ETH zürich

Mechanized Data Consistency Models for Distributed Database Transactions

Shabnam Ghasemirad

Master’s Thesis Presentation

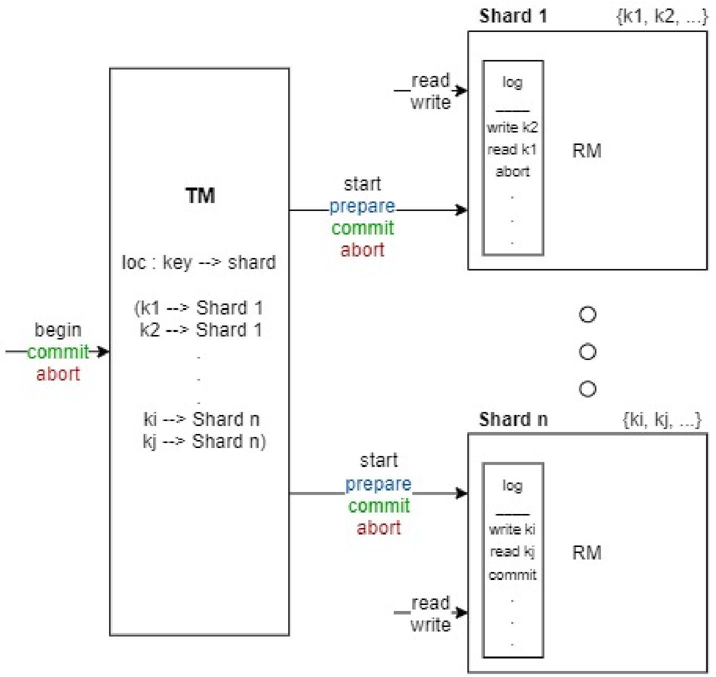
9 November 2022, Information Security Group, ETH Zürich

Table of contents

Preliminaries

Distributed database systems

- Consists of independent components,
  spread across different machines or locations.
  - A common model:
    Transaction Manager (TM) and Resource Managers (RM)
  - TM coordinates (sends commands to) RMs
  - Each RM has a shard of database or a log
  - Each RM responsible for a subset of keys



- CAP theorem [12]: Distributed database systems can only provide two of the following guarantees:
  - (strong) Consistency
  - Availability
  - Partition tolerance

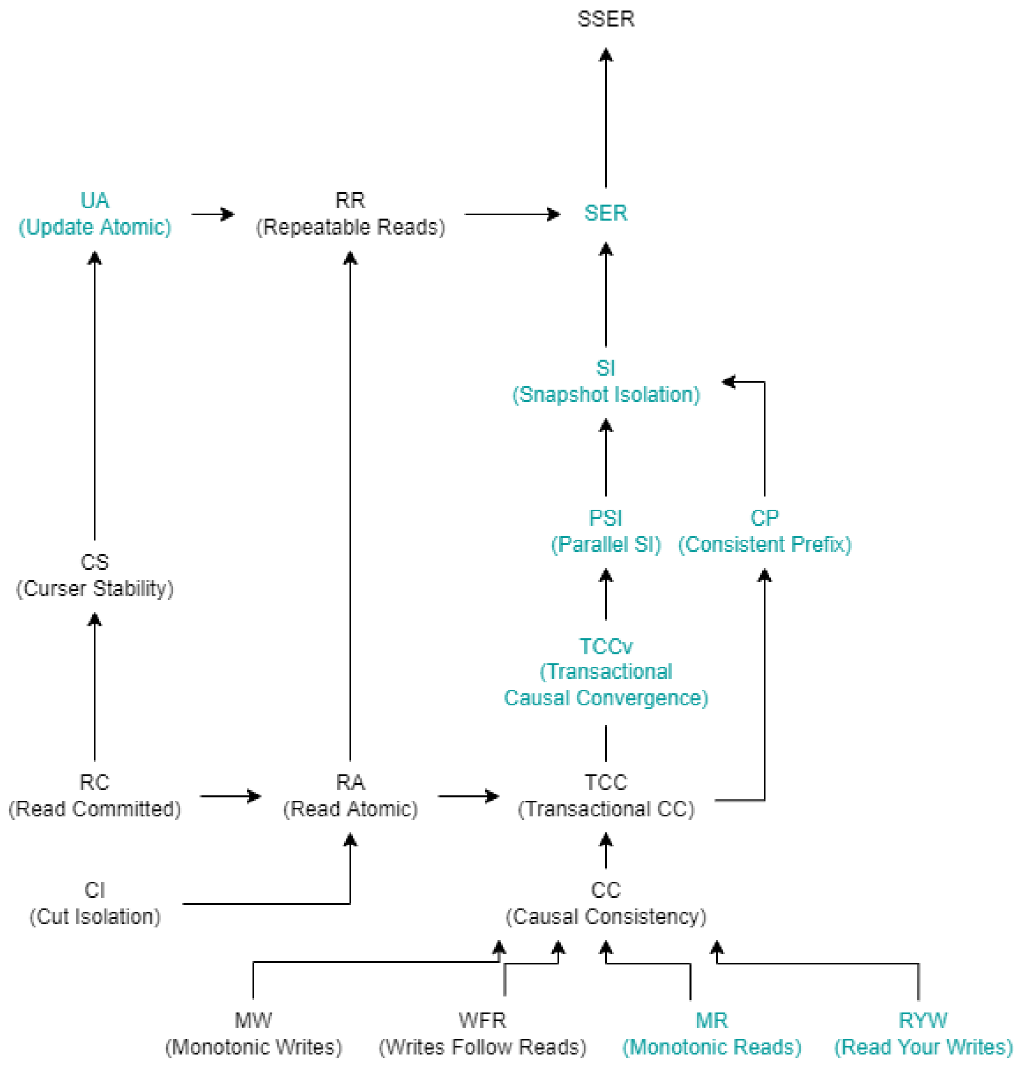
- High scalability and availability → weaker transactional consistency guarantees (consistency models)

Data consistency models

Definition: A contract between a distributed data store and user processes with certain transactional consistency guarantees.

Some of the most widely used Consistency Models

- Read Atomicity
  - a transaction's writes become visible atomically
- Causal Consistency
  - Causal relationships between transactions are preserved.
  - t visible ➔ what t sees also visible.
- Parallel Snapshot Isolation
  - CC + no write conflicts.
- (Strict) Serializability
  - Transactions are totally ordered, mimicking sequential execution.



