Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's thesis

Ordering Transactions on Distributed Persistent Objects

Valentin Simon Nepomuk König

Course of Study:	M.Sc. Informatik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Lukas Epple, M.Sc.
Commenced:	June 21, 2023
Completed:	December 21, 2023

Abstract

Resilient and consistent data storage is critical for applications and services across all industries. As such, various persistent storage layers, such as database storage systems, have been developed. These either require complex data translations, commonly tackled through Object-Relational Mapping (ORM), or force the application to use data types specifically developed for persistent storage (e. g., Intel PMDK). Persistent objects describe a novel storage system that is able to persist and transactionally modify application-defined objects. With persistent objects, applications transactionally execute arbitrary functions on their existing data objects. However, they only allow local transactional modifications on individual objects, as there are no consistency guarantees across sequences of transactions.

In this work, we present a novel distributed transaction coordination algorithm called Two-Phase Ordering (2PO) that provides isolation between concurrently executing clients. The algorithm uses application-specific knowledge, which becomes available through the use of application-native objects in the local persistent storage layer. Moreover, the distributed transaction execution order of 2PO depends entirely on the user's desired semantics. Thus, the system accomodates both linearizable consistency as well as arbitrary interleavings of non-transactional interactions at the same time. By proving the consistency guarantees of 2PO in the general case, we show that our work is applicable to traditional transactional storage systems (e. g., databases) as well.

Our evaluations show that 2PO supports linearizability as well as highly available transaction consistency models, such as Monotonic Atomic View (MAV), both with high commit rates (above 97 % under high contention), all while scaling linearly with the number of nodes.

Contents

Acronyms

2PL Two-Phase Locking. 9

2PO Two-Phase Ordering. 3, 9

MAV Monotonic Atomic View. 3, 9

MVCC Multi-Version Concurrency Control. 17

NVRAM Non-Volatile Random Access Memory. 12

OCC Optimistic Concurrency Control. 10

ORM Object-Relational Mapping. 3

1 Introduction

Consistent and robust storage of data is a fundamental prerequisite for data processing across diverse applications and industries [BBC+11; CDE+13; MAN+14]. Database systems are a core technology in such settings. While databases excel at providing transactional consistency, they often do so at the cost of introducing a gap between the data storage layout and its application-native object representation [TGPM17]. Applications have to use internal data representations on which they execute operations, and a second layout in the database to persistently store the data.

Persistent objects are a novel concept that enables a tight integration of persistence into the application's existing data representation. With these, applications can reuse their existing data objects in persistent storage. Using application-defined objects in the persistent storage layer can avoid costly computational overhead associated with type-related translation, query language processing and transaction implementation abstractions [TGPM17]. Moreover, they enable novel approaches to distributed transaction coordination. However, persistent objects so far only allow local transactional modifications. There are no consistency guarantees across sequences of transactions or multi-client concurrent object access.

To enable the transformation of local persistent objects to distributed persistent objects, we present a new and fully-distributed transaction coordination algorithm that ensures the consistency of distributed object access. Our algorithm, which we call Two-Phase Ordering (2PO), uses application specific knowledge about the data interactions to determine a conflict-free schedule beforehand without requiring distributed locking. We prove its correctness in the general case, hence, it is applicable to traditional transactional storage systems (e. g., databases) in the same way. Unlike traditional distributed transaction systems, our algorithm does not rely on a centralized transaction manager, thereby reducing bottlenecks and enhancing scalability. We adopt a shared memory synchronization model and use it to describe ordering constraints [BA08]. This comprises the definition of consistency semantics that grasp the ordering requirements of different functions. Such ordering constraints apply to concurrent object access.

2PO simultaneously supports different consistency models in a distributed setting. The current implementation provides linearizability and Monotonic Atomic View (MAV) [BDF+13], but is user-customizable. Using 2PO, linearizability is implemented through virtual object locks with which we realize Two-Phase Locking (2PL). We show that virtual locking results in high commit rates (above 97 % under high contention) while scaling linearly with the number of nodes.

The contributions of this work are as follows:

- We develop a novel fully-distributed coordination algorithm called 2PO that computes a conflict-free execution schedule at runtime, without requiring transaction rollbacks for conflict resolution.
- We prove the algorithm's correctness in the general case, which makes it applicable to existing transactional data storage systems.
- We present measurements that show 2PO scales linearly with the number of participating nodes and improves commit rates over traditional implementations of Optimistic Concurrency Control (OCC) or 2PL.

The remainder of this work is structured as follows: In Chapter 2, we present a system model for local execution and communication, and formalize local transactional execution. In Chapter 3, we introduce our concept of distributed transactions, develop a formal proof for conflict detection and derive our coordination algorithm. Chapter 4 presents implementation details for the previously presented concepts. In Chapter 5, we introduce related work. In Chapter 6, we evaluate our approach. Chapter 7 concludes the thesis.

2 Transactional Objects

We describe each processor P as an automaton whose transition relation defines the set of legal sequences of steps that P can execute. W.l.o.g. we assume every processor executes an infinite sequence of steps. Termination of a processor, i. e., terminal states are modeled as a single state with a "no-op" transition to itself [HT94]. The states of the automaton are given by the processor's program: Every processor P executes a pre-defined program comprising a sequence of instructions.

Programs generally contain two types of expressions: Local operations, and communication instructions. Local expressions comprise instructions a processor executes that modify the processor's local state and side-effects are not directly visible to other processors. Such changes can only become visible to other processors through communication.

When a processor P executes an instruction c of its program, it generates a corresponding event e, we write

$$P(c) = e.$$

The execution of a processor's program induces a partial order of events. We write

$$e \xrightarrow{P} e'$$

to denote that the processor P generated the event e before it generated the event e'. This partial order of events is the processor's observable behavior. An event is generated if a processor reads or writes a local variable, sends a message or receives a message. This notion of each processor generating events allows reasoning about the correctness of the ordering of an execution later.

We assume the underlying communication network to be a point-to-point network. A point-to-point network can be modeled as an undirected graph (P, L) where P is the set of processors and L the set of communication links between the processors [HT94]. Any pair of processors connected by an edge are able to communicate according to the communication primitives defined below: Let $P_i, P_j \in P$ two processors and assume $\{P_i, P_j\} \in L$, i. e., there is a link between the two processors. Associated with each such link are in- and outgoing message buffers at each of the processors. When the processor P_i invokes $\textsc{Send}(P_j, m)$, it inserts m in the link's outgoing message buffer. Once P_i returns from that call, we say it completes the sending of m to P_j. The link then transports the message to the incoming message buffer, where P_j can now receive m. When P_j returns from the execution of $\textsc{Receive}(P_i)$ with a message m as the return value, we say P_j receives m.

With this, individual processors are able to communicate with each other. Thus, the observable behavior of the system has become more complex than that of the individual processors. We extend the idea of a processor emitting events during the execution of its program as follows. The distributed execution of all processors creates a set of events E. This is the union of all events generated by the participating processors. The distributed execution, once again, induces a partial

order over the events in E. This has been well studied in literature [KKS18; Lam78; Mat88] and has applications throughout the distributed systems field of study. Lamport first formalized an ordering with the following relation [Lam78].

Definition 2.1 (Happened-before relation)
For two events $a, b \in E$, we say a *happened before* b, denoted by $a \longrightarrow b$, if and only if

1. a, b are executed by the same processor P and P emits a before it emits b, that is $a \overset{P}{\longrightarrow} b$, or
2. a is the sending of a message m and b is the delivery of m, or
3. there is an intermediary step c, s.t. $a \longrightarrow c$ and $c \longrightarrow b$. ♦

The happened-before relation is acyclic because the transmission of a message precedes the delivery of the same message in real time. By the third clause, the relation is also transitive. Therefore, it is a partial order over the events. An execution of a distributed system can be denoted by $(E, \longrightarrow)$ [KKS18]. Events a, b for which neither $a \longrightarrow b$ nor $b \longrightarrow a$ holds are called *concurrent*; we write $a \parallel b$.

2.1 Persistent Objects

Persistent objects describe a novel access strategy for locally-attached persistent memory. Persistent objects encompass a programming model for transactional modifications of application-defined objects. They tightly integrate persistency into the application's existing data representation. Using application-defined objects in the persistent storage layer can avoid costly computational overhead associated with type-related translation, query language processing and transaction implementation abstractions [TGPM17]. In contrast to existing programming models on persistent main memory (e. g., Intel PMDK [pme]), this does not require applications to be designed specifically for Non-Volatile Random Access Memory (NVRAM) (e. g., [Int; SBF+15]).

Transactions are a powerful abstraction for reliable data management in persistent storage. The properties of transactions are commonly described by the four ACID properties: A transaction groups a set of instructions which then appear to execute as a single atomic unit within the system (Atomicity). This is the key to data integrity, a transaction transforms the system's state from one consistent state to another (Consistency). Intermediate and as such possibly inconsistent values need to remain invisible to other clients. Specifically, concurrent transactions are executed such that they are unaware of other concurrently executing transactions (Isolation). Finally, once a transaction's unit of work is finished (committed), its changes remain permanent and persist a power loss, system crash, or other type of failure (Durability).

With objects directly stored in persistent memory, the concept of transactions becomes crucial for ensuring the integrity and consistency of their state. As they are stored permanently in non-volatile memory, inconsistencies would remain persistent. Such inconsistencies could then easily transition the application into an irrecoverable state. Hence, we use transactions to ensure that the persistent storage remains in a valid state, and all modifications of objects are atomic. Specifically, on persistent objects, there is no unit of work *smaller* than a transaction.

2.2 Modifying Persistent Objects

In our storage system each processor stores zero or more objects on their machine; we say the processor owns the objects it stores. Let object o be owned by a processor P. Then P has local access to the object's persistent representation and can access that object arbitrarily. Specifically, only P can perform read and write accesses on o. Every modification of o happens in a transaction. For a transaction T we define $\mathsf{modifies}(T)$ as the object modified by the transaction T. When the processor P initiates a transaction by executing an instruction a, it emits the event $[_T$ (read T begins). The processor that executes the transaction is called the *executor*; we write $\mathsf{executor}(T) = P$. Every transaction has a terminating instruction b, i. e., either commit or abort. When P executes this terminating instruction, it emits the event $]_T$ (read T ends). We define the *body of* T ($\mathsf{body}(T)$) as the set of instructions between a and b (in program order).

To achieve ACID properties for the execution of T, we require two properties: Firstly, the execution of the body must not be reordered relative to the surrounding begin and end instructions, or more formally it must hold

$$\forall c \in \mathsf{body}(T): \quad \big([_T \rightarrow P(c)\big) \wedge \big(P(c) \rightarrow]_T\big). \tag{T1}$$

Secondly, a processor may not concurrently execute two transactions that modify the same object. Moreover, read operations on an object must be ordered before or after any transaction's execution but not concurrently. We formalize this with the following definition.

Definition 2.2 (Mutual exclusion)
Let P a processor that owns an object o. We call the object o *exclusive* (or the *access to o mutually exclusive*) if for every execution the following conditions hold

1. Let $\mathcal{P}$ the set or all instructions in the program of P. For every transaction T with $\mathsf{executor}(T) = P$ and $o = \mathsf{modifies}(T)$ it must hold that every instruction $c \in \mathcal{P} \setminus \mathsf{body}(T)$ that accesses o must be executed such that

   $$\big(P(c) \rightarrow [_T\big) \vee \big(\,]_T \rightarrow P(c)\big). \tag{MX1}$$

   That is, any instruction c that accesses the same object as a transaction T must be executed either before T begins or after T ends.

2. For all transactions T, T' with $\mathsf{executor}(T) = \mathsf{executor}(T') = P$ and $\mathsf{modifies}(T) = \mathsf{modifies}(T')$ it holds

   $$[_T \nparallel [_{T'}, \tag{MX2a}$$

   and

   $$[_T \rightarrow [_{T'} \implies]_T \rightarrow [_{T'}. \tag{MX2b}$$

   In words, no two transactions on the same object can begin concurrently, and if transaction T begins before T', then T must end before T' can start. ♦

Mutual exclusion, as the name implies, describes a form of isolation between concurrent modifications. Specifically, modifications on an exclusive object have to be serialized. This serialization of transactions on each object induces a total order of transactions for each object. We call this order the modification order.

Definition 2.3 (Modification order)
Let o an exclusive object. The *modification order of o* is a totally ordered sequence of transactions that modify o. We denote the modification order of object o by $<_o$. ♦

The modification order is unique per execution.

Theorem 1:
For every exclusive object o there exists a totally ordered modification order per execution. Let o owned by processor P, and let T_i, T_j a pair of transactions that modify o. The notation $T_i \rightarrow T_j$ denotes that all instructions of T_i happen before the instructions of T_j. This extends the happened-before relation and it always holds $T_i \not\rightarrow T_j$. Then for such transaction T_i, T_j, the modification order is given by

$$T_i \rightarrow T_j \Longleftrightarrow T_i <_o T_j. \tag{MO}$$

◇

Proof We first show that $T_i \not\rightarrow T_j$. Assume $T_i \parallel T_j$. There are two cases: (1) a pair of begin or end events is concurrent or (2) an instruction from the body of T is concurrent with an instruction of T'. Equations (MX2a) and (MX2b) eliminate (1). Hence, let $t_i \in \mathsf{body}(T_i), t_j \in T_j$ with $P(t_i) \parallel P(t_j)$. For these two instructions Equation (T1) gives

$$[_{T_i} \rightarrow P(t_i) \rightarrow]_{T_i} \text{ and } [_{T_j} \rightarrow P(t_j) \rightarrow]_{T_j}.$$

Now, Equation (MX2a) gives w.l.o.g. $[_{T_i} \rightarrow [_{T_j}$. However, Equation (MX2b) together with the result above implies

$$[_{T_i} \rightarrow P(t_i) \rightarrow]_{T_i} \rightarrow [_{T_j} \rightarrow P(t_j) \rightarrow]_{T_j}.$$

which contradicts $P(t_i) \parallel P(t_j)$. Thus, $T_i \not\rightarrow T_j$ always holds. Anti-symmetry of $T_i \rightarrow T_j$ follows directly from the underlying happened-before relation. $T_i \rightarrow T_j$ is well-defined.

Uniqueness of $<_o$ is given by the $\rightarrow$ relation of the execution.

