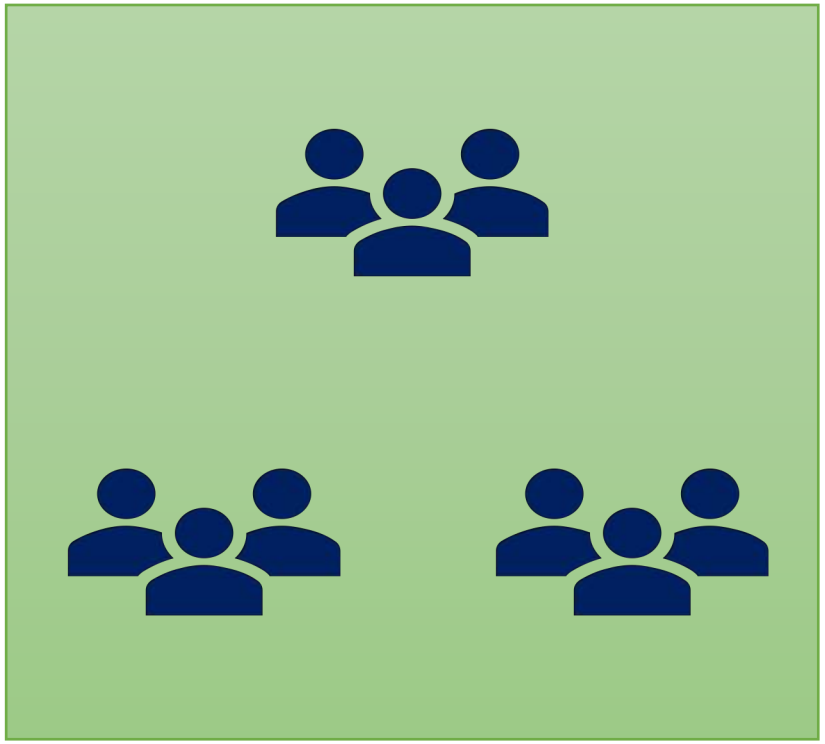
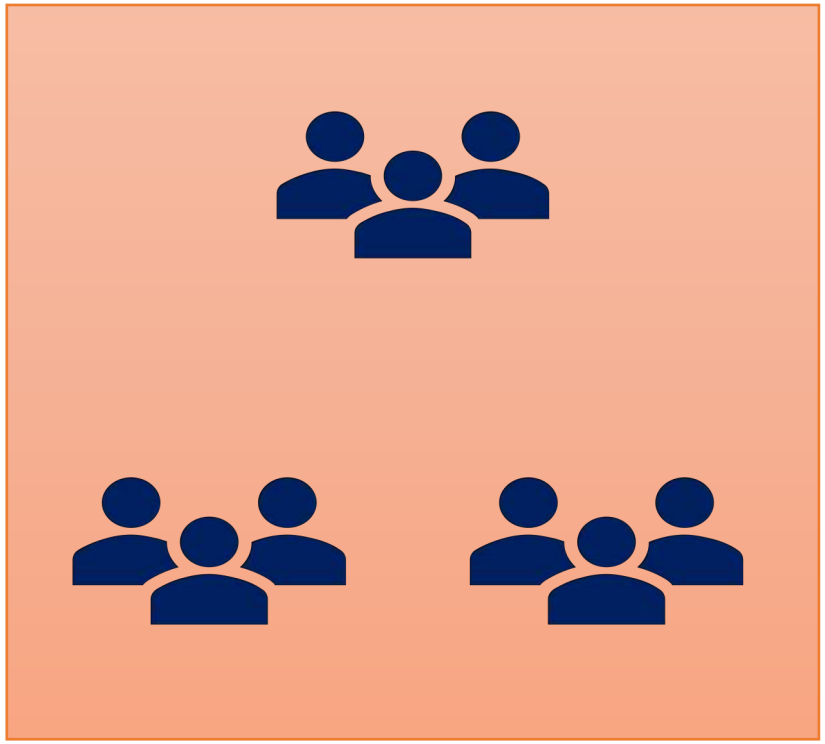
MixT: A Language for Mixing Consistency in Geo-distributed Transactions

