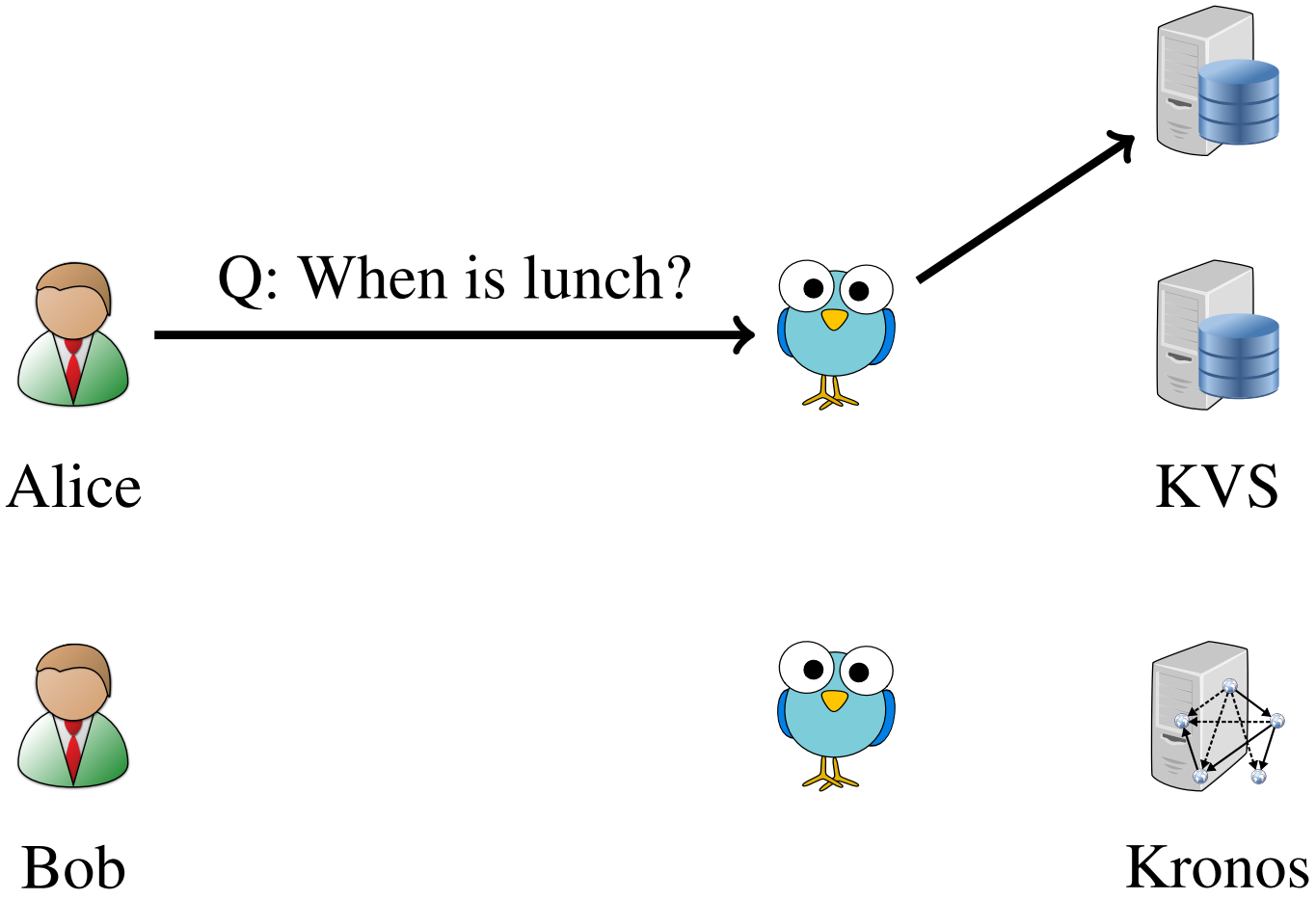
API: Query Operations



- `query_order(A,B);`
  $A \rightsquigarrow B$
- `query_order(A,D);`
  $A \ ?\ D$ // concurrent
- `query_order([(A,B),(C,A)]);`
  $[A \rightsquigarrow B,\ \ A \rightsquigarrow C]$

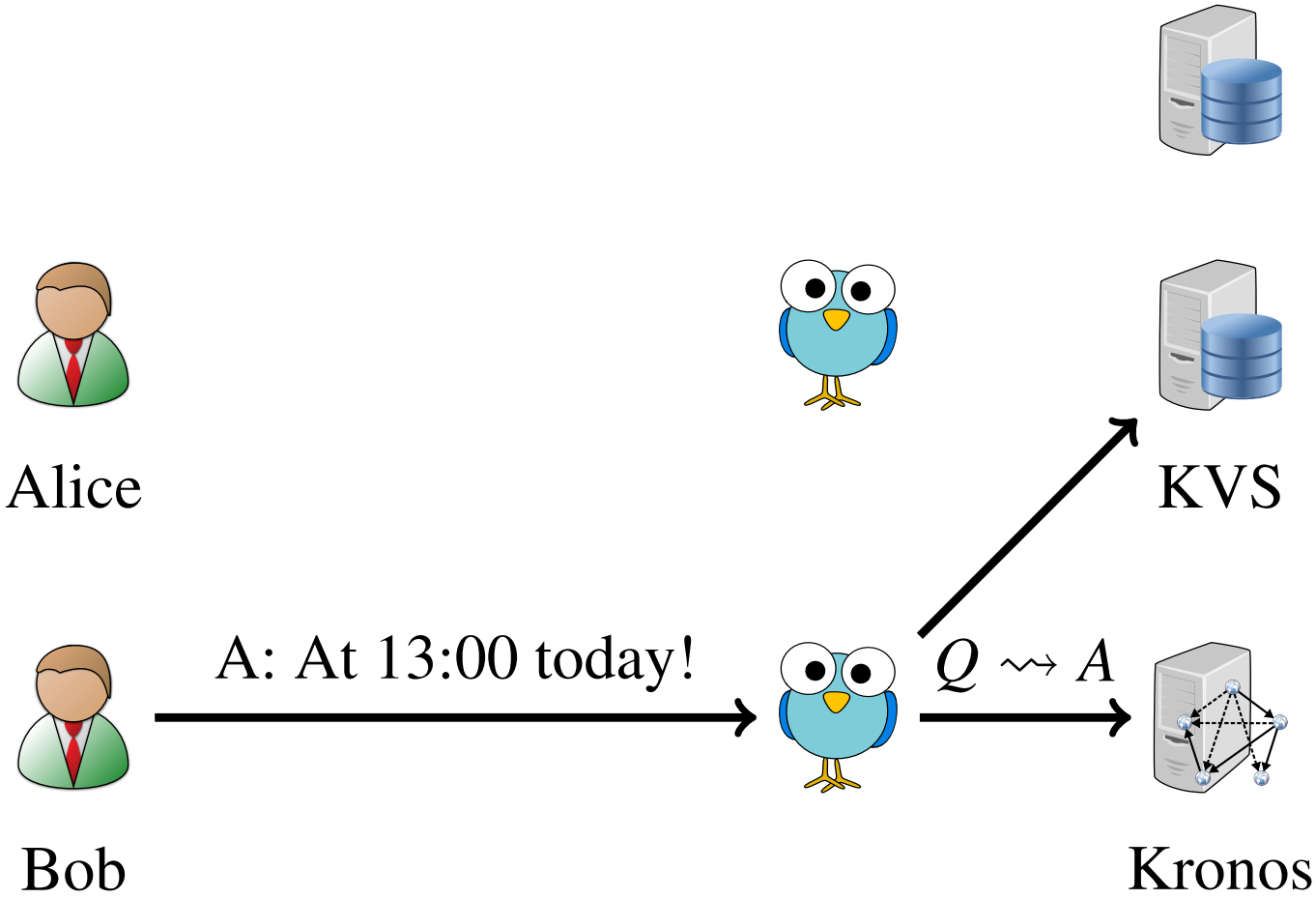
Queries may be submitted in bulk, retrieving multiple results simultaneously.

Twitter Clone



Alice posts a message to the social network, which stores it in the key-value store.

Twitter Clone



Bob's reply gets stored in the key-value store, and its dependency on Alice's message gets stored in Kronos.

Twitter Clone: Posting

```
def post_message(user, message):
    e = kronos.create_event()
    for friend in friends_of(user):
        enqueue_in_timeline(timeline=friend,
                            source=user,
                            message=message,
                            event=e)
```

Twitter Clone: Replying

```
def reply_to_message(user, message,
                     in_reply_to):
    e = kronos.create_event()
    kronos.assign_order(
        [(in_reply_to, '->', e, 'must')])
    for friend in friends_of(user):
        enqueue_in_timeline(timeline=friend,
                            source=user,
                            message=message,
                            event=e)
```

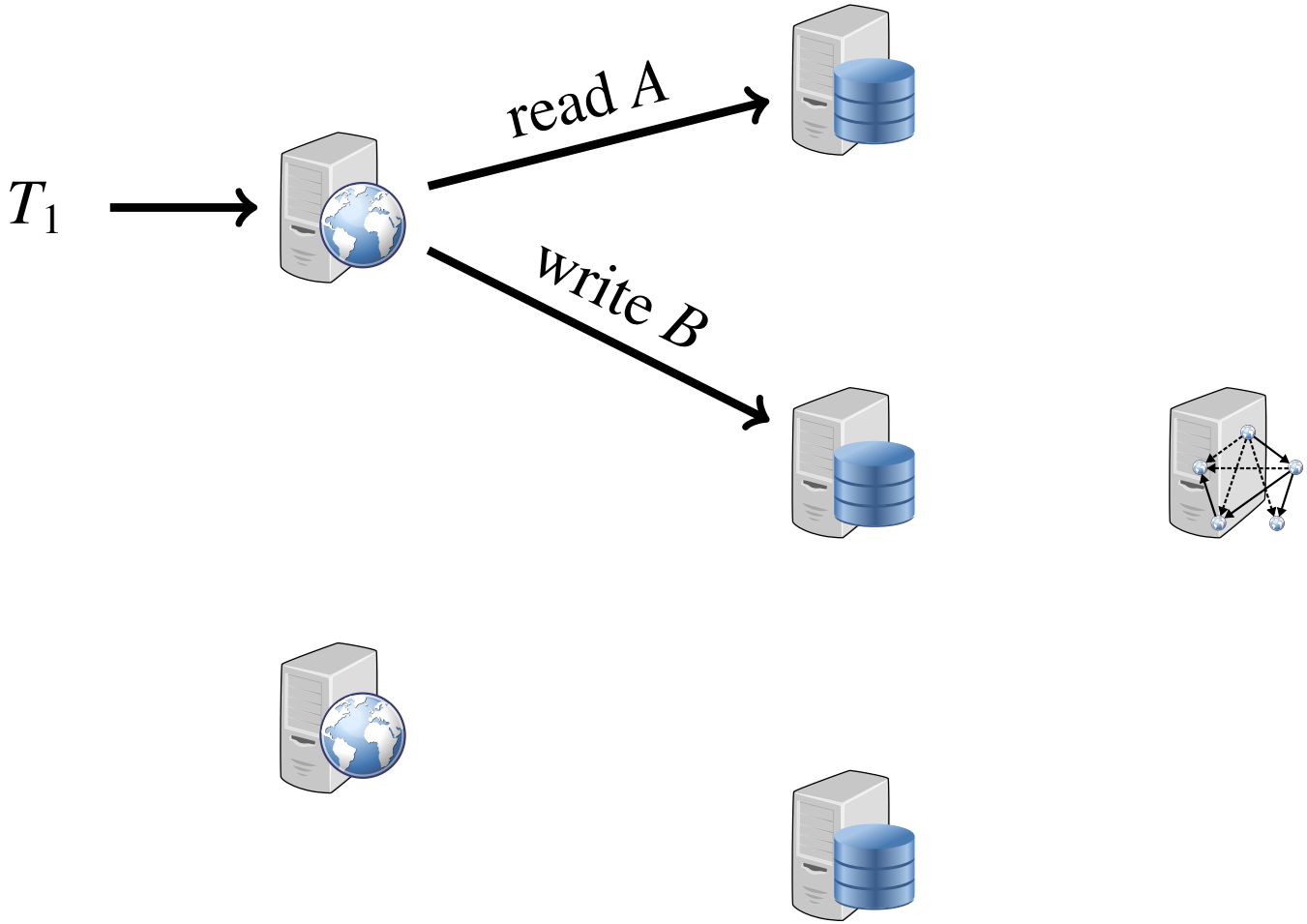
Twitter Clone: Displaying

```
def render_timeline(user):
    messages = get_enqueued_for(timeline=user)
    pairs = all_pairs([m.id for m in messages])
    orderings = kronos.query_order(pairs)
    return topological_sort(messages, orderings)
```