Data consistency models

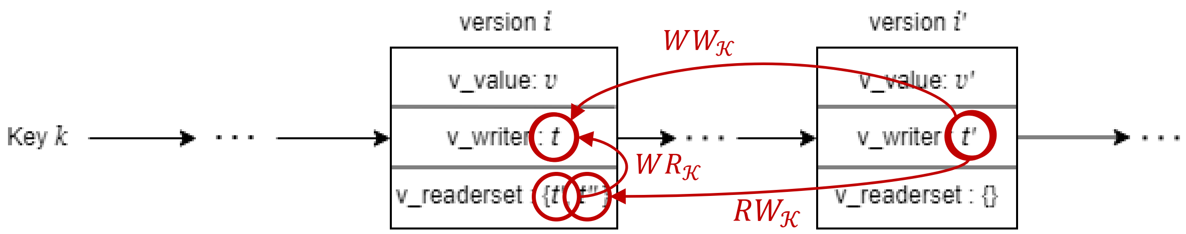
- Originally defined by engineers and quite informal

- General definitions, independent of a particular implementation
  - Declarative semantics
    - dependency graphs [8]
    - abstract executions [9]

  - Operational semantics: Xiong et al. [10] → centralized model: key-value store
    - General semantics for weak consistency models
    - Verify reference implementations
    - Analyse the behavior of client programs with respect to a particular consistency model

Dependency relations

- Proposed by Adya's PhD thesis [8] and used for dependency graphs in distributed database transactions.
- Dependency relations are a basis for **formalizing consistency models**.
- Three possible dependency relations between two transactions in kv-store $\mathcal{K}$ on key k:
  1. **write-read (WR)**: t' reads a version of k written by t
  2. **write-write (WW):** t' writes a newer version of k written by t
  3. **read-write (RW):** t' writes a version of k and t reads an older version (anti-dependency)



The operational framework of consistency models

Centralized Operational Semantics

- One global Key-Value store: $\mathcal{K}: KEY \rightarrow List\ (VERSION)$
  - A mapping from keys to version lists (a value, a writer, and a reader-set for each version)
  - In reality, the database may be sharded and replicated
  - Each client may have a different view on its current content
- Client view: $u \in VIEWS(\mathcal{K}) \triangleq KEY \rightarrow \mathcal{P}(\mathbb{N})$
  - Subset of versions of each key that a given client sees
  - Explicitly represented in the centralized model
- Configuration:
  - A pair $(\mathcal{K}, \mathcal{U})$ where $\mathcal{K}$ is a key-value store and $\mathcal{U}$ is a function from clients to their views
- Transactions:
  - Update the kv-store and client view using an atomic transaction → subject to execution test

version
v_value: v
v_writer : t
v_readerset : $\{t', t''\}$

Centralized Operational Semantics

- Fingerprints: $\mathcal{F}: KEY \times \{R, W\} \rightarrow VALUE$
  - A partial function from key and type of operation (Read/Write) to value
  - Can be constructed for a transaction to capture its effect on the kv-store
    - Only the **first read** of a key is stored, only if it happens before a write (snapshot property)
    - Only the **last write** of a key is stored (last-write-wins policy)

Limitations and assumptions

1. A ***last-write-wins*** policy is hard-wired in this framework.
2. Four assumptions for well-formedness of *key-value stores*, including the **snapshot property:** Each transaction reads at most one and writes at most one version of each key.
3. Two assumptions for well-formedness of *client views*, including **atomic view**: All versions written by one transaction for different keys are all in the view or all absent from it, which reflects baseline RA.

We have proven these properties as **invariants** of our model; they are preserved through events.

Execution Tests

An execution test, **ET**, is a client local transition for committing a transaction t:

$$\frac{\mathbf{canCommit_{ET}}\ (\mathcal{K},\ u,\ \mathcal{F}) \quad \mathbf{vShift_{ET}}\ (\mathcal{K},\ u,\ \mathcal{K}',\ u') \quad t \in \mathrm{NextTxID}\ (cl,\ \mathcal{K}) \qquad \mathcal{K}' = \mathrm{UpdateKV}\ (\mathcal{K},\ u,\ \mathcal{F},\ t) \quad \forall k, v.\ \ \mathcal{F}(k, R) = v\ \ \rightarrow \mathcal{K}(k,\ max_{<}(u(k))) = v}{cl \vdash (\mathcal{K},\ u) \xrightarrow{\mathcal{F}}_{ET} (\mathcal{K}',\ u')}$$

- $\mathbf{canCommit_{ET}}$: a condition that must be satisfied for a **commit** to take place, parametrized by **ET**
- $\mathbf{vShift_{ET}}$: a constraint on the possible **updates** to the committing **client's view,** parameterized by **ET**
- The kv-store is updated by applying the fingerprint $\mathcal{F}$ to the old kv-store, using fresh TID t.
- Read value in the fingerprint is the newest version in the client's view (**fingerprint property**)

Execution Tests

- Set of visible transactions for a client with view u in the key-value store $\mathcal{K}$:

$$visTx(\mathcal{K}, u) = \{v\_writer(\mathcal{K}(k,i)) \mid i \in u(k)\}$$

- $\mathbf{canCommit_{ET}}(\mathcal{K}, u, \mathcal{F})$ is defined as a ***view closure*** for u with respect to $\mathcal{K}$, and a relation R_{ET} $\triangleq$ $\mathbf{closed}(\mathcal{K}, u, R_{ET})$ and it holds if and only if: $visTx(\mathcal{K}, u) = (R_{ET}^{*})^{-1}(visTx(\mathcal{K}, u)) \backslash$(read-only txns)