It remains to show that $<_o$ is a total order i. e., there does not exist a pair of transactions T_i, T_j with $o = \mathsf{modifies}(T_i) = \mathsf{modifies}(T_j)$ for which neither $T_i <_o T_j$ nor $T_j <_o T_i$ holds. We have just shown that such transactions cannot be concurrent, hence $T_i \not\rightarrow T_j$. This implies w.l.o.g. it holds $T_i \rightarrow T_j$. ■

This shows how concurrent modifications are serialized and that there exist a single order of all modifications on an object.

Finally, we prove that read operations always see a consistent state of objects on a node. Specifically, we will see that the modification order describes the order in which a continuously reading reader observes changes on an object.

Definition 2.4 (Modification order coherence)
Let o an object. An execution of a distributed system is called *coherent (with the modification order of o)* if it fulfills the following properties:

1. (Read-Read) Let a, b read events on $o \in \mathbb{O}$, where $a \rightarrow b$ and a reads the value stored by a transaction T_1, then the value read during b is either the same value or the value of a transaction T_2 with $T_1 <_o T_2$.

2. (Read-Write) Let a be a read event and T_2 a transaction, both regarding the object o. If $a \rightarrow T_2$ then a must read a value that was written by a transaction T_1 with $T_1 <_o T_2$.
3. (Write-Read) Let T_1 a transaction that modifies o and a a read operation on the same object. If $T_1 \rightarrow a$ then the value read by a is either the value stored by T_1 or the value stored by a transaction T_2 with $T_1 <_o T_2$. ♦

These definitions for the modification order of objects is equivalent to the modification order of atomic variables in the C++ memory model [ISO17].

Theorem 2:
Every execution of a distributed system is coherent with the modification order of an exclusive object. Thus, all read operations observe changes in the modification order. ◊

Proof Directly follows from $<_o$ being a total order and any access to o being mutually exclusive. Read operations are guaranteed to happen either before a transaction begins or after a transaction ends, cf. Equation (MX1). ■

Corollary (Slow memory consistency): All processes observe the modifications of a given process to a given object in the same order. This fulfills slow memory consistency [HA90] or PerObjectPRAM as described in [VV16]. ◊

Corollary (Per-object single-order consistency): The modification order describes the single order in which all processes observe changes to an object. This gives PerObjectSingleOrder consistency as described in [VV16], originally defined as per-record timeline consistency in [CRS+08]. ◊

Viotti and Vukolić [VV16] combine these two properties into PerObjectSequential consistency. This consistency ensures that all processors observe the same order of operations of a certain process on each object. Moreover, there is a global ordering of all operations invoked per object.

Corollary (Per-object sequential consistency): Our system fulfills PerObjectSequential consistency. ◊

3 Distributed Transaction Ordering

In this work, we use an *atomic state transition protocol* that is able to transactionally execute arbitrary functions on single objects stored in persistent memory. A modification performed through the atomic state transition protocol is atomic. Consistency is guaranteed because we always ensure there is at most one valid version of each object in persistent storage. The protocol isolates concurrent transactions by locking the object, thus serializing concurrent access. Durability is given, as we keep one version of each object in its persistent representation in persistent storage at all times.

However, the concept of atomic state transitions alone is insufficient for distributed shared objects. This is because we have no transactional guarantees across sequences of transactions yet. Specifically, when multiple clients access a set of shared objects, local single-object state transitions alone are not sufficient to provide clients a consistent view of data. The execution of multiple clients' transaction sequences may overlap, violating isolation. Hence, in this section, we discuss how we achieve isolation between clients accessing the same object(s).

Traditional database systems combine multiple (sub-)transactions into larger transactions. To ensure transactional properties of these, one typically employs a transaction coordination algorithm such as two-phase commit [BBC+11; BHG+87; MCZ+14]. Moreover, to achieve a linearizable execution ordering of the sub-transactions contained in such a larger transaction, variations of 2PL, OCC [KR81] or Multi-Version Concurrency Control (MVCC) are often used [BBC+11; BHG+87; CDE+13; MAN+14]. These approaches all either start to execute the transactions, and then check whether they were allowed to make the changes (OCC, MVCC), or they start to acquire locks and then check wheter the lock acquisitions are deadlock-free (2PL). If either check fails, they have to abort the transaction, undo their changes and start over.

In this work, we propose an alternative approach. We use application-specific knowledge in a novel algorithm that determines a conflict-free schedule of transactions prior to their execution. This ordering isolates concurrently executing clients on the object-level without requiring traditional distributed locking strategies. This ensures the execution schedule is valid before executing any transactions. The basis for this ordering algorithm is a concept from local shared memory data synchronization. With it, we realize isolation among concurrently executing clients. The degree of isolation together with the ACID properties of local transactions implies consistency guarantees for the data access.

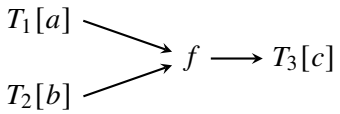
We use the model of flushing and acquiring instructions as a means of synchronization (e. g., [BA08]). In this model, a processor flushes (i. e., stores) information about the changes it has performed. An acquiring processor reads this flushed information and synchronizes its local state accordingly. Typically, a write to a variable first executes the store, followed by a flush. A read instruction first synchronizes its state to acquire all flushed changes, then reads the variable. The local setting requires this synchronization because the caches of different threads may be inconsistent. We utilize this procedure to synchronize clients concurrently accessing the same objects.

Traditional database systems or distributed transaction coordinators in general have no application knowledge and are only able to use information about data access as the application issues them. In contrast, our algorithm is able to exploit application-specific knowledge about the causal future of data interactions. We use this knowledge while computing a final execution ordering. While we apply this approach to distributed persistent objects, it is applicable to traditional database systems as well and could simplify their coordination efforts. For our system, we adopt this model and use a flush to describe semantic relationships between transactions of the *same* client. With this, an acquisition forms an isolating constraint on the final execution order. This constraint limits the valid execution order of *different* clients' transactions.

3.1 Client Operations

We use information about the application's (i. e., the client's) logical interactions with the storage system. Thus, to represent such an interaction, we define *client operations*. A client operation is a logical group of one or more object-level transactions. The intention is that each client operation fulfills a piece of the application's business process. Client operations are the way for clients to interact with any persistent object stored on the processors. Thus, the client combines its object-modifying transactions into client operations. The assumption is that transactions inside the same operation are causally related, whereas transactions of different client operations are logically independent. We refrain from calling client operations transactions as well, as we do not require a linearizable ordering for all client operations.

In general, a client operation O may initiate a transaction T on an object located on a remote server. By $T[m]$ we denote the execution of transaction T that accesses object m. Similarly, we write e. g., f for the local function execution of a client. We use the notation $T, f \in O$ to denote that transaction T or the local function f is a part of the operation O. Hence, we use O to denote that both the set of functions and the operation itself. However, given the context, it is always clear which of the two is used. The client operation may contain concurrent transactions or concurrent local instructions. For example, a client operation that reads two remote objects a and b and writes a combination of the two in a third object c might issue two reads in parallel. Thus, the client operation describes its execution as a partial order over its instructions. The previous example could be represented as



Which means that c can only be modified after both reads on a and b are done.

More formally, a client operation can be represented as the partial order $(O, \prec_O)$. The transitive sequenced-before relation $I \prec_O I'$ describes that the instruction $I \in O$ must be executed before $I' \in O$. We say I is sequenced before I' or conversely, I' is sequenced after I. We sometimes also use simply $\prec$ instead of the more verbose $\prec_O$ if the operation is either irrelevant or clear from the surroundings. The $\prec$ relation is anti-symmetric, transitive and fully defined by the source code of the client. The orderings of $\prec$ are given by the programmer's intentions and describe data, or, more generally, causal dependencies in the application. Our representation of these dependencies is influenced by serializability theory [BHG+87].

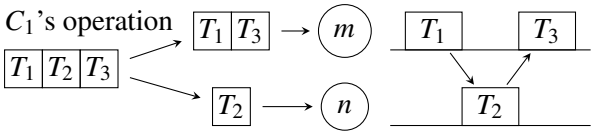


Figure 3.1: A client C_1 executes an operation comprising three transactions. Transactions T_1, T_3 modify object m and T_2 modifies object n. The transactions must be executed in the order of appearance because they are causally dependent.

The sequenced-before relation is the formalization of an instruction's *causal surroundings*. The causal surroundings describe both the causal precedence as well as future transactions sequenced after the current one. Specifically, the causal future of transaction T contains transactions that are sequenced after T. In non-distributed programs, the compiler derives such data dependencies and ensures semantically correct ordering of the instructions. However, in a distributed execution the data dependencies are hidden and implicitly given by modifications of remote objects. Therefore, we need to make such data-dependencies between operations explicit and available to the distributed execution.

We introduce a data flow-oriented representation of distributed client operations. A *segment* represents either a local function call or a remote object transaction. Consequently, a segment that represents a transaction is called a *remote segment*, other ones are *local*. Then, we represent an operation as a directed graph of segments that are connected according to their data-dependencies. An edge (s_1, s_2) between two segments means s_1 is an *input segment* to s_2. That is, the segment s_2 gets the return value of s_1's function call as input parameter. Hence, before the execution of s_2 can start, it must await the completion of s_1 and any other such input segments. The segment edge (s_1, s_2) thus directly translates to the sequencing relation $s_1 \prec s_2$. This effectively limits parallelism and reordering possibilities but retains the programmer's intended program semantics.

To sort the transactions of an operation O by the object they modify, we define the restriction

$$\forall m \in \mathbb{O}: \quad O|_m := \{T \in O \mid \mathsf{modifies}(T) = m\}.$$

In other words $O|_m$ is the set of all transactions in the body of O that modify the object m. We use the set

$$\mathsf{future}(T) := \{T\} \cup \{K' \mid T \prec K'\}$$

to denote the transactions K' that cannot precede a transaction T in the execution order, i.e., have to be executed in the future. This gives us information about the local causal order of coherent transactions. Figure 3.1 shows an operation with intrinsic ordering restrictions ($T_1 \prec T_2 \prec T_3$). This means T_1 must precede T_2 and T_2 must precede T_3 in the final execution. Neither T_1 nor T_3 may execute concurrently (in real time) to T_2, even though they modify different objects. The sequencing of transactions constrains the execution of transactions inside the same operation. As such, for transactions T, T' accessing the same object o, we require

$$T \prec T' \implies T <_o T'.$$

In a distributed setting, there may be concurrent object access. The order in which the server executes the three transactions is crucial to the consistency we can guarantee. Going forward, we use the term of *interference* to describe this need for an ordering. For a transaction T, we use the time stamps $btime, etime$ as the time stamps of begin and end (in an ideal global time).

Definition 3.1 (Interfering transactions)
Two transactions T_1, T_2 are called *interfering*, if they both operate on the same object

$$\mathsf{modifies}(T_1) = \mathsf{modifies}(T_2)$$

and their executions overlap in time

$$T_1.btime < T_2.etime \wedge T_2.btime < T_1.etime. \qquad \blacklozenge$$

1 Lemma:
Let T_1, T_2 two transactions from the same operation O with $\mathsf{modifies}(T_1) = \mathsf{modifies}(T_2)$. If T_1 interferes with T_2 then they are concurrent, i. e., $T_1 \parallel T_2$ which means $T_1 \not\prec T_2$ and $T_2 \not\prec T_1$. $\diamond$

Proof Assume $T_1 \prec T_2$, then the data dependency between the transactions implies T_1 needs to return before T_2 can start. Thus, $T_1.etime < T_2.btime$. ■

Hence, we can also correctly say transactions only interfere if they are not ordered by a sequencing relation. While a and b in the example modify the same object, they can not interfere in their ordering as the execution order is always explicitly given by the surrounding operation. The correct ordering is always $a \rightarrow b$ because in operation O_1, a is sequenced before b. The execution ordering $b \rightarrow a$ is impossible because of a data dependency from b to a. Thus, we use the term of interference for two transactions of which the order is initially indeterminate or different orderings may be possible but some orderings should be disallowed. Determining a *correct* ordering of interfering transactions is the main goal of any transaction control algorithm. Hereby, correctness of the interleaving depends on the desired consistency. Our system uses a technique inspired by shared-memory to determine the correct ordering of interfering transactions. The following section describes this in more detail.

3.2 Shared-Memory Synchronization in a Distributed Setting

User-defined constraints describe inter-operation execution orderings. We allow the programmer to give such constraints as follows. Let T be a transaction of an operation O. The users describe semantic relations between T and other transactions of O by defining the set $\mathsf{flush}(T)$ (subset of O's transactions). Intuitively, if another transaction L which is not part of O sees the effects of T, it should also see the effects of all $K \in \mathsf{flush}(T)$. We say K is a flushed transaction, and T flushes K. Users do not define the flush sets directly, but implicitly through annotating the transactions with semantics. We give more detail on such semantics in Section 3.4.6. Only the intuitive understanding of the set $\mathsf{flush}(T)$ is relevant for this discussion.

Note that, at this point, we deviate slightly from the analogous intuition of shared-memory flushing and acquiring instructions. To reiterate, in shared memory computations, a processor flushes modifications that were performed prior to the current instruction. In our model, we allow the user to flush transactions that may not have happened yet at this point in time. That is, for $K \in \mathsf{flush}(T)$, the transaction K may very well follow T in the logical order ($\prec$) of their operation. We only require that these two transactions T and K appear finished to *other* operations, if the effects of T are visible. This model allows for powerful yet intuitive ordering constraints in the system.

From the flush set, the system deduces the set $\mathsf{release}(T)$ which expands $\mathsf{flush}(T)$ to all causal dependencies of the flushed transactions. If $K \in \mathsf{flush}(T)$, then $\mathsf{release}(T)$ contains the causal precedence (inside the operation) of K. Moreover, $\mathsf{release}(T)$ contains the transitive closure of flushing. This expansion is necessary to ensure the effects of flushing align with the intuitive understanding. In the remainder of this section, we detail this deduction from flush set to release set.

