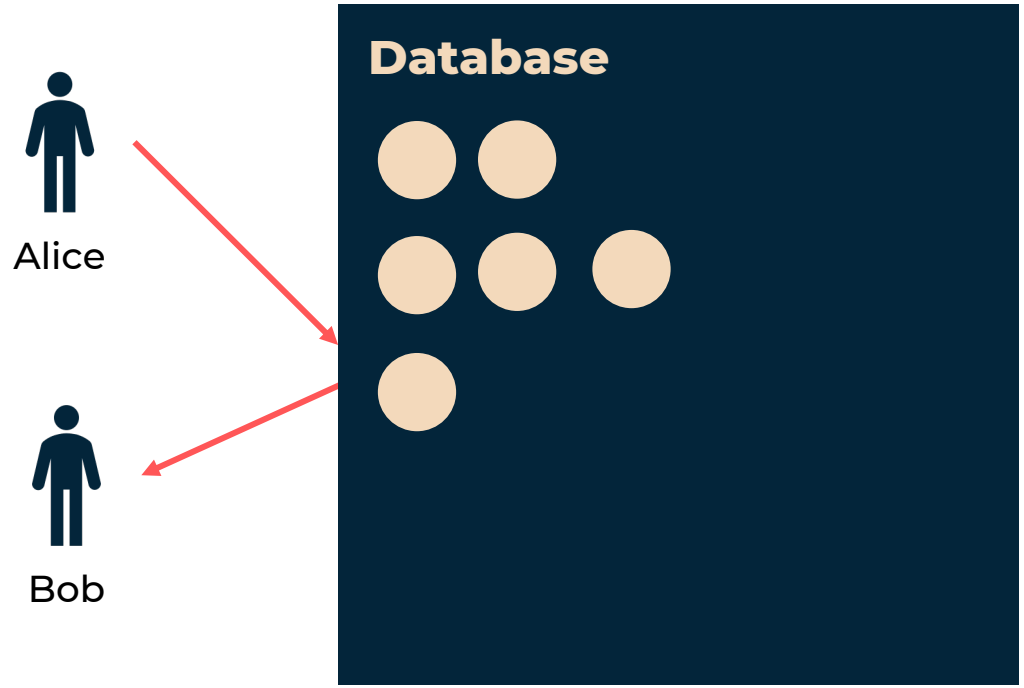
Towards correct high-performance database backend

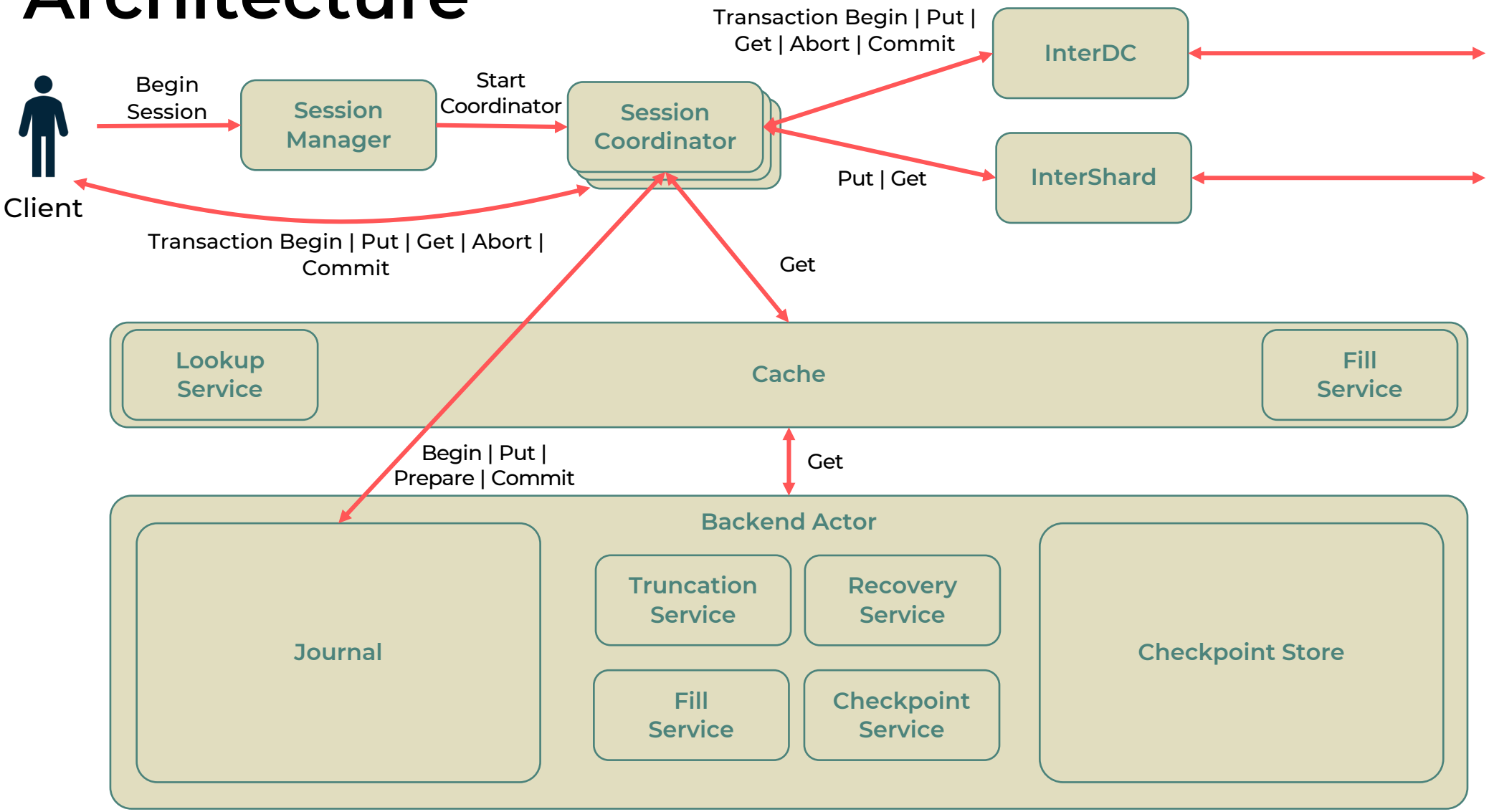
Saalik Hatia - Annette Bieniusa - Carla Ferreira - Gustavo Petri - Marc Shapiro

What is a database ?