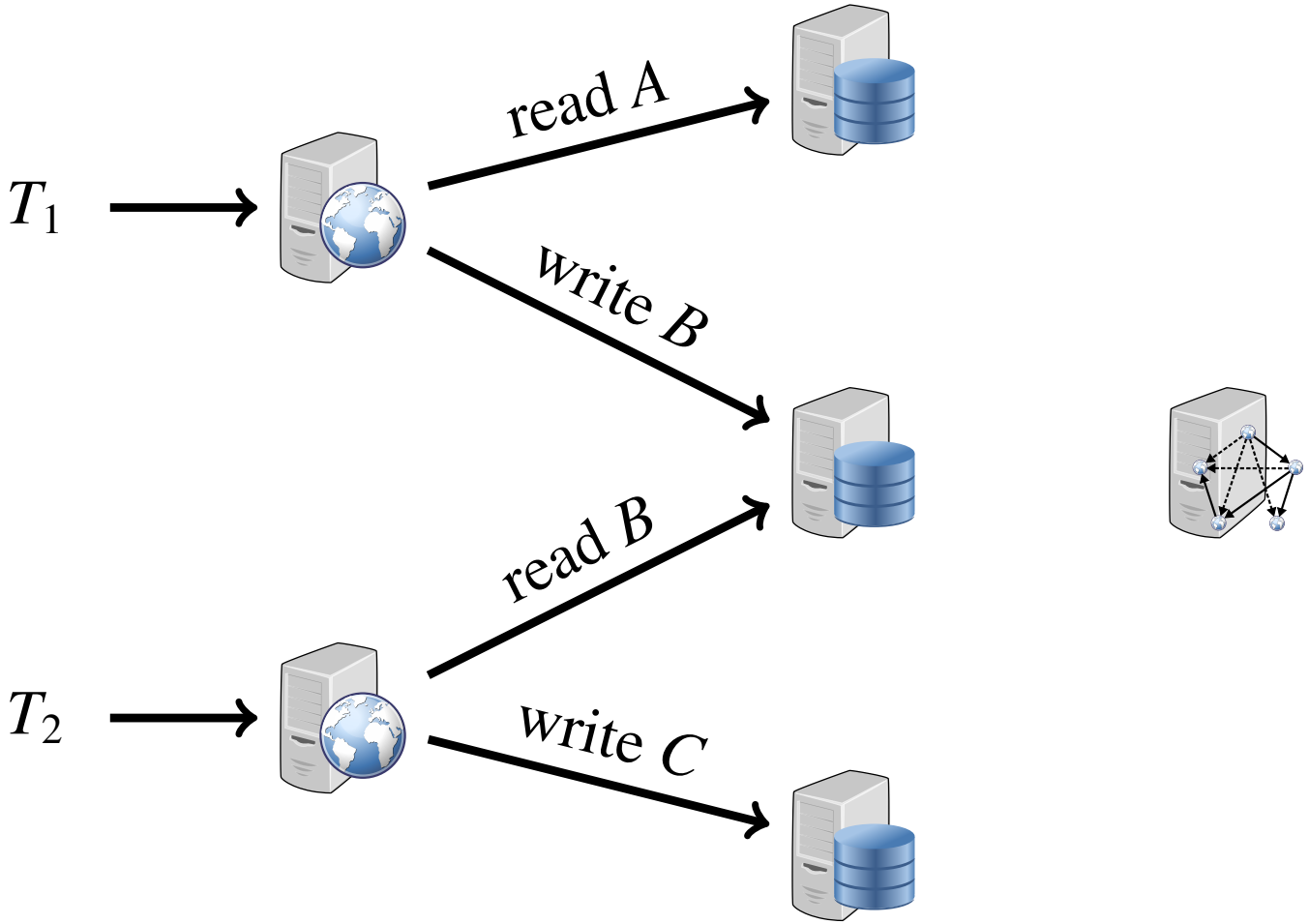
Transactional Key-Value Store

- ACID transactions: update multiple objects atomically
- Transactions that read/write the same keys ordered by Kronos
- Kronos ensures a serializable order across all transactions

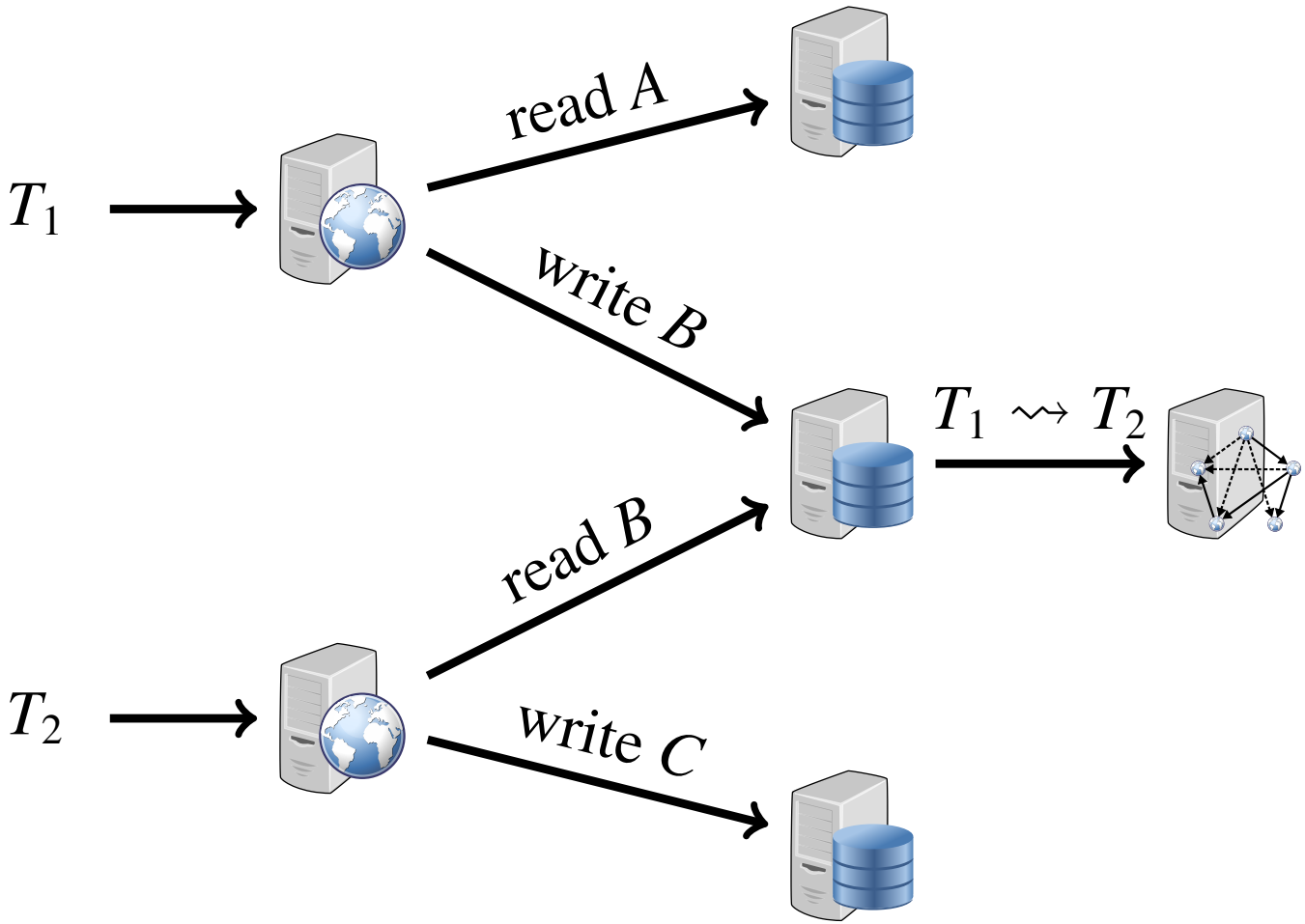
Transactional Key-Value Store



Transactional Key-Value Store



Transactional Key-Value Store

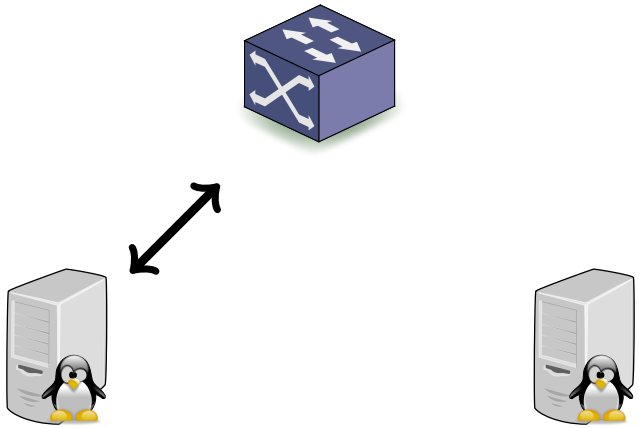


KronoGraph

KronoGraph is an online graph store.