We first express the flush set as a partial order flush over the operation. For T, T' in the same operation it holds $(T, T') \in \mathsf{flush}$ if $T' \in \mathsf{flush}(T)$. By flush^{-1} we denote the inverse *flushed by* relation. Consider an operation O with $T, A, B \in O$, where all three transactions modify m and it holds $T \prec A \prec B$. Moreover $(T, B) \in \mathsf{flush}$ and a transaction K acquires T. Then, we required $\{T, B\} \subseteq \mathsf{acquire}(K)$ that is to K it should appear as if T and B have already been executed. Because $A \prec B$, the execution of B directly depends on A; B can only start if A has already finished. This implies K acquires A as well. The same argument applies to any transaction X with $X \prec B$ because the execution of B requires X to be finished. We get

$$\bigcup_{F \in \mathsf{flush}(T)} \{T' \mid T' \prec F \wedge T' \not\prec T\} \subseteq \mathsf{acquire}(K).$$

Moreover, in the case of $\{(T, A), (A, B)\} \in \mathsf{flush}$, transaction K that acquires T has to see the effects of A. Then, it must also see the flush set of A, which implies K acquires A as well. Thus, acquiring imposes a transitivity on the flush relation. We get the lower bound

$$\{L \mid (T, L) \in \mathsf{flush}^+\} \subseteq \mathsf{acquire}(K).$$

Hereby flush^+ is the transitive closure of the flush relation.

We call these properties *flush set expansion*. Figure 3.2 shows an example of the effects of flush set expansion. Even if the flush set only contains few transactions, the effects of the flush set might expand to other transactions which are not in the flush set. Ordering constraints need to respect all transactions after flush set expansion. Thus, if a transaction K acquires T, the following acquire set must be respected in the execution ordering

$$\bigcup_{F:(T,F) \in \mathsf{flush}^+} \{F\} \cup \{T' \mid T' \prec F \wedge T' \not\prec T\} =: \mathsf{release}(T) \subseteq \mathsf{acquire}(K).$$

Figure 3.3 shows a possible incorrect ordering caused by omitting flush set expansion. The specific details will be discussed later, for now it is enough to understand that flush set expansion helps reduce the algorithm complexity and simplify the detection of incorrect orderings.

3.3 Forming Constraints on the Execution

Intuitively, if a transaction K acquires T, we require that, to K, it should appear as if all transactions in $\mathsf{release}(T)$ have already been executed. If K acquires T it must hold $\mathsf{release}(T) \subseteq \mathsf{acquire}(K)$. We formalize this to the following ordering constraint.

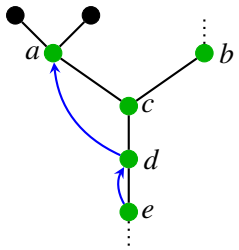


Figure 3.2: All transactions marked green must be acquired when acquiring a, whereas only $d \in \mathsf{flush}(a)$. The $\prec$ relation is from top to bottom along the edges. The transactions c, b (and anything before b) must be acquired because they are required (by the $\prec$ relation) for the execution of d. Transaction e has to be acquired because $(a, e) \in \mathsf{flush}^+$. The transactions above a do not need to be part of the acquire set as their execution is already guaranteed to be finished.

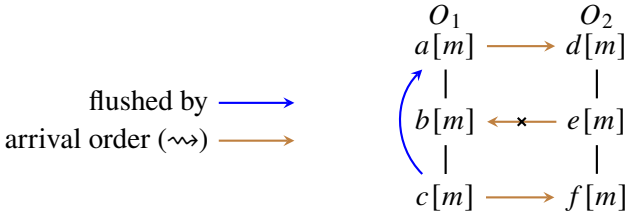


Figure 3.3: Incorrect intermediate ordering caused by using $\{c\} = \mathsf{flush}(a)$ instead of $\{a, b, c\} = \mathsf{release}(a)$ in the constraint. Transaction d acquires a and thus, has to defer its future (e, f). If, rather than the release set, the constraint would use the flush set of a, undetectable incorrect orderings can arise. The figure shows such an ordering: e is ordered before b and by transitivity before c. However, the algorithm is unable to detect the incorrect ordering $e <_m b$ because we omitted flush set expansion.

Definition 3.2 (Acquire ordering constraint)
Let T, K interfering transactions on the object m. By $\mathsf{acquire}(K)$ we denote the set of all transactions K must synchronize with. This synchronization means, the ordering must fulfill for all objects o

$$\forall L \in \mathsf{acquire}(K)|_o,\ \forall K' \in \mathsf{future}(K)|_o :\quad L <_o K',$$

where

$$\mathsf{future}(K) = \{K\} \cup \{K' \mid K \prec K'\}.$$

In our case, K *acquires* T if $T <_m K$, that is T appears before K in the modification order of m. If K acquires T, it must hold $\mathsf{flush}(T) \subseteq \mathsf{release}(T) \subseteq \mathsf{acquire}(K)$. ♦

In words, $\mathsf{future}(K)$ is the set of all transactions that are either K itself or follow K in the sequencing relation of K's operation. Recall that $\alpha|_o$ is the restriction of the set of transactions α to only those transactions that modify the object o. For transactions $L \in \mathsf{acquire}(K)$ and a K' with $K \prec K'$, the acquire constraint ensures K' already sees the effects of L. Hereby, we do not expect $L \to K$ (which is a constraint in global time) but only the slightly weaker $L <_o K'$ (which is a constraint in object-local precedence). Thus, to an acquiring transaction K it *appears as if* the execution of the

flush set $\mathsf{flush}(T)$ happens at the same time as T. An acquisition realizes an ordering restriction based on constraints intrinsic to an operation given by $\mathsf{future}(T)$, and the user-defined ordering constraints given by $\mathsf{release}(K)$.

An acquisition between two operations O_1, O_2 essentially orders some of the transactions of O_1 before some of the transactions in O_2. Specifically, if $K \in O_2$ acquires $T \in O_1$, we get a pair-wise ordering between the transactions in $\mathsf{release}(T) \subseteq \mathsf{acquire}(K)$ and $\mathsf{future}(K)$.

With this, we define the prefix set for K as the set of transactions that must precede transactions of K's operation because of previous acquisitions.

Definition 3.3 (Prefix set)
Let K a transaction. We define the *prefix set of* K as the set

$$\mathsf{prefix}(K) := \bigcup_{T:K \in \mathsf{future}(T)} \mathsf{acquire}(T).$$

♦

Note, how the prefix set carries through the sequencing of an operation.

2 Lemma:
Let T, K transactions from the same operation with $T \prec K$. Then it holds

$$\mathsf{prefix}(T) \subseteq \mathsf{prefix}(K).$$

◇

Proof Because $T \prec K$, we have

$$\forall L: \ T \in \mathsf{future}(L) \implies K \in \mathsf{future}(L).$$

Therefore, we have

$$\begin{aligned}\mathsf{prefix}(T) &= \bigcup_{L:T \in \mathsf{future}(L)} \mathsf{acquire}(L) \\ &\subseteq \bigcup_{T:K \in \mathsf{future}(T)} \mathsf{acquire}(T) \\ &= \mathsf{prefix}(K)\end{aligned}$$

■

The prefix represents the ordering constraints of an acquisition. As such, the execution order must respect the prefix set as follows.

3 Lemma:
Let T a transactions and $K \in \mathsf{prefix}(T)$. If the two transactions modify the same object $m = \mathsf{modifies}(T) = \mathsf{modifies}(K)$, then there exists an acquire ordering constraint which requires

$$K <_m T.$$

◇

Proof Because $K \in \mathsf{prefix}(T)$, there exists a transaction $L \prec T$ (alternatively $L = T$) with $K \in \mathsf{acquire}(L)$. As such, $T \in \mathsf{future}(L)$. Therefore, the acquire constraint requires the execution order $K <_m T$. ■

Figure 3.4 shows a scenario with two clients concurrently modifying the objects m, n. The first execution schedule (1) shows the possibility of interleaving executions if there are no additional constraints in the form of acquisitions. If, however, K_1 performs an acquisition such that $T_3 \in \mathsf{prefix}(K_1)$ the executions can no longer interleave, resulting in the execution schedule (2).

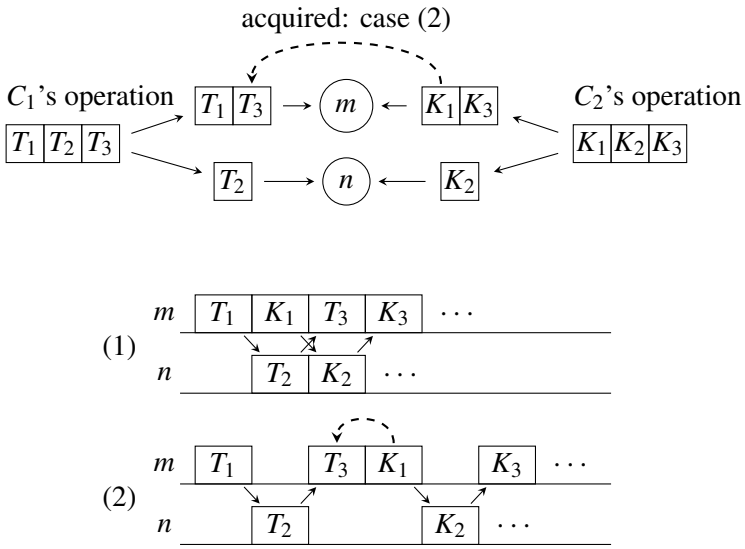


Figure 3.4: Two clients C_1, C_2 each execute an operation comprising three transactions. Without acquire constraints, the execution of the two operations may interleave (1). However, an acquisition between the two operations can give isolation between concurrent operations (2).

Longer chains of $<_m$ orderings can emerge if there are multiple acquisitions.

4 Lemma (Acquisition chains):
For two transactions T, K which both modify the object m, it holds that $T <_m K$ if and only if there exists a (possibly trivial) sequence of transactions $\{T_i\}_{i=1}^{n}$ on object m such that $T \in \mathsf{prefix}(T_1)$, $T_i \in \mathsf{prefix}(T_{i+1})$ and $T_n \in \mathsf{prefix}(K)$. ◇

Proof ($\Leftarrow$): Directly follows from the definition of the acquire constraint and the prefix set. For each pair T_i, T_{i+1}, the conditions of Lemma 3 apply. The same applies to the orderings between T, T_1 and T_n, K.

($\Rightarrow$): Given a constraint $K <_m T$, we have to construct a sequence of transactions T_i. Because $<_m$ is a total order, there exists a (possibly empty) sequence of transactions $S = (T_1, T_2, \ldots, T_n)$, $\mathsf{modifies}(T_i) = m$ with the constraints

$$T <_m T_1 <_m T_2 <_m \ldots <_m T_n <_m K.$$

We assume S is maximal, i. e., there are no transactions constrained in between the T_i but not in S. We will now show that this sequence is exactly the sequence of the proposition. Specifically, we will show that $T_i \in \mathsf{prefix}(T_{i+1})$. Because there is no transaction constrained between T_i and T_{i+1}, there must exist an acquisition between the operations of T_i and T_{i+1}. As such, by Definition 3.2, there exists a transaction A with

$$T_i \in \mathsf{acquire}(A)|_m \text{ and } T_{i+1} \in \mathsf{future}(A)|_m.$$

This already gives us everything we need to prove the statement. We have

$$T_i \in \mathsf{acquire}(A) \subseteq \mathsf{prefix}(A).$$

Moreover, because $T_{i+1} \in \mathsf{future}(A)$, we get $T_{i+1} = A$ or $A \prec T_{i+1}$. In the latter case, we can apply Lemma 2 and finally get

$$T_i \in \mathsf{prefix}(A) \subseteq \mathsf{prefix}(T_{i+1}).$$

This proves the statement. ■

Such chains of acquisitions formally describe constraints on valid execution orders. Acquisitions form the central building block on the path to determining the final execution order.

The first preliminary execution order we will consider is the order in which the transactions' messages arrive at the corresponding server. We call these messages the announcements for the transactions.

Definition 3.4 (Arrival order)
For two interfering transactions T_1, T_2 (i. e., they execute on the same server) we define the *arrival order* as $T_1 \rightsquigarrow T_2$ if the server received the announcement of T_1 before it received the announcement of T_2. We say T_1 *arrived before* T_2. ⧫

The arrival order is a total order of all transactions modifying the same object. Without any ordering constraints, the arrival order could be used as the final execution order.

The first constraint is more of a technical nuisance than a problem regarding transaction ordering. If two transactions $T \prec K$ arrive *out of order*, that is $K \rightsquigarrow T$, then the server would be unable to execute the transactions. This is because the execution of K has a data dependency to T if $T \prec K$. This data would be missing if K were to execute before T. Thus, we require the arrival order to be consistent with the sequencing order. Otherwise, the arrival order would be an impossible execution ordering.

Definition 3.5 (Arrival order consistency)
Let $m \in \mathbb{O}$ an object and O a client operation. The arrival order $\rightsquigarrow$ is called *consistent (with the sequencing)* if for any pair of transactions $T, K \in O|_m$ it holds

$$T \prec_O K \implies T \rightsquigarrow K.$$ ⧫

However, as we have mentioned previously, this is only a minor technical nuisance; ensuring consistency of the arrival order is trivial.

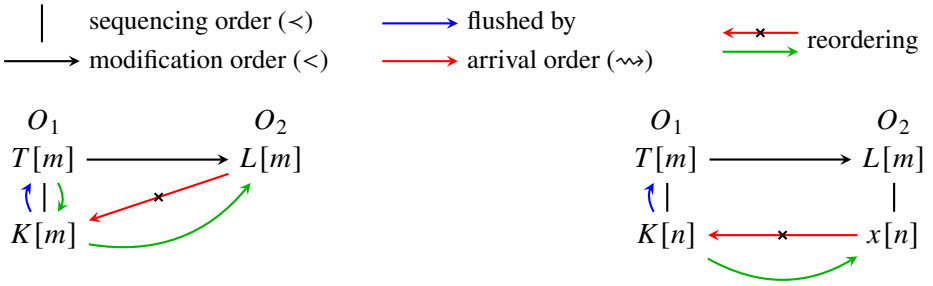
5 Lemma:
If the communication channel between client and server is a FIFO channel and the client announces transactions in $\prec$ order, every induced arrival order is consistent. ◇

From now on, we assume the arrival order to be consistent.

The second problem with the ordering of $\rightsquigarrow$ that prevents us from directly executing transactions in arrival order is the fact that it may violate acquire constraints. Arrival order consistency only ensures the preliminary execution order is executable at all. However, this does not mean the resulting interleaving is correct with regard to the programmer's defined flush sets and the resulting acquire constraints.