ET	$\mathtt{canCommit_{ET}}(\mathcal{K}, u, \mathcal{F}) \triangleq \mathtt{closed}(\mathcal{K}, u, R_{\mathtt{ET}})$	$\mathtt{vShift_{ET}}(\mathcal{K}, u, \mathcal{K}', u')$
MR	true	$u \sqsubseteq u'$
RYW	true	$\forall t \in \mathcal{K}' \setminus \mathcal{K}. \forall k, i. (\mathsf{w}(\mathcal{K}'(k,i)), t) \in \mathsf{SO}^{?} \Rightarrow i \in u'(k)$
CC	$R_{\mathtt{CC}} \triangleq \mathsf{SO} \cup \mathsf{WR}_{\mathcal{K}}$	$\mathtt{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
UA	$R_{\mathtt{UA}} \triangleq \bigcup_{(\mathtt{W}, k, \_) \in \mathcal{F}} \mathsf{WW}_{\mathcal{K}}^{-1}(k)$	true
PSI	$R_{\mathtt{PSI}} \triangleq R_{\mathtt{UA}} \cup R_{\mathtt{CC}} \cup \mathsf{WW}_{\mathcal{K}}$	$\mathtt{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
CP	$R_{\mathtt{CP}} \triangleq \mathsf{SO}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{WR}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}}^{?} \cup \mathsf{WW}_{\mathcal{K}}$	$\mathtt{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SI	$R_{\mathtt{SI}} \triangleq R_{\mathtt{UA}} \cup R_{\mathtt{CP}} \cup (\mathsf{WW}_{\mathcal{K}}; \mathsf{RW}_{\mathcal{K}})$	$\mathtt{vShift_{MR \cap RYW}}(\mathcal{K}, u, \mathcal{K}', u')$
SER	$R_{\mathtt{SER}} \triangleq \mathsf{WW}_{\mathcal{K}}^{-1}$	true

Xiong et al.[10] – Figure 6

Programming language

Transition rule for an atomic transaction $[T]$:

$\rightsquigarrow^*$: Transactional multi-step

Implies that $\mathcal{F}$ has the ***fingerprint property***

$$\frac{u \sqsubseteq u'' \quad \sigma = snapshot(\mathcal{K},\ u'') \quad (s, \sigma, \emptyset),\ T \rightsquigarrow^* (s',\ \_,\ \mathcal{F}),\ skip \quad \mathbf{canCommit_{ET}}\ (\mathcal{K},\ u'',\ \mathcal{F}) \qquad t \in \mathrm{NextTxID}\ (cl,\ \mathcal{K}) \quad \mathcal{K}' = \mathrm{UpdateKV}\ (\mathcal{K},\ u'',\ \mathcal{F},\ t) \quad \mathbf{vShift_{ET}}\ (\mathcal{K},\ u'',\ \mathcal{K}',\ u')}{cl \vdash (\mathcal{K},\ u,\ s),\ [T] \xrightarrow{(cl,\ u'',\ \mathcal{F})}_{ET} (\mathcal{K}',\ u',\ s'),\ skip}$$

- For modeling **client programs**, in addition to the abstract model state (configuration), we need ***client-local stacks*** for storing values in variables and expression evaluations.
- A more concrete model of the execution test, including how the fingerprint $\mathcal{F}$ is built by the client's program.
- We proved that the programming language model **refines** the abstract execution test model.
- This implies *property* preservation, so the well-formedness of the key-value stores and client views is preserved for the programming language model.

Protocol verification by refinement

- Many different concurrency control mechanisms in distributed systems protocols
  - RAMP protocol [1] $\xrightarrow{\text{satisfies}}$ RA
  - COPS protocol [6] $\xrightarrow{\text{satisfies}}$ CC
  - Eiger-PORT [10] $\xrightarrow{\text{satisfies}}$ CC — Modeled in Isabelle/HOL, refinement proof in progress
  - Two-Phase Locking $\xrightarrow{\text{satisfies}}$ SER — Modeled and proven in the Isabelle/HOL framework

- The complex concurrent behaviors of these protocols call for formal verification
- To verify if the protocol satisfies the desired consistency model

- Idea: Verifying database concurrency control protocols by proving that they **refine** the formalized execution test of the desired consistency model.

Protocol verification by refinement

- Refinement guarantees trace inclusion, hence **property preservation.**
- If a protocol refines a **consistency model execution test**, it preserves its consistency guarantees

  ➔ satisfies the consistency model

- Protocol verification by refinement consists of two steps:
  - modeling the protocol as an **event system**.
  - proving that the modeled protocol refines the **execution test** of the desired consistency model.

For the refinement proof of each protocol the following must be done:

- The abstract state is reconstructed from the concrete state.
- Prove that the events that refine ***skip*** of the abstract model do not change the state.
- For the events refining the ***abstract commit*** (execution test transition rule) all the **ET** rule premises are satisfied by the reconstructed abstract state (**5** conditions + **3** view well-formedness)

1^{st} protocol verification
2PL + 2PC satisfying serializability

2PL + 2PC protocol

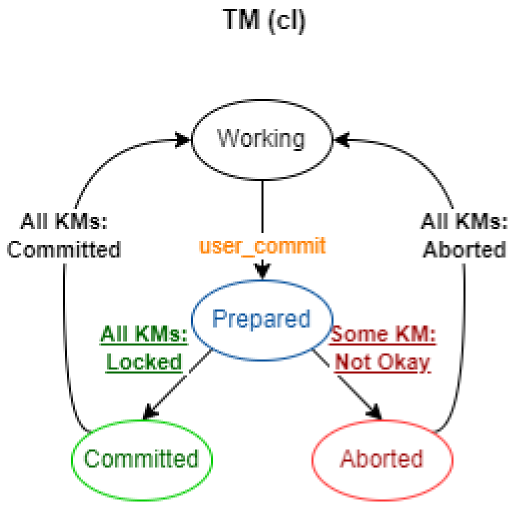
The two-phase locking (2PL) protocol works on top of a two-phase commit (2PC) protocol.