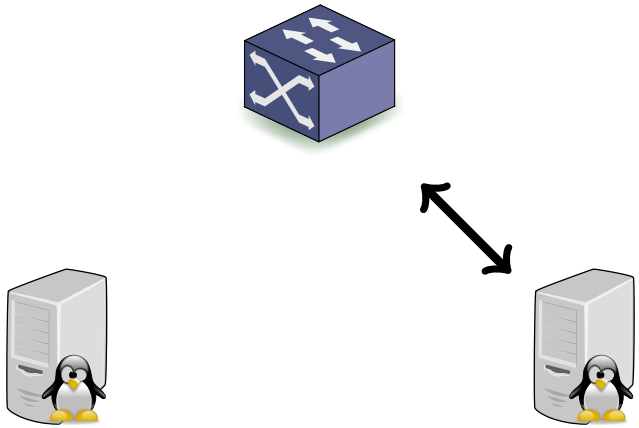
- The graph may change as queries execute
- The correctness of queries relies upon seeing a correct state of the graph
- Because the graph may be quite big, queries or updates could span multiple hosts

KronoGraph Motivation



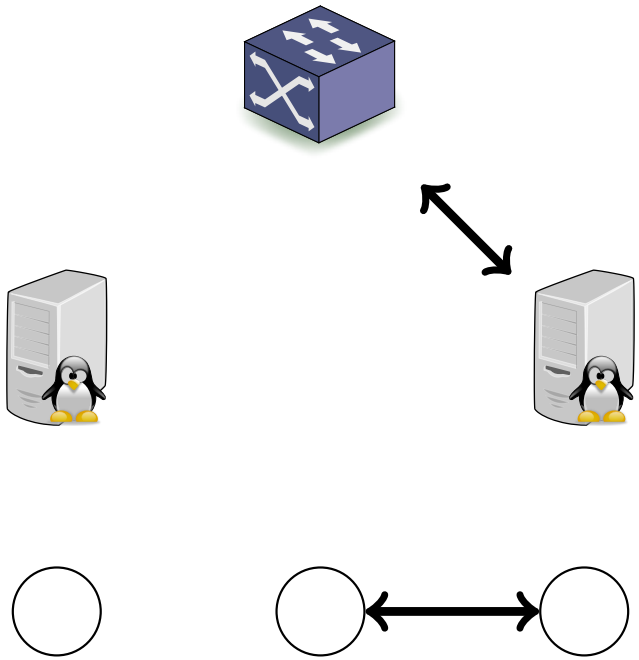
In a networked environment, it sometimes is necessary to change the network configuration.

KronoGraph Motivation



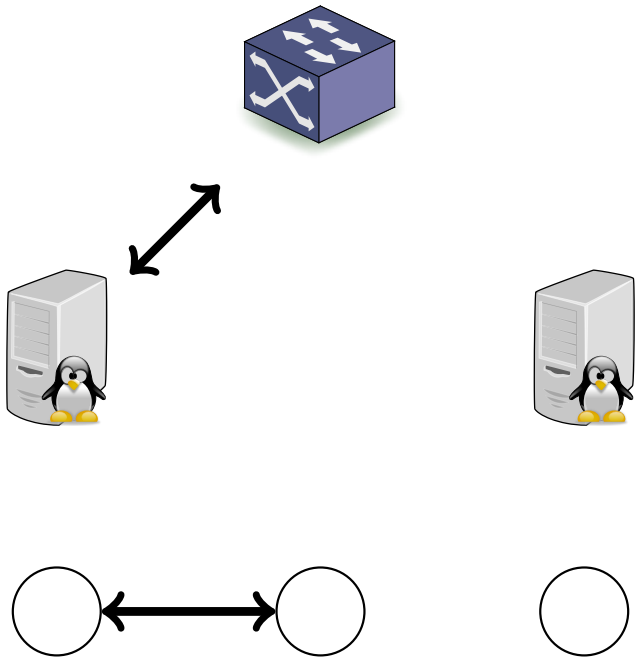
In a networked environment, it sometimes is necessary to change the network configuration.

KronoGraph Motivation



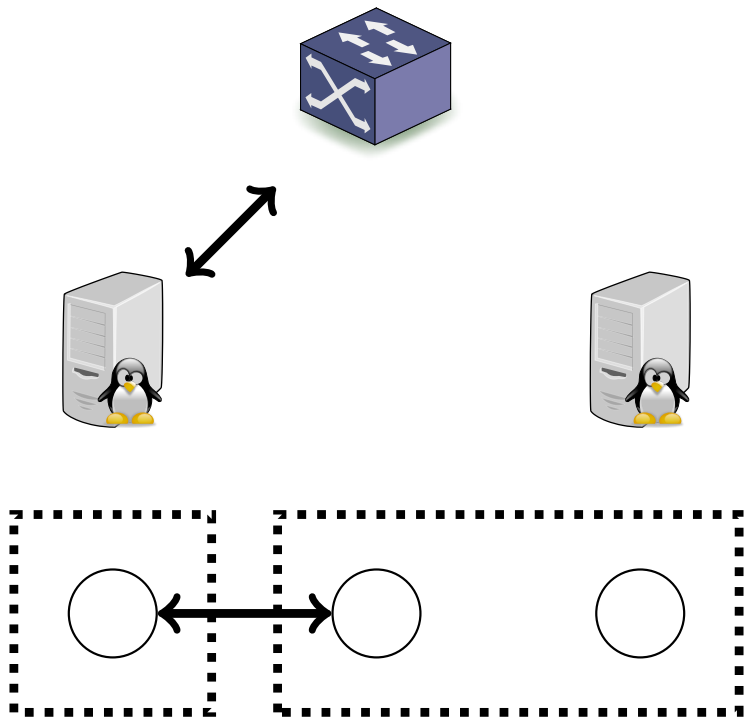
The control platform for this environment could store the topology in a graph store for easy route discover.

KronoGraph Motivation



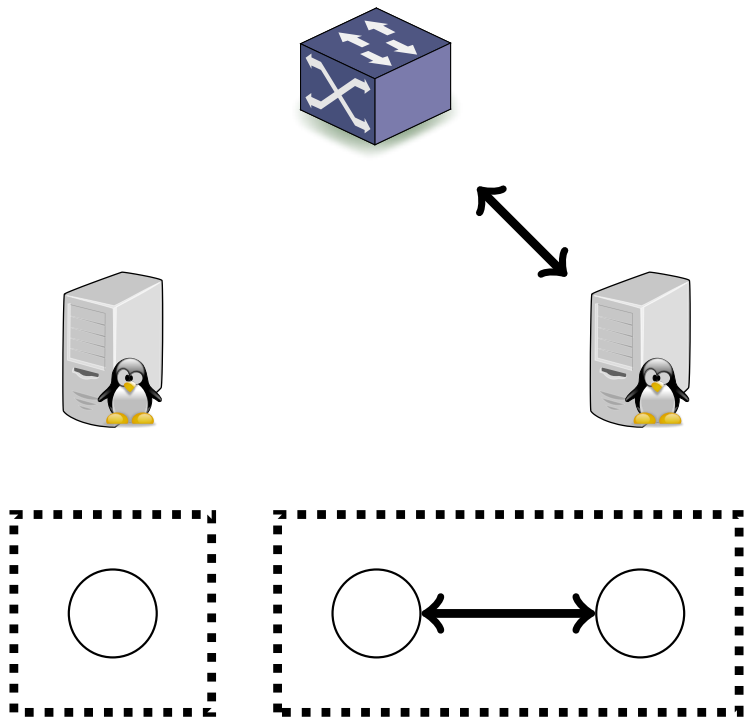
Such topology changes could be atomic, otherwise it would be possible to discover routes that never existed.

KronoGraph Motivation



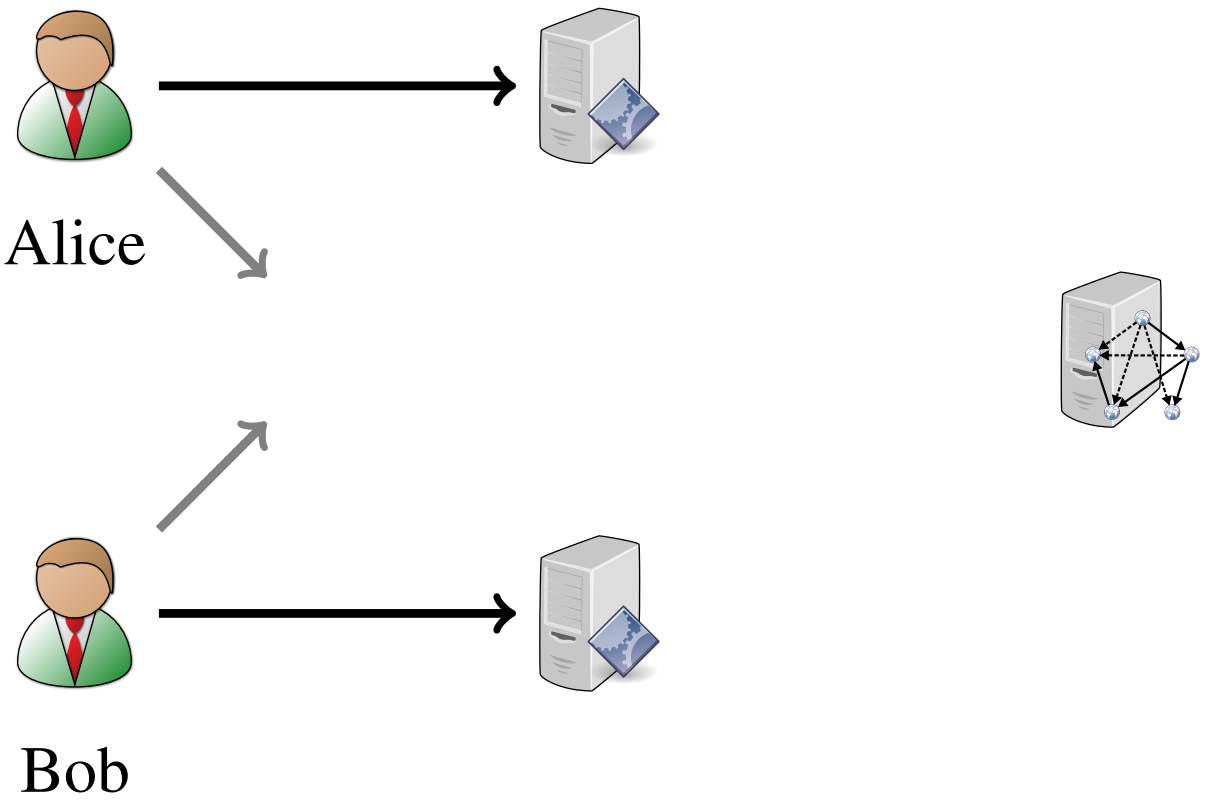
Such topology changes could be atomic, otherwise it would be possible to discover routes that never existed.