- Store data
- Retrieve data
- Share data



Architecture



Target database - Antidote

Geo-distributed data centers (DC)

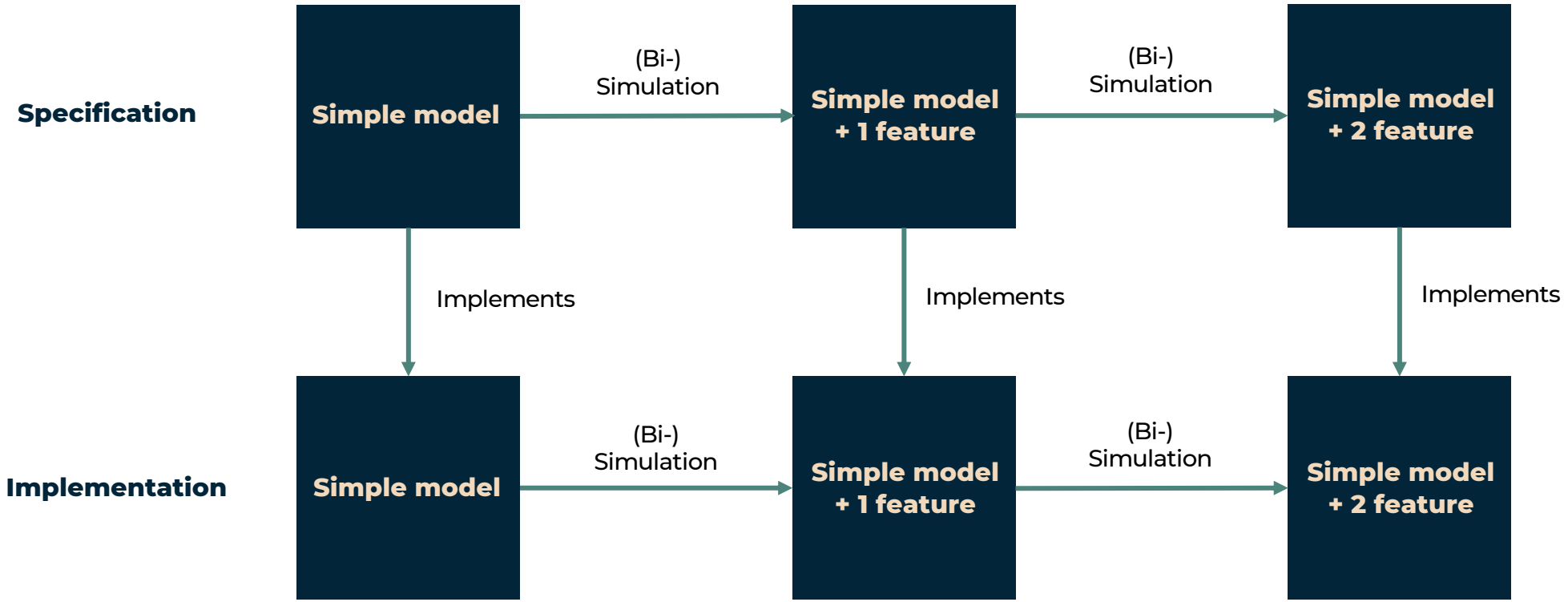
Full replication

Causal consistency between DCs

In a DC:

- Total order per DC *[Akkoorath et al. ‘16]*
- Sharding *[DeCandia et al ‘07]*
- Multiversion Concurrency Control (MVCC) with partial order
- Journal of operations
- Safe journal truncation
- States stored in a Checkpoint Store

Adding one feature at a time



Our plan

Baseline:

- Transaction
- Causal consistency
- Multi-version concurrency control
- Store all versions

Incrementally add features:

- Bounded versions
- Fault tolerance
- Journal
- Etc..

Specification of the model

Operational semantics

Invariants

$$\textsc{Read Local}\ \frac{k \in \mathsf{dom}(\mathcal{E}) \qquad \mathcal{E}(k) = v}{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, \bot) \xrightarrow{\mathsf{read}_\tau(k,\mathtt{dt}) \Rightarrow \{v\}} (\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, \bot)} \qquad \textsc{Read}\ \frac{k \notin \mathsf{dom}(\mathcal{E})}{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, \bot) \xrightarrow{\mathsf{read}_\tau(k,\mathtt{dt}) \Rightarrow V} (\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R} \cup \{k\}, \bot)}$$

$$\textsc{Effect}\ \frac{}{(\tau, \mathtt{dt}, \mathcal{E}, \mathcal{R}, \bot) \xrightarrow{\mathsf{effect}_\tau(k,v)} (\tau, \mathtt{dt}, \mathcal{E}[k \leftarrow v], \mathcal{R}, \bot)}$$

Figure 5: Transaction Semantics

$$\textsc{Step}\ \frac{\mathcal{T} = \mathcal{T}'' \cup \{\tau\} \qquad \tau \xrightarrow{\alpha_\tau} \tau' \qquad \mathcal{T}' = \mathcal{T}'' \cup \{\tau'\}}{(\mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}) \xrightarrow[\tau]{\alpha} (\mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}')} \qquad \textsc{Abort}\ \frac{\mathcal{T} = \mathcal{T}' \cup \{\tau\} \qquad \mathcal{T}'_\mathrm{a} = \mathcal{T}_\mathrm{a} \cup \{\tau\}}{(\mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}) \xrightarrow[\tau]{\mathsf{abort}_\tau} (\mathcal{T}'_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}')}$$

$$\textsc{Begin Transaction}\ \frac{\tau \notin \mathcal{T}_\mathrm{a} \cup \mathcal{T}_\mathrm{c} \cup \mathcal{T} \qquad \mathcal{CM}_{\mathrm{begin}}(\mathtt{dt}, \_, \mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}) \qquad \mathcal{T}' = \mathcal{T} \cup \{(\tau, \mathtt{dt}, \emptyset, \emptyset, \bot)\}}{(\mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}) \xrightarrow[\tau]{\mathsf{begin}(\mathtt{dt})} (\mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}')}$$

$$\textsc{Commit}\ \frac{\mathcal{T} = \mathcal{T}' \cup \{\tau\} \qquad \mathcal{CM}_{\mathrm{commit}}(\tau, \_, \mathcal{T}_\mathrm{c}, \mathtt{ct}) \qquad \tau = (\_, \_, \mathcal{E}, \_, \_) \qquad \mathcal{T}'_\mathrm{c} = \mathcal{T}_\mathrm{c} \cup \{\tau[\bot \leftarrow \mathtt{ct}]\}}{(\mathcal{T}_\mathrm{a}, \mathcal{T}_\mathrm{c}, \mathcal{T}) \xrightarrow[\tau]{\mathsf{commit}_\tau(\mathtt{ct}, \mathcal{E})} (\mathcal{T}_\mathrm{a}, \mathcal{T}'_\mathrm{c}, \mathcal{T}')}$$