Definition 3.6 (Incorrectly ordered before)
Let T be a transaction, $K \in \mathsf{prefix}(T)$, with T and K both modifying the same object. If $T \rightsquigarrow K$ then T *is incorrectly ordered before* K. ♦

Figures 3.5a and 3.5b visualize the two cases of incorrectly ordered before. An ordering algorithm must detect such constraint violations and avoid them. The figures also show how we resolve the incorrect ordering in these simple cases. If x is incorrectly ordered before K (where x may be equal to L), we flip the execution order from $x \rightsquigarrow K$ to $K <_* x$.



(a) L is incorrectly ordered before K because $T <_m L$ and $L \rightsquigarrow K$. The reordering is $T <_m K <_m L$.

(b) x is incorrectly ordered before L because $T <_m L$, $L \prec_{O_2} x$ and $x \rightsquigarrow K$. The reordering is $K <_n x$

Figure 3.5: Reordering strategies to resolve *incorrectly ordered before*. Recall, the notation $T[m]$ means transaction T modifies object m. The algorithm is able to detect an incorrect ordering and perform a reordering.

However, the constraints may be ill-formed, that is, constraints may produce cyclical dependencies. We formalize these as follows.

Definition 3.7 (Cyclic Acquisition)
A non-empty set $C = \{T_i\}_{i=1}^{n}$ of transactions is called a *cyclic acquisition* if it holds

$$\forall i \in \{1, \ldots, n-1\} : \quad T_i \in \mathsf{prefix}(T_{i+1})$$

and $T_n \in \mathsf{prefix}(T_1)$. ♦

Note, a cyclical acquisition by itself does not necessarily mean the operations' constraints produce a cycle in the execution order. Transactions are impossible to execute if they are *unorderable*.

Definition 3.8 (Unorderable transactions)
A set of transactions $U = \{U_1, U_2, \ldots, U_n\}$ is called *unorderable* if for all arrival orders $\rightsquigarrow$, there exist at least one pair of indices $i \neq j$ s.t. U_i is incorrectly ordered before U_j. ♦

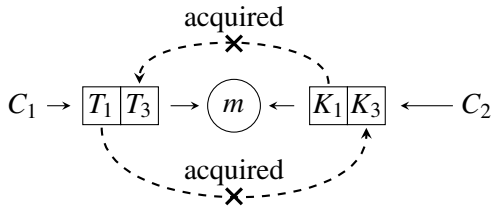


Figure 3.6: Two or more acquisitions can form a cycle. This can make all possible execution orderings invalid. The depicted transactions are unorderable.

If there exists no executable arrival order for the transactions in U, there can be no modification order that orders the transactions. This is because the arrival order describes an arbitrary ordering of transactions modifying the same object. The set of valid modification orders is a subset of all arrival orders.

Figure 3.6 shows an example of two transactions modifying m. If there are two acquisitions forming a cycle, we would have $K_3 \in \mathsf{prefix}(T_1), T_3 \in \mathsf{prefix}(K_1)$. This forms a cyclic acquisition. Moreover, the depicted cyclic acquisition shows an unorderable set of transactions. There exists no execution order which fulfills both acquire constraints.

In the remainder of this section, we show whether a transaction is unorderable depends on the modified objects and on whether they are part of a cyclical acquisition. We discuss the necessary conditions for a set of transactions to become unorderable. This shows how our transaction ordering algorithm detects unorderability and avoids deadlocks at the execution time.

The following theorem proves the intuitive understanding that a set of transactions modifying the same object is unorderable if the transactions have cyclical ordering dependencies.

Theorem 3 (Same-object conflicts):
Let $C = \{T_i\}_{i=1}^{n}$ be a cyclic acquisition. If there exists an object m such that $\mathsf{modifies}(T_i) = m$ for all i, then C is unorderable. ◊

Proof As we have already shown in Lemma 4, the prefix constrains the modification order for transactions modifying the same object. As such, the modification order $<_m$ would form a cycle. This is impossible because $<_m$ is a total order. ■

We strengthen this statement as follows.

6 Lemma:
A set of transactions U is unorderable if and only if there exists a subset of transactions $S \subseteq U$ where all transactions in S modify the same object and S is unorderable. ◊

Proof We partition U into a set of classes $[U]_{m_i}$, where

$$[U]_{m_i} = \{T \in U \mid \mathsf{modifies}(T) = m_i\}.$$

These sets group all transactions by the object they modify. Because U is unorderable, at least one of the $[U]_{m_i}$ contains two or more transactions. It remains to show that there exists an m_i such that $[U]_{m_i}$ is unorderable.

An arrival order $\rightsquigarrow$ only orders transactions operating on the same object. If we envision $\rightsquigarrow$ as the edges of a directed graph, where the transactions are vertices, then $[U]_{m_i}$ produce a fully disconnected graph partition. That is, there exists no edge $(c, c') \in \rightsquigarrow$ with $c \in [U]_{m_i}, c' \in [U]_{m_j}$ if $m_i \neq m_j$.

We define a restriction of $\rightsquigarrow$ on the partition as

$$\rightsquigarrow_{[U]_{m_i}} := \{(c, c') \in \rightsquigarrow \mid c, c' \in [U]_{m_i}\},$$

in words, $\rightsquigarrow_{[U]_{m_i}}$ connects transactions in the same partition. Note that the union over all $\rightsquigarrow_{[U]_{m_i}}$ is equal to $\rightsquigarrow$. Consequently, if there exists no executable arrival order for U, then there must be an m_i where no $\rightsquigarrow_{[U]_{m_i}}$ corresponds to an executable modification order.

$U \supseteq S$ remains unorderable if S is unorderable. ■

We can extend this lemma: The unorderable subset contains a cyclic acquisition as the root cause of unorderability.

Theorem 4 (Multi-object conflicts):
A set of transactions U is unorderable if and only if there exists a subset $C \subseteq U$ such that C is an unorderable cyclic acquisition and all transactions in C modify the same object. ◊

Proof We partition the set U as we did in the proof of Lemma 6. Thus, let $S := [U]_{m_i}$ be an unorderable subset of U. It remains to show that there exists a subset $C \subseteq S \subseteq U$ such that C is a cyclic acquisition.

For this, we construct the set of transaction pairs R which contains (T, K) if and only if there exists an arrival order $\rightsquigarrow$ where $T \rightsquigarrow K$ is incorrectly ordered

$$R = \{(T, K) \in S^2 \mid \exists \rightsquigarrow\colon K \text{ incorrectly ordered before } T \text{ given } \rightsquigarrow\}.$$

This defines the directed graph $G = (S, R)$. Because $(T, K) \in R$ means in particular, $T \in \mathsf{prefix}(K)$, it remains to show that G contains a cycle.

Graph G contains only corrections of incorrect orderings. That is, if two transactions $T, K \in S$ are not connected by a path ($(T, K) \notin R^+$), then the execution order of T and K can be arbitrary. Specifically, neither $T \rightsquigarrow K$ nor $K \rightsquigarrow T$ produces an incorrect ordering.

Assume G is cycle-free. We can then construct an arrival order $\rightsquigarrow$ that is equivalent to an executable modification order. Since there are no cycles in the connected regions of the graph and unconnected transactions can be ordered arbitrarily, we can create a path through the graph that traverses all vertices. The emerging path is an executable arrival order. This contradicts S being unorderable.

Hence, G contains a cycle, which we denote by C. The transactions in C form a cyclic acquisition; C is unorderable by Theorem 3.

The other direction follows directly. U remains unorderable, even if the set of transactions ordered by $\rightsquigarrow$ grows. ■

This theorem shows how the root cause of a conflict is always a cyclic acquisition over same-object transactions. Our ordering algorithm uses this property to detect unorderable transactions. The next section presents our ordering algorithm.

3.4 Ordering Transactions in a Two-Phase Algorithm

To determine a valid execution schedule of transactions on distributed persistent object, we implement a pessimistic distributed scheduler called 2PO. This scheduler constructs a valid execution schedule based on causal dependencies and user-defined ordering criteria (the release sets). It does this in two communication phases: Announcement and Execution. Figure 3.7 visualizes the two phases for a client issuing the transaction T which is part of a client operation O.

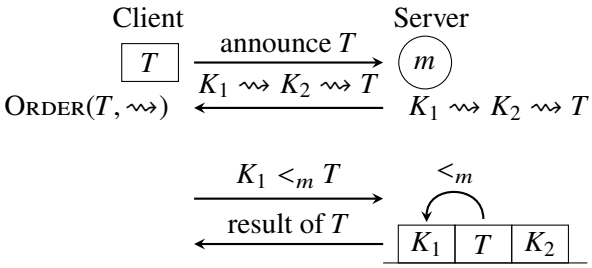


Figure 3.7: Two phases of 2PO visualized for one transaction T modifying the object m. The client deduces K_1 must precede T to produce a correct execution schedule.

3.4.1 Announcement Phase

The clients announce the operation to the servers which later execute the transactions. To announce an operation, the client sends an announcement message for every transaction in the operation to the server that will later execute the transaction. An announcement message contains three pieces of information about the announced transaction: The message contains metadata about the transaction itself (operation identifier, information about the sequencing of transactions, etc.), the transaction's ordering constraint (the $\mathsf{release}(T)$ set), and, finally, the set P_T. The set P_T stores the client's currently known $\mathsf{prefix}(T)$ at the time of announcement. Upon receiving an announcement for a transaction T, the server first stores the announced information. Then, it responds with the announcement information of all transactions that precede T in the arrival order.

Clients announce transactions in sequencing order. If $T \prec K$, the client announces T synchronously before K. The servers respond with their current local transaction backlog, i. e., the current arrival order of transactions ($\rightsquigarrow$). In Section 3.4.4, we discuss how and when to discard announcement information from the server's storage. With this response, a client then locally and deterministically compute the ordering of T relative to its known transactions.

To give a brief overview of the ordering algorithm, the client announcing a transaction T is able to calculate the set $\mathsf{prefix}(K)$ for all transactions K with $K \rightsquigarrow T$. The announcement of a transaction K includes the transactions in $\mathsf{prefix}(K')$ for the K' immediately preceding K in its operation ($K' \prec K$). We call these the *announced acquisitions* of K. With this information, other clients only have to deduce the set of acquisitions K performs directly. We call these the *immediate acquisitions* of K. This calculation is possible because the server makes the arrival order available to other clients. The combination of announced acquisitions and immediate acquisitions of a K corresponds exactly to the $\mathsf{prefix}(K)$. In the ordering algorithm, T acquires K if $T \notin \mathsf{prefix}(K)$.

The algorithm tracks dependency paths by performing a cycle check on the graph induced by the prefix sets. The client that would close a cyclic acquisition through an acquisition aborts its ordering. This breaks the cycle for all other clients on the path. Aborting the ordering has minimal negative effects, as no transactions were executed yet.

The following section describes in detail how the algorithm uses the announcement data to determine an execution order.

3.4.2 The Ordering Algorithm

This section discusses how each of the clients determines a part of the global execution order. The ordering algorithm is deterministic and fully-distributed across all participating clients. Fairness between the clients is guaranteed because the algorithm uses the arrival order of transactions at the server as the preliminary execution order. This inherently randomizes the execution order across all clients. A client orders transactions by consolidating the announcement information for every transaction it knows. It first detects whether the arrival order violates a constraint. If so, it avoids the conflict by deferring its own transactions accordingly. Otherwise, the algorithm uses the arrival order as the execution order and checks for new constraints. It tries to issue ordering instructions to fulfill the constraints immediately. Thereby, it may detect an unorderable cycle in the constraints. If this happens, it aborts the announcement phase and restarts before executing a single transaction. In this way, the transaction ordering algorithm is pessimistic. However, this approach has two main advantages over an optimistic scheduler. Firstly, a single client aborting breaks the cycle for other clients. Secondly, an abort of the announcement phase is fast, because no transactions need to be undone. Once the announcement phase has finished, the resulting execution order fulfills all constraints.

The remainder of this section discusses the details of the ordering algorithm. The proof of correctness follows in Section 3.4.3. Algorithm 3.1 shows the first phase of 2PO. It shows how a client executing an operation O iterates over the transactions $T \in O$, sends the announcements and subsequently initiates the ordering algorithm with the received announcement information. The ordering algorithm updates the locally known prefix for a transaction and the operation's ordering graph. This ordering graph represents a local view of the global execution order.

Definition 3.9 (Ordering graph)
Let C a client executing an operation O. The corresponding *ordering graph* is the graph $G_O = (\Gamma_O, <_O)$. The set Γ_O describes the transactions O locally orders. For $T, K \in \Gamma_O$, the relation $T <_O K$ represents T should execute before K. ♦

The ordering algorithm sets edges in the ordering graph. Algorithm 3.2 shows the ordering algorithm used in 2PO. The client C that initiated the transaction T executes the algorithm once. Therein, it iterates the transactions K that precede T in the arrival order of the modified object m. There are two main decision points in the algorithm. Firstly, C has to decide whether the execution order $K <_m T$ is valid considering the active acquire constraints. Secondly, C detects if the transaction T is part of a cyclic acquisition. In short, the algorithm needs to detect both, constraint violations, and unsolvable constraints. These two parts are critical to the correctness of the resulting execution order.

```
function PhaseOne(O)
    AA(T) = ∅ ∀T ∈ O
    G_O = (O, ≺)
    for T ∈ O do // Iterate in order of ≺
        ⇝_m = Announce(T, AA(T), release(T))
        Order(G_O, T, AA, ⇝_m) // Updates AA and G_O
```

Algorithm 3.1: High-level overview of the first 2PO phase.

```
1:  function Order(G_O, T, AA, ⇝_m)
2:      IA = CalcIA(AA, ⇝_m)
3:      for K : K ⇝_m T, K ∉ O do
4:          if T ∉ AA(K) ∪ IA(K) then
5:              // T acquires K
6:              AA(T') = AA(T) ∪ release(K)  ∀T' ∈ future(T)
7:              for L ∈ release(K) do
8:                  for T' ∈ future(T), modifies(T') = modifies(L) do
9:                      <_O = <_O ∪ {(L, T')}
10:             if CheckCycle() then
11:                 Abort // Cyclic acquisition detected
12:         else
13:             // No acquisition: K incorrectly ordered before T
```

Algorithm 3.2: Ordering algorithm of the announcement phase of 2PO.

To do this, the algorithm tracks three sets of transactions: Firstly, the algorithm tracks the set of *announced acquisitions* of transaction K, denoted by $AA(K)$. This set contains the released transactions of acquisitions preceding K, cf. Algorithm 3.2 line 6. As the name implies, the announcement message of transaction K contains the set $AA(K)$.