KronoGraph Motivation



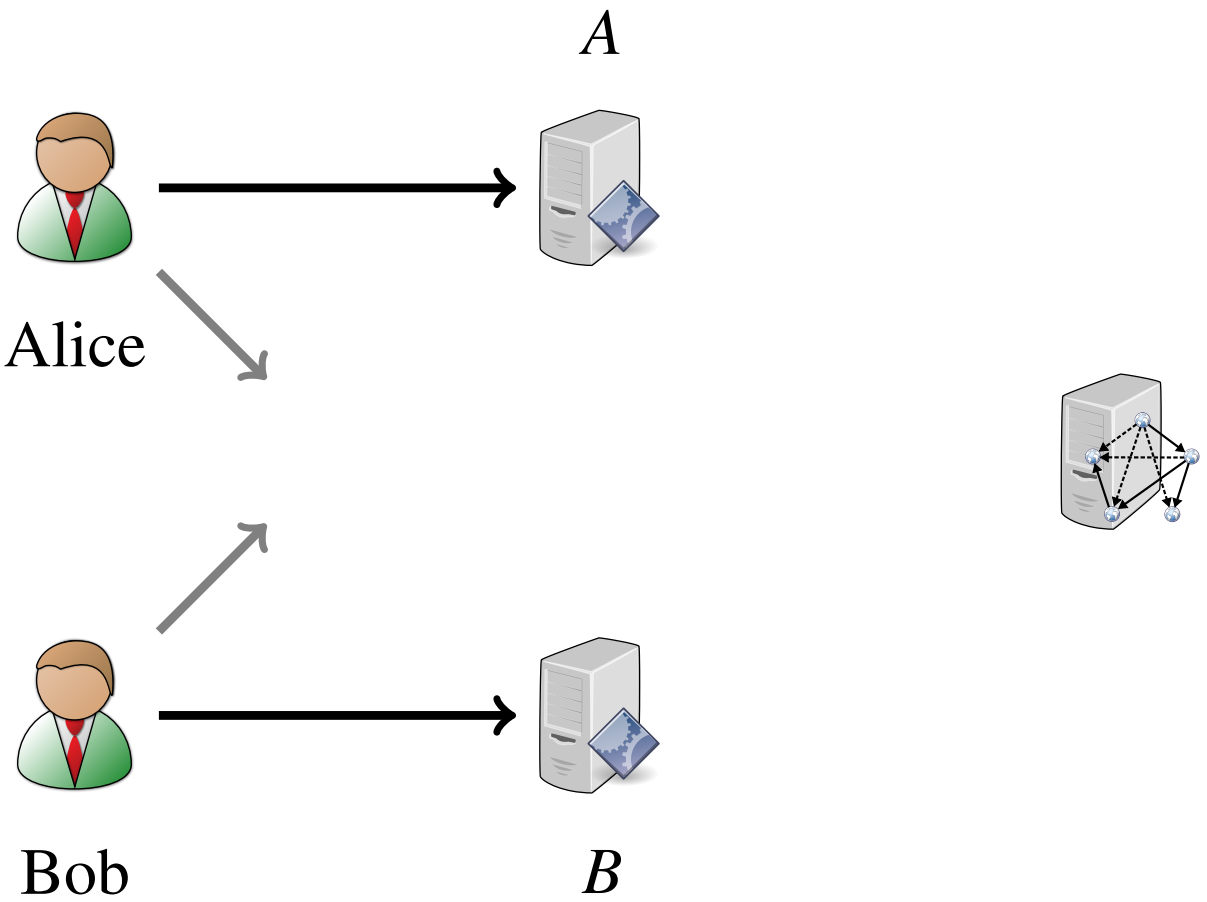
If the graph changes while the query traverses across the shard boundaries, it could arrive after the change to the graph is made.

KronoGraph Example



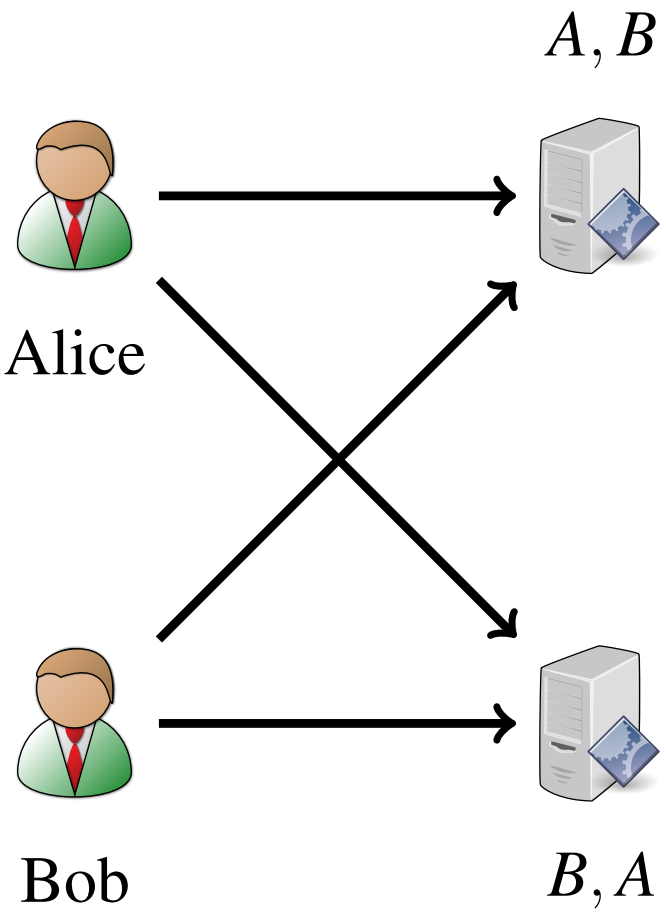
Clients issuing requests to multiple servers can see these requests cross and re-order in the network.

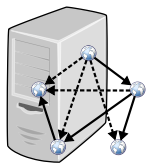
KronoGraph Example



Clients issuing requests to multiple servers can see these requests cross and re-order in the network.

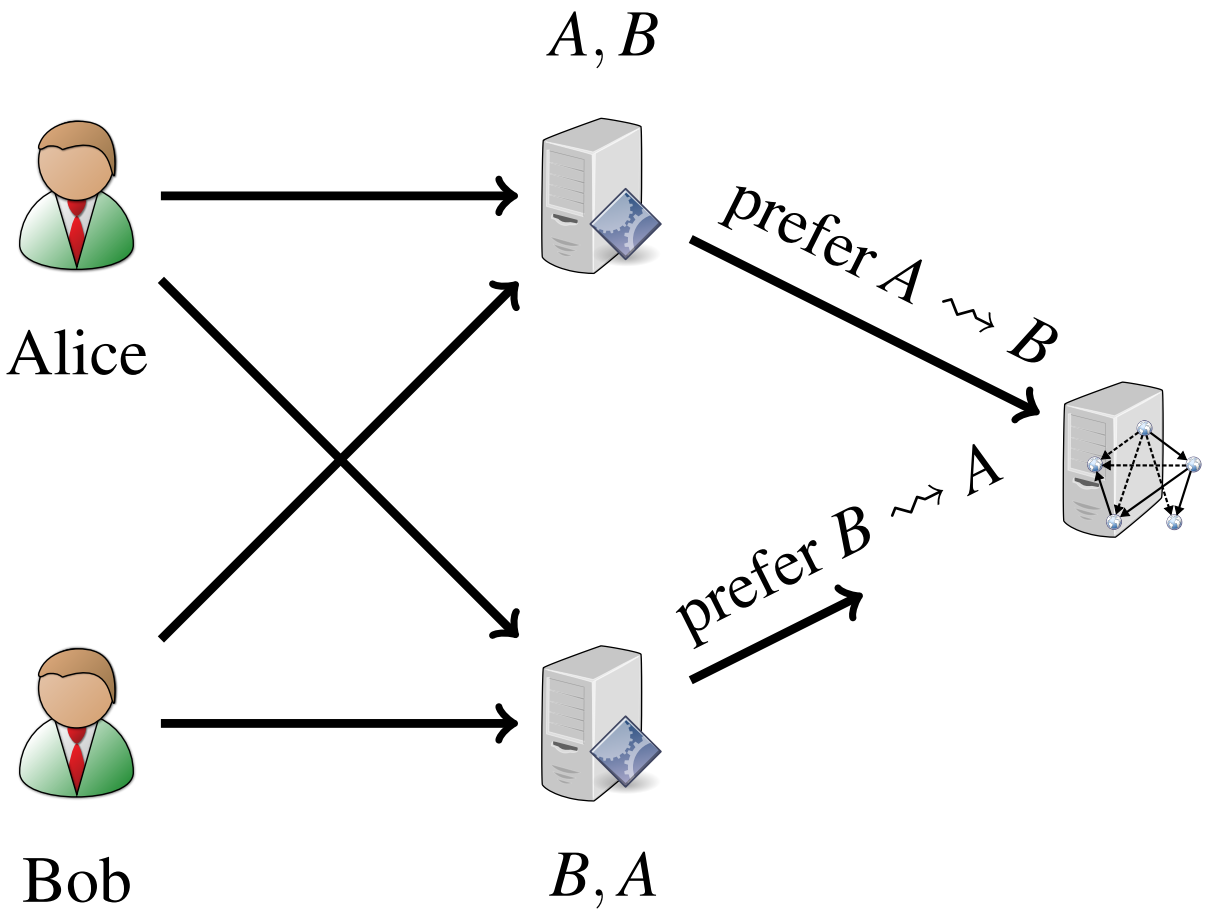
KronoGraph Example





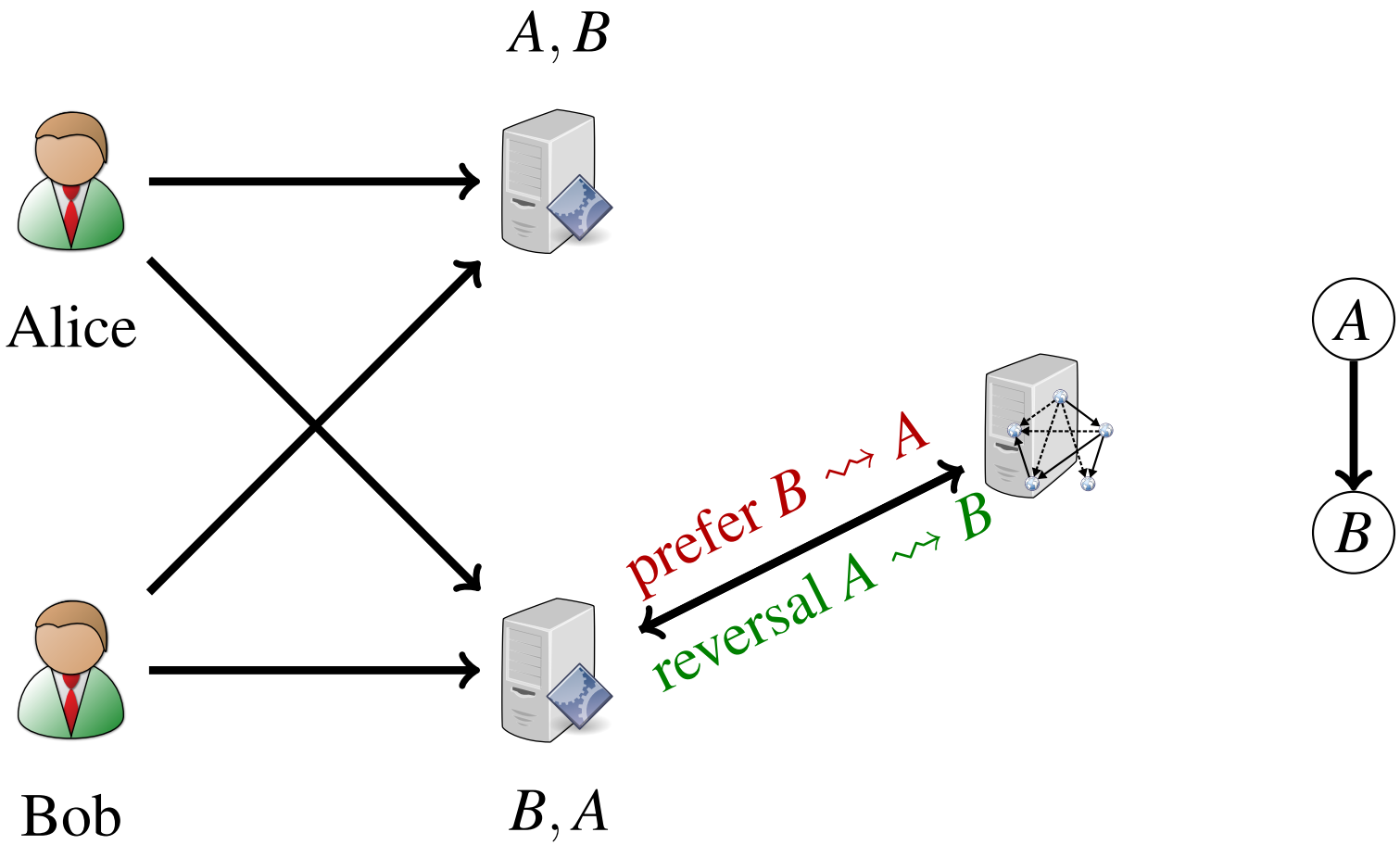
Clients issuing requests to multiple servers can see these requests cross and re-order in the network.