Figure 6: Transaction System

Starting simple

Client API

- $begin(dt)$ initialize a transaction with a snapshot
- $read(key)$ blob returns the object key from the snapshot Dt
- $effect(key,\ value)$ assigns the value blob to the key
- $commit(\ )$ assign a commit timestamp to the transaction
- $abort(\ )$ abort the live transaction

Implementation

We take our specification to be as close as possible to the model

We represent a transaction with the following pattern:

$$(\tau_i, \mathtt{dt}, \mathcal{E}, \mathcal{R}, \mathtt{ct})$$

with the following interpretation:

- τ_i: transaction identifier (unique by definition),
- $\mathtt{dt}$: dependency timestamp (identifies the transactions this one depends upon),
- $\mathcal{E}$: effect map (records the content of the transaction's writes),
- $\mathcal{R}$: read set (records what objects the transaction has read),
- $\mathtt{ct}$: commit timestamp (set at termination time). In the case where the transaction has not been finalized we have that $\mathtt{ct} = \perp$.

Implementation

We take our specification to be as close as possible to the model

```
public class Transaction {
    // Unique by definition
    private TransactionID id;
    // Identifies the transactions this one depends upon
    private Timestamps dependency;
    // Records the content of the transaction's writes
    private HashMap<String, Value> effectMap;
    // Records what objects the transaction has read
    private CopyOnWriteArraySet<String> readSet;
    // Time of commit
    private Timestamps commit;
```

Implementation

Then we add all the necessary invariants

Begin transaction:

- Only one active transaction per client
- Dependency timestamp must be valid

Implementation

Then we add all the necessary invariants.

```
@Override
public void startTransaction(TransactionID dependency) {
    checkArgument(tr == null, "Transaction already started");
    if (backend.dependencyIsValid(dependency)){
        tr = new Transaction(backend, dependency);
    }else{
        System.out.println("Dependency is not valid ! " +
                "Please retry with a correct transaction identifier");
    }
}
```

Implementation

Then we add all the necessary invariants.

```
@Override
public void startTransaction(TransactionID dependency) {
    checkArgument(tr == null, "Transaction already started");
    if (backend.dependencyIsValid(dependency)){
        tr = new Transaction(backend, dependency);
    }else{
        System.out.println("Dependency is not valid ! " +
                "Please retry with a correct transaction identifier");
    }
}
```

Implementation

Then we add all the necessary invariants.

```
@Override
public void startTransaction(TransactionID dependency) {
    checkArgument(tr == null, "Transaction already started");
    if (backend.dependencyIsValid(dependency)){
        tr = new Transaction(backend, dependency);
    }else{
        System.out.println("Dependency is not valid ! " +
                "Please retry with a correct transaction identifier");
    }
}
```

Adding a feature – Bounded Versions

Once everything is working, we add a single feature:

- Bounded versions
  - Write the specification
  - Implement the new version
  - Comparison with the previous model

Bounded Versions – Memory invariants

We define an arbitrary limitation on the size of the memory used by the system.

New system values:

$$M_{used}$$
$$M_{limit}$$

New invariant:

$$M_{used} \leq M_{limit}$$

Bounded Versions – Memory invariants

```
public void effect(String key, Value value) {
    checkArgument(backend.sizeAvailable(), "Memory size limit reached");

    @Nullable
    Value newValue = null;
    try {
        if (effectMap.containsKey(key)) {
            Value oldValue = effectMap.get(key);
            newValue = Value.merge(oldValue ,value);
            effectMap.put(key, newValue);
        } else {
            Bounded.Value oldValue = backend.getValue(key, dependency);
            if (oldValue != null) {
                newValue = Value.merge(oldValue ,value);
                effectMap.put(key, newValue);
            } else {
                effectMap.put(key, new Value(id, value));
            }
        }
    } finally {
        checkArgument(effectMap.containsValue(newValue), "Value was not added to effectmap");
    }
}
```

Bounded Versions – Memory invariants

```
public void effect(String key, Value value) {
    checkArgument(backend.sizeAvailable(), "Memory size limit reached");

    @Nullable
    Value newValue = null;
    try {
        if (effectMap.containsKey(key)) {
            Value oldValue = effectMap.get(key);
            newValue = Value.merge(oldValue ,value);
            effectMap.put(key, newValue);
        } else {
            Bounded.Value oldValue = backend.getValue(key, dependency);
            if (oldValue != null) {
                newValue = Value.merge(oldValue ,value);
                effectMap.put(key, newValue);
            } else {
                effectMap.put(key, new Value(id, value));
            }
        }
    } finally {
        checkArgument(effectMap.containsValue(newValue), "Value was not added to effectmap");
    }
}
```

Bounded Versions – Memory invariants

```
public void effect(String key, Value value) {
    checkArgument(backend.sizeAvailable(), "Memory size limit reached");

    @Nullable
    Value newValue = null;
    try {
        if (effectMap.containsKey(key)) {
            Value oldValue = effectMap.get(key);
            newValue = Value.merge(oldValue ,value);
            effectMap.put(key, newValue);
        } else {
            Bounded.Value oldValue = backend.getValue(key, dependency);
            if (oldValue != null) {
                newValue = Value.merge(oldValue ,value);
                effectMap.put(key, newValue);
            } else {
                effectMap.put(key, new Value(id, value));
            }
        }
    } finally {
        checkArgument(effectMap.containsValue(newValue), "Value was not added to effectmap");
    }
}
```

Bounded Versions – Memory invariants

```
public void effect(String key, Value value) {
    checkArgument(backend.sizeAvailable(), "Memory size limit reached");

    @Nullable
    Value newValue = null;
    try {
        if (effectMap.containsKey(key)) {
```

```
public boolean sizeAvailable() {
    int mUsed = backend.size();
    return mUsed <= MLIMIT;
}
```