Using this set, other clients can deduce the set of transactions *immediately acquired* by K, denoted by $IA(K)$. If transaction K acquires another transaction, say L, this acquisition is not present in $AA(K)$. That is because $AA(K)$ is calculated prior to sending the announcement of K, whereas the acquisition of L happens after the announcement of K. Therefore, the announcement of K does not include $IA(K)$. However, because K and L must modify the same object and and $L \rightsquigarrow K$, any transaction following K in the arrival order knows both K, L and can identify if K would acquire L. Hence, it is possible to deduce $IA(K)$ from the arrival order and $AA(K)$. Algorithm 3.3 shows the algorithm that calculates the sets $IA(K)$.

These two sets allow other clients to determine the correct execution order between two transactions T and K. In Algorithm 3.2 line 4, the algorithm decides whether K is incorrectly ordered before T, or whether T should acquire K. Specifically, the algorithm determines $K <_m T$ is the correct modification order, and as such T acquires K, if

$$T \notin AA(K) \cup IA(K). \tag{3.1}$$

We call this the acquire condition. If the algorithm decides to acquire a K, it first issues ordering instructions into the local ordering graph G_O. These ordering instructions realize an acquire constraint between T and K (cf. Definition 3.2).

function CalcIA($AA, \rightsquigarrow_m$)
  for $\forall K : K \rightsquigarrow_m T$ **do** // Iterate in order of $\rightsquigarrow_m$
    $IA(K) = \{M \in \mathsf{release}(L) \mid L \rightsquigarrow_m K \wedge K \notin AA(L) \cup IA(L)\}$

Algorithm 3.3: Calculation of the IA sets in 2PO's ordering algorithm. Invoked as part of Order for transaction T in the operation O.

Finally, we have seen that some acquisitions may produce unsatisfiable constraints. Therefore, while creating the constraint in Algorithm 3.2 line 10, we check whether there exists a cyclic acquisition containing T through the acquired transaction K. If so, the announcement phase has to be aborted. This has minimal negative side-effects has no transactions were executed yet. Aborting the announcement phase means the client notifies the executing server of its abort. The server disregards the aborted operation during execution. This way, other clients do not have to be notified of the abort.

The following section discusses the correctness of this ordering algorithm.

3.4.3 Correctness of Two-Phase Ordering

The announced acquisitions and the calculated immediate acquisitions of each transaction are the basis for deciding whether a new acquire constraint should be created. We will see that an invocation of Order for transaction T can calculate $\mathsf{prefix}(K)$ for transactions K preceding T in the arrival order.

Theorem 5 (Acquire condition correctness):
In the ordering algorithm invocation for a transaction T with $K \rightsquigarrow_m T$ it holds

$$AA(K) \cup IA(K) = \mathsf{prefix}(K). \qquad \diamond$$

Proof The ordering algorithm calculates the AA sets as follows (cf. Algorithm 3.2 line 6)

$$AA(K) = \bigcup_{K' \neq K:\ K \in \mathsf{future}(K')} \mathsf{acquire}(K').$$

It remains to show that the algorithm calculates

$$IA(K) = \mathsf{acquire}(K).$$

We do this by induction over the sequence induced by $\rightsquigarrow_m$. Let $(T_1, T_2, \ldots, T_n)$ the sequence s.t. $T_i \rightsquigarrow_m T_{i+1}$.

($i = 1$): For T_1, there is no transaction L with $L \rightsquigarrow_m T_1$. Therefore, it holds

$$\mathsf{acquire}(T_1) = IA(K) = \emptyset.$$

($i \rightarrow i + 1$): Let S the set of transactions acquired by T_{i+1}. Then we have

$$\mathsf{acquire}(T_{i+1}) = \bigcup_{L \in S} \mathsf{release}(L).$$

It is clear that for every $L \in S$ it must hold $L \rightsquigarrow_m T_{i+1}$. Therefore, by induction it holds $IA(L) = \mathsf{acquire}(L)$.

The transaction T_{i+1} acquires T_j with $j \leq i$ if $T_{i+1} \notin \mathsf{prefix}(T_j)$ (cf. Definition 3.6). As such we get

$$S = \{L \mid L \rightsquigarrow_m T_{i+1} \wedge T_{i+1} \notin \mathsf{prefix}(L)\}$$

Specifically, this gives the final proposition of the theorem. Algorithm 3.3 calculates

$$\begin{aligned} IA(T_{i+1}) &= \{M \in \mathsf{release}(L) \mid L \rightsquigarrow_m T_{i+1} \wedge T_{i+1} \notin AA(L) \cup IA(L)\} \\ &= \{M \in \mathsf{release}(L) \mid L \rightsquigarrow_m T_{i+1} \wedge T_{i+1} \notin \mathsf{prefix}(L)\} \\ &= \mathsf{acquire}(T_{i+1}). \end{aligned}$$

■

Corollary (Acquire condition): Let T, K two transactions on object m. The acquire condition (cf. Algorithm 3.2 line 4) is true if

$$T \notin \mathsf{prefix}(K).$$

◊

Semantic correctness depends on the programmer's constraint semantics. Such semantics are given as ordering constraints which can then be applied to individual transactions. Programmers can specify both the constraint and the transactions they apply to. The execution ordering of a constrained transaction must then fulfill the constraint. The constraint's ordering requirements are fully described by the prefix. Hence, with the prefix set of other surrounding transactions, the algorithm can issue correct ordering instructions. Now, we show how a client detects if a transaction is incorrectly ordered and avoids issuing a violating ordering instruction.

7 Lemma (Local correctness):
Let O an operation. For every $T \in O$ and an arbitrary transaction K that modifies the same object m as T, the following holds. If K is incorrectly ordered before T, it holds $K \not\prec_O T$. ◊

That is, if a pair of transactions is incorrectly ordered, the algorithm does not issue ordering instructions that require the execution of transactions in an incorrect order.

Proof By Definition 3.6, if K is incorrectly ordered before T, it holds $T \in \mathsf{prefix}(K)$. Then, there exists a set

$$A = \{K' \mid K' \in \mathsf{future}(K)\}$$

with which it holds

$$T \in \mathsf{prefix}(K) = \bigcup_{K' \in A} \mathsf{acquire}(K').$$

Specifically, there must be transaction K' that acquired a transaction L where $T \in \mathsf{release}(L)$.

Thus, let L the first transaction (in sequencing order $\prec$) for which $T \in \mathsf{release}(L)$ holds. Say K' acquires L, then the algorithm sets $T \in \mathsf{release}(L) \subseteq AA(K'')$ for $K' \prec K''$. Thus, any transaction $L' \in \mathsf{future}(L)$ with $K' \rightsquigarrow L'$ will not acquire L' or any other transaction in A that may cause an incorrect ordering.

Finally, it must hold $T \neq L$ because we have seen in Corollary 4, a transaction T acquires K only if $T \notin \mathsf{prefix}(K)$.

This proves the statement. ■

This alone does not prove that a client outputs correct ordering instructions. Specifically, an empty ordering graph would technically fulfill the requirements. As such, we prove the algorithm issues ordering instructions for all relevant prefixes.

8 Lemma (Local completeness):
Let C a client, O an operation initiated by C and $G_O = (\Gamma_O, <_O)$ the associated ordering graph. Given $\prec_O \subseteq <_O$, the ORDER algorithm issues the following ordering instructions

$$\forall T \in O, \forall K \in \mathsf{prefix}(T): \quad \mathsf{modifies}(K) = \mathsf{modifies}(T) \implies K <_O T. \qquad \diamond$$

In other words, if the ordering graph respects the sequencing relation (i.e., the implicit execution order of transactions issued by the same client), the ordering graph G_O orders transactions $K \in \mathsf{prefix}(T)$ before the execution of T.

PROOF

$$K \in \mathsf{prefix}(T) = \bigcup_{\forall T' \colon T \in \mathsf{future}(T')} \mathsf{acquire}(T')$$

for which there exists a corresponding $K' <_m T'$ s.t.

$$= \bigcup_{\forall T' \colon T \in \mathsf{future}(T')} \mathsf{release}(K').$$

Specifically, this implies there was an acquisition between K' and T'. Thus, the ORDER invocation on T' issues the ordering instruction $K <_O T$ in Algorithm 3.2 line 9. ∎

We can further extend this proposition to an equivalence.

9 Lemma:
For any pair of transactions T, K the following equivalence holds

$$K <_O T \Leftrightarrow K \in \mathsf{prefix}(T). \qquad \diamond$$

PROOF ($\Rightarrow$): If there exists such an ordering $K <_O T$, then there must have been a transaction T' with $T \in \mathsf{future}(T')$, for which the acquire condition on a K' with $K \in \mathsf{release}(K')$ was true. With that, it holds

$$K \in \mathsf{release}(K') \subseteq \mathsf{acquire}(T') \subseteq \mathsf{prefix}(T').$$

Because $T \in \mathsf{future}(T')$, we get $T' \prec T$. Now, we can apply Lemma 2 and get the proposition

$$K \in \mathsf{prefix}(T') \subseteq \mathsf{prefix}(T).$$

($\Leftarrow$): Proven by Lemma 8. ∎

Thus, each client calculates a part of the global execution order according to the prefix. It remains to show that the combination of such parts does not produce conflicting ordering information. Recall how multiple acquisitions can make a set of transactions unorderable. This means, there exists no execution order that fulfills all constraints. If two clients produce such conflicting ordering instructions, the server runs into a deadlock, trying to execute the transactions with cyclical dependencies. We have yet to see that this does not happen. That is, the combination of all clients' ordering graphs is conflict-free.

Definition 3.10 (Conflict-free ordering graphs)
Let $\{G_1 = (\Gamma_1, <_1), G_2 = (\Gamma_2, <_2), \ldots, G_n = (\Gamma_n, <_n)\}$ a set of ordering graphs. These ordering graphs are called *conflict-free* if the graph

$$\hat{G} = (\hat{\Gamma}, \hat{<}) = \Big(\bigcup_{i \in \{1,\ldots n\}} \Gamma_i, \quad \bigcup_{i \in \{1,\ldots n\}} <_i \Big)$$

is acyclic. ♦

Luckily, we have seen in Theorem 4, that an unorderable set of transactions exists if and only if it contains a cyclic acquisition over same-object transactions. A cycle in the ordering graph $\hat{G}$ is the result of a cyclic acquisition.

10 Lemma (Conflicting ordering graphs):
A set of ordering graphs contains a cycle if and only if there exists an unorderable set of transactions in the graph. ◊

Proof As shown in Theorem 4, an unorderable set of transactions exists if and only if there exists a cyclic acquisition over same-object transactions. Thus, the lemma directly follows from Lemma 9. ■

The next theorem shows that the ordering algorithm is able to detect cyclic acquisitions and abort the operation.

Theorem 6 (Cycle detection):
If $C = \{T_1, T_2, \ldots, T_n\}$ is a cyclic acquisition and all T_i modify the same object m, then there exists a client C that detects the cycle. The client C initiated one of the transactions T_i. ◊

Proof Assume $\rightsquigarrow_m$ is the arrival order of m. This totally orders all T_i where the permutation $\pi(i)$ describes the position of T_i in the total order $\rightsquigarrow_m$. We show that the client C executing the transaction T_{last} with $last = \pi^{-1}(n)$ (the last in the arrival order) detects the cycle and aborts its announcement.

By Theorem 5, the client C can calculate $\mathsf{prefix}(T_i)$ for all T_i. Specifically, C can create the transitive closure over the $\mathsf{prefix}(T_i)$ and detect whether it contains T_{last}. ■

A single aborting client can make previously unorderable sets of transactions orderable. Thus, conflicts caused by cyclic constraints can be avoided using a single abort.

Corollary (Conflict Avoidance): Let $C = \{T_1, T_2, \ldots, T_n\}$ a minimal cyclic acquisition and all T_i modify the same object m. After a client C aborts its transaction T_j, the set

$$\{T_1, T_2, \ldots, T_n\} \setminus \{T_j\}$$

has an execution order that fulfills all constraints. ◊

Proof The execution order

$$T_{j+1} <_m T_{j+2} <_m \ldots <_m T_n <_m T_1 <_m T_2 <_m \ldots <_m T_{j-1}$$

is valid. ■

This ensures the global execution graph is executable.

Corollary (Global Correctness): The ordering graphs of non-aborted client operations are conflict-free. ◊

Proof An aborting client issues an empty ordering graph. ■

In conclusion, we have proven that each unorderable set of transactions contains a cyclic acquisition over same-object transactions (Theorem 4). The ordering algorithm then detects such cycles and causes one client to abort and retry (Theorem 6). The abort allows the rest of the transactions to become orderable (Corollary 5). Specifically, this ensures the global ordering graph is conflict-free and as such executable.

3.4.4 Server-Side Space Complexity

The server needs to store the arrival order of transactions T along with the sets AA and $\mathsf{release}(T)$. The server sends its stored information to a client if there might be an incorrect ordering or acquisition between the client's and one of the stored transactions. This information can quickly grow arbitrarily large, if the server does not discard information from its storage.

11 Lemma (Overlapping Announcement Phases):
Let K, X two interfering transactions. If X is incorrectly ordered before K, the announcement phases of $\mathsf{oper}(K)$ and $\mathsf{oper}(X)$ must overlap in time. ◊

Proof Let $m = \mathsf{modifies}(K)$. By definition (cf. Definition 3.6), we have $X \rightsquigarrow K$; and there must be a transaction $T \in \mathsf{oper}(K)$ with $K \in \mathsf{release}(T)$. Finally, there must be a transaction $L \in \mathsf{oper}(X)$ with $T <_m L$. To get to this execution ordering of $T <_m L$, either the algorithm used the preliminary ordering $T \rightsquigarrow L$ or L was incorrectly ordered before T. If L was incorrectly ordered before T, apply the proof to these two transactions. This recursion is finite because a client operation is a finite sequences of transactions. Thus, let us now assume $T \rightsquigarrow L$. We have either

1. $T \rightsquigarrow X \rightsquigarrow K$, or
2. $T \rightsquigarrow L \prec X \rightsquigarrow K$

which shows the announcement phases must overlap in time. ■

As soon as a client C sends the first execution message for a transaction of operation O, the server may discard all transactions $T \in O$ from its temporary storage. At this point, no transaction $T \in O$ can acquire other transactions, and as such, the ordering of transactions in O can no longer change. A transaction K that arrives after the last transaction in O can no longer induce a reordering of a transaction $T \in O$ after K.