KronoGraph Example



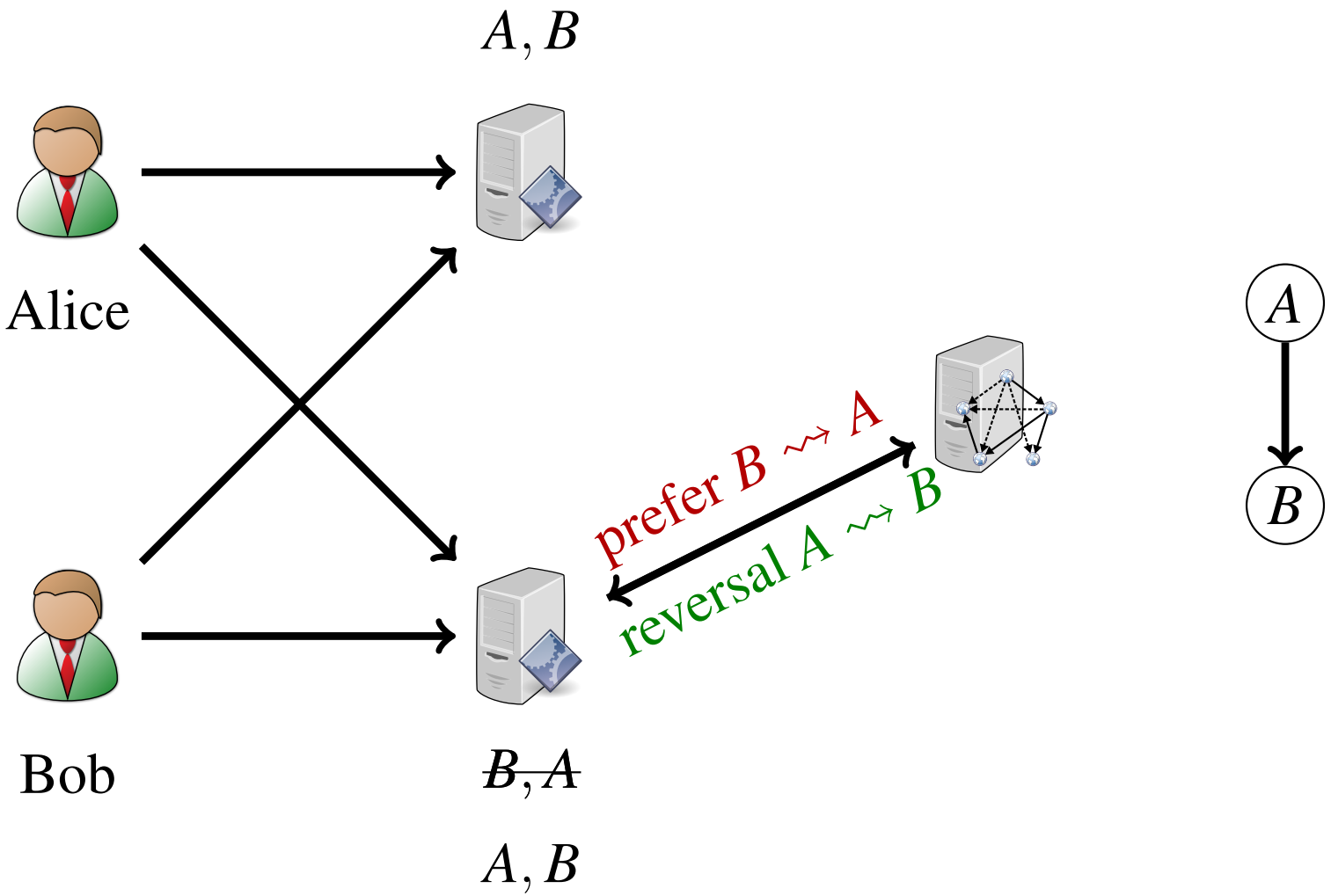
Servers may use Kronos to disambiguate the true order of events, preferring the natural arrival order where possible.

KronoGraph Example



Kronos answers the prefer request with a reversal and the true order between the events.

KronoGraph Example

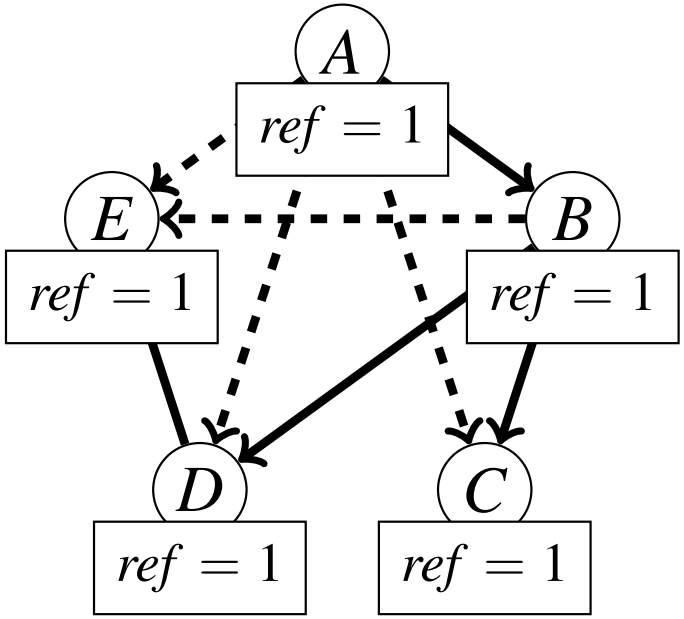


The shard server can re-order the execution of operations when Kronos indicates a reversal.

Garbage Collection

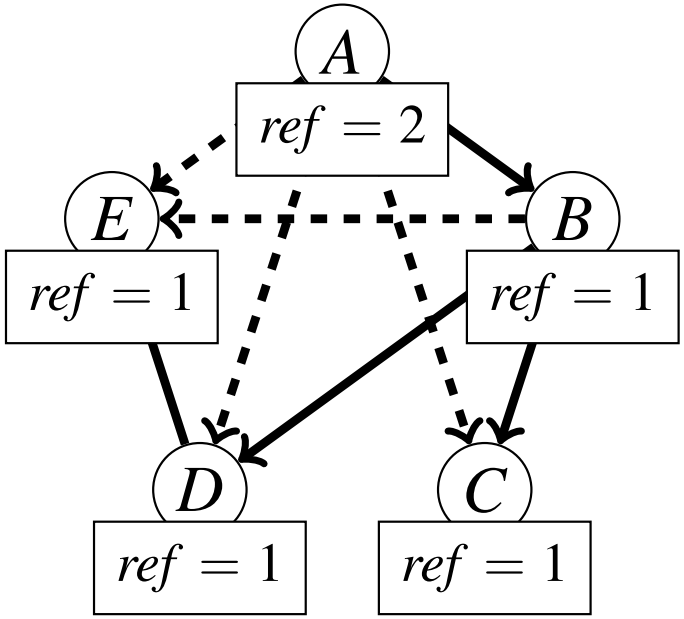
Garbage collection keeps the size of the event dependency graph proportional to the working set of events.

Garbage Collection



Kronos associates with each event a reference count. Clients manually acquire and release references.

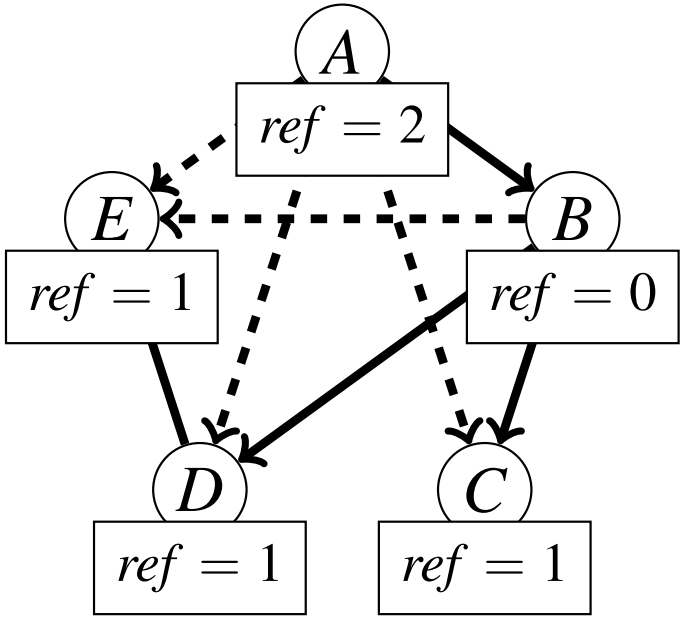
Garbage Collection



- `acquire_ref('A');`

Acquiring a reference increases the reference count.

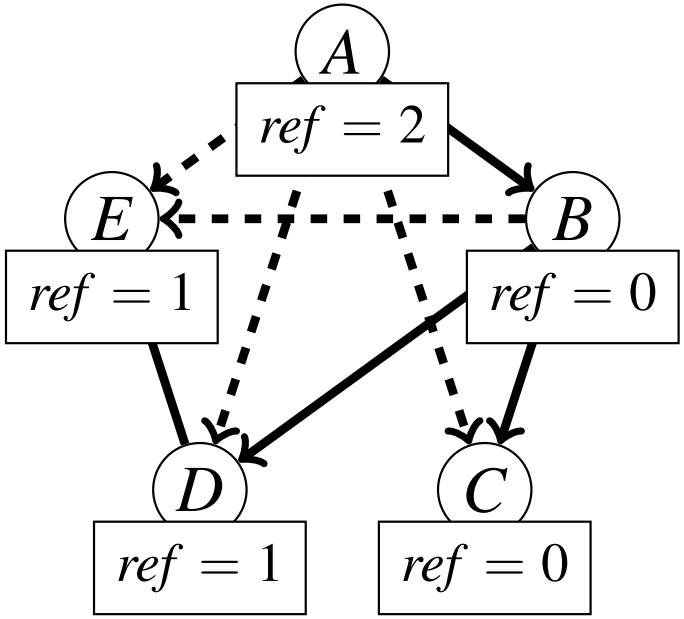
Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`

When a reference count goes to zero, the associated event is ready for garbage collection.