Matthew Milano, Andrew C. Myers

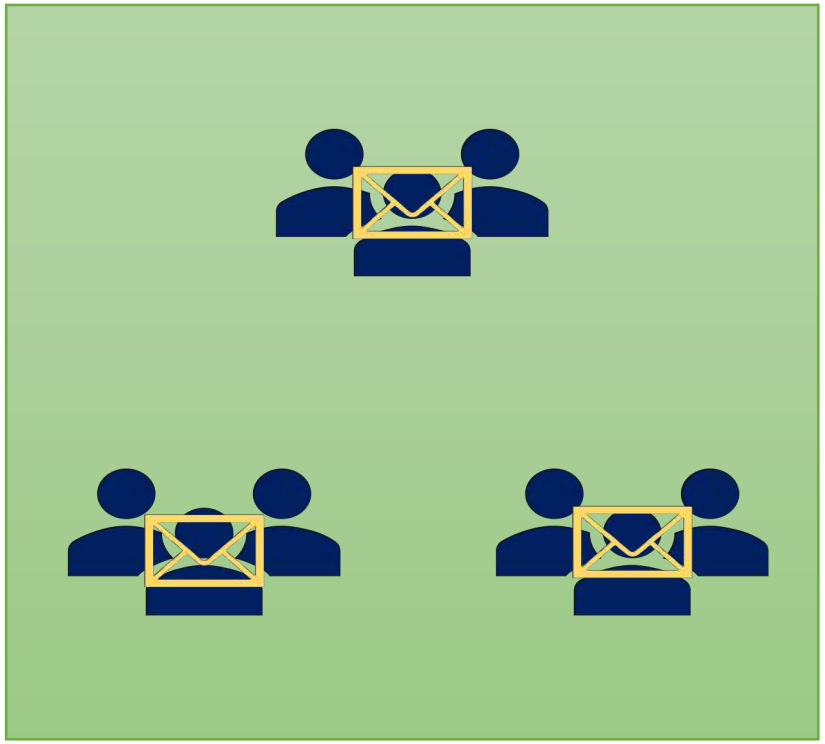
Presented By: Natasha Mittal

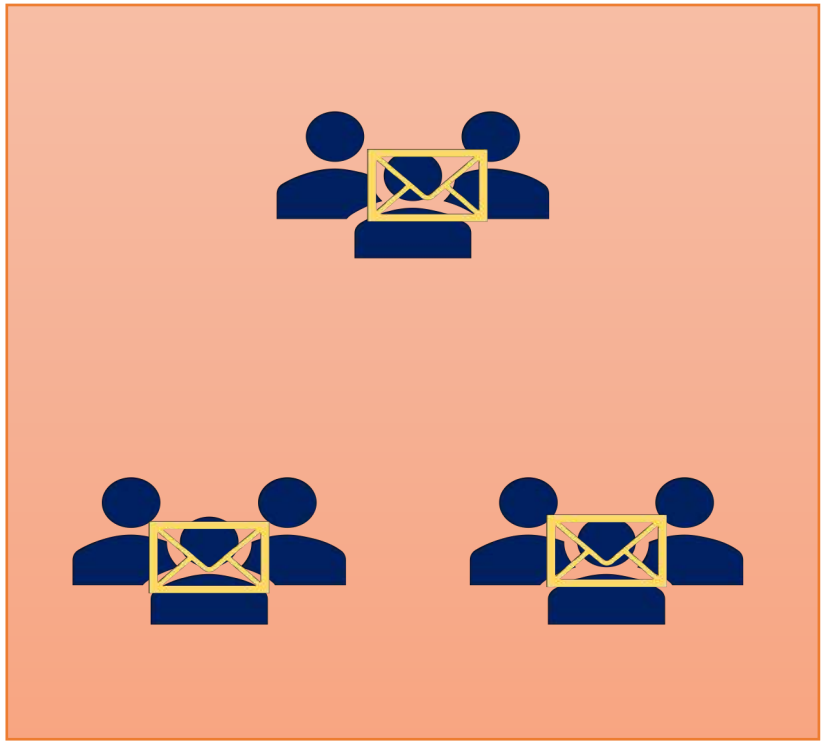
Mailing Delivery System





Mailing Delivery System



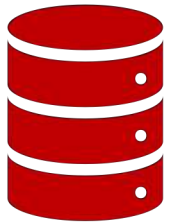


Implementing Message Group System

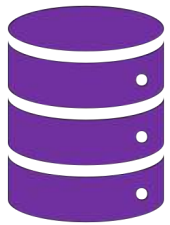
```
// Deliver message to mailing list
while(iterator.isValid()) {
      log.append(iterator.inbox.insert(post)),
      iterator = iterator.next
}
```

Mix it Up: Message Group System

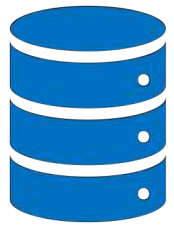
```
// Deliver message to mailing list
while(iterator.isValid()) {
      log.append(iterator.inbox.insert(post)),
      iterator = iterator.next
}
```