Each resource manager is responsible for one key ➔ Key Manager (KM)

Transaction Manager

TM state
- Status
- Sequence Number (last sqn of client)
- View (of client on the kv-store)

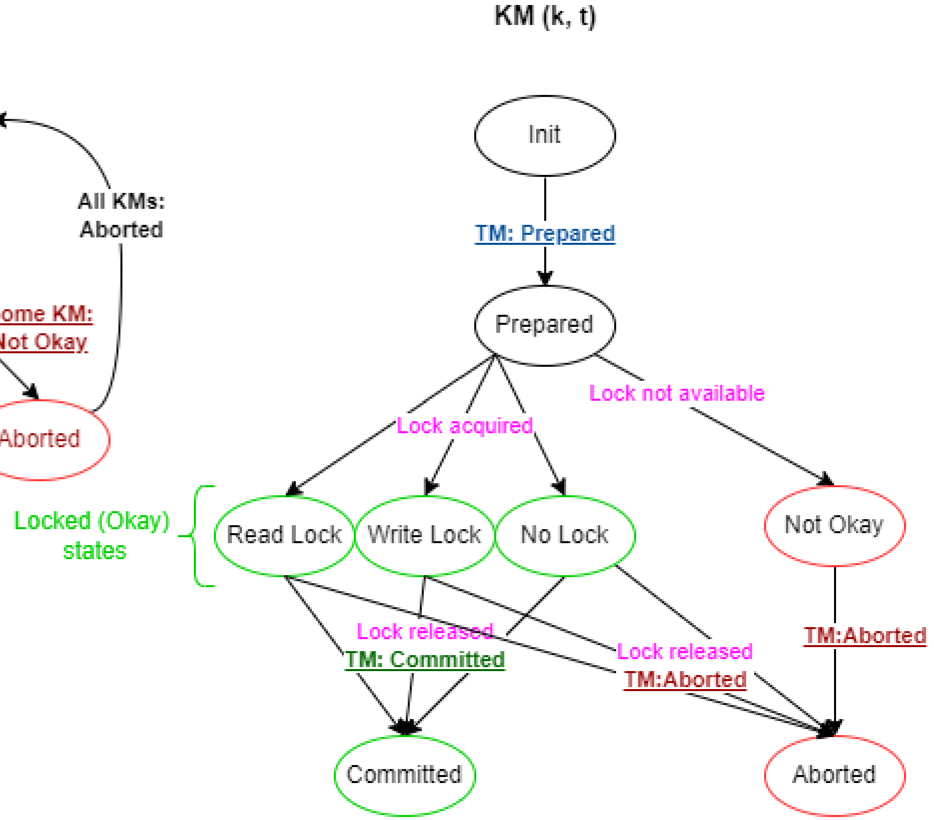
One for each client.



Key Manager

KM state
- Status
- Local Key Version List
- Local Key Fingerprint

One for each key and transaction pair.



Verifying 2PL + 2PC protocol satisfying SER

The proof is done by **refinement :**

The protocol refines the transaction commit under the serializability execution test

- **Mediator function**: TM_commit refines the *abstract commit*, the rest of the events refine *skip*
- **Simulation function** (mapping the state):

  The state consists of the global kv-store and a mapping from clients to their views

  - KV-store: A function that applies the **pending read and writes** prematurely to the kv-store, as soon as the KM is in one of the OK states and TM has committed, to simulate the effect of these updates on the kv-store.
  - Client views: We keep track of the view for each client as it grows and make sure it will be extended to the full (global) view after each commit.

Proof obligations

- The desired consistency model is **serializability**
- $\mathbf{vShift_{SER}} = \mathbf{true}$
- $\mathbf{canCommit_{SER}}(\mathcal{K}, u, \mathcal{F}) \triangleq \mathbf{closed}(\mathcal{K}, u, R_{SER})$ where $R_{SER} \triangleq WW_K^{-1}$

  $\Leftrightarrow\ visTx(\mathcal{K}, u) = WW_{\mathcal{K}}^{*}\ (visTx(\mathcal{K}, u)) \backslash$(read-only txns)

  $\Longrightarrow$ *Intuition*: All writes that are overwritten by the visible transactions, should be visible.

  $\Longrightarrow$ all writer transactions are visible $\Longrightarrow$ view is **complete**, and the client sees everything in the key-value store.
- For **view well-formedness** we show that a kv-store expansion preserves the view well-formedness.
- For the **fingerprint property** and **transaction id freshness** we prove invariants.
- For the **update** to the kv-store ($UpdateKV$) we show that the protocol commit makes the same changes.

2nd protocol verification
Eiger-PORT+ satisfying causal+ consistency

The Eiger-PORT protocol

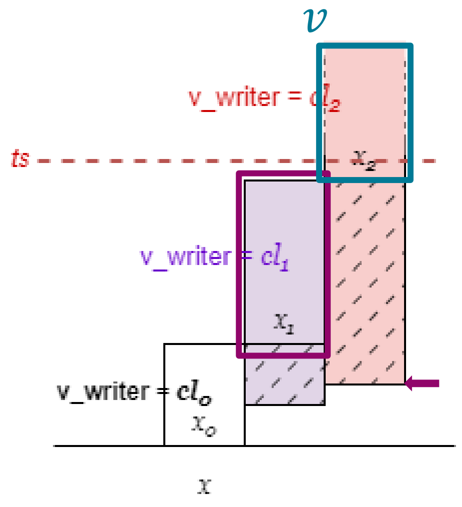
- The Eiger-PORT protocol is a performance-optimized version of an older protocol Eiger [22], that supports read-only and write-only transactions.
- Eiger uses **Lamport clock**s [17] to assign a timestamp to each committed write.
- **Lamport clocks** can create a partial order on events (used to capture causality relations).
- ***Write-only transactions*** in Eiger follow a variation of the ***2PC*** protocol that ***always commits***.
- ***Read-only transactions*** are optimized in Eiger-PORT to be non-blocking, need one round of communication, and have constant metadata.
- Eiger-PORT must also satisfy *Read Your Writes* (RYW) and *Read atomicity* (RA) as they are weaker than CC.

The Eiger-PORT protocol

A read-only transaction in Eiger-PORT:

- Each server has a *local safe time* (lst), and clients keep record of these lsts in a lst_map and update their *global safe time* (gst) to the minimum of those each time before reading.
- The steps for reading a version at the read timestamp ts are as follows:

1. The responsible server retrieves the last available version v committed before or at the timestamp ts.
2. Then it scans all versions **newer** than v to find a version written by **the same client**.
3. If such a version is found it is returned to the client to satisfy **RYW**.
4. Otherwise, the originally retrieved v returned if it is **not written by the same client**.
5. If v is written by the same client, a function ***find_isolation*** is called that looks for any **older** versions than v that has been committed after the start of v by **another client** and returns it if found. (intended for write-isolation)



The Eiger-PORT protocol

The global state consists of the client state and the server state: (transition labels are events)

Client state:

- **`txn_state`**
- **`txn_sn`**
- **`gst`**
- **`lst_map`**
- **`cl_view`**

$$\texttt{txn\_state} \left[\begin{array}{l} \textbf{\textit{Idle}} \\ \textbf{\textit{RtxnInProg}}\ keys\ kv\_map \\ \textbf{\textit{WtxnPrep}}\ kv\_map \\ \textbf{\textit{WtxnCommit}}\ ts\ kv\_map \end{array} \right.$$

One for each client.