3.4.5 Execution Phase

In the execution phase, the clients have finished constructing their local ordering graph, that is, they calculated ordering specifications for every transaction they issued They send the execution instructions along with this ordering information to the executing server. Because all cycles have been broken in the announcement phase, we know for certain that instructing the server to execute a transaction with the derived ordering instructions can not produce a cyclic dependency. Thus, the server does not need to perform deadlock avoidance.

Moreover, the orderings produced by 2PO only describe histories. A client instructing the server to execute a transaction T sends its ordering relation $<_C$ to the server. If a client operation contains the transactions $T_1, T_2, \ldots$, then its ordering relation $<_C$ will only contain edges where the T_i appear on the right side. A server receiving the execution instruction for T_j along with the ordering instructions $<_C$, can immediately execute T_j if all transactions K_j with $(K_j, T_j) \in <_C$ are finished. The execution of a transaction with yet unfulfilled precedence can be deterministically deferred. Specifically, a server does not need to wait for *all* outstanding execution instructions to arrive before being able to execute any transaction. This is because no client issues ordering instructions stating one of its own transactions should *precede* another client's transaction. Hence, the server does not need to consolidate the execution orders $<_C$ of all clients C into the global execution order graph. This ensures forward progress of the execution.

Regarding failure modes of the system, the atomic state transition protocol ensures we do not partially apply a transaction's effects, which could potentially lead to corruption or data loss. Recall that we do not require transactions to be idempotent. Therefore, to ensure network failures do not interfere with the processing of a sequence of transactions, the underlying network protocol needs to prevent duplicate messages (i. e., we use TCP).

3.4.6 Consistency of Two-Phase Ordering

The consistency guarantees of our approach depend on the programmer's chosen release semantics. Programmers may allow a more relaxed consistency model if the operation's business logic does not require a stricter consistency level. To define the release semantics, we introduce *semantic constraints* which automatically deduce the set $\mathsf{release}(T)$ of a transaction T given the surroundings of T. Programmers select semantic constraints on a per-transaction granularity and use them as annotations on the transactions. As part of this work, we present two semantic constraints we used as part of our evaluation. Figure 3.8 shows the flushing relationship for the two consistency models described below.

Logical object locks Firstly, the *ObjectLock* describes a virtual distributed lock on a given persistent object. We call it virtual, because the semantic does not actually create a lock on the object. Rather, the execution of the individual transactions is reordered such that no access may be concurrent to a locking operation. By annotating a transaction T as `ObjectLock<T>`, the set $\mathsf{flush}(T)$ contains the corresponding `ObjectUnlock`. Any other transaction K trying to access the object $m = \mathsf{modifies}(T)$ after the locking transaction was executed will perform an acquisition on T. As such, the execution of K will be deferred after the `ObjectUnlock`. This makes transactions between an `ObjectLock` and the corresponding `ObjectUnlock` appear atomic to other transactions.

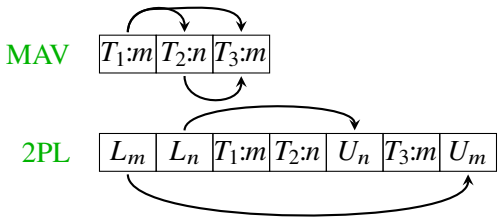


Figure 3.8: Flush set of the same operation, using MAV and 2PL consistency semantics. An arrow from A to B means $B \in \mathsf{flush}(A)$.

Using our ObjectLock semantic constraint, we can implement 2PL. Our ordering algorithm 2PO internally resolves a *wait-for graph* in the form of the prefix paths. 2PO detects cycles in the graph; that is, 2PO detects possible deadlocks before any locks were even issued. This avoids complex distributed lock management strategies. To realize a linearizable ordering using 2PL, the operation must adhere to the two phases of 2PL: That is, it must not issue new ObjectLocks after an ObjectUnlock transaction was executed. Thus, it may require additional (no-op) transactions which are only there for ordering purposes.

Monotonic atomic view To avoid additional transactions and to highlight that the system also supports weaker consistency models, we implement the *TotalRelease* semantic constraint. A transaction annotated as `TotalRelease` flushes all succeeding transactions in the operation. With this, `TotalRelease` realizes MAV [BDF+13] isolation. This falls under the class of consistency models especially suitable for highly available systems [BDF+13]. MAV informally means once an operation O_j sees parts of the effects of another operation O_i, the operation O_j must observe all effects of transactions in O_j.

4 Implementation Details

This section highlights implementation details of three core components of the system. Firstly, we give more insight on the implementation of a remote transaction call and the steps this process entails. Secondly, we introduce how we represent client operations inside our system. Specifically, we present a representation for a transaction's causal surroundings. Lastly, we give a brief introduction to semantic constraints and how they use such causal surroundings.

4.1 Remote Transaction Calls

To execute a transaction remotely, we first have to address the object modified by the transaction. Secondly, to allow a location-independent execution of the modifying function, we need to address the modifying function on a remote server. Remote references solve the first of these two problem. Moreover, a remote reference stores additional metadata about an object which provides additional quality of life improvements for the programmer. We approach the problem of function addressing via a bind-and-dispatching architecture. The following sections detail these two concepts.

4.1.1 Remote References

A client does not know the physical address of a persistent object located on a remote server. Therefore, the client addresses persistent objects using virtual addresses we call *remote references*. Remote reference can be resolved to the server that stores the object using a reference lookup scheme (e. g., [KK03; SML+03]). This server can then resolve a remote reference to the physical memory address of the object.

In addition to hiding the physical location of an object, remote references represent a crucial building block in the client-side type safety of the distributed system. Figure 4.1 shows exemplary remote references and the types of access allowed using them. Remote references store the type of the persistent object they represent. Thus, a remote reference ensures even an object access over the network is type safe at compile-time. Moreover, remote references can be const-qualified. We generate a compile error if a client tries to modify a persistent object through a const-qualified remote reference.

Using a remote reference, we address individual objects in the distributed storage system. To access an object's data or to modify its state, we use remote transactions. The following section presents our remote transaction call architecture.

```
class A {};
class B {};

persistence::TReference<A> ref_a;             // can modify objects of type A
persistence::TReference<const A> const_ref_a; // can only read objects of type A

persistence::TReference<B> ref_b;             // can modify objects of type B
persistence::TReference<const B> const_ref_b; // can only read objects of type B
```

Figure 4.1: Remote references can be const-qualified. Trying to modify a remote object through a const-qualified reference will raise a compile error.

4.1.2 Bind and Dispatch

In remote data processing, we differentiate two interaction models [SW16]. The data-flow model only allows clients to read and write data on remote servers. The control-flow model on the other hand allows the control flow of a program to virtually move between client and remote server. Specifically, remote servers can execute arbitrary functions on the data. A remote transaction can then be thought of as passing the control flow from the client to remote servers which execute a local transaction and return the control back to the client. The control flow model can emulate the data-flow model and allows heavy-weight data processing, even when the client is less powerful. A client does not need to provide compute capabilities yet is still able to execute complex operations on the servers. Therefore, we allow greater freedom for the physical object location without limiting the processing capabilities.

For these reasons, our system supports the control-flow model of interaction with objects. Programmers can balance the load of their applications across several machines. The programmer decides the degree with which work is distributed among the client and server. For that, programmers create *remotely executable* methods. Specifically, we support arbitrary function executions on a given remote object. Remotely executable methods can be seen as stored procedures that access one data object at a time. To create such a remotely executable method, a programmer defines a stateless hashable struct with a call operator. The call operator implements the logic of the remotely executable function. Figure 4.2 shows the remotely executable function `GaussSum` as an example. To the client, the struct defines the remote interface. The call operator's first argument is a pointer to a persistent object. That is, the first argument must have the type `T* const` where `T` is a persistable object type and the asterisk represents the pointer type. We call this first argument the *object pointer*. Each remote call operates on a persistent object, the object pointer gives the function access to that object. The arguments following the object pointer are the caller arguments. They can be any data with the only restriction being that they must be serializable. The client's remote reference is resolved to an object pointer at the executor-side.

To be able to call a remotely executable function as a transaction on a persistent object, we first need to *bind* it to the system. This creates a link between the function's identifier (a hash value) to the executable function on the executor. Specifically, it allows the executor to execute the function just from an identifier, without any further type information. Once the function has been bound, the client can call it on an object. To call this function on an object, a client uses the object's remote reference to address the object. On a high-level view, a remote function call entails four steps:

```
// Main node class as interface to the storage system
Node<...> node;

// Remotely executable function that accesses an integer persistent object
class GaussSum {
public:
    auto operator()(int * const obj, int arg) noexcept -> int
    {
        auto g_sum = (arg * (arg + 1)) / 2;
        *obj = g_sum;
        return g_sum;
    }
};

// Bind the function to the node
node.bindFunction<GaussSum>();

...

// Dispatch the function execution on the object pointed to by ref.
persistence::TReference<int> ref = ...;
int arg = 5;
auto result = co_await node.callRemote<GaussSum>(ref, arg);
```

Figure 4.2: Remotely executable function that operates on an integer object, accessed through the object pointer `obj`, and an integer `arg` as caller argument. A client needs to supply both, a non-const remote reference of integer type to address the object behind `obj`, and a second integer for `arg`. If either is missing or wrongly typed, the compiler will issue an error.

Firstly, a client invokes the lookup scheme and resolves the remote reference to the node that stores the corresponding persistent object. Secondly, the client sends the remote reference, the function's hash along with any function arguments to that node. The hash simply reduces the message size required to address such a function on a remote server. Thirdly, the server resolves the remote reference to the object pointer in memory, selects the method with the requested function hash and locally calls it with the given arguments. Finally, it sends the return value of the function call as a response to the client.

The interface definition in combination with the typed remote reference enables full compile-time type- and qualifier-safety even for remote transaction executions. Because our system is type-safe at compile-time, we neither have to check correctness of the types at runtime nor do we have to attach type information to serialized objects. Hence, our system has minimal overhead for a remote call because it does not require any runtime type checks and there is no intermediate type processing or type-related metadata storage.

4.1.3 ACID Properties of Function Calls

If the server were to experience a crash failure in the middle of the function execution, any intermediate modifications to the object would be retained. This can easily leave the persistent memory in an inconsistent or invalid state. Therefore, let us now briefly discuss atomicity of a remote function call on persistent objects. The exact implementation details of the persistence strategy are outside the scope of this work. Here, it is enough to limit ourselves to three concepts of the underlying persistent storage layer. Firstly, for each persistent object, we use a reader/writer lock that ensures mutual exclusion of concurrent writes. Secondly, for each reference to a persistent object, the underlying layer can provide a pointer to a temporary representation of the object. Finally, there exists a `persist(...)` method which atomically commits any changes from a temporary object copy to persistent storage.

Upon receiving a call request that contains an object reference o, the function's hash h and any further arguments a, the server does the following: Retrieve the corresponding pointer p to a temporary representation of the object referenced by o. Find the call operator function f that corresponds to the function hash h. Acquire the exclusive lock on object o. Invoke $f(p, a...)$, call `persist(o)`, and finally, release the lock on o.

Corollary: This implementation ensures the access to o is mutually exclusive, that is, the object is exclusive according to Definition 2.2. ◊

Proof The locking of o ensures no two modifications can happen at the same time. Similarly, because the lock is exclusive, no read can happen during the modification performed by f. Finally, the reader/writer lock ensures no write can modify the object while there are still readers in the critical section. ■

Then, given the execution of f transitions the persistent (i. e., durable) object from one consistent state to another, the ACID properties of this implementation are trivially fulfilled. We now have a good understanding of the transactions and how they modify objects. As we have proven before (Theorem 2), the objects' exclusivity ensures modification order coherence.

This concludes the access to remote objects. However, as part of this work, we introduced the concept of client operations. Client operations allow sequences of transactions to be executed and ordered such that isolation between different client operations can be achieved. The next section introduces some implementation details on client operations.

4.2 Client Operations

A client operation is a logical group of one or more causally related object-level transactions. Client operations give the ordering algorithm application-specific knowledge about a client's data access. We describe the operation as a directed graph of segments which are connected by their causal dependencies. This section details segments and the additional information we store to give access to application-specific knowledge.

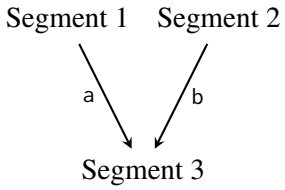
4.2.1 Segments

Segments are the vertices of the operation's execution graph. Operations may contain local segments (that are local non-transactional function executions) and remote segments (that are remote object-level transactions). Figure 4.3 shows an example where two local segments execute in parallel and then join their output values into a third segment. This example also gives a first introduction to the interface programmers can use to define client operations. In this interface the `||` operator specifies possible parallelism between segments, as there exist no data-dependencies between the two. The `|` operator creates data dependency edges ($\prec$) between segments.

Our implementation checks all data dependencies at compile-time. This includes checks for completeness and type-compatibility of the outputs and arguments. The implementation checks dependencies between local and remote segments in the same way. For example trying to call a remote transaction which operates on an integer persistent object using a float remote reference would output a compile error. Similarly, the compiler checks whether all function arguments are correctly typed, independent of the location where the function will be executed. This robust type-safety minimizes the potential for type-related errors, improving both reliability and security of the entire system. In contrast to traditional systems, our system then requires no type-checks at runtime.

```
(
    operation::begin() |
    []() -> coro::Task<int> {
        std::cout << "running 1" << std::endl;
        co_return 1;
    } // Segment 1
    ||
    []() -> coro::Task<int> {
        std::cout << "running 2" << std::endl;
        co_return 2;
    } // Segment 2
)
|
[](int a, int b) -> Task<int> {
    assert(a == 1 and b == 2);
    co_return a + b;
}; // Segment 3
```

(a) Definition of the operation using our interface.



(b) Logical segment graph representing the data dependencies.

Figure 4.3: Two local segments execute concurrently without a pre-defined order (parallel operator `||`). Then, they join their output values into a third local segment (pipe operator `|`). The order of the parameters `a,b` is fixed independent of the execution order of the first segments.