Linearizable



Causal

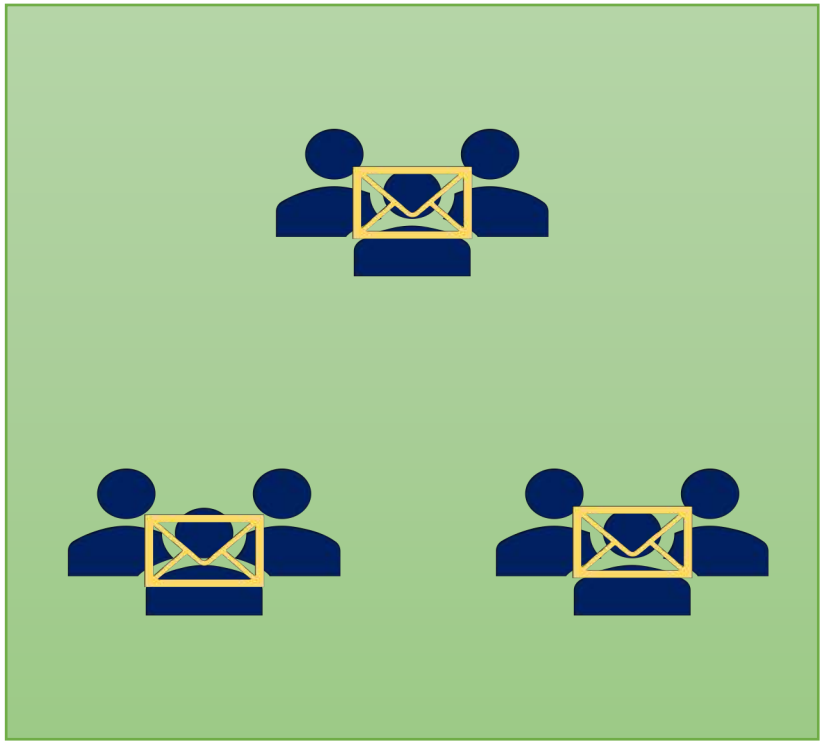


Eventual

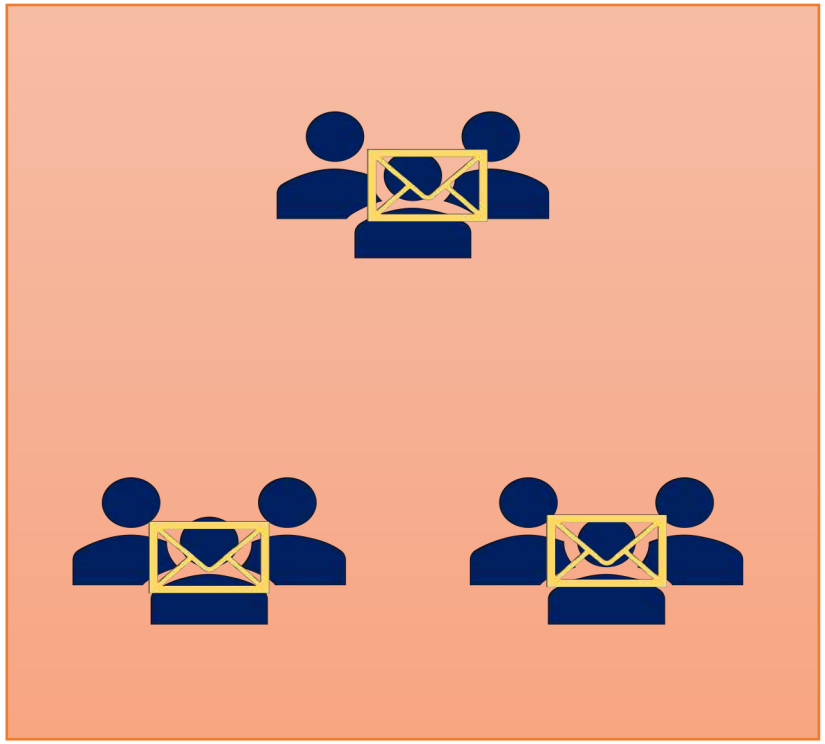
Mixing Consistency Breaks Strong Consistency

- Causal consistency and linearizability offer well-defined consistency guarantees.
- But trying to mix these levels in the same application can break the guarantees that both levels claim to offer.

Mailing Delivery System: Contest

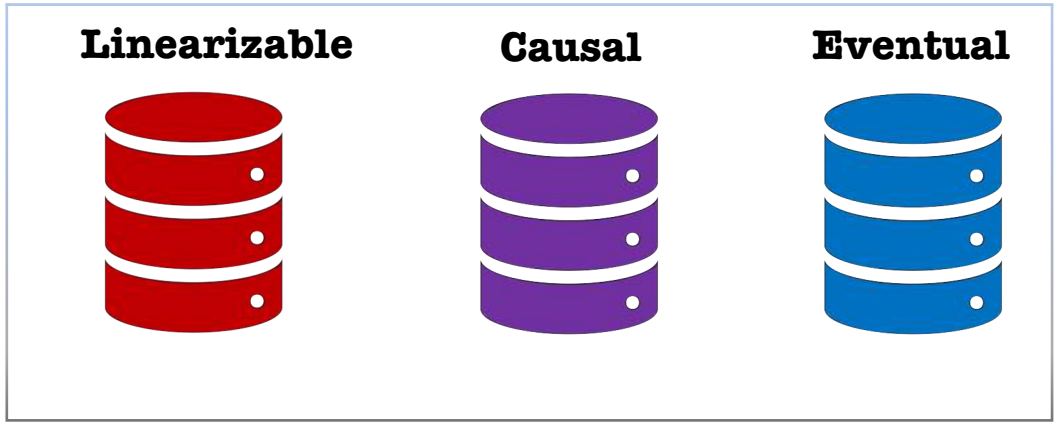


A



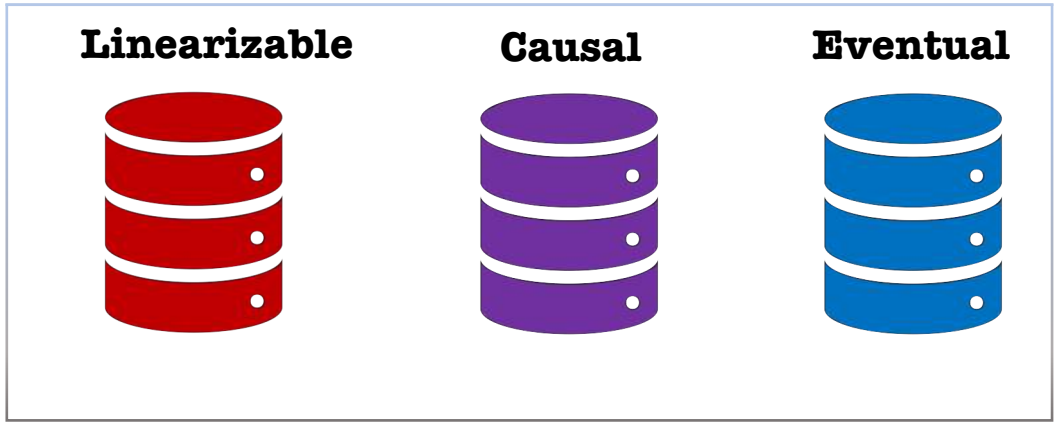
B

Implementing Message Group Contest




```
if (a.inbox.size() >= 100K &&
b.inbox.size() < 100K ) {

        a.declare_winner()

} else if (a.inbox.size() < 100K &&
b.inbox.size() >= 100K) {

        b.declare_winner()

}
```