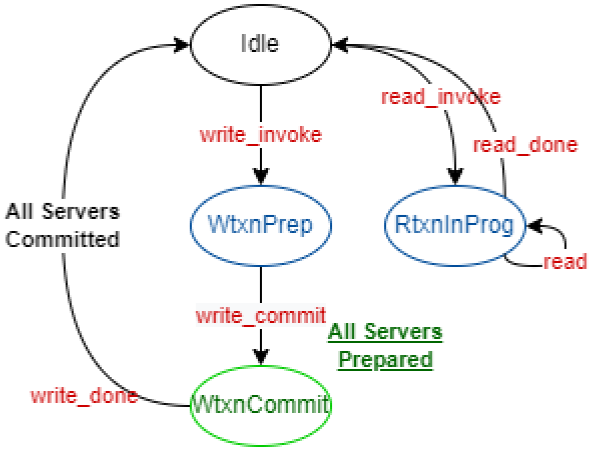
Server state:

- **`wtxn_state`**
- **`clock`**
- **`lst`**
- **`pending_wtxns`**
- **`DS`**

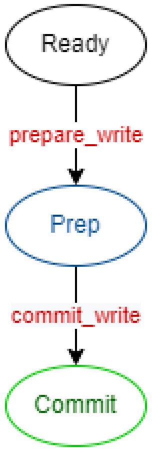
$$\texttt{wtxn\_state} \left[\begin{array}{l} \textbf{\textit{Ready}} \\ \textbf{\textit{Prep}}\ ts\ v_{id} \\ \textbf{\textit{Commit}} \end{array} \right.$$

One for each server(key) and transaction id pair.

Client - txn_state



Server - wtxn_state
(given a key and transaction id)



Server - registering reads on the server
(given a key and transaction id)

Not modeled by a state variable, but by a search on the kv-store

Protocol formalization overview

- We noticed that Xiong et al.[10] framework implicitly assumes a **total order** on non-causally-ordered writes to a key, by having fixed version indices (identifiers) that denote order.
- All clients see the versions of a key's version list in the same order ➔ **stronger** than CC.
- This is called **causal+ consistency (CC+)** or **causal convergence (CCv)** in the literature.
- It is the *de facto* of production CC database systems, e.g., used in Cure [23] and MongoDB [24].
- The Eiger-PORT protocol [7] satisfies plain CC, so we **optimized** it to satisfy CC+ instead.
- The optimization is also expected to improve performance by removing the usage of the *find_isolation* function.
- This optimization does not break write isolation and preserves atomic visibility.

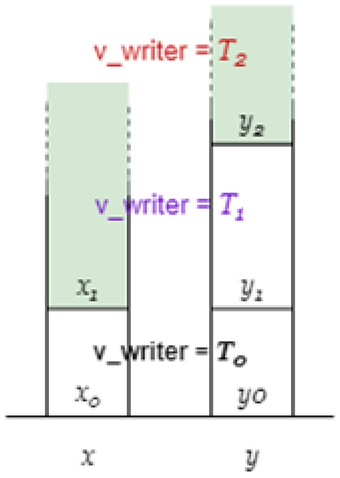
Fractured reads and atomic visibility

Let us review the definition of atomic visibility based on the *anomaly* it avoids: (taken from [1])

Fractured Reads: A transaction T_j exhibits the fractured reads phenomenon, if transaction T_i writes versions x_a of x and y_b of y, T_j reads version x_a of x and version y_c of y, and $c < b$ (y_c is older than y_b).

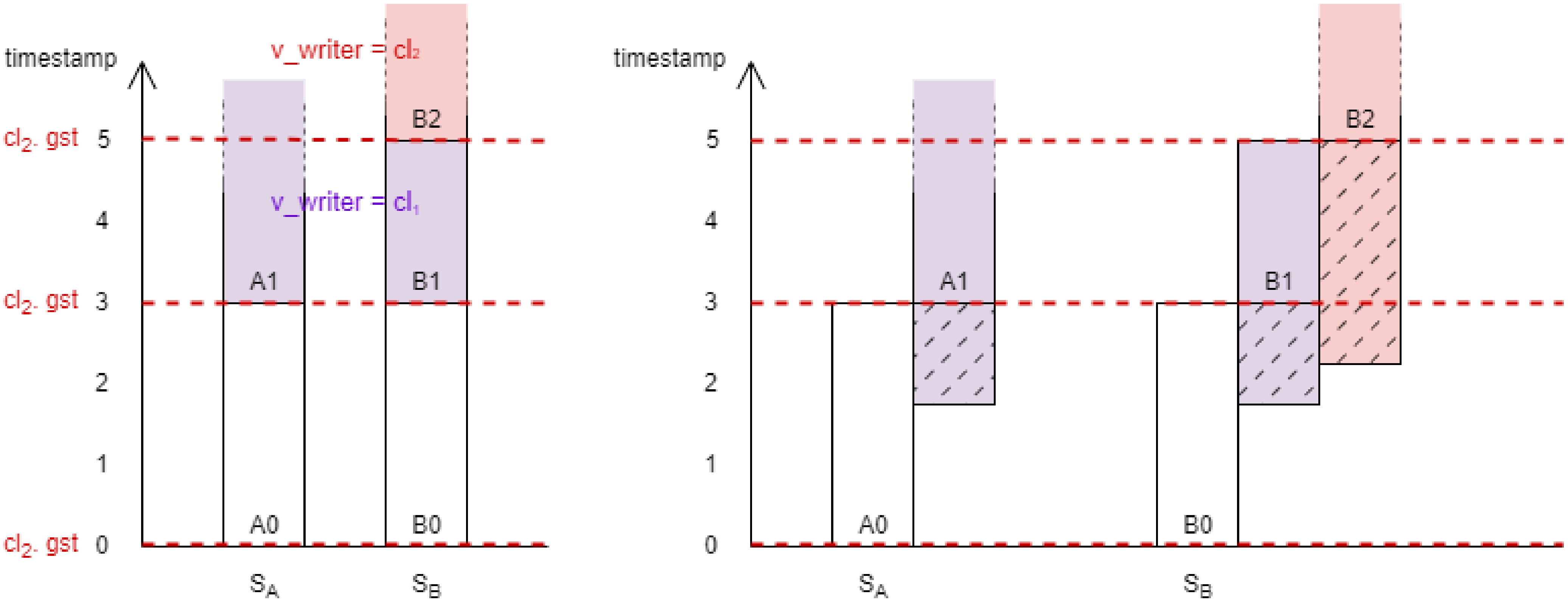
For example, for two objects x and y, if T_0 writes x_0 and y_0, T_1 writes x_1 and y_1, and T_2 writes only y_2:

- (x_0, y_0) is not a fractured read
- (x_0, y_1) is a fractured read
- (x_0, y_2) is **not** a fractured read
- (x_1, y_0) is a fractured read
- (x_1, y_1) is not a fractured read
- (x_1, y_2) is **not** a fractured read



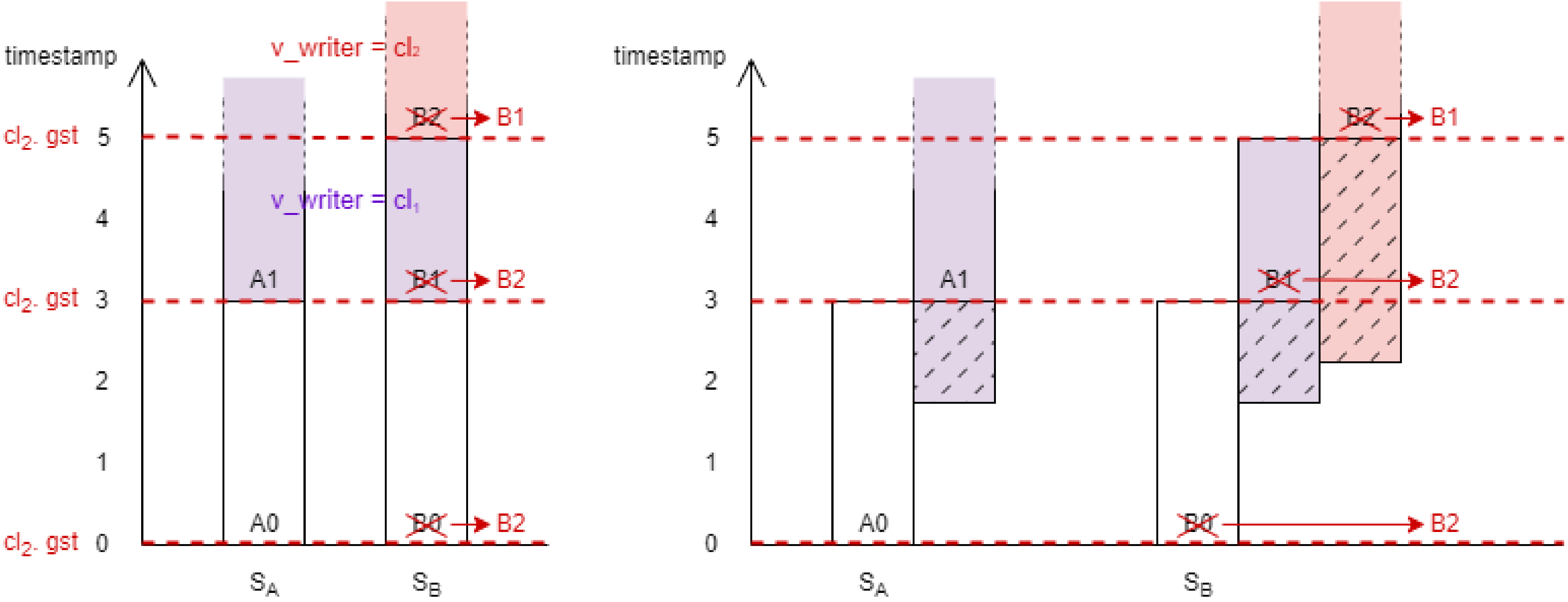
Example

- Two servers S_A and S_B each responsible for one key: A and B



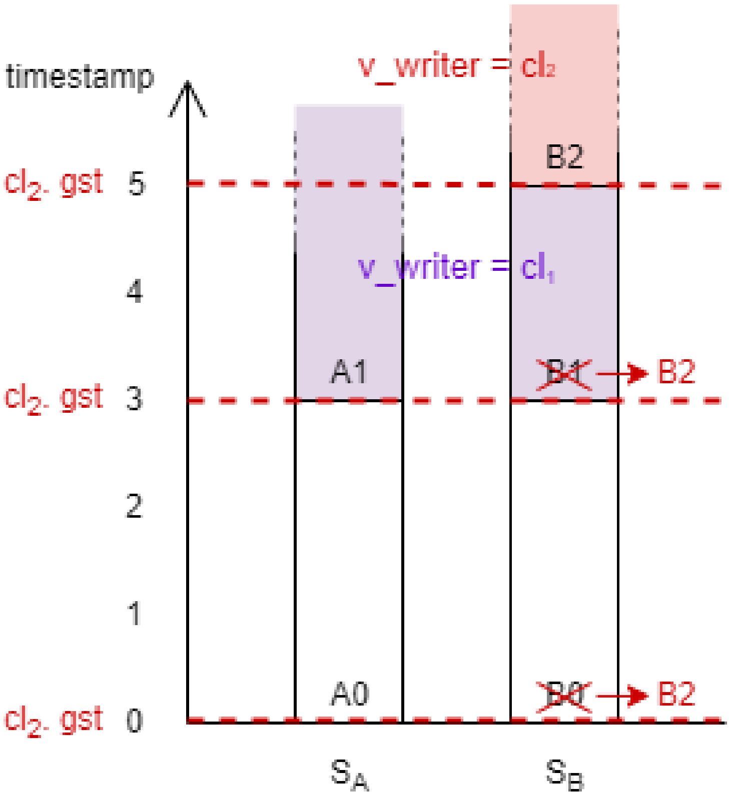
Eiger-PORT, client cl_2's perspective of the example