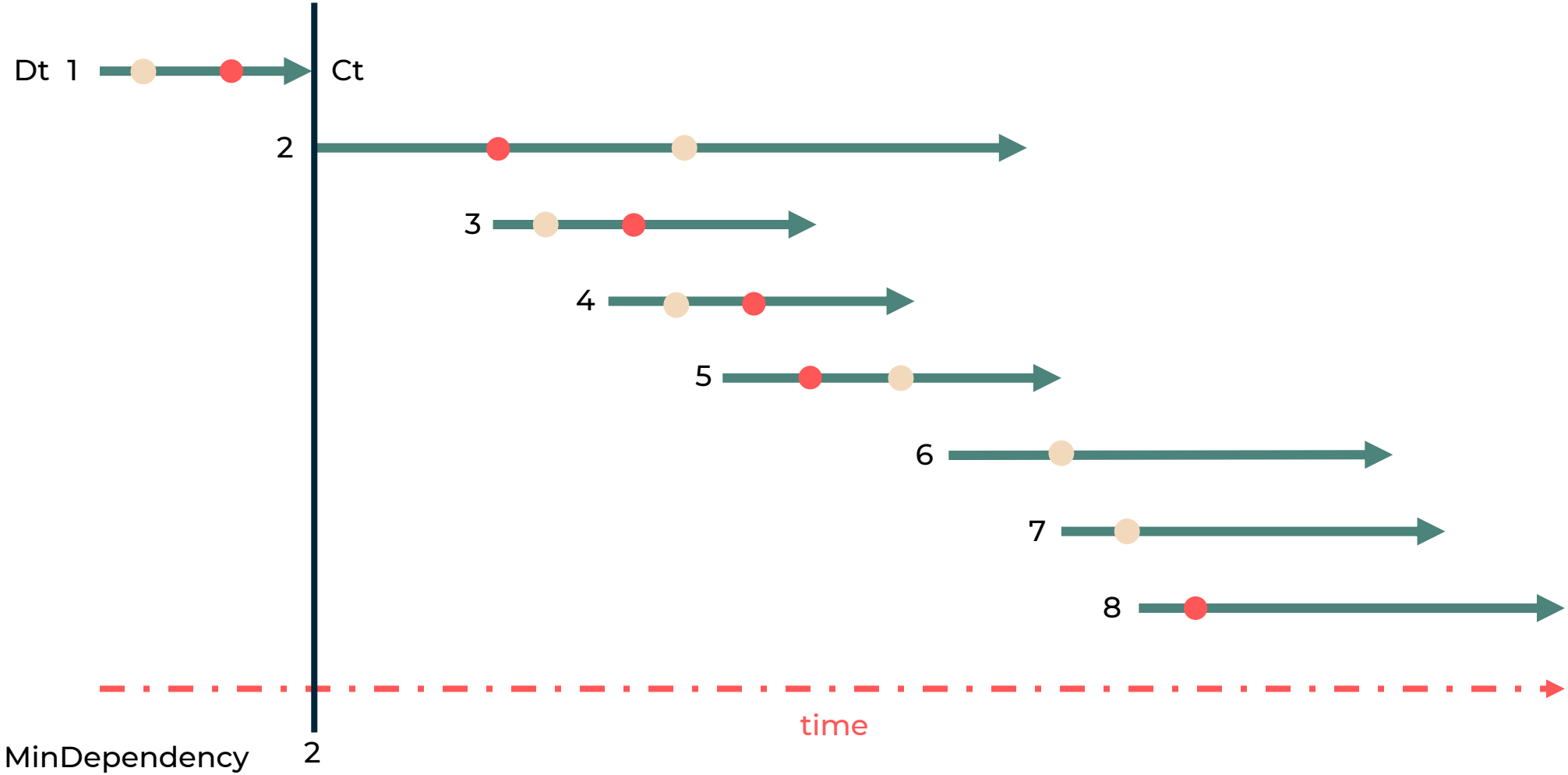
```
            } else {
                effectMap.put(key, new Value(id, value));
            }
        }
    } finally {
        checkArgument(effectMap.containsValue(newValue), "Value was not added to effectmap");
    }
}
```

Bounded memory – Garbage collection

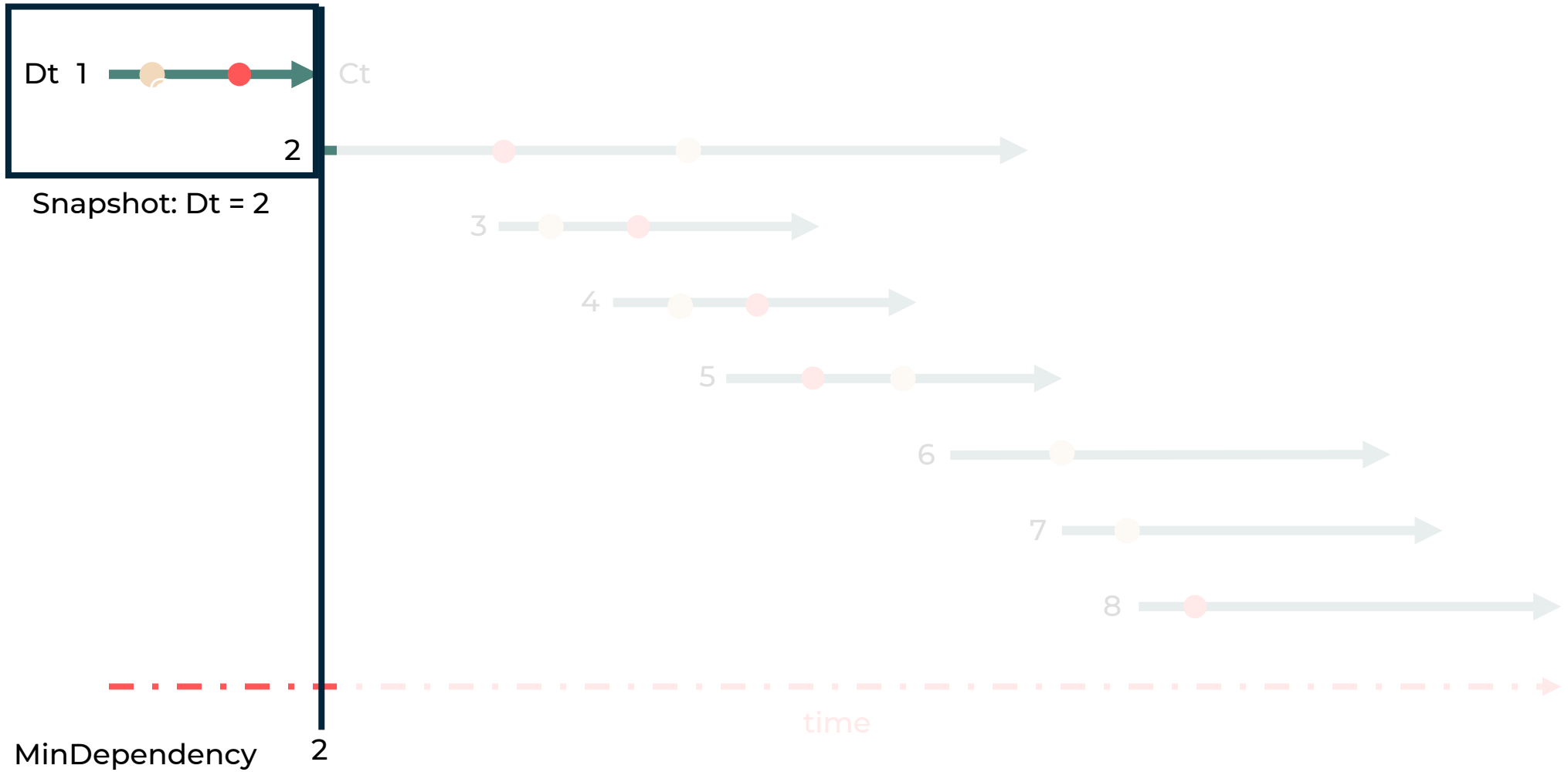
How do we know what we can throw away ?