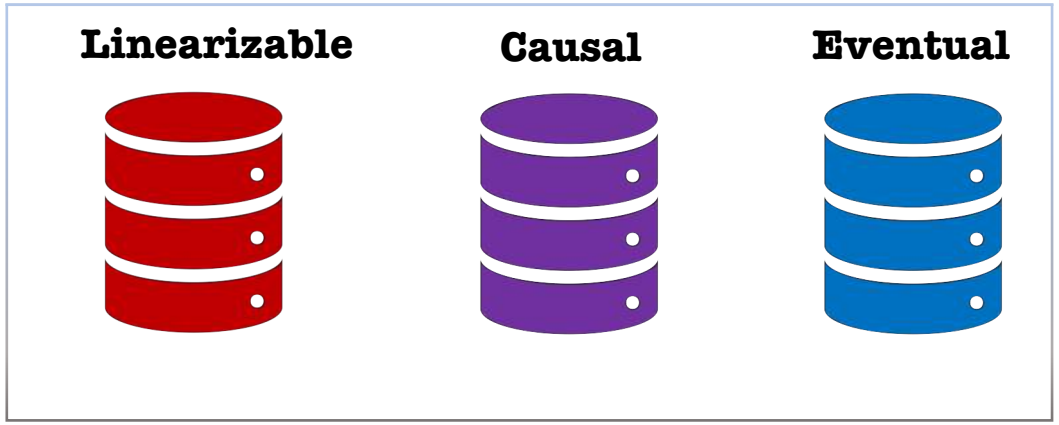
Implementing Message Group Contest




```
if (a.inbox.size() >= 100K &&
b.inbox.size() < 100K ) {
        a.declare_winner()
} else if (a.inbox.size() < 100K &&
b.inbox.size() >= 100K) {
        b.declare_winner()
}
```

Both groups can be declared winner

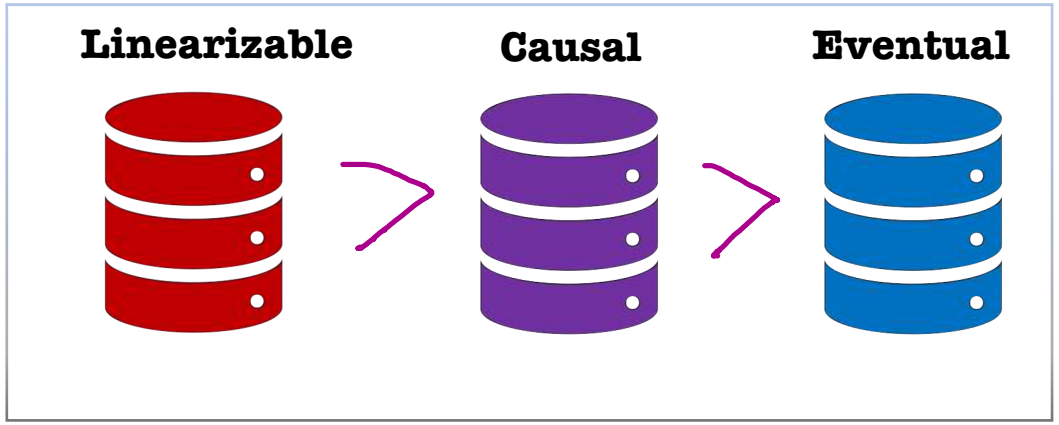
Implementing Message Group Contest




```
if (a.inbox.size() >= 100K &&
b.inbox.size() < 100K ) {
        a.declare_winner()
} else if (a.inbox.size() < 100K &&
b.inbox.size() >= 100K) {
        b.declare_winner()
}
```

Both groups can be declared winner
Because Inboxes can get out of sync

Implementing Message Group Contest




```
if (a.inbox.size() >= 100K &&
b.inbox.size() < 100K ) {
    a.declare_winner()
} else if (a.inbox.size() < 100K &&
b.inbox.size() >= 100K) {
    b.declare_winner()
}
```

Strong depends on weak

Use Information Flow to enable transactions which operate over multiple consistency models

Key Observation: Consistency is a property of information itself and not only of operations that use this information

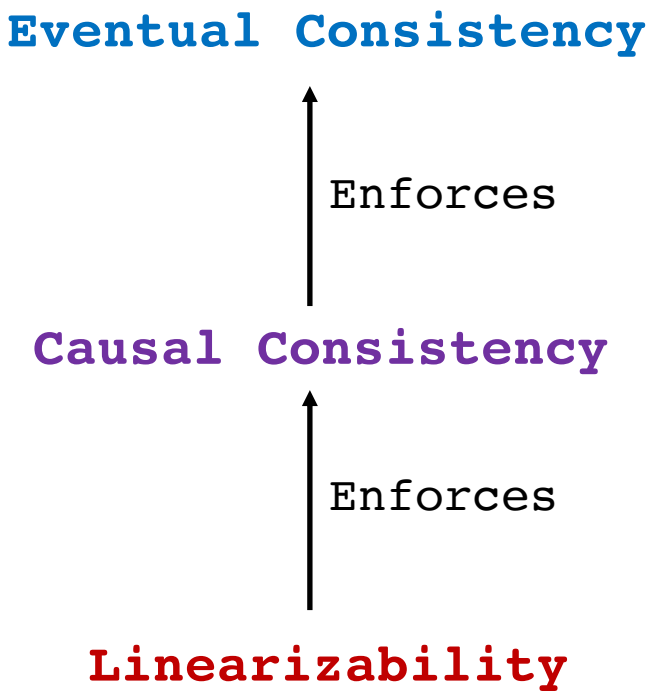
Requirement: Guarantee

- **Data keeps** its consistency **guarantees**
- **Atomicity:** All mutations appear to occur simultaneously
- **Low Latency:** Operations run at the fastest safe consistency level
- **Legacy system** support: Use the data which already exists in legacy systems

What is MixT?