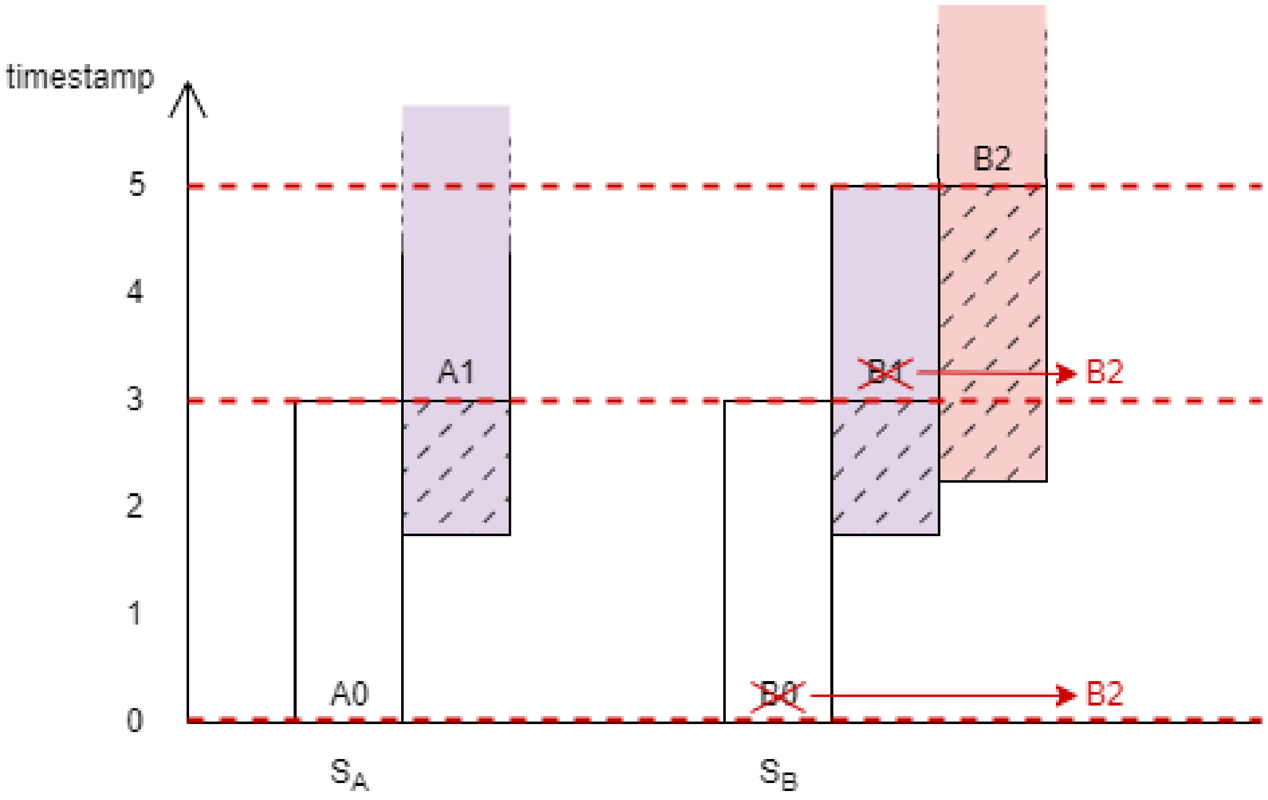
- Order of versions for client cl_1: $B0 \rightarrow B1 \rightarrow B2$, for client cl_2: $B0 \rightarrow B2 \rightarrow B1$.



Eiger-PORT+, client cl_2's perspective of the example

- Order of versions for client cl_1: $B0 \rightarrow B1 \rightarrow B2$, for client cl_2: $B0 \rightarrow B2$.





Verifying Eiger-PORT+ protocol satisfying CC+

The proof is done by **refinement :**

The protocol refines the transaction commit under the causal+ consistency execution test

- **Mediator function**: $read\_done$ and $write\_commit$ refine the *abstract commit*, the rest of the events refine *skip*
- **Simulation function** (mapping the state):

  The state consists of the global kv-store and a mapping from clients to their views

  - KV-store: Two filters are used to remove **pending writes** unless the client indicates a *commit*, and **pending reads**. Extra fields (version $v\_is\_pending$, v_ts, and v_gst) are also removed.
  - Client views: We keep track of the view for each client as it grows and make sure it corresponds to the abstract model client's view after each commit.

Invariants for the refinement proof

1. **Monotonicity lemmas:** *Lamport clocks ,* local safe time (lst) of servers, the client's lst_map and gst all increase monotonically. It is given for clocks and all the others follow eachother in increasing.

2. **Timestamps inequality:** for a given client always holds:

$$\forall svr.\, gst(cl) \leq lst\_map[svr](cl) \leq lst(svr) \leq clock(svr)$$

3. **KV-store invariants:** Establish that the kv-store is always non-empty and has at least one committed version.

4. **System state for past and future write TIDs:** Past transactions with a sequence number smaller than `txn_sn` are committed and future transactions with a larger sequence number than `txn_sn` are ready.

A list of all invariants available on the code base of the thesis.

Conclusions

Conclusion

Contributions

- ✓ Formalizing a general framework of consistency models
- ✓ Proving the assumptions of the framework as invariants in our model
- ✓ Using the framework for protocol verification: The first general and fully mechanized correctness proof of a concurrency control protocol in such a framework
- ✓ Verification of Two-Phase Locking + Two-Phase Commit protocol satisfying Serializability
- ✓ Formalizing a candidate state-of-the-art protocol which is the representative implementation of its consistency model (Eiger-PORT for CC)
- ✓ Optimizing the protocol (Eiger-PORT+) to satisfy a stronger consistency model CC+

Conclusion

Future Work

- Finishing the verification proof of the optimized Eiger-PORT+ protocol and analyzing the resulting performance improvement
- Developing a stronger and more generic operational framework of consistency models by relaxing some restrictions established by the framework of Xiong et al.[10], for example, by:
  1. Relaxing the **snapshot property** to deal with:
     a) Weaker consistency models: Monotonic Atomic View (MAV) and Read Committed (RC)
     b) Recent transaction algorithms which allow reads to fetch prepared-only versions: RAMP or LORA
  2. Relaxing the **last-write-wins** policy, to handle algorithms like RAMP-1PW.
- Refining the operational framework of consistency models to achieve a more generic (parameterized) distributed model.
- Continuing the refinement of the protocols in order to connect it to the Igloo framework and get further refinements into Igloo I/O specifications.