- Keep track of the lowest dependency of a running transaction (MinDependency)
  - Among the versions contained in the snapshot
  - At the minimum, keep the most recent version of every key

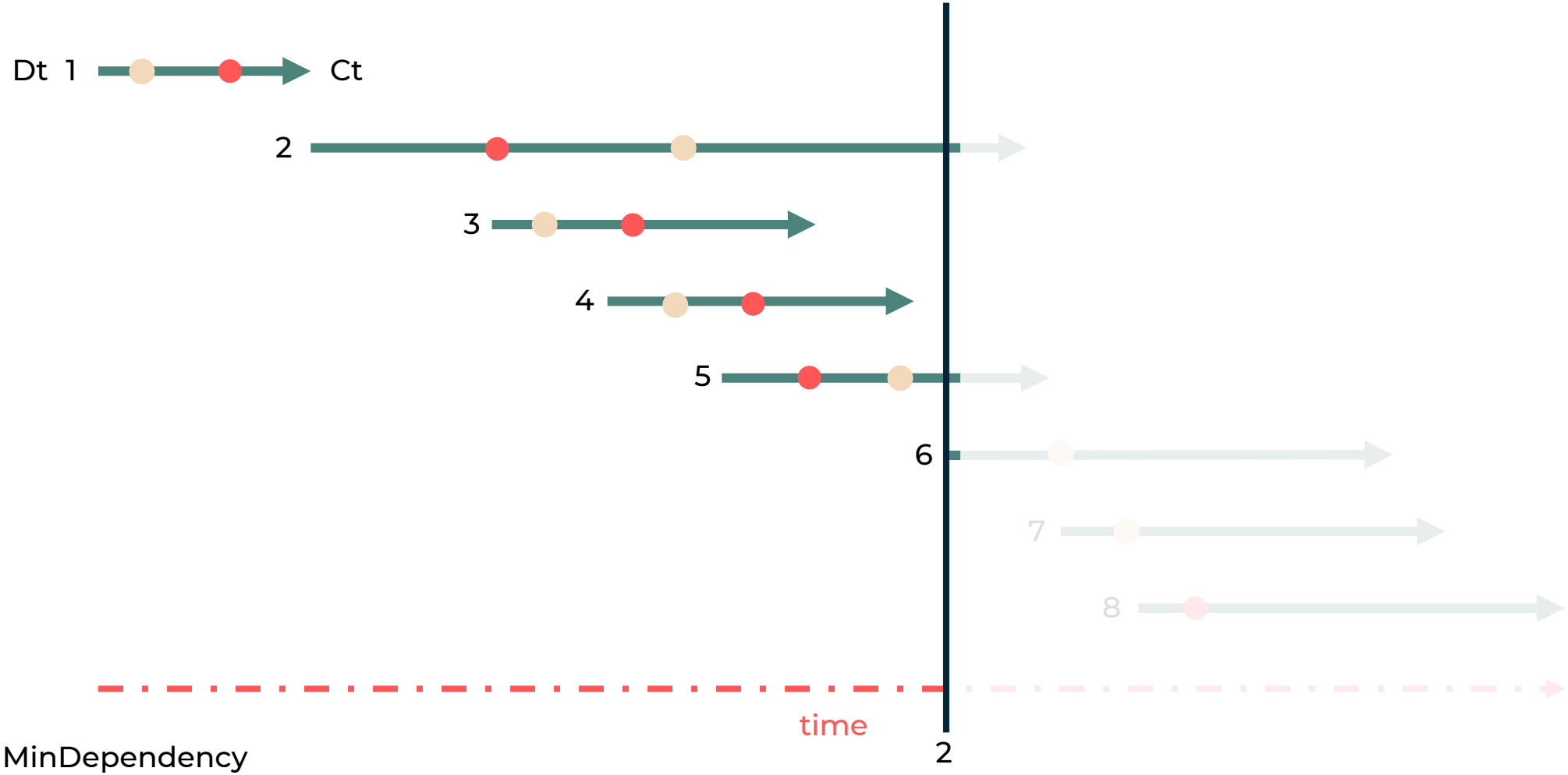
Bounded memory – Garbage collection