- A **Domain Specific Language (DSL)** for mixed consistency
  - mixed consistency transactions
  - associates **consistency** with **data**
  - forbids weakly-consistent influence on strong data via **information flow**
  - **legacy system** support

Forbids Invalid Influence



Influence is invalid when:

- A mutation at level α
- Depends on a read at level β
- When β cannot enforce α

MixT surface syntax

$x \in \mathbf{Var} \qquad f \in \mathbf{Operation}$

$\oplus \in \mathbf{Binop} \qquad \ominus \in \mathbf{Unop}$

$(Location)\ m ::= x \mid *e \mid e.x \mid e \texttt{->} x$

$(Expr)\ e ::= m \mid e_1 \oplus e_2 \mid \ominus e \mid e_0.f(e_1, \ldots, e_n)$

$(Stmt)\ s ::= \texttt{var}\ x = e \mid m = e \mid \texttt{return}\ e$

$\mid \texttt{while}\ (e)\ s \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \{s_1, \ldots, s_n\}$

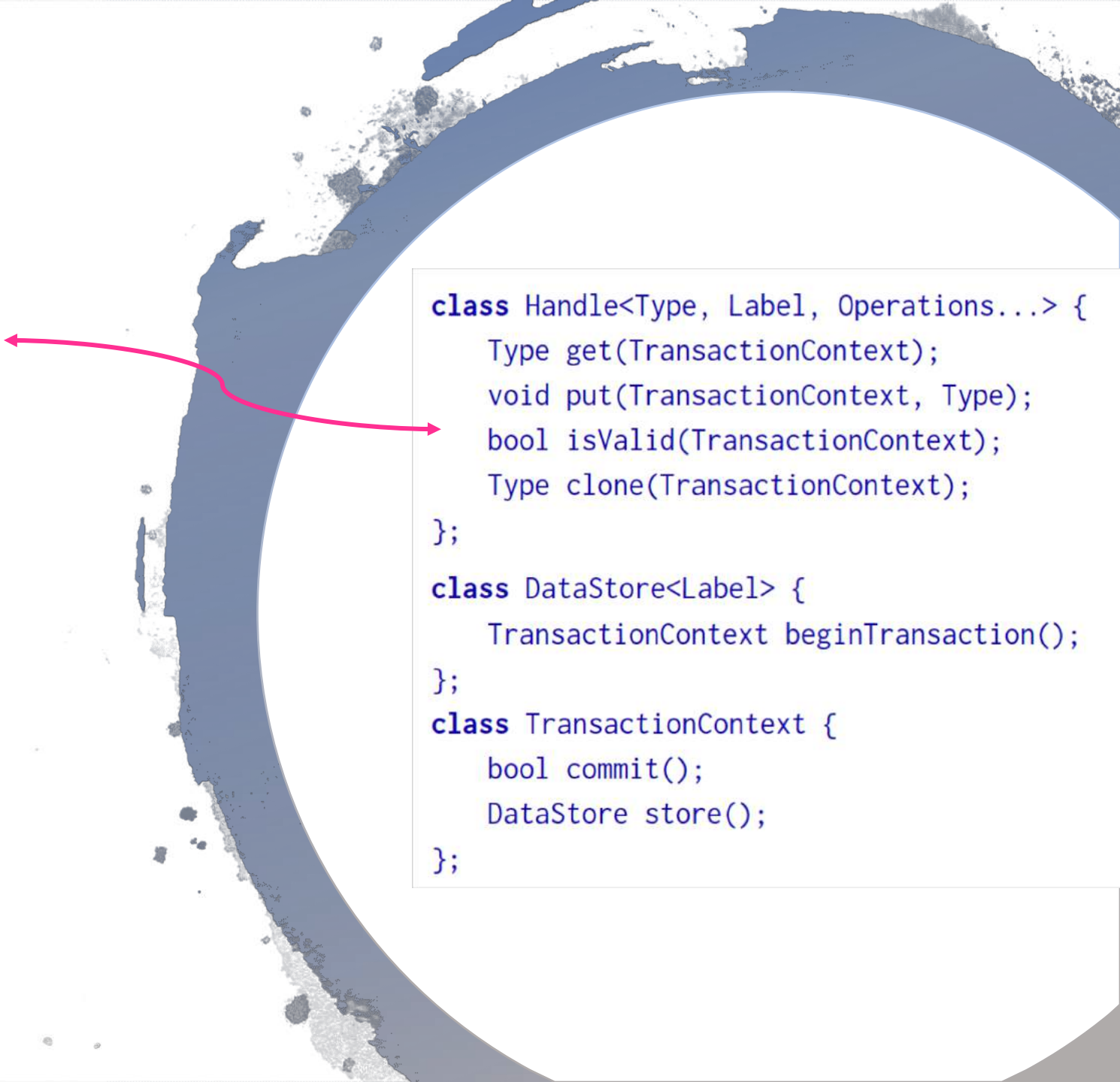
Handles behave like pointers

this.method()

MixT API

Handle Interface:

- Consists of get/put/check APIs for accessing underlying data
- Consists set of routines for accessing and using the datastore from which Handle originated