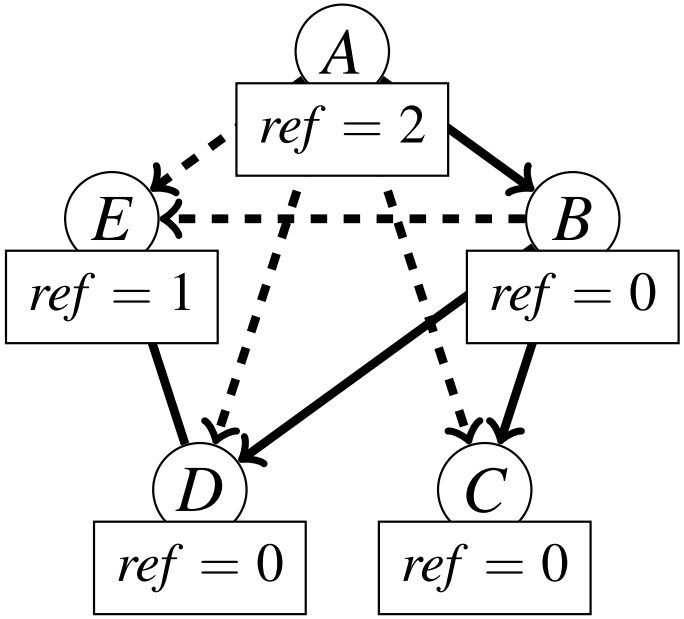
Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`

Events that are ready for garbage collection stay in the graph until they have no more incoming edges.

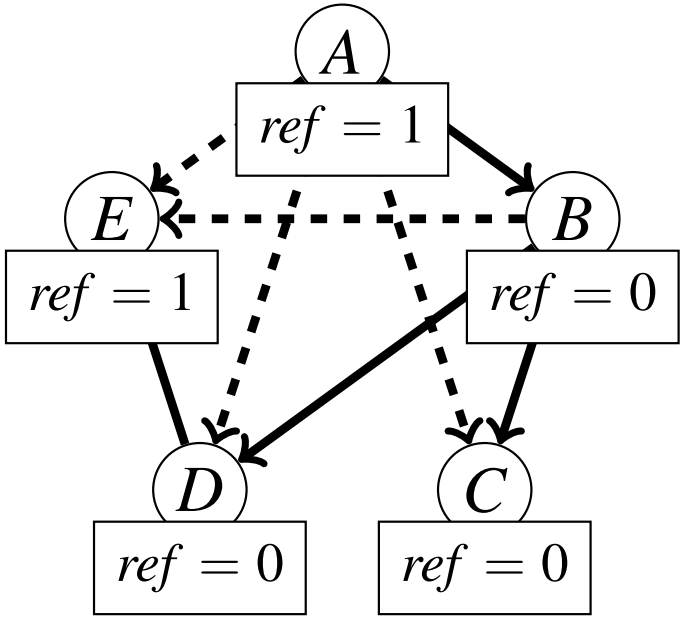
Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`
- `release_ref('D');`

Events that are ready for garbage collection stay in the graph until they have no more incoming edges.

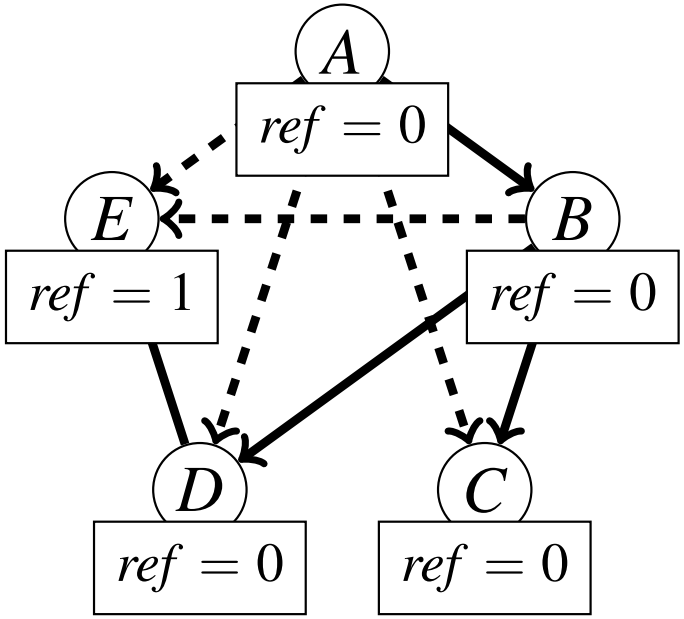
Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`
- `release_ref('D');`
- `release_ref('A');`

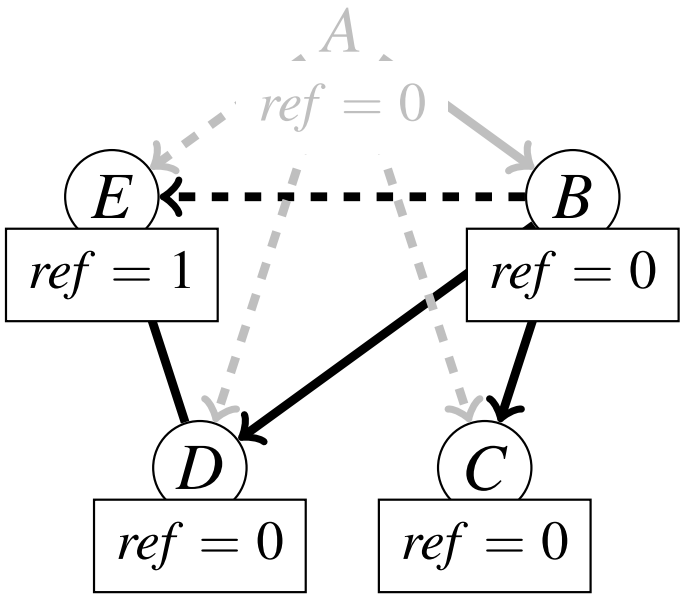
Events that are ready for garbage collection stay in the graph until they have no more incoming edges.

Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`
- `release_ref('D');`
- `release_ref('A');`
- `release_ref('A');`

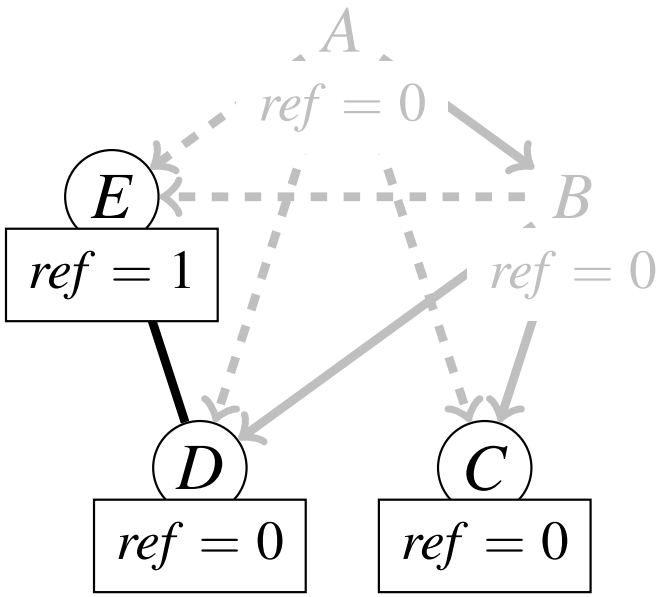
Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`
- `release_ref('D');`
- `release_ref('A');`
- `release_ref('A');`

Once a reference count goes to zero, and has no incoming edges, the event will be garbage collected

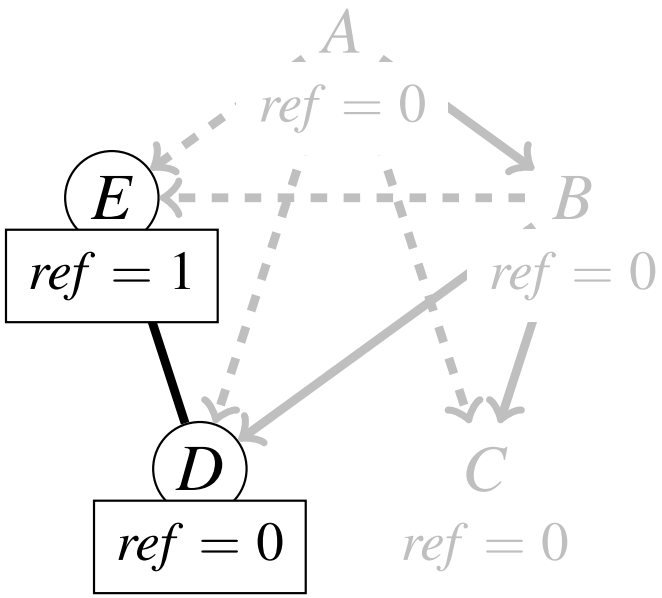
Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`
- `release_ref('D');`
- `release_ref('A');`
- `release_ref('A');`

Once a reference count goes to zero, and has no incoming edges, the event will be garbage collected

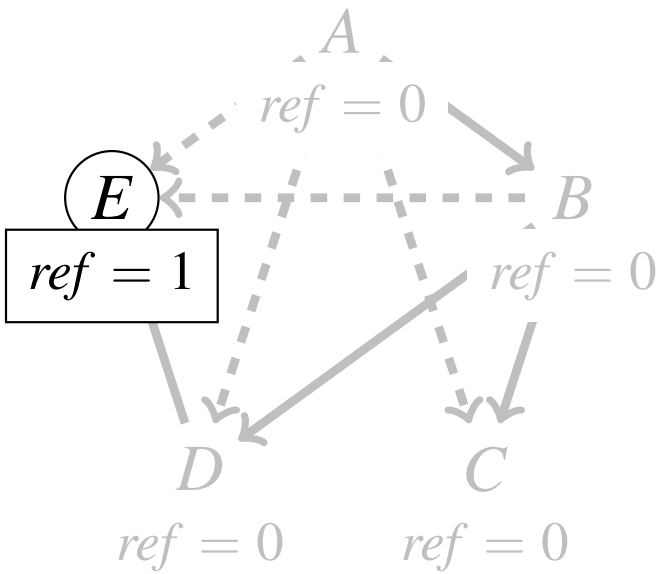
Garbage Collection



- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`
- `release_ref('D');`
- `release_ref('A');`
- `release_ref('A');`

Once a reference count goes to zero, and has no incoming edges, the event will be garbage collected

