On the Serializability of **Transactions** in **Shared Dataspaces** with **Temporary Data**

```
[take(job);
  result=Exec(job);
  write(result,df);]
```

Gianluigi Zavattaro  Department of Computer Science
Joint work with Nadia Busi  University of Bologna

---

**Plan of the Talk**

- **Temporary Data** in Shared Dataspaces
- **Transactions** in Shared Dataspaces
- Transaction **Serializability**
- Locking Policies for Dataspaces
- **New Problem:**
  - Transactions and Temporary Data
- **Solutions:**
  - Enhanced locking policies
  - Loose Expired Data Collector

---

**A Typical Problem:**
**Outdated Information**

- Data are shared from an **unpredictable** amount of processes
- After its generation, a datum has a lifetime **independent** from its producer

Accumulation of **outdated** information (especially in highly dynamic open systems)

---

**A Possible Solution:**
**Temporary Data**

- The lifetime of a datum is **indicated** by its producer
- The expiration time of a datum is defined when produced

```
write(a,10)  t=10
```

- On expiration, a datum is **removed**

---

**Plan of the Talk**

- **Temporary Data** in Shared Dataspaces
- **Transactions** in Shared Dataspaces
- Transaction **Serializability**
- Locking Policies for Dataspaces
- **New Problem:**
  - Transactions and Temporary Data
- **Solutions:**
  - Enhanced locking policies
  - Loose Expired Data Collector
A Typical Problem: Update

\[
take(d); \\
write(c); \rightarrow d
\]

- Datum \(d\) should be **updated** to \(c\)
- A first solution: **NO GOOD** if
  - The updating process caches (no **atomicity**)
  - Other processes test the absence of both \(c\) and \(d\) (no **isolation**)

A Typical Solution: Transactions

\[
\begin{align*}
take(d); \\
write(c); &\leftarrow d
\end{align*}
\]

- Datum \(d\) should be **updated** to \(c\)
- The **typical** solution
  - The two operations are grouped inside a transaction to be performed according to some transactional semantics

Plan of the Talk

- **Temporary Data** in Shared Dataspaces
- **Transactions** in Shared Dataspaces
- **Transaction Serializability**
- **Locking Policies** for Dataspaces
- **New Problem**:
  - Transactions and Temporary Data
- **Solutions**:
  - Enhanced locking policies
  - Loose Expired Data Collector

Serializability

- **Serial** schedule
  - If any action of transaction \(x\) precedes any action of transaction \(y\), then all actions of \(x\) precede all actions of \(y\)
- **Serializable** schedule
  - Its effect on the state of the dataspace is the same as that of some **serial** schedule
  - A schedule policy satisfies serializability, if it allows only serializable schedules

Example: a non-serializable schedule

\[
\begin{align*}
take(a); &\rightarrow write(c) \\
\text{[read(a); take(c)]}
\end{align*}
\]

- A non-serializable schedule
  - \(read(a)\) take(\(c\)) \(write(c)\) take(\(c\))

Plan of the Talk

- **Temporary Data** in Shared Dataspaces
- **Transactions** in Shared Dataspaces
- Transaction **Serializability**
- **Locking Policies** for Dataspaces
- **New Problem**:
  - Transactions and Temporary Data
- **Solutions**:
  - Enhanced locking policies
  - Loose Expired Data Collector
Locking Policies

- Serializability in databases is ensured by, e.g., **two phase locking**
- Similarly, in dataspaces we can **enhance** the semantics of primitives with
  - **Read** locks
  - **Take** locks
  - **Absence** locks

Read Locks

- Solution: when read under a bxn, an entry cannot be taken outside the bxn
  - [take(job): write(res)]

- Read locks **forbid** the execution of the non-serializable computation

Plan of the Talk

- **Temporary Data** in Shared Dataspaces
- **Transactions** in Shared Dataspaces
- **Transaction Serializability**
- **Locking Policies** for Dataspaces
- **New Problem:**
  - Transactions and Temporary Data
- **Solutions:**
  - Enhanced locking policies
  - Loose Expired Data Collector

A Process Calculus

- Embed the coordination primitives in a process calculus

\[ P ::= 0 \mid m.P \mid n?P \mid l.P \mid K \]

- `m ::= write(a,u) \mid read(a) \mid take(a) \mid create(x) \mid commit(x)`

- `n ::= read³(a) \mid take³(a)`

Serialization in the Calculus

- Any computation has an equivalent **serialized** computation sequence
- For example
  - If \( P \rightarrow Q \rightarrow R \rightarrow S \) then
    - either \( P \rightarrow Q' \rightarrow R' \rightarrow S \)
    - or \( P \rightarrow Q'' \rightarrow R'' \rightarrow S \)
**write - read - take**

- We have modeled a (JavaSpaces like) strict expired data collection policy
- When a datum expires, it becomes immediately unavailable
- **Serializability Theorem**
  All computations are serializable
- **Corollary**
  The “standard” locks ensure serializability even in presence of temporary data

**Adding test-for-absence**

```
[read(x), read(y)]  
\[ x \rightarrow \$ \]  
[read(x) ? \_write(x)]
```

- This computation is **non-serializable**

**Plan of the Talk**

- **Temporary Data** in Shared Dataspaces
- **Transactions** in Shared Dataspaces
- Transaction Serializability
- Locking Policies for Dataspaces
- **New Problem**:
  - Transactions and Temporary Data
- **Solutions**:
  - Enhanced locking policies
  - Loose Expired Data Collector

**A First Possible Solution**

- **Expired data locks**
  - When a datum read under a transaction expires,
  - No test-for-absence of that datum can be executed until the transaction terminates

**Expired Data Locks in Action**

```
[read(x), read(y)]  
\[ x \rightarrow \$ \]  
[read(x) ? \_write(x)]
```

- The expired data locks **forbid** the execution of the test for absence

**Expired data locks \( \Rightarrow \) Serializability?**

```
[read(x), read(y)]  
\[ x \rightarrow \$ \]  
[read(x) ? \_write(x)]
```

- The expired data locks are **not** enough
Enhanced Expired Data Locks

- When a datum read under a transaction expires,
  - No test-for-absence of all those data which expire subsequently can be executed until the transaction terminates
- Under this enhanced lock the serializability theorem returns to hold
- This is a quite complex/restrictive policy!

An Alternative Solution:
Loose Expired Data Collection

- Data may remain available for reading or consumption even after expiration
- Advantages
  - The lock policy requires to record about expired data
  - The information could be accessed by processes
  - The temporal dependency can be easily broken
  - Expired data could be collected in any order
- Under this policy, the simple expired data locks are sufficient to ensure serializability

The Scheduling is now Serializable

\[ \text{[read(C), read(C)]} \]

- Under loose expired data collection the previous computation becomes \text{serializable}

Conclusion

- Lesson learned
  - When introducing temporary data the transactional semantics should be \text{revisited}
- Future work
  - \text{Deadlock} free policies
  - Notions alternative to \text{serializability}