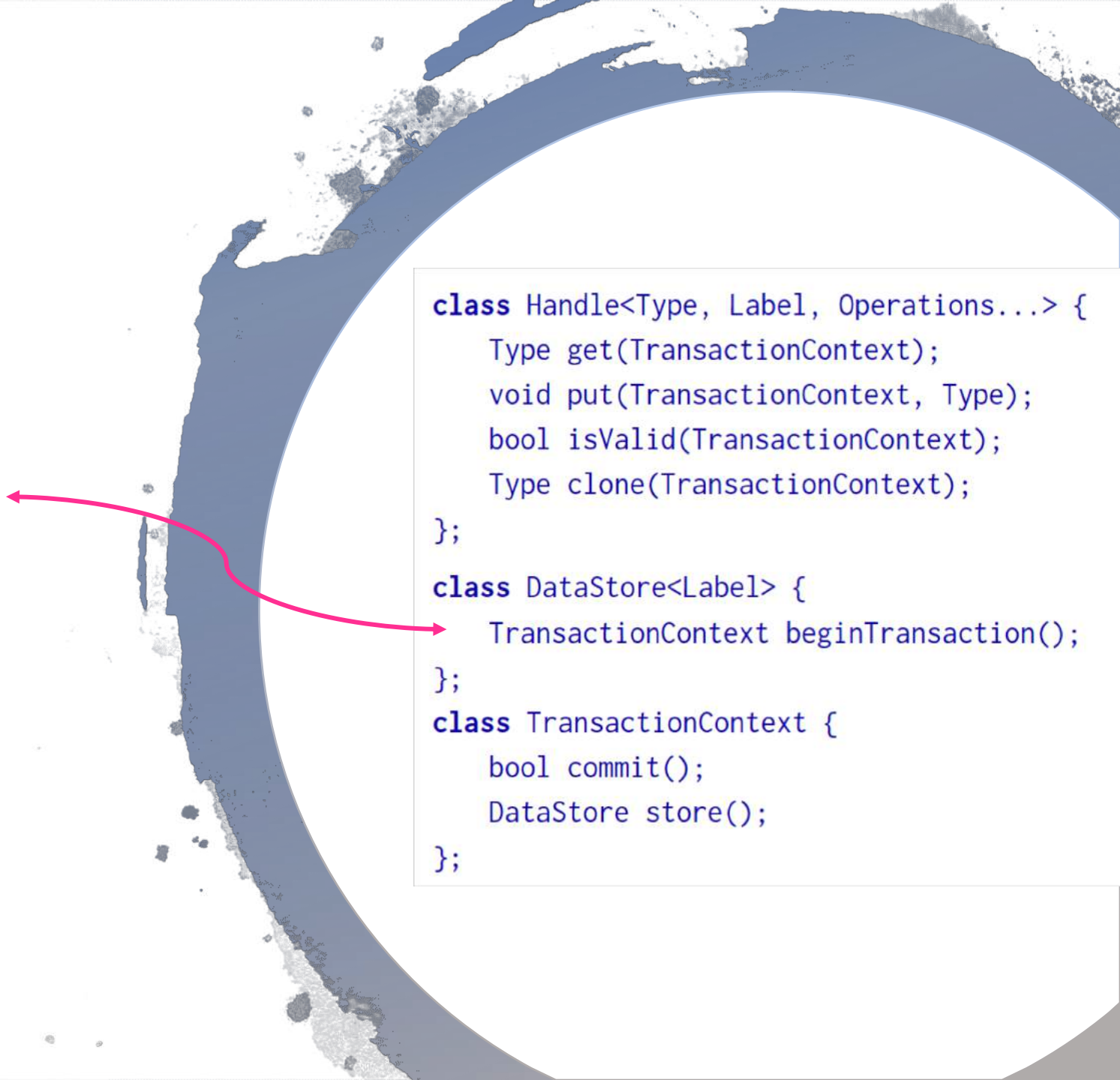


```
class Handle<Type, Label, Operations...> {
    Type get(TransactionContext);
    void put(TransactionContext, Type);
    bool isValid(TransactionContext);
    Type clone(TransactionContext);
};
class DataStore<Label> {
    TransactionContext beginTransaction();
};
class TransactionContext {
    bool commit();
    DataStore store();
};
```

MixT API

DataStore Interface:

- Serves as an entry point to underlying storage system
- Always associated with a consistency label (level) and a specific implementation of **Handle**



```
class Handle<Type, Label, Operations...> {
    Type get(TransactionContext);
    void put(TransactionContext, Type);
    bool isValid(TransactionContext);
    Type clone(TransactionContext);
};
class DataStore<Label> {
    TransactionContext beginTransaction();
};
class TransactionContext {
    bool commit();
    DataStore store();
};
```

MixT API

TransactionContext Interface:

- Used to commit or abort the transaction
- Can be extended to supply **DataStore** specific transaction interactions options

```
class Handle<Type, Label, Operations...> {
    Type get(TransactionContext);
    void put(TransactionContext, Type);
    bool isValid(TransactionContext);
    Type clone(TransactionContext);
};
class DataStore<Label> {
    TransactionContext beginTransaction();
};
class TransactionContext {
    bool commit();
    DataStore store();
};
```

MixT Message Delivery Implementation

```
1  class user {
2    Handle<set<string>, causal, supports<insert>> inbox;
3  };

5  class group {
6    RemoteList<Handle<user, causal>, linearizable> users;
7    Handle<Log, eventual, supports<append>> log;
8    mixt_method(add_post) (post) mixt_captures(users,log) (
9      var iterator = users;
10     while (iterator.isValid()) {
11       log.append(iterator->v.inbox.insert(post)),
12       iterator = iterator->next
13     }
14   )
15 };
```

Handles

MixT Code

MixT Message Delivery Implementation

```
1 class user {
2   Handle<set<string>, causal, supports<insert>> inbox;
3 };

5 class group {
6   RemoteList<Handle<user, causal>, linearizable> users;
7   Handle<Log, eventual, supports<append>> log;
8   mixt_method(add_post) (post) mixt_captures(users,log) (
9     var iterator = users,
10    while (iterator.isValid()) {
11      log.append(iterator->v.inbox.insert(post)),
12      iterator = iterator->next
13    }
14  )
15 };
```

MixT Code

- Inbox is a set of strings
- Stored at causal consistency
- Custom operation insert

Job of the causal store to ensure that insert operations from different clients are merged with causal consistency

What are the desired semantics of mixed consistency?