Garbage Collection

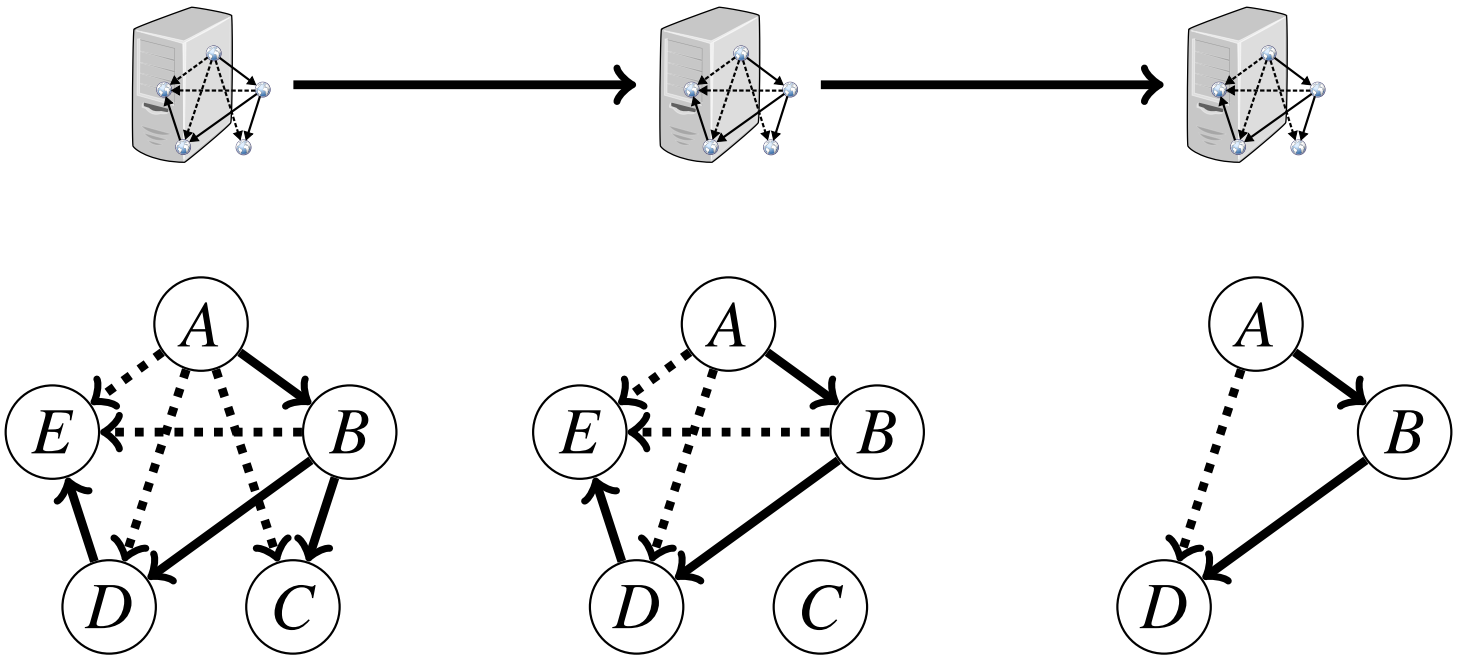


- `acquire_ref('A');`
- `release_ref('B');`
- `release_ref('C');`
- `release_ref('D');`
- `release_ref('A');`
- `release_ref('A');`

Once a reference count goes to zero, and has no incoming edges, the event will be garbage collected

Fault Tolerance

- Replicated using Chain Replication
- Tolerate f failures with $f + 1$ replicas
- Could easily use any state machine replication technique



Typical Optimizations

- Out-of-date replicas may be used for queries that return a happens-before relationship
- No cache invalidation necessary for happens-before relationships
  - Caching becomes near free
  - Cache within Kronos
  - Cache within clients
- Exploit batching for `assign_order` and `query_order` calls
- Optimistically order events to improve batching by looking ahead at what events may need order

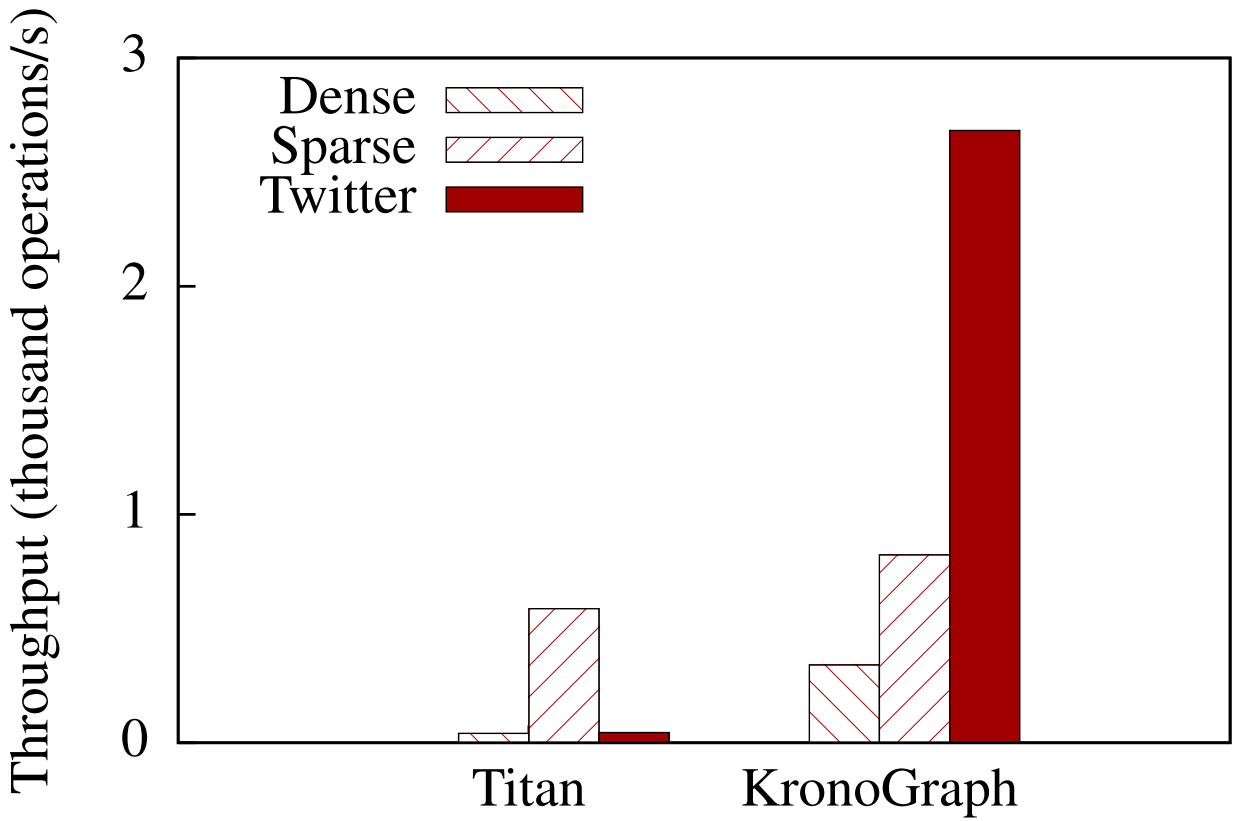
Experimental Setup

- What is the performance of our Kronos applications?
- How scalable is Kronos?
- How large a graph can be stored?
- What are the costs associated with garbage collection?

Experimental Setup

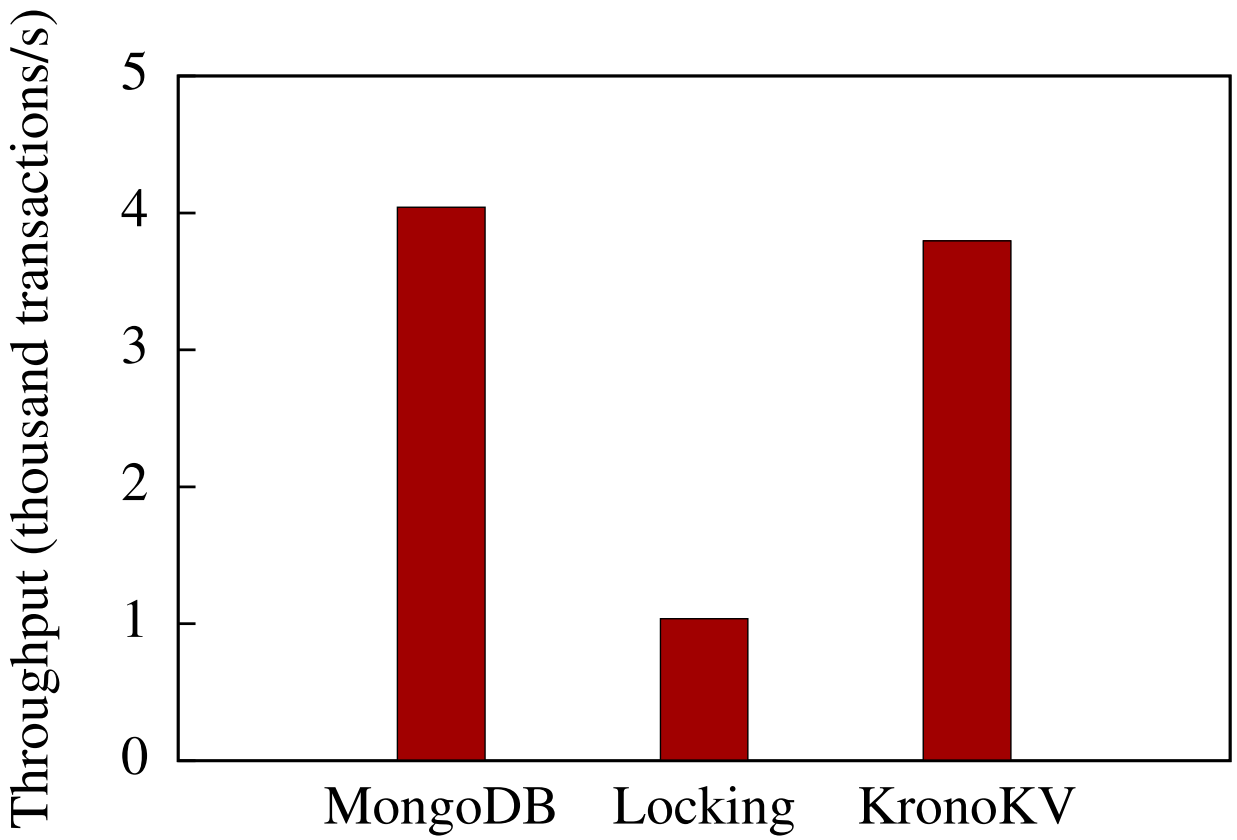
- 14 Machines for applications
- Intel Xeon 2.5 GHz E5420 $\times$ 2
- 16 GB RAM
- 500 GB SATA HDD
- Debian 7.0
- Linux 3.2