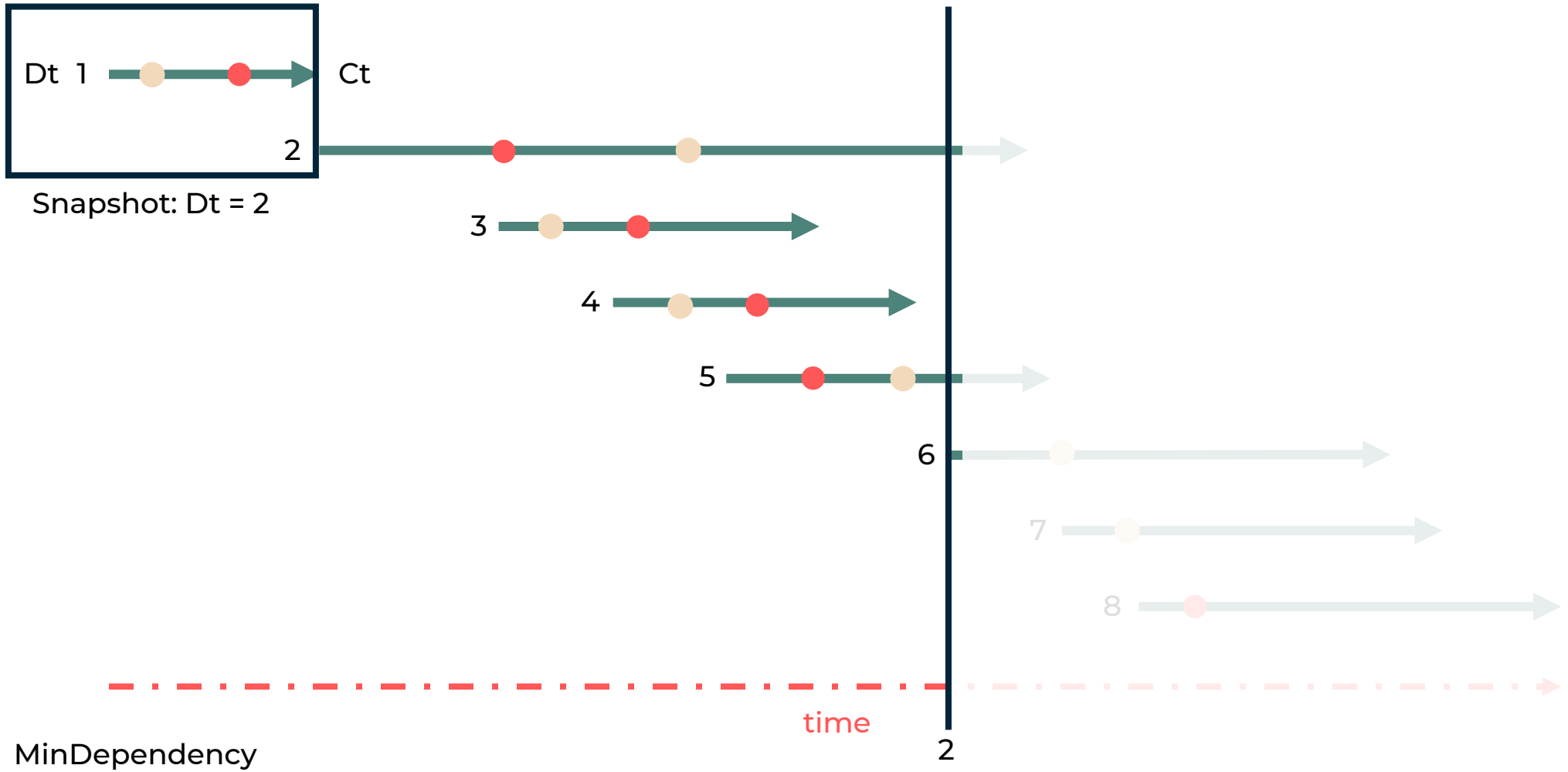
Bounded memory – Garbage collection



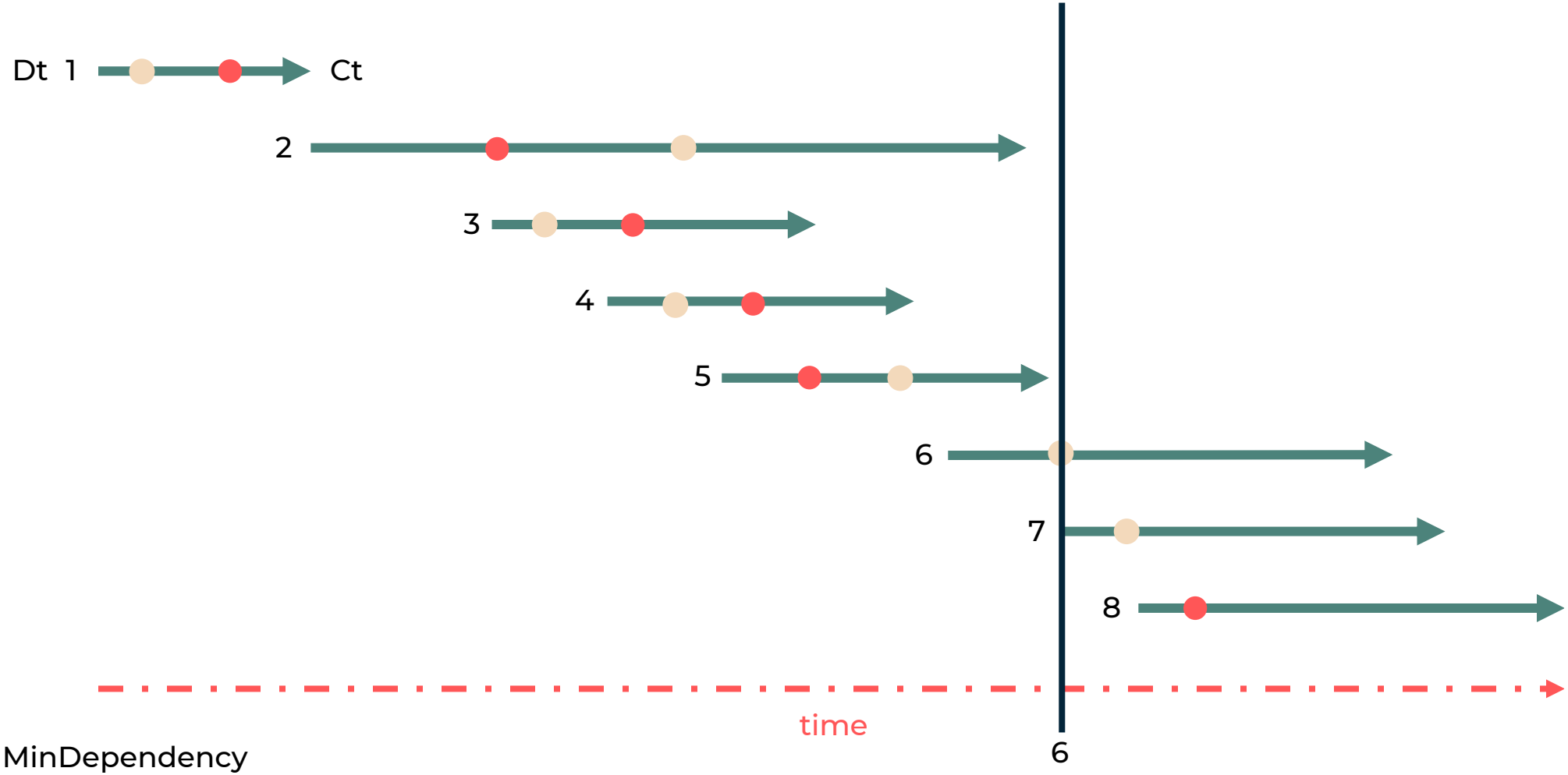
Bounded memory – Garbage collection



Bounded memory – Garbage collection

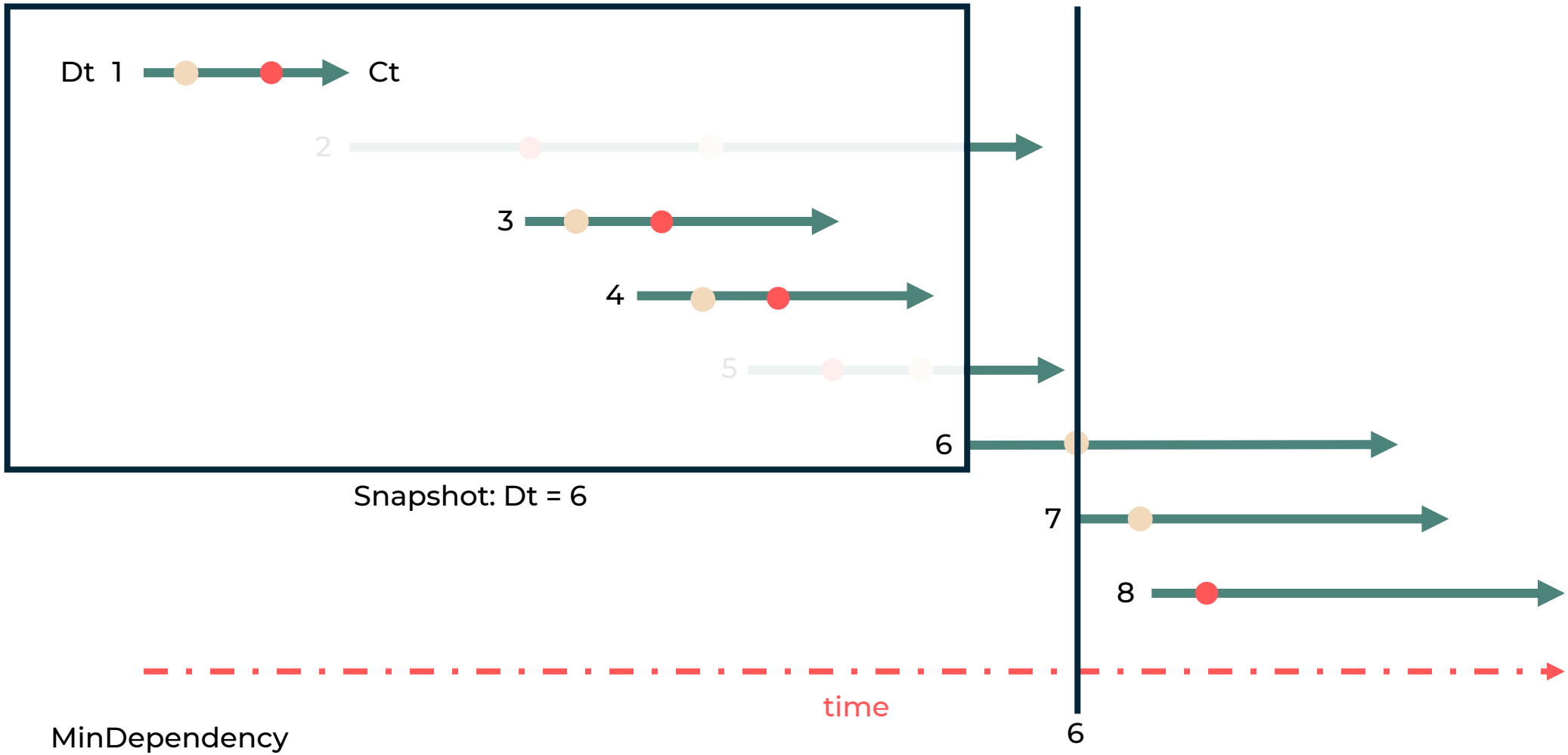


Bounded memory – Garbage collection

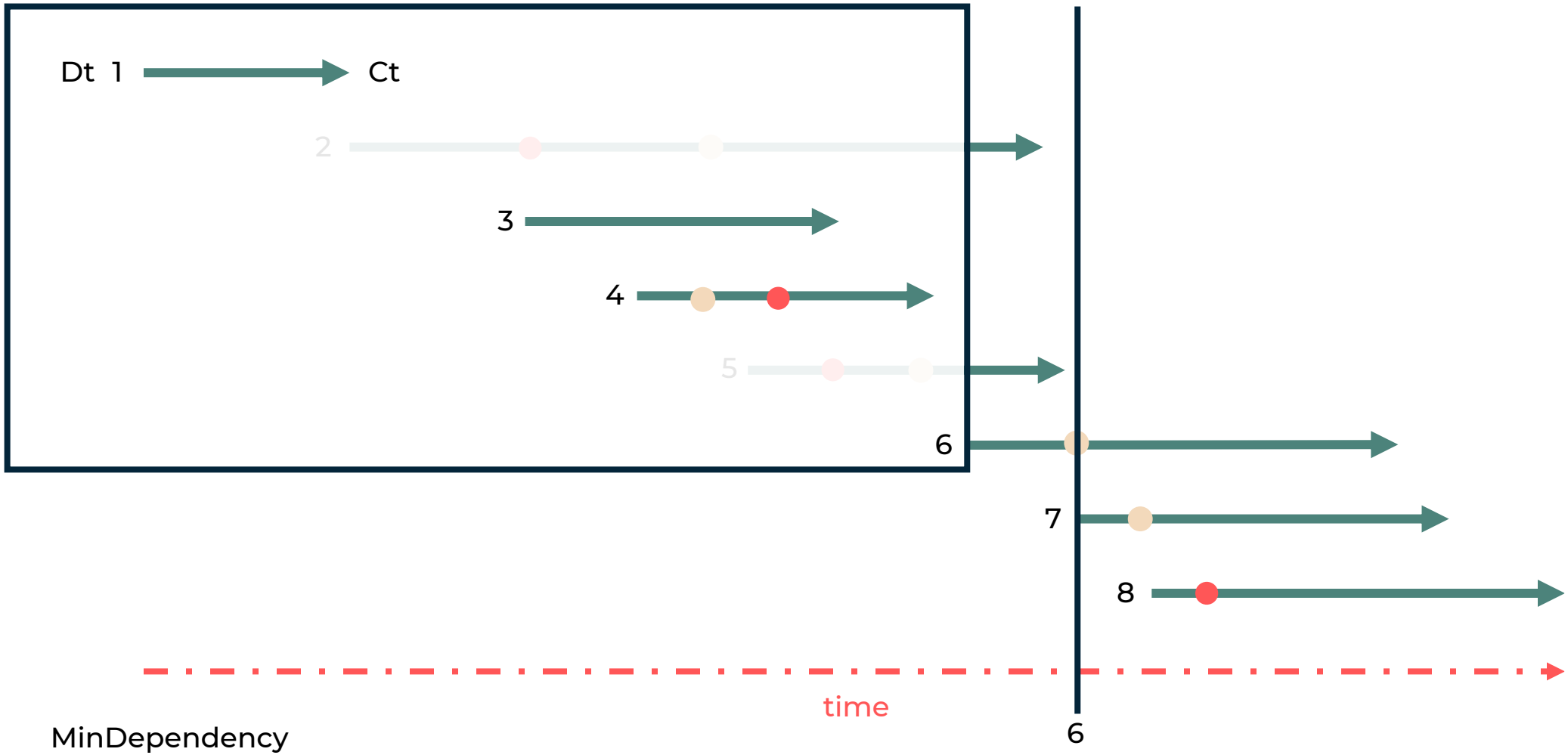