In a system with both eventual consistency and linearizability, **execution traces** involving any subset of of objects should adhere **at least** to **eventual consistency** and traces involving **only linearizable** objects should **respect linearizability**.

What are the desired semantics of mixed consistency?

$$l \sqsubseteq l' \Leftrightarrow T_l \subseteq T_{l'}$$

- T_l is a set of traces with consistency level l
- $T_l \subseteq T_{l'} \rightarrow T_l$ provides more guarantees than $T_{l'}$
- Each trace $t \in T$ is a sequence of events e

What are the desired semantics of mixed consistency?

- An event e is a 5-tuple (a, o, l, v, S) where
  - a is the action corresponding to this event
  - o is the exact memory location or object referenced by this event
  - l is the consistency level of the store for this event's location
  - v is the tuple of values processed by this event
  - S is the client session in which this event occurred
  - $consistency((a, o, l, v, S)) = l$

What are the desired semantics of mixed consistency?

Trace Projection:

$$t \downarrow l = [\, e \mid e \in t \wedge consistency(e) \sqsubseteq l$$

Given a trace t, the events relevant to a given consistency level l are those whose consistency level is **at least** as **strong**.

What are the desired semantics of mixed consistency?

Trace Projection:

$$t \downarrow l = [\, e \mid e \in t \wedge consistency(e) \sqsubseteq l$$

Mixed Consistency:

$$\forall l, t \downarrow l \in T_l$$

A trace t exhibits mixed consistency if it **satisfies** every consistency model T_l when projected onto that consistency level

There is always some minimum consistency model onto which all events can be projected.

Noninterference

- Given a **policy lattice** of **security labels**, program is **secure** when program behavior at one point in the policy lattice **cannot influence** behavior at levels that are **lower** in the lattice.

- When using **noninterference** to enforce privacy or confidentiality, two runs of a program that **differ** only in **weakly consistent inputs** should have **identical strongly consistent** observable **behavior.**

- To determine if any weakly consistent data can influence any strongly consistent data, sets of possible runs of transactions are considered:

  - by keeping the **deterministic** program **inputs fixed**
  - but **varying** the **nondeterministic choices** made by the program

Example?

Generalized Noninterference

Varying weakly consistent data should only affect strongly consistent values in ways already permitted by the inherent nondeterminism of the system

MixT Transaction Splitting

```
while(iterator.isValid()) {
        record(iterator)
        iterator = iterator→next
}
for(it: recorded(iterator)) {
        tmp = it.inbox.insert(post),
        record(tmp)
}
for(tmp': recorded(tmp)) {
        log.append(tmp')
}
```

Linearizability

Causal Consistency

Eventual Consistency

MixT Transaction Splitting

```
while(iterator.isValid()) {
        record(iterator)
        iterator = iterator→next
}
for(it: recorded(iterator)) {
        tmp = it.inbox.insert(post),
        record(tmp)
}

for(tmp': recorded(tmp)) {
        log.append(tmp')
}
```

results of each conditional test for the loop is recorded

replaying them during loop execution in subsequent phases

Splitting does not guarantee atomicity

Commits to strong stores happens before commits to weaker stores!

Witness

- During each phase's execution,
  - **write witness object** is created for each mutation
  - indicates that a **lock** has been acquired on object being mutated
- At the end of each phase,
  - **commit witness object** indicates that all locks acquired during this transaction have been released
- If MixT transaction encounters write witness, it must suspend execution until it encounters the corresponding commit witness

By creating an explicit object during each transaction and blocking future progress until it has appeared, guarantees atomicity

MixT Flattened Syntax

$$
\begin{aligned}
(\mathit{Location})\ m &::= x \mid m.x \\
(\mathit{Expr})\ e &::= m \mid x_1 \oplus x_2 \mid \ominus x \mid x_0.f(x_1, \ldots, x_n) \\
(\mathit{Stmt})\ s &::= \texttt{var}\ x\ =\ e\ \texttt{in}\ s \mid \texttt{remote}\ x\ =\ e\ \texttt{in}\ s \\
&\quad \mid m\ =\ x \mid \texttt{return}\ x \mid \texttt{while}\ (x)\ s \\
&\quad \mid \texttt{if}\ (x)\ s_1\ \texttt{else}\ s_2 \mid \{s_1, \ldots, s_n\}
\end{aligned}
$$

Instead of using pointer like syntax, remote variables are used which directly correspond to referenced location on underlying store

Selected consistency typing rules for MixT

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell_1 \qquad \Gamma \vdash e : \tau \qquad \Delta, x_V : \ell \mid \Gamma, x_V : \tau \mid pc \vdash s : \ell_2 \qquad \ell_1 \sqsubseteq \ell \qquad x \notin \Gamma \qquad x \notin \Delta}{\Delta \mid \Gamma \mid pc \vdash \texttt{var}\ x\ \texttt{=}\ e\ \texttt{in}\ s : \ell}$$

$$\frac{\Gamma \vdash e : \texttt{Handle}\langle \tau, \ell_1 \rangle \qquad \Delta \mid \Gamma \mid pc \vdash e : \ell_2 \qquad \Delta, x_R : \ell \mid \Gamma, x_R : \tau \mid pc \vdash s : \ell' \qquad \ell_1 \sqcup \ell_2 \sqsubseteq \ell \qquad x \notin \Gamma \qquad x \notin \Delta}{\Delta \mid \Gamma \mid pc \vdash \texttt{remote}\ x\ \texttt{=}\ e\ \texttt{in}\ s : \ell}$$

REMOTE-READ

$$\frac{pc \sqsubseteq \ell}{\Delta, x_R : \ell \mid \Gamma \mid pc \vdash x : \ell} \qquad \Delta, x_V : \ell \mid \Gamma \mid pc \vdash x : \ell$$