4.2.2 Transaction Indices

The sequencing relation describes the internal execution ordering constraints of the data dependencies inside the operation. Other external constraints, e.g., those given by the programmer can also use the sequencing relation for their description. Specifically, a programmer's constraints can influence the execution ordering of a transaction's causal surroundings relative to other transactions. Hence, we describe the causal surroundings to make them available for the ordering algorithm.

For that, we implement *transaction indices*. Each transaction in an operation has a transaction index which uniquely identifies the remote segment inside the operation. The transaction index only depends on the structure of the operation's dependency graph. Specifically, the transaction index is a vector time stamp (cf. [Mat88]) encoding the dependencies. Moreover, this makes it possible to uniquely identify every transaction in an operation. We identify an execution instance of an operation by its *operation ID*. The operation ID is a global identifier of the operation as a whole. Every execution of a segment inside the same operation uses the same operation ID. Every transaction execution can be uniquely identified globally by its operation ID and its transaction index.

The remainder of this section explains the deduction rules for transaction indices in an operation. Figure 4.4 shows the resulting transaction indices of an operation with eight segments for reference.

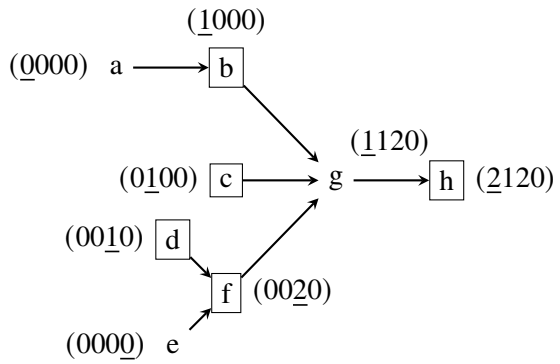


Figure 4.4: A client operation with eight segments. Remote segments are drawn with a box, local segments without. Each segment is annotated with its transaction index vector clock. Moreover, we mark the vector indices v_s of each segment with an underline.

The size of the vector is equal to the number of leaves in the dependency tree. The deduction of transaction indices is given by induction over the edges. We begin at the leaves of the dependency tree. Each leaf segment corresponds to a vector index in the vector time stamp. For a segment s, we denote its index in the vector as i_s. Let s a leaf segment. If s is a local segment its transaction index T_s is the zero vector. If s is a remote segment then it is

$$T_s = (t_0, t_1, \ldots, t_n), \quad t_j = \begin{cases} 1 & j = i_s, \\ 0 & \text{else.} \end{cases}$$

For the induction step, let s an inner segment with input segments

$$I = \{s' \mid (s', s) \in E\}$$

where E is the set of edges in the dependency graph. Then we initialize the transaction index of s with the merged transaction indices of the input segments

$$T_s = \sum_{s' \in I} T_{s'}.$$

Moreover, the index i_s of this segment is

$$i_s = \min\{i_{s'} \mid s' \in I\}.$$

If the segment s is a remote segment, we tick the vector clock by one at the index v_s.

The size of the vector depends on the complexity and degree of parallelism in the operation. It is possible to encode the transaction indices in a single integer without loss of information. Kshemkalyani et al. present an adaption of vector clocks that encode the vector using a product of prime numbers [KKS18]. They realize the clock's tick operation by the multiplication with a prime number. If every tick globally selects a new prime, we can identify if an event is contained in the encoded time stamp by checking if the event's prime number is contained in the product. The main problem with this approach is the product's space complexity; the size grows linear in the number of events. However, in our case, the total number of primes required to represent all transaction indices is fixed and known a priori. We can even select the first n prime numbers rather than particularly large primes. Thus, we can reduce the space complexity of transaction indices to (effectively) constant.

The transaction index is an operation-local identifier specific to a single remote segment. Moreover, by comparing the indices of two segments it is possible to determine whether there exists a data dependency between them. Specifically, a transaction can derive its causal precedence from its own transaction index. With access to all transaction indices of an operation, we can even identify the *causal future* of a transaction. Semantic constraints use these exact properties to determine allowed interleavings of concurrent operations. The following sections introduce semantic constraints for object transactions on distributed persistent objects and how programmers can influence the ordering of concurrent operations.

4.3 Semantic Constraints

Semantic constraints use the causal surroundings of a transaction to influence the final execution ordering of concurrent transactions. They define the flush sets of transactions inside an operation. Programmers do not directly define the flush set, but annotate their transactions with semantics. As part of this work, we use the `Release` and `ObjectLock` semantics with which we realize MAV or 2PL respectively. Figure 4.5 shows a simplified order transaction example, inspired by the TPC-C benchmark [Tra]. The MAV implementation requires fewer overall transactions than 2PL because 2PL needs to keep the acquiring and shrinking phase separate to ensure linearizability [BHG+87]. The figure also shows how programmers can specify remote segments in our interface. The placeholders `_1`, `_2`, etc. allow executing the same operation on different sets of remote objects. The implementation of the placeholders still ensures compile-time type safety when using placeholders. With placeholders, users can supply object references at call time of the operation, rather than at the definition time.

```
(
    operation::begin() |
    operation::execute_on<Release<ReserveCart>>(_1)
    ||
    operation::execute_on<Release<GetCustomerData>>(_3)
)
| createOrderlist
| operation::execute_on<Release<InsertOrderLines>>(_2);
```

(a) MAV

```
(
    operation::begin() |
    operation::execute_on<ObjectLock<ReserveCart>>(_1)
    ||
    operation::execute_on<ObjectLock<GetCustomerData>>(_3)
)
| createOrderlist
| operation::execute_on<InsertOrderLines>(_2)
| operation::execute_on<ObjectUnlock<NoOp<Customer>>>(_3)
| operation::execute_on<ObjectUnlock<NoOp<Warehouse>>>(_1);
```

(b) 2PL

Figure 4.5: Usage of semantic constraints in an operation.

It is possible to implement custom synchronization semantics. The system exposes the causal surroundings to an instance of the semantics. With this, programmers can apply customized flush set rules. The system automatically applies flush set expansion and automatically deduces the release set based on the given flush instructions.

5 Related Work

Transaction Coordination The state-of-the-art approach is to wrap the client's interactions with the entire system in a (global) transaction. Then, to ensure data consistency, the systems guarantee serializability (cf. [BHG+87]) of the global transaction's operations using an operation scheduler. There are two basic approaches to transaction scheduling that ensures serializability of distributed transactions: optimistic and pessimistic schedulers (cf. [BHG+87] who differentiate aggressive and conservative schedulers). An optimistic scheduler tries to execute a transaction's operations as soon as possible. It does not reserve time to reorder the execution to a later point in time. Without this possibility to reorder, the scheduler might later find that it scheduled two operations in a non-serializable manner. Thus, it must resort to aborting the transactions to recover serializability. Pessimistic schedulers on the other hand deliberately delay operations to avoid aborts.

The performance differences of optimistic and pessimistic schedulers depends on the application's degree of conflicting concurrency. An optimistic scheduler is called so, because it is optimistic that only few operations will conflict. On the other hand, the pessimistic scheduler works under the assumption that there will be a conflict almost every time. Consequently, applications that rarely issue conflicting operations might perform better under an optimistic scheduler. Since conflicts are rare, optimistic schedulers would not need to reject operations often. Conversely, applications that often concurrently issue conflicting transactions might benefit from a pessimistic scheduler. The pessimistic scheduler would then reserve enough potential for reordering such that only few transactions need to be aborted.

2PL is a pessimistic concurrency control algorithm which serializes the execution of transactions that contain conflicting operations [BHG+87; BMT+19; MCZ+14]. It does so by acquiring locks on all objects the transaction intends to access. When two (or more) transactions concurrently acquire locks, deadlocks can arise if each transaction requires a lock held by the other transaction. Consequently 2PL sometimes aborts transactions to prevent such deadlocks [CDE+13; MCZ+14]. OCC [KR81] is an optimistic scheduler that executes transactions without acquiring locks on the accessed data. Then, before committing, a transaction must verify that no other transaction modified the data it has read. A failed check reveals a non-serializable execution ordering. Thus, the conflicting transactions need to abort. With high contention, both OCC and 2PL suffer low commit rates [HAMS18; MCZ+14]; throughput can be below 10 % of the maximum [MCZ+14]. This also causes transaction latency to increase with contention.

There have been numerous studies on concurrency control methods: [DKG18; MAN+14; MCZ+14; TDW+12; XSL+15] develop new algorithms, Mahmoud et al. [MAN+14] avoid distributed locking in transaction coordination. The authors of [LLZ+22; LMP17] move the coordination efforts from the servers into the network by utilizing programmable switches. In contrast to these approaches, we exploit the knowledge of client's object access. Thus, the clients know their order of operations and cooperate in finding the execution schedule. By using this approach, we achieve a higher degree of decentralization. Mu et al. [MCZ+14] present a similar approach to ours, as they perform

a pre-execution reordering of transactions. However, they use a centralized coordinator, and if a conflict has to be reordered, it has the same communication complexity as 2PL or OCC. In comparison, our algorithm, 2PO, distributes dependency information through the objects, not through a centralized server. The clients hold just enough information to be able to deterministically calculate a consistency model-compliant execution order in a fully distributed manner. This avoids both centralization and further communication efforts. Similar to our approach, Calvin eliminates concurrency control by using a deterministic algorithm to calculate the execution order beforehand [TDW+12]. Its schedulers are co-located with each storage partition. In our system, each client schedules its own transactions globally, improving the parallelization of the system.

Consistency models Bailis et al. [BDF+13] analyze weaker consistency models targeted towards highly available transactions. They provide a formalization for many consistency models, including MAV, which we use. Herlihy and Wing [HW90] describe linearizability theory.

Ordering and causal dependencies Lamport [Lam78] first defined the happened-before ($\rightarrow$) relation as we are using it here. Lamport's logical clocks create a total order on all events of a distributed system. However, processors could not yet identify causal relationships between events. [Mat88] later introduced vector clocks that capture causal precedence. Vector clocks capture a logical time stamp for every process. In our system, we use it to represent dependencies between several transactions that execute as part of the same logical operation. The vector time stamps then represent causal dependencies between such transactions. As such, our vector timestamps have static values per operation. Specifically, the entries do not change over time. This allows vector representation optimizations such as in [KKS18].

6 Evaluation

In this section, we evaluate our approach in two settings. Firstly, we use a local benchmark, where we compare our local transaction throughput to that of MySQL. Secondly, we evaluate the distributed scaling of our transaction coordination algorithm under varying levels of contention.

6.1 Local Throughput

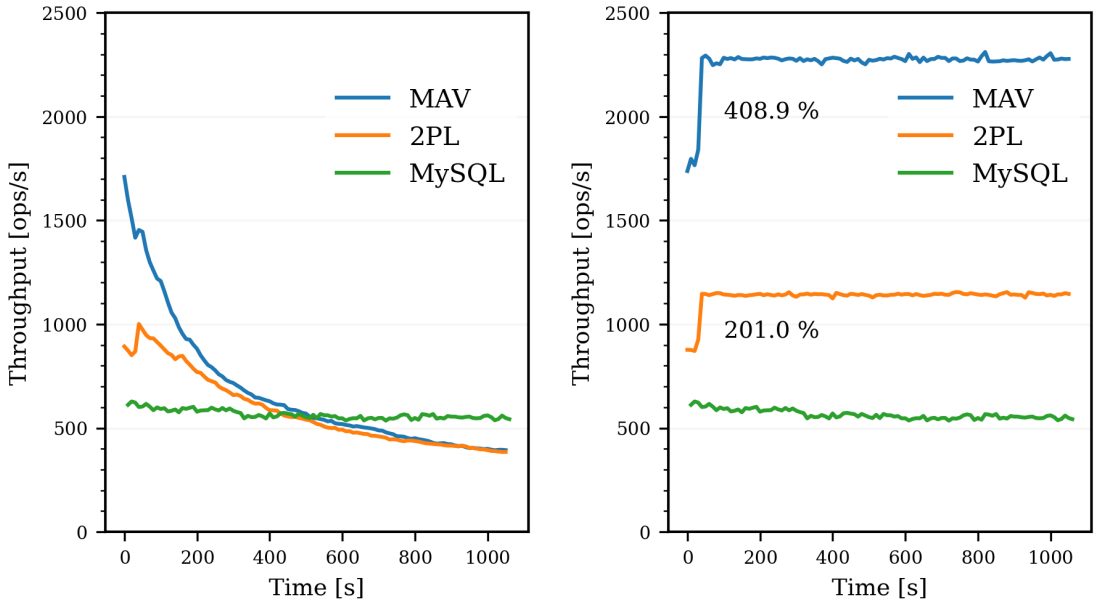
We ran our local measurements on a machine equipped with an Intel Xeon Gold 6338, 64 GB of DDR4-3200 main memory along with 512 GB of Intel Optane NVRAM DIMMs. We use a Samsung PM9A3 PCIe 4.0 x4 NVMe SSD for our baseline measurements on persistent storage. In this setting, we use the NVRAM as stable storage.

We measure the transaction throughput using the TPC-C benchmark's new order transaction [Tra]. As a baseline, we use a MySQL instance that uses 10 warehouses and 10 connections. To measure our system, we implement a logical equivalent of the new order transaction and execute it on 10 warehouse objects, using 10 clients. Figure 6.1 shows the throughput of MySQL compared to our system.

In Figure 6.1a, we see the impact of object size on transaction throughput in our system. While MySQL achieves a near-constant transaction throughput, our system suffers the copy overhead of the increasing order list size. This is entirely caused by the underlying atomic state transition protocol used to execute local transactions. At approx. 500 s runtime, we reach the critical object size after which MySQL outperforms our system.

In a second experiment, we modify the new order transaction such that it only uses constant-size objects (cf. Figure 6.1b). In this setting, our system's throughput remains constant at the initial values over the course of the entire measurement. We achieve 201 % the throughput of MySQL while still providing a linearizable transaction ordering (2PL). If we weaken the consistency model to MAV, our system even achieves 409 % throughput. This is difference is because 2PL requires the execution of more transactions than MAV to keep the growing and shrinking phases separated. In fact, MAV has no significant overhead compared to no synchronization at all in this setting.

This shows that, when operating on constant-sized objects, our system outperforms relational database systems in the local transaction throughput. In the following section, we execute the modified TPC-C new order transaction in a distributed storage setting and analyze the scalability of our transaction coordination scheme 2PO.