Thank you for your attention!

Questions?

References

- [1] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Scalable atomic visibility with ramp transactions," ACM Transactions on Database Systems (TODS), vol. 41, no. 3, pp. 1-45, 2016.
- [2] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 401-416.
- [3] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011, pp. 385-400.
- [4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," ACM SIGMOD Record, vol. 24, no. 2, pp. 1-10, 1995.
- [5] C. H. Papadimitriou, "The serializability of concurrent database updates," Journal of the ACM (JACM), vol. 26, no. 4, pp. 631-653, 1979.
- [6] R. Rajan, J. Boyle, A. Sastry, R. Cohen, D. Durham, and S. Herzog, "The COPS (Common Open Policy Service) Protocol," RFC 2748, Jan. 2000. [Online]. Available: https://www.rfc-editor.org/info/rfc2748
- [7] H. Lu, S. Sen, and W. Lloyd, "Performance-Optimal Read-Only transactions," in 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, Nov. 2020, pp. 333-349. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/lu
- [8] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.

References

- [9] A. Cerone, G. Bernardi, and A. Gotsman, "A framework for transactional consistency models with atomic visibility," in 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015, ser. LIPIcs, L. Aceto and D. de Frutos-Escrig, Eds., vol. 42. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 58-71. [Online]. Available: https://doi.org/10.4230/LIPIcs.CONCUR.2015.58
- [10] S. Xiong, A. Cerone, A. Raad, and P. Gardner, "Data consistency in transactional storage systems: A centralised semantics," in 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference), ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 21:1-21:31. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2020.21
- [11] C. Sprenger, T. Klenze, M. Eilers, F. Wolf, P. Müller, M. Clochard, and D. Basin, "Igloo: Soundly linking compositional refinement and separation logic for distributed systems verification," in ACM Program. Lang. 4, OOPSLA, Article 152, 2020. [Online]. Available: https://doi.org/10.1145/3428220
- [12] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," ACM Sigact News, vol. 33, no. 2, pp. 51-59, 2002.
- [13] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," Proc. VLDB Endow., vol. 7, no. 3, p. 181-192, nov 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237
- [14] ANSI X3.135-1992, American National Standard for Information Systems -- Database Language -- SQL (includes ANSI X3.168-1989). November, 1992.
- [15] S. Xiong, "Parametric operational semantics for consistency models," Ph.D. dissertation, Imperial College London, 2019.

References

- [16] M. Van Steen and A. S. Tanenbaum, Distributed systems. Maarten van Steen Leiden, The Netherlands, 2017.
- [17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system,"Commun. ACM, vol. 21, no. 7, p. 558-565, jul 1978. [Online]. Available: https: //doi.org/10.1145/359545.359563
- [18] B. Liskov and R. Ladin, "Highly available distributed services and fault-tolerant distributed garbage collection," in Proceedings of the fifth annual ACM symposium on Principles of distributed computing, 1986, pp. 29-39.
- [19] T. Nipkow, M. Wenzel, and L. C. Paulson, Eds., Isabelle/HOL - A Proof Assistant for Higher-Order Logic. Springer Berlin Heidelberg, 2002. [Online]. Available: https://doi.org/10.1007%2F3-540-45949-9
- [20] T. Nipkow and G. Klein, Concrete semantics: with Isabelle/HOL. Springer, 2014. [Online]. Available: https://doi.org/10.1007%2F978-3-319-10542-0
- [21] E. W. Weisstein, "invariant." from mathworld-a wolfram web resource," https://mathworld.wolfram.com/Invariant.html.
- [22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for Low-Latency Geo-Replicated storage," in 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). Lombard, IL: USENIX Association, Apr. 2013, pp. 313-328. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd
- [23] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguica, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2016, pp. 405-414.
- [24] H. Ouyang, H. Wei, and Y. Huang, "Checking causal consistency of mongodb," in Proceedings of the 12th Asia-Pacific Symposium on *Internetware*, 2020, pp. 209-216.

References

- [25] P. S. Group, "Eiger-port," https://github.com/princeton-sns/Eiger-PORT, 2020.
- [26] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv, "Eventually consistent transactions," in Programming Languages and Systems, H. Seidl, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 67-86.
- [27] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," ACM Computing Surveys (CSUR), vol. 13, no. 2, pp. 185-221, 1981.
- [28] A. Cerone, A. Gotsman, and H. Yang, "Algebraic Laws for Weak Consistency," in 28th International Conference on Concurrency Theory (CONCUR 2017), ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Meyer and U. Nestmann, Eds., vol. 85. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017, pp. 26:1-26:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/779447
- [29] N. Crooks, Y. Pu, L. Alvisi, and A. Clement, "Seeing is believing: A client-centric specification of database isolation," in Proceedings of the ACM Symposium on Principles of Distributed Computing, ser. PODC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 73-82. [Online]. Available: https://doi.org/10.1145/3087801.3087802
- [30] G. Kaki, K. Nagar, M. Najafzadeh, and S. Jagannathan, "Alone together: Compositional reasoning and inference for weak isolation," Proc. ACM Program. Lang., vol. 2, no. POPL, dec 2017. [Online]. Available: https://doi.org/10.1145/3158115
- [31] K. Nagar and S. Jagannathan, "Automated Detection of Serializability Violations Under Weak Consistency," in 29th International Conference on Concurrency Theory (CONCUR 2018), ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Schewe and L. Zhang, Eds., vol. 118. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018, pp. 41:1-41:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9579

References

- [32] S. Liu, P. C. Ölveczky, M. Zhang, Q. Wang, and J. Meseguer, "Automatic analysis of consistency properties of distributed transaction systems in maude," in Tools and Algorithms for the Construction and Analysis of Systems, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 40-57.
- [33] S. Liu, "All in one: Design, verification, and implementation of snow-optimal read atomic transactions," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 31, no. 3, pp. 1-44, 2022.

ETH zürich

Shabnam Ghasemirad
Information Security Group
sghasemirad@student.ethz.ch

Mechanized Data Consistency Models for Distributed Database Transactions

Supervisors:
Dr. Christoph Sprenger
Dr. Si Liu

Professor:
Prof. Dr. David Basin

ETH Zürich