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell \qquad \Delta \mid \Gamma \mid pc \sqcup \ell \vdash s : \ell'}{\Delta \mid \Gamma \mid pc \vdash \texttt{while}\ (e)\ s : \ell}$$

- **Δ** : keeps track of labels
- **Γ** : keeps track of types of variables
- V : local variable
- R : remote variable
- pc : program counter

Selected transaction splitting rules for MixT

$[\![\cdot]\!]_\ell$: generates code for transaction phase at consistency level l

(*Expr*) $e ::= \ldots \mid \texttt{rand()} \mid \texttt{peek}(x)$

(*Stmt*) $s ::= \ldots \mid \texttt{release\_all}(n)$
$\mid \texttt{acquire}(x, n, \ell_1, \ldots, \ell_n)$
$\mid \texttt{advance}(x) \mid \texttt{advance remote}(x)$

$$[\![\texttt{remote } x = e \texttt{ in } s : \ell]\!]_\ell \triangleq \texttt{remote } x = e \texttt{ in } [\![s]\!]_\ell$$

$$\frac{\ell \not\sqsubseteq \ell'}{[\![\texttt{remote } x = e \texttt{ in } s : \ell]\!]_{\ell'} \triangleq [\![s]\!]_{\ell'}}$$

$$\frac{\ell \sqsubseteq \ell'}{[\![\texttt{remote } x = e \texttt{ in } s : \ell]\!]_{\ell'} \triangleq \{\texttt{advance binding}(x), [\![s]\!]_{\ell'}\}}$$

$$[\![\texttt{while } (e)\ stmt : \ell]\!]_\ell \triangleq \texttt{while } (e)\ [\![stmt]\!]_\ell$$

$$\frac{\ell \not\sqsubseteq \ell'}{[\![\texttt{while } (e)\ stmt : \ell]\!]_{\ell'} \triangleq \{\}} \qquad \frac{\ell \neq \ell' \qquad \ell \sqsubseteq \ell'}{[\![x = e : \ell]\!]_{\ell'} \triangleq \texttt{advance}(x)}$$

$$\frac{\ell \neq \ell' \qquad \ell \sqsubseteq \ell'}{[\![x : \ell]\!]_{\ell'} \triangleq \texttt{peek}(x)} \qquad [\![x : \ell]\!]_\ell \triangleq x$$

Satisfying Mixed Consistency

Mixed Consistency:

$$\forall l, t \upharpoonright l \in T_l$$

A trace t exhibits mixed consistency if it **satisfies** every consistency model T_l when projected onto that consistency level

Satisfying Mixed Consistency

Mixed Consistency:

$$\forall l, t \upharpoonright l \in T_l$$

A trace t exhibits mixed consistency if it **satisfies** every consistency model T_l when projected onto that consistency level

How to verify?

Satisfying Mixed Consistency

Mixed Consistency:

$$\forall l, t \upharpoonright l \in T_l$$

A trace t exhibits mixed consistency if it **satisfies** every consistency model T_l when projected onto that consistency level

- MixT associates each data **object/memory** location with a **single** consistency model
- All **reads** of some value from memory location at consistency level ℓ must be **paired** with **write** to that location at level ℓ
- For all **reads** in a **trace**, the $\upharpoonright$ **preserves** all **matching writes** in that **trace**
- $\upharpoonright$ selects **sets** of **memory locations**

Satisfying Mixed Consistency

Mixed Consistency:

$$\forall l, t \upharpoonright l \in T_l$$

A trace t exhibits mixed consistency if it **satisfies** every consistency model T_l when projected onto that consistency level

- MixT associates **each data object**/memory location with a **single** consistency model
- All **reads** of some value from memory location at consistency level ℓ must be **paired** with **write** to that location at level ℓ
- For all **reads** in a **trace**, the $\upharpoonright$ **preserves** all **matching writes** in that **trace**
- $\upharpoonright$ selects **sets** of **memory locations**

No proof!

Endorsement

```
    if ((a.inbox.strong_read().size() >= 1000000 &&
         b.inbox.strong_read().size() < 1000000)
          .endorse(strong)) {
      a.declare_winner()
  } else
    if ((a.inbox.strong_read().size() < 1000000 &&
         b.inbox.strong_read().size() >= 1000000)
          .endorse(strong)) {
      b.declare_winner()
  }
```

Endorsement

```
    if ((a.inbox.strong_read().size() >= 1000000 &&
         b.inbox.strong_read().size() < 1000000)
          .endorse(strong)) {
      a.declare_winner()
  } else
    if ((a.inbox.strong_read().size() < 1000000 &&
         b.inbox.strong_read().size() >= 1000000)
          .endorse(strong)) {
      b.declare_winner()
  }
```

What's wrong?

Endorsement

```
if (((a.inbox.size() >= 1000000 &&
      b.inbox.size() < 1000000)
  && (a.inbox.strong_read().size() >= 1000000 &&
      b.inbox.strong_read().size() < 1000000))
       .endorse(strong)) {
    a.declare_winner()
} else
if (((a.inbox.size() < 1000000 &&
      b.inbox.size() >= 1000000)
  && (a.inbox.strong_read().size() < 1000000 &&
      b.inbox.strong_read().size() >= 1000000))
       .endorse(strong)){
    b.declare_winner()
  }
```

Discussion

What are the similarities and differences with the IPA paper from last week?

Discussion

What are the similarities and differences with the IPA paper from last week?

What if sometimes you want to access a particular data object in a more consistent way, and other times a less consistent way?

Example?

How to integrate this in MixT?

Discussion

What are the similarities and differences with the IPA paper from last week?

What if sometimes you want to access a particular data object in a more consistent way, and other times a less consistent way?

What if you want the latency bound that IPA offered?

How to integrate this in MixT?