KronoGraph Evaluation



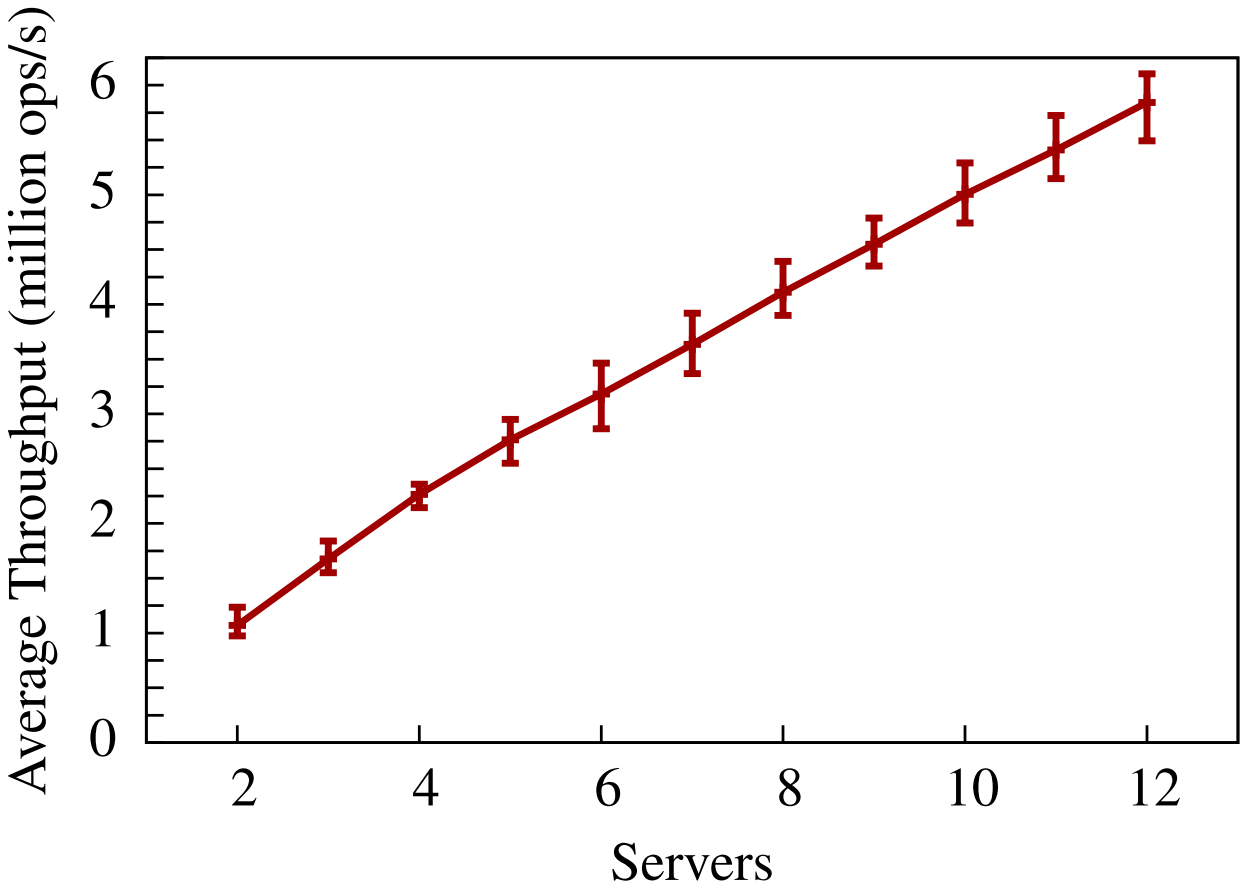
KronoGraph outperforms Titan, an online graph store that employs lock-based techniques.

KronoKV Evaluation



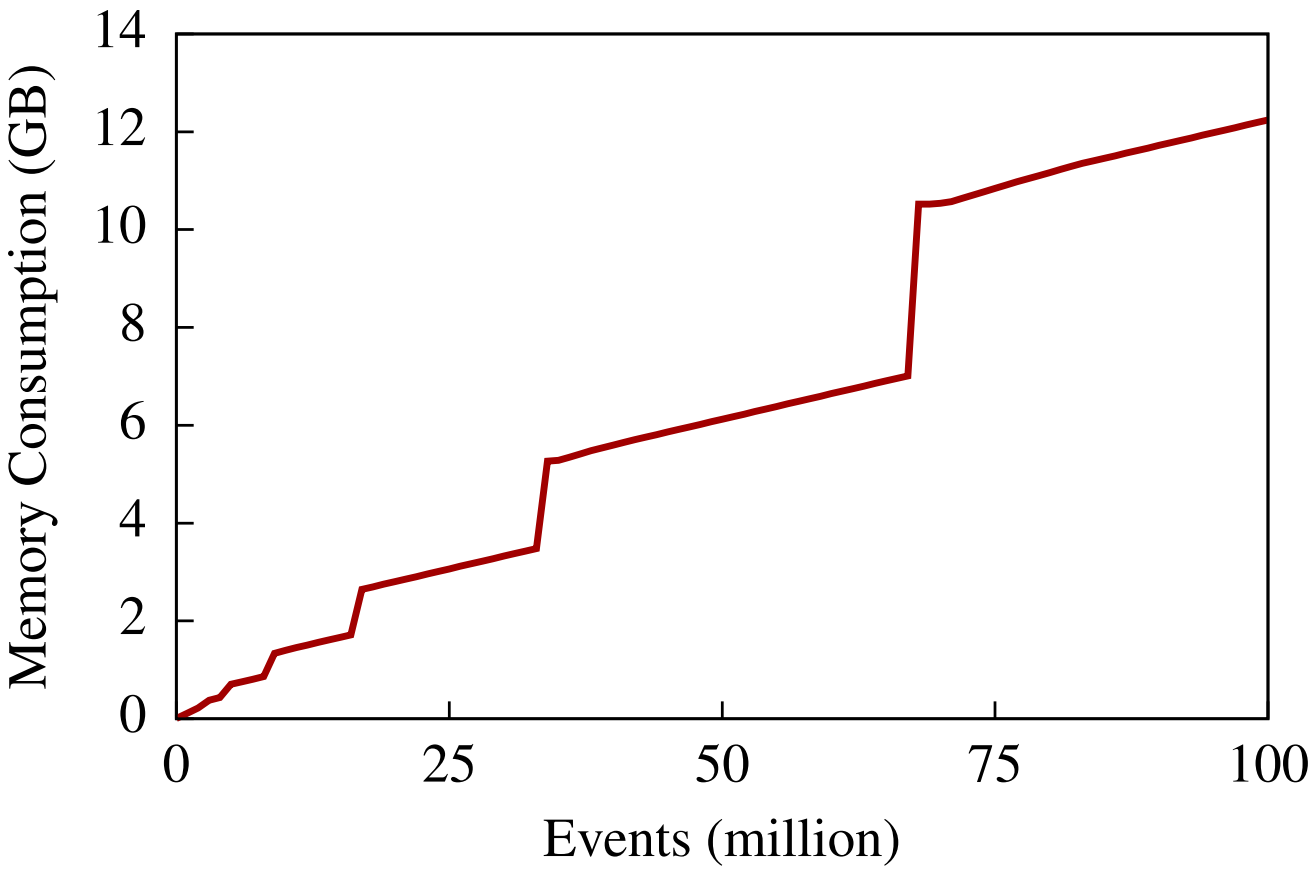
KronoKV performs better than locking approaches and is on par with popular industry solutions

Microbenchmark: Scalability



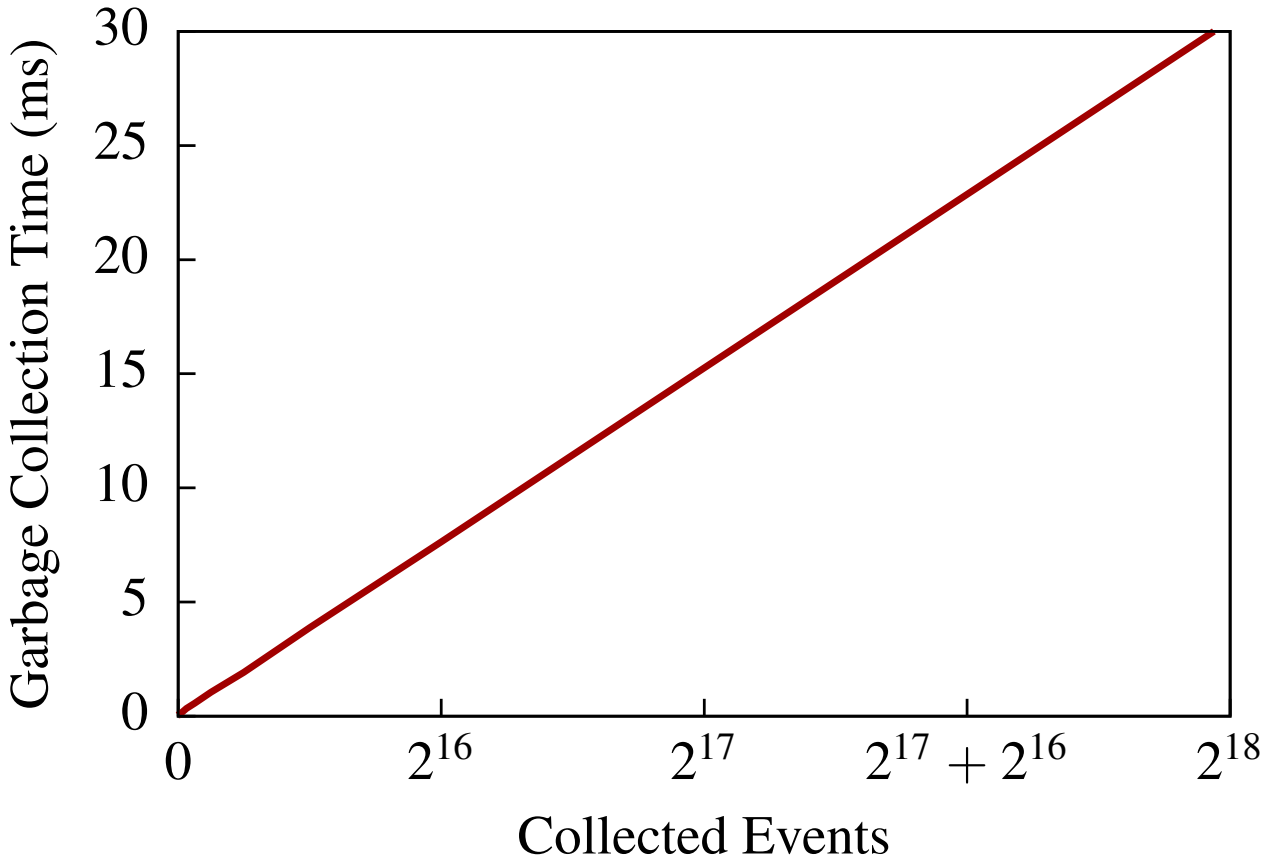
Scalability on a sparse random graph

Microbenchmark: Memory Usage



Memory usage remains proportional to the number of vertices in the graph

Microbenchmark: Garbage Collection



Garbage collection time is proportional to the number of events collected

Conclusion

- Kronos is a time oracle for distributed systems
- A time oracle allows high performance systems that uphold strong guarantees