MinDependency

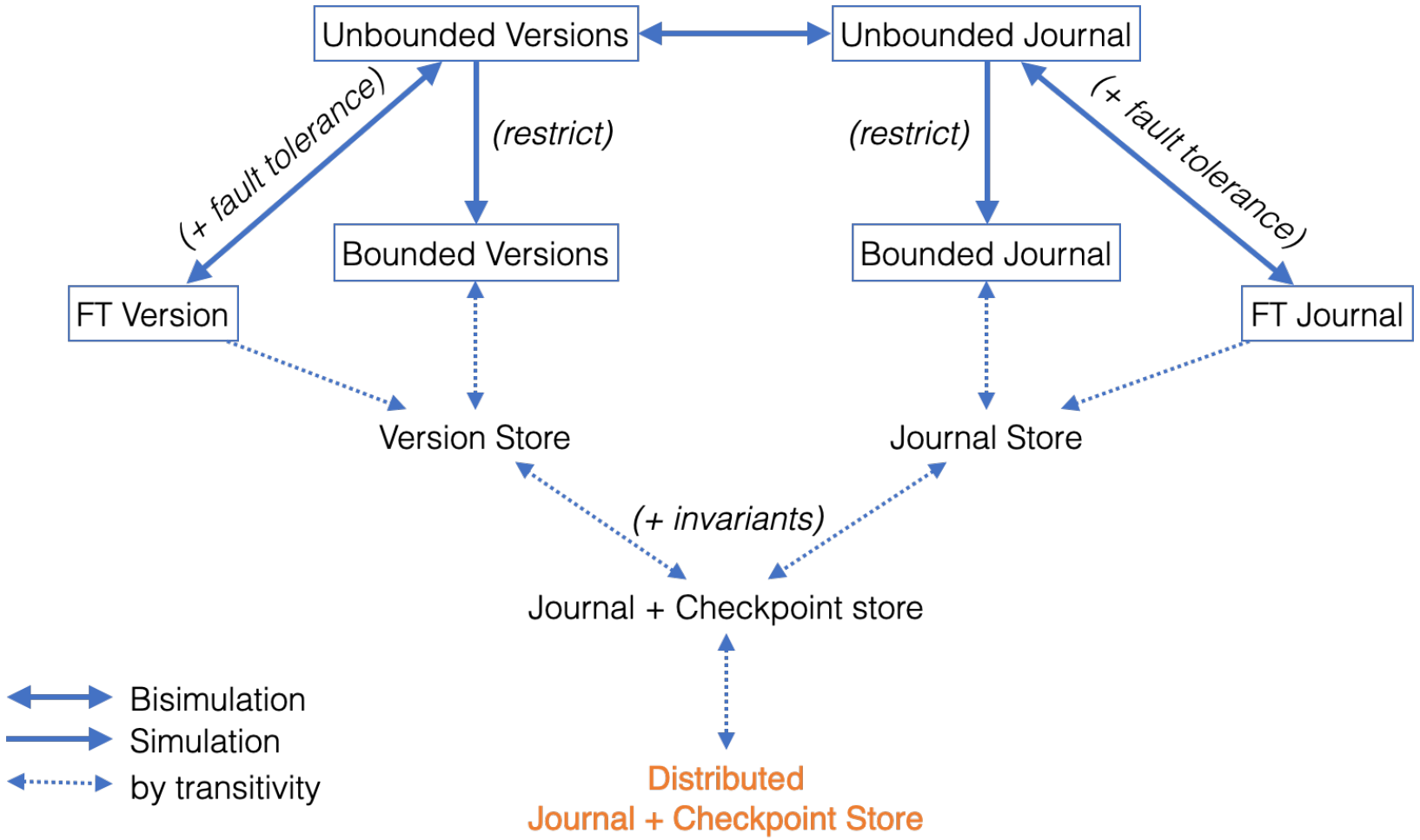
Bounded memory – Garbage collection



Bounded memory – Garbage collection



Roadmap


Unbounded Versions
Unbounded Journal
(+ fault tolerance)
(restrict)
(restrict)
(+ fault tolerance)
Bounded Versions
Bounded Journal
FT Version
FT Journal
Version Store
Journal Store
(+ invariants)
Journal + Checkpoint store
Bisimulation
Simulation
by transitivity
Distributed
Journal + Checkpoint Store

Ongoing and future work

Finishing all implementations

Testing invariants

Performance comparison

Static analysis