(a) Increasing object size

(b) Near-constant object size

Figure 6.1: Local throughput compared to MySQL

6.2 Distributed Scaling

In our distributed configuration, the system utilizes a cluster of five nodes. Among these, three nodes use an Intel Xeon E5-1650 V4 processor, each with 16 GB RAM. Complementing this setup, the remaining two nodes use Intel Xeon E3-1245 V2 processors and 8 GB RAM each. On this cluster, we evaluate our distributed transaction coordination algorithm. We execute the modified TPC-C benchmark's new order transaction (constant object size) with up to 20 storage nodes. In the following measurements, we are only concerned about the scaling of the coordination algorithm. Hence, we disable the objects' persistence by placing them in RAM. In doing so, we only measure the coordination overhead, as transaction execution becomes a negligible overhead.

First, we evaluate scalability under a constant medium contention level. Contention is constant as we keep the number of nodes equal to the number of objects. Figure 6.2 shows the results of this measurement. Our algorithm proves to be near-perfect linearly scalable. To double the performance, we have to approx. triple the number of nodes. Even though the probability for a concurrent object access is high (above 99 % for 4 nodes and above), our algorithm is able to reorder transactions, avoiding most aborts. This results in a high commit rate of more than 99 % across the entire measurement. Under MAV, we observe zero aborts during the measurement, which is expected as the operation cannot create cycles.

In a second experiment, we evaluate the influence of contention on the coordination overhead. For that, we use the maximum of 20 nodes and vary the number of objects in the storage system, measuring global throughput. Object access is randomized over all available objects. Thus, contention decreases with increasing number of objects. Figure 6.3 shows the results of this measurement. With high contention, MAV is mainly constrained by the compute power of the

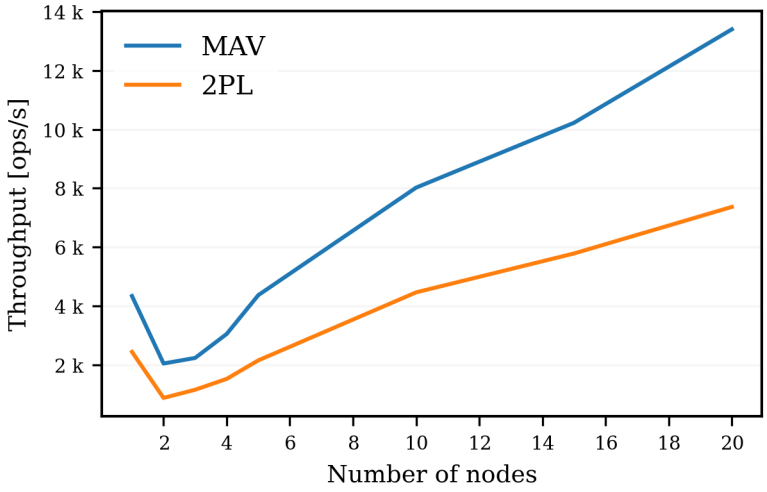


Figure 6.2: Throughput of our system in a distributed setting, with the storage system spread across multiple compute nodes.

executor. Once the load is distributed across three or more servers, the overall throughput remains near-constant and is limited by the clients. For 2PL, we see the coordination overhead under high contention limits the possible throughput. However, the throughput is not limited by a low commit rate, as is common for traditional implementations of either OCC or 2PL [MCZ+14]. Our scheduler avoids aborts as much as possible: Even when all clients access the same object, our commit rate is above 97 %. With a distribution level of 50 % or higher (10 objects), the throughput of our 2PL implementation using 2PO remains near-constant as well.

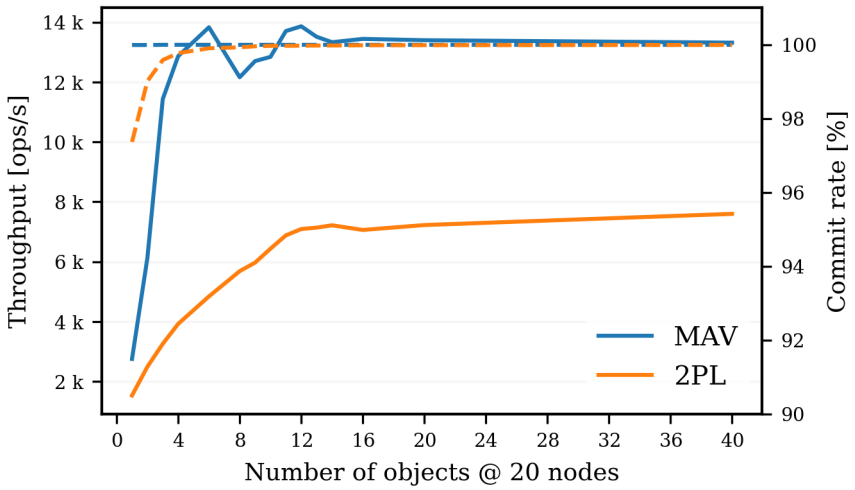


Figure 6.3: Throughput with varying levels of distribution and contention. Dashed lines show the corresponding abort rate. 1 Object means all client operations access the same object. Access to the available objects is random.

In conclusion, we have shown that, under constant contention, throughput with 2PO scales linearly with the number of globally participating nodes. With increasing contention, the global coordination effort increases, decreasing throughput. However, 2PO successfully achieves high commit rates under any contention level. As such, throughput under high contention is not limited by transactions aborting and undoing their changes. Hence, 2PO is capable of a higher overall system throughput than conventional coordination approaches.

7 Conclusion

In this work, we presented a novel distributed transaction coordination algorithm called 2PO. 2PO provides consistency guarantees among multiple clients accessing the same persistent objects concurrently. It accommodates both linearizable consistency as well as arbitrary interleavings of non-transactional interactions at the same time. We exploit application-specific knowledge to describe the causal future of a transaction. Using this knowledge, we move the coordination of transactions to a point in time prior to execution. Moreover, the coordination effort is distributed across all accessing clients. With 2PO, clients deduce a conflict-free transaction schedule before executing any of their transactions. At execution time, we do not need any deadlock-avoidance or conflict detection strategies, as clients are proven to issue conflict-free orderings.

This defines a new way of synchronizing concurrent access to distributed data. We showed that 2PO significantly improves commit rates over traditional transaction coordinators while also proving to be linearly scalable. The underlying coordination concept is not specific to our storage system, and, thus, is applicable to other transactional storage systems (e. g., databases) as well.

For future work, we analyze how 2PO can be extended to support replication, thereby improving its fault tolerance on the executor-side. We believe that knowledge about causally related transactions can also be leveraged to develop novel replication strategies.

Bibliography

[BA08] H.-J. Boehm, S. V. Adve. “Foundations of the C++ Concurrency Memory Model”. In: *SIGPLAN Not.* 43.6 (June 2008), pp. 68–78. ISSN: 0362-1340. DOI: 10.1145/1379022.1375591. URL: https://doi.org/10.1145/1379022.1375591 (cit. on pp. 9, 17).

[BBC+11] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, V. Yushprakh. “Megastore: Providing Scalable, Highly Available Storage for Interactive Services”. In: *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 2011, pp. 223–234. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf (cit. on pp. 9, 17).

[BDF+13] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, I. Stoica. *Highly Available Transactions: Virtues and Limitations (Extended Version)*. 2013. arXiv: 1302.0309 [cs.DB] (cit. on pp. 9, 38, 48).

[BHG+87] P. A. Bernstein, V. Hadzilacos, N. Goodman, et al. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading, 1987 (cit. on pp. 17, 18, 45, 47).

[BMT+19] C. Barthels, I. Müller, K. Taranov, G. Alonso, T. Hoefler. “Strong Consistency is Not Hard to Get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores”. In: *Proc. VLDB Endow.* 12.13 (Sept. 2019), pp. 2325–2338. ISSN: 2150-8097. DOI: 10.14778/3358701.3358702. URL: https://doi.org/10.14778/3358701.3358702 (cit. on p. 47).

[CDE+13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford. “Spanner: Google’s Globally Distributed Database”. In: *ACM Trans. Comput. Syst.* 31.3 (Aug. 2013). ISSN: 0734-2071. DOI: 10.1145/2491245. URL: https://doi.org/10.1145/2491245 (cit. on pp. 9, 17, 47).

[CRS+08] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, R. Yerneni. “PNUTS: Yahoo!’s Hosted Data Serving Platform”. In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. ISSN: 2150-8097. DOI: 10.14778/1454159.1454167. URL: https://doi.org/10.14778/1454159.1454167 (cit. on p. 15).

[DKG18] B. Ding, L. Kot, J. Gehrke. “Improving optimistic concurrency control through transaction batching and operation reordering”. In: *Proceedings of the VLDB Endowment* 12.2 (2018), pp. 169–182 (cit. on p. 47).

[HA90] P. Hutto, M. Ahamad. "Slow memory: weakening consistency to enhance concurrency in distributed shared memories". In: *Proceedings.,10th International Conference on Distributed Computing Systems*. Los Alamitos, CA, USA: IEEE Computer Society, June 1990, pp. 302–309. DOI: 10.1109/ICDCS.1990.89297. URL: https://doi.ieeecomputersociety.org/10.1109/ICDCS.1990.89297 (cit. on p. 15).

[HAMS18] S. Harizopoulos, D. J. Abadi, S. Madden, M. Stonebraker. "OLTP through the Looking Glass, and What We Found There". In: *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*. Association for Computing Machinery and Morgan & Claypool, 2018, pp. 409–439. ISBN: 9781947487192. URL: https://doi.org/10.1145/3226595.3226635 (cit. on p. 47).

[HT94] V. Hadzilacos, S. Toueg. *A modular approach to fault-tolerant broadcasts and related problems*. Tech. rep. Cornell University, 1994 (cit. on p. 11).

[HW90] M. P. Herlihy, J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972 (cit. on p. 48).

[Int] Intel. *libpmemobj-cpp* (cit. on p. 12).

[ISO17] ISO. *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. Fifth. Geneva, Switzerland: International Organization for Standardization, Dec. 2017, p. 1605. URL: https://www.iso.org/standard/68564.html (cit. on p. 15).

[KK03] M. F. Kaashoek, D. R. Karger. "Koorde: A Simple Degree-Optimal Distributed Hash Table". In: *Peer-to-Peer Systems II*. Ed. by M. F. Kaashoek, I. Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 98–107. ISBN: 978-3-540-45172-3 (cit. on p. 39).

[KKS18] A. D. Kshemkalyani, A. Khokhar, M. Shen. "Encoded Vector Clock: Using Primes to Characterize Causality in Distributed Systems". In: *Proceedings of the 19th International Conference on Distributed Computing and Networking*. ICDCN '18. Varanasi, India: Association for Computing Machinery, 2018. ISBN: 9781450363723. DOI: 10.1145/3154273.3154305. URL: https://doi.org/10.1145/3154273.3154305 (cit. on pp. 12, 45, 48).

[KR81] H. T. Kung, J. T. Robinson. "On Optimistic Methods for Concurrency Control". In: *ACM Trans. Database Syst.* 6.2 (June 1981), pp. 213–226. ISSN: 0362-5915. DOI: 10.1145/319566.319567. URL: https://doi.org/10.1145/319566.319567 (cit. on pp. 17, 47).

[Lam78] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM*. Ed. by R. S. Gaines. Vol. 21. 7. July 1978, pp. 558–565 (cit. on pp. 12, 48).

[LLZ+22] J. Li, Y. Lu, Y. Zhang, Q. Wang, Z. Cheng, K. Huang, J. Shu. "SwitchTx: Scalable in-Network Coordination for Distributed Transaction Processing". In: *Proc. VLDB Endow.* 15.11 (July 2022), pp. 2881–2894. ISSN: 2150-8097. DOI: 10.14778/3551793.3551838. URL: https://doi.org/10.14778/3551793.3551838 (cit. on p. 47).

[LMP17] J. Li, E. Michael, D. R. Ports. "Eris: Coordination-free consistent transactions using in-network concurrency control". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 104–120 (cit. on p. 47).

[MAN+14] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, A. El Abbadi. "MaaT: Effective and Scalable Coordination of Distributed Transactions in the Cloud". In: *Proc. VLDB Endow.* 7.5 (Jan. 2014), pp. 329–340. ISSN: 2150-8097. DOI: 10.14778/2732269.2732270. URL: https://doi.org/10.14778/2732269.2732270 (cit. on pp. 9, 17, 47).

[Mat88] F. Mattern. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988 (cit. on pp. 12, 44, 48).

[MCZ+14] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, J. Li. "Extracting More Concurrency from Distributed Transactions". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 479–494 (cit. on pp. 17, 47, 51).

[pme] pmem.io. *https://github.com/pmem/pmdk/*. URL: https://github.com/pmem/pmdk/ (cit. on p. 12).

[SBF+15] D. Schwalb, T. Berning, M. Faust, M. Dreseler, H. Plattner. "nvm malloc: Memory Allocation for NVRAM." In: *Adms Vldb* 15 (2015), pp. 61–72 (cit. on p. 12).

[SML+03] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on networking* 11.1 (2003), pp. 17–32 (cit. on p. 39).

[SW16] K. Siek, P. T. Wojciechowski. "Atomic RMI: A distributed Transactional Memory Framework". In: *International Journal of Parallel Programming*. Vol. 44. June 2016, pp. 598–619 (cit. on p. 40).

[TDW+12] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, D. J. Abadi. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 2012, pp. 1–12 (cit. on pp. 47, 48).

[TGPM17] A. Torres, R. Galante, M. S. Pimenta, A. J. B. Martins. "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design". In: *Information and Software Technology* 82 (2017), pp. 1–18. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.09.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584916301859 (cit. on pp. 9, 12).

[Tra] Transaction Processing Performance Council. *TPC-C Benchmark Revision 5.11.0*. Tech. rep. (cit. on pp. 45, 49).

[VV16] P. Viotti, M. Vukolić. "Consistency in Non-Transactional Distributed Storage Systems". In: *ACM Comput. Surv.* 49.1 (June 2016). ISSN: 0360-0300. DOI: 10.1145/2926965. URL: https://doi.org/10.1145/2926965 (cit. on p. 15).

[XSL+15] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, Y. Wang. "High-Performance ACID via Modular Concurrency Control". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 279–294. ISBN: 9781450338349. DOI: 10.1145/2815400.2815430. URL: https://doi.org/10.1145/2815400.2815430 (cit. on p. 47).

All links were last followed on December 17, 2023.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature