Multi-client Transactions in Distributed Publish/Subscribe Systems

Martin Jergler  
TU München, Germany  
jergler@in.tum.de

Kaiwen Zhang  
École de technologie supérieure, Canada  
kaiwen.zhang@etsmtl.ca

Hans-Arno Jacobsen  
Middleware Systems Research Group  
arno.jacobsen@msrg.org

Abstract

Transactional operation processing among clients is increasingly required of publish/subscribe (pub/sub) systems in enterprise settings. For instance, in workflow management, dispatching or consolidating process instances requires a set of publications and (un-) subscriptions by different clients to be executed according to ACID semantics. As pub/sub systems are usually optimized for performance and scalability, such properties are often neglected, which results in unexpected system behavior such as misrouted events.

In this paper, we provide a model for supporting multi-client transactions in pub/sub. We formalize ACID properties for pub/sub, and define a consistency model and isolation level required to support the aforementioned scenarios. We present three approaches for two transaction types: S-TX, where a coordinator has full static knowledge about all operations in a transaction, and D-TX/D-TXNI, where operations by other clients are dynamic and unknown to the coordinator. We describe algorithms to realize these approaches and experimentally evaluate our implementations by comparing them to a baseline mechanism, which simulates these guarantees partially by introducing manual waits between operations.

Our results show that the uncertainty introduced by the dynamic behavior renders D-TX/D-TXNI costly, and suitable only for small configurations or rare occasions. S-TX, in contrast, offers enriched semantics for many applications in a scalable manner without disrupting regular event routing.

CCS Concepts  • Computing methodologies → Distributed algorithms;  • Networks → Application layer protocols;  • Applied computing → Enterprise information systems;

Keywords  Publish/Subscribe; transactions; workflow management;

ACM Reference format:

1 Introduction

Motivation – The integration of large-scale applications in enterprise systems is still a challenging task. Often, such applications comprise numerous components, reveal plenty of dependencies, and show complex interaction patterns [13]. Moreover, application systems are dynamic and require adaptations or elastic provisioning [11]. Hence, components must be added, removed, or adjusted ad-hoc without disrupting the service. Middleware services, such as message queues and pub/sub, are used in this context as a coordination mechanism for individual components [3, 18]. On one hand, a middleware should support non-functional requirements like scalability and availability, on the other hand, it must express the required degree of coordination—which is often a trade-off.

Large-scale applications often coordinate using a distributed pub/sub middleware service to improve scalability [7]. In this paper, we study the distributed content-based pub/sub system model [12]. An overlay network of brokers forwards subscriptions and publications according to their content. Each broker performs matching and routing functions to disseminate publications and subscriptions to intended recipients.

In this work, we consider workflow management systems (WFMS) as one class of enterprise-grade applications that are frequently realized in a distributed fashion using event-based coordination [9–11, 13]. Some WFMSs, for instance, consist of multiple components for data access and control flow computations [9]. The execution of a workflow requires these components to coordinate according to a protocol that depends on the atomicity and consistent order of a set of operations [10, 13].

 Often, a single workflow instance involves significant communication with its environment and it is not uncommon for a WFMS to handle thousands of instances at a time. To provide an elastically scaling service, industry-strength WFMSs assign individual instances to workflow agents [11]. Each agent is a replica of the WFMS dedicated to handle a subset of instances. A load balancer monitors new instances and dispatches them across available agents for further processing (scale-out). Similarly, the load balancer consolidates instances from multiple replicas if the overall load decreases (scale-in). Again, the dispatching or consolidation procedure requires a set of operations to be executed atomically and consistent with a protocol-specific order. In general, this load balancing mechanism can be considered a design pattern for elasticity in many PaaS cloud services (e.g., using Docker containers [11]).

The above workflow scenarios require the transactional execution of a sequence of operations, which is not possible today using common pub/sub systems. We therefore propose a model for defining pub/sub transactions and a design for supporting them. Challenges – To the best of our knowledge, there exists no definition of ACID semantics in the context of pub/sub. Adapting the ACID properties from databases to pub/sub is challenging because both types of systems fundamentally differ in their interaction paradigm, operation sets, and processing model. Yet, a precise formulation of the ACID properties is crucial to reason about an execution model for pub/sub transactions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference’17, Washington, DC, USA  
© 2017 ACM. 978-1-xxx-xxxx-x/YY/MM... $15.00  
DOI: 10.1145/nmnnnn.nmnnnn
Furthermore, distributed pub/sub systems introduce a high degree of concurrency in managing the state of various brokers. Modelling consistency and isolation is non-trivial since transactions affect the state of each broker differently, depending on how pub/sub operations are routed through the network. Thus, the ACID semantics must take into consideration the fact that pub/sub brokers are not designed to be perfect replicas of one another.

In addition, it is challenging for several pub/sub clients, which are fundamentally decoupled in nature, to be able to express a working order of operations within the context of a single (multi-client) transaction. This is normally not an issue for database systems where transactions are single-client, or involves clients which directly communicate with one another. In some cases, the order of operations of a multi-client transaction is determined in an ad-hoc fashion, without relying on prior knowledge of all involved parties. In other cases, the challenge is to identify viable system assumptions, i.e., which prior knowledge can be assumed for clients in a given scenario. Based on these assumptions, we develop algorithms that provide an efficient and scalable integration of the multi-client transaction model into a distributed pub/sub service.

In this paper, we study the following scenarios stemming from the above industry-inspired application [11]. It can be realized over a pub/sub architecture as depicted in Figure 1(a), where a network of five brokers (B1 - B5) connects all clients involved. The WFMS is realized by two workflow Agents (A1 and A2) and the Environment clients (E1 and E2) invoke new or update existing instances.

**Use case 1: Dispatching** – In this scenario, a Dispatcher (D) dispatches new instances to one of the agents. The sequence diagram in Figure 1(b) depicts the dispatching of an instance x1 for process p1, created by E2, through publication pub(p1, x1, 0). This publication is routed to D, which, in turn, sends an assign publication pub(assign, A1, p1, x1) to A1. Now, A1 subscribes to updates for that process instance (sub(p1, x1)), while, at the same time, D un-subscribes to such events (usub(p1, x1)). The expected system behavior is that now all update events for x1 (e.g., pub(p1, x1, 1)) are only delivered to A1. However, during the dispatching transition, the destination of update events to x1 is not guaranteed to be A1. This can happen because the subscription has not been propagated through the whole overlay, i.e., received by all brokers on the path from A1 to E2. Moreover, duplicate delivery of events can occur when the un-subscription by D has not fully been propagated.

**Use case 2: Consolidation** – Similar to dispatching, workflow instance consolidation in pub/sub can be modeled as depicted in Figure 1(c). In this scenario, a Consolidator client (C) acquires the execution of existing instances from two Agent clients (A1 and A2). First, C creates two consolidate publications (pub(cons, x1) and pub(cons, x2)) that are delivered to A1 and A2, respectively. After receiving the publication, agents unsubscribe to the corresponding instances x1 and x2. Last, C itself subscribes to both instances. Now, the expected system behavior is that all events for x1 and x2 are delivered to C, which, again, might be violated if not all (un)subscriptions are fully processed yet.

In summary, the problem is that in both scenarios all operations need to be executed atomically, in the order imposed by multiple clients, and isolated from any other concurrent operation, e.g., successive publications to the same instance. These requirements can be captured as ACID semantics for pub/sub.

Note that both scenarios describe transitory management operations which occur on a live system. The transactional handling of these operations should not disrupt the flow of regular pub/sub events which are non-transactional in nature.

**Structure** – This paper provides the following contributions:

1. We present a formal model for supporting transactions of pub/sub operations (Section 3). We propose different levels of ACID semantics for expressing multi-client transactions with varying guarantees and requirements with respect to a priori knowledge.
2. We propose D-TX and D-TX_{NII}, our first set of solutions for supporting transactions in the context of a distributed content-based pub/sub system (Sections 4, 5). D-TX_{NII} allows a set of operations to be defined at run-time, provides sequential consistency, and atomicity. In addition to that, D-TX also provides strong isolation (serializability).
3. We propose S-TX, our third distributed solution (Section 6). S-TX relies on static knowledge of all operations included in a transaction, provides weak isolation (application level), and sequential consistency (using conflict-free replicated datatypes), and atomicity.
4. We provide implementations of D-TX, D-TX_{NII}, and S-TX in a distributed pub/sub system, and a comprehensive evaluation that compares the strengths of our solutions with a baseline solution and analyzes event traffic disruption (Section 7).

The paper continues with Section 2, with background information on related concepts and distributed pub/sub. Related works are described in Section 8, after the core sections outlined above.
2 Background

2.1 ACID Semantics in Distributed Databases

A database transaction is defined as a sequence of atomic operations on data objects forming an atomic unit of work. Operations are typically issued by a single client together with a distinct begin and end operation and executed on one or multiple servers, while providing atomicity, consistency, isolation, and durability—also referred to as the ACID properties. Within a database management system, the ACID properties are maintained by a transaction manager containing at least: (1) a recovery mechanism to guarantee atomicity, (2) a scheduler to guarantee consistency, and (3) a concurrency control mechanism to guarantee isolation. [27]

**Distributed atomic commitment** – A distributed transaction involves the access to transactional resources on multiple hosts in a networked environment, e.g., a database that is partitioned or replicated across several servers. Here, the transaction manager synchronizes the enforcement of the ACID properties among all participating database servers.

Core challenges in distributed transactions originate from the atomicity and isolation properties. Both require distributed agreements: atomicity, in order to decide whether to commit or abort the transaction; and isolation, to either agree on the conflict/commit order, or to propagate locks on data objects. A well-established solution to solve this problem is the 2-phase commit protocol (2-PC) [27] combined with commitment ordering [19] to impose an order on transactions. Our D-TX approach adopts these concepts to distributed pub/sub systems.

**Consistent replication** – In addition to the “C” in the ACID properties, which requires a transaction to transition into a new correct system state, consistent replication is another, yet different, challenge in distributed databases. In this context, a consistency model defines a contract between a distributed data store and a set of participating database servers.

2.2 Distributed Publish/Subscribe

The foundation of our approach is the distributed content-based pub/sub model including an advertisement optimization for subscription routing [7, 8] that we formalize next.

**Event space** – The basis of the content-based pub/sub model is a d-dimensional event space $E_d$, where each dimension is representing an attribute $A_i$ with domain $\text{dom}(A_i)$.

$$E_d = A_1 \times A_2 \times A_3 \times \cdots \times A_d$$

The domain $\text{dom}(A_i)$ is predefined and ordered; the lowest and highest values are denoted by $l_i$ and $u_i$, respectively.

A filter $F$ on $E_d$ is defined as a set of predicates $p_1, \ldots, p_d$, such that the $i$th predicate represents an interval of values $[p^l_i, p^u_i]$ for attribute $A_i$ within $\text{dom}(A_i)$, where $p^l_i \geq l_i$ and $p^u_i \leq u_i$. With $F_p$, we refer to a special filter called point filter, if and only if, every predicate $p_1 \in F_p$ represents a single value from $\text{dom}(A)$, i.e., $\forall p_i \in F : p^l_i = p^u_i$. Otherwise, we refer to a filter as a range filter, denoted by $F_r$, if at least a single predicate $p_1 \in F$, defines an interval with more than a single value, i.e., $\exists p_i \in F : p^l_i < p^u_i$.

Two filter relations enable the expression of a matching and a conflict relation among operations: overlap ($\sigma$: Definition 2.1) and subsume ($\supset$: Definition 2.2).

**Definition 2.1. Filter overlap:** The overlap of filters $F_g$ and $F_h$ is defined as a Boolean function $\sigma(F_g, F_h)$ that returns true, if and only if $\exists A_i \in E_d$ such that $F_g(p_i) \cap F_h(p_i) \neq \emptyset$; i.e., if at least the predicates for a single attribute overlap.

**Definition 2.2. Filter subsumption:** The subsumption of a filter $F_h$ by $F_g$ is defined by a Boolean function $\sigma(F_g, F_h)$ that returns true, if and only if $\forall p_i \in F_g, F_h : F_h(p^l_i) \geq F_g(p^l_i) \land F_h(p^u_i) \leq F_g(p^u_i)$; i.e., each predicate in $F_g$ is an interval containing the predicate in the other filter ($F_h$).

**Elementary operations** – Based on the event space formalization $E_d$ and the filter concept $F$, we now formalize the set of elementary pub/sub operations. We assume that an operation is issued by a particular client $c_i$, issuing operation $\text{op} \in O_{PS}$, we write $\text{op}(c_i)(F)$; operations have the following semantics:

- $\text{pub}(F_p)$ – publishes an event that is represented by a point filter $F_p$ on $E_d$, i.e., it describes a single value from the domain of each attribute $A_i \in E_d$.
- $\text{adv}(F)$ – advertises events a client will publish in the future. The set of advertisements, $\mathcal{A}_c$, a client issues forms the client’s publication space: $F_{\text{pub}}^c = \bigcup_{F \in \mathcal{A}_c} F$.
- $\text{sub}(F)$ – subscribes to publications from different clients, $\text{pub}(F_p)$, that match the subscription, i.e., $\sigma(F, F_p)$ returns true. The set of subscriptions, $\mathcal{S}_c$, a client issued forms the clients subscription space: $F_{\text{sub}}^c = \bigcup_{F \in \mathcal{S}_c} F$.

A notification represents the delivery of a publication, $\text{pub}(F_p)$, to an interested client $c$ if $\sigma(F_{\text{pub}}^c, F_p)$ returns true. The set of notifications a client received over time is represented by $\mathcal{N}_c$. 

\[ E_d = A_1 \times A_2 \times A_3 \times \cdots \times A_d \]
We now present our formal transaction model for distributed content-delivered to the various brokers, which communicate only using overlay links. Advertisements, adv\((F_a, F_b)\), are broadcast to all brokers and stored in a Subscription Routing Table (SRT), i.e., a list of \([\text{adv, lastHop}]\)-pairs, where lastHop points to a broker or client that sent the operation. A subscription sub\((F_s)\) is matched against all adv\((F_a)\) ∈ SRT at a broker, where a successful match is defined by \(\sigma(F_s, F_a) = T\). Matching subscriptions are stored in a Publication Routing Table (PRT), which is a list of \([\text{sub, lastHop}]\)-pairs, and forwarded to the lastHops of the matching advertisements. Non-matching subscriptions are buffered until they match a later advertisement and are forwarded to the lastHop points to a broker or client that sent the operation.

We define the ACID properties. We then focus on two specific properties, consistency and isolation, and demonstrate how they can be supported in a distributed pub/sub system with multi-client transactions.

### 3 Transactional Pub/Sub Model

We now present our formal transaction model for distributed content-based pub/sub systems (cf. Section 2.2). We first present our definition of transactions in the context of pub/sub and the ACID properties. We then focus on two specific properties, consistency and isolation, and demonstrate how they can be supported in a distributed pub/sub system with multi-client transactions.

#### 3.1 Definition and Properties of a Transaction

A pub/sub transaction, \(\mathcal{T}_{PS}\), consists of a sequence of elementary pub/sub operations \(\sigma_1, \ldots, \sigma_n\), where \(\sigma_i \in \mathcal{O}_{PS}\). Each operation is a pub/sub operation, which originates from any client in the system. In this way, a transaction can involve multiple clients, with operations originating from different sources. Additionally, each transaction is distributed, since the operations must be applied at various brokers, which communicate only using overlay links.

In Definition 3.1, we define the ACID semantics for a pub/sub transaction \(\mathcal{T}_{PS} = [\sigma_1, \ldots, \sigma_n]\), \(\sigma_i \in \mathcal{O}_{PS}\), executed on a pub/sub system \(\Pi = (\mathcal{B}, \mathcal{C})\), where a set of clients \(\mathcal{C} = \{c_1, \ldots, c_m\}\) is connected to a broker network \(\mathcal{B} = \{b_1, \ldots, b_k\}\).

**Definition 3.1.** ACID semantics in pub/sub. We define the ACID semantics for a transaction \(\mathcal{T}_{PS}\) as follows:

- **atomicity**: either all operations \(\sigma \in \mathcal{T}_{PS}\) are successfully processed by \(\Pi\) or none of them. In particular, the SRTs of brokers are updated with the adv/adv operations, the PRTs are updated with the sub/sub operations, and clients are notified with publications pub \(\in \mathcal{T}_{PS}\).
- **consistency**: \(\mathcal{T}_{PS}\) transitions a correct pub/sub system state \(\Sigma\) into another correct state \(\Sigma'\) (cf. Definition 3.2). A correct state transition by \(\mathcal{T}_{PS}\) is defined through internal and external consistency:
  1. **internal consistency**: every operation \(\sigma \in \mathcal{T}_{PS}\) is executed on a consistent state \(\Sigma\) of \(\Pi\). In particular, the states of SRTs and PRTs are consistent across all brokers \(b \in \mathcal{B}\) while processing \(\sigma\).

1To better understand internal consistency, consider an overlay with two brokers \(b_1\) and \(b_2\). A publication, \(p_i\), first arrives at \(b_1\), where it is processed based on the PRT of \(b_1\), and is then forwarded to \(b_2\). If the PRT of \(b_2\) represents a different state compared to the PRT of \(b_1\), consistency is violated. A different state might be reached because a subscription was concurrently processed by \(b_2\) and modified its PRT.

- **external consistency**: the order in which the operations of \(\mathcal{T}_{PS}\) are processed by \(\Pi\) is prescribed by the order expressed by the application semantics.
- **isolation**: \(\mathcal{T}_{PS}\) only reads and writes to the latest committed state. In particular, any sub/sub only reads the SRTs and any pub only reads the PRTs of brokers constructed by the latest committed transaction (serializability).
- **durability**: a committed transaction, i.e., all routing state changes in \(\mathcal{B}\) and all event notifications in \(\mathcal{C}\) are durable and survive node and network failures.

**Definition 3.2.** Consistent pub/sub system state: A consistent pub/sub system state \(\Sigma\) for \(\Pi = (\mathcal{B}, \mathcal{C})\) is defined as the state that is reached at all brokers \(b_i \in \mathcal{B}\), i.e., SRTs and PRTs, and all clients \(c_i \in \mathcal{C}\), i.e., \(\mathcal{F}_{pub}^c, \mathcal{F}_{sub}^c\), and \(\mathcal{N}_{c}\), after completely applying a single operation \(\sigma \in \mathcal{O}_{PS}\).

Atomicity is taken into consideration in our designs (cf. Section 4 and 6). For durability, we employ existing techniques for tolerating broker failures such as [20], which will not be described in this paper. For consistency, the main challenge is maintaining consistency for multi-client transactions, since pub/sub clients are acting in an asynchronous and decoupled manner. For isolation, the main challenge comes from the distributed nature of the pub/sub system, which can be modeled as a replicated database, where each broker maintains a partial copy of the pub/sub state.

#### 3.2 Multi-Client Consistency

An inherent prerequisite to reason about consistency in multi-client transactions is a definition of a working order between operations from different clients. Therefore, we first introduce a special operation \(tcm(F_p)\) (transaction control message) and add it to the set of elementary pub/sub operations, \(\mathcal{F}_{pub} \cup \mathcal{F}_{sub}\). A \(tcm\) is a special type of publication specified to trigger other operations at the receiving client, e.g., a client receiving a \(tcm\) issues a subscription sub. Our model provides three different mechanisms to express an order between operations, as specified in Definition 3.3.

**Definition 3.3.** Transaction construction: A transaction \(\mathcal{T}_{PS}\) has a coordinator TXC \(\in \mathcal{C}\) and identifier \(txID\). The TXC issues the first operation together with \(txID\); then, the complete operation sequence is constructed by:

1. **TXC operation**: the TXC issues more operations \(\sigma \in \mathcal{O}_{PS}\) to \(\Pi\) using \(txID\).
2. **TCM-triggered operation**: a client that received a \(tcm\) for \(txID\) issues its own operations using \(txID\).
3. **Internally-triggered operation**: a broker that received an adv (or adv) operation matching a buffered subscription forwards the sub (or usub) using \(txID\).

**Consistency**: In our model, a consistent state transfer is defined by internal and external consistency. While internal consistency is already precisely described, external consistency requires a consistency model. Essentially, all operations should be executed according to an application-specific order, i.e., the sequential order, in which clients issued them (cf. Definition 3.4).
Definition 3.4. Sequential consistency: External consistency in TS1 is defined as a sequential order relation among operations, denoted ≤SO, by the following rules:

1. Thread—For any two operations a, b ∈ TS1 issued by the same client: if a happened before b, then a ≤SO b.

2. Trigger—For two operations a, b ∈ TS1: If a triggers b, where b is either an internally-triggered-operation at a broker or a TCM-triggered-operation at a different client, then a ≤SO b.

3. Trigger’—For three operations a, b, and c: If a ≤SO b according to rule Trigger, and if a ≤SO c according to rule Thread, then b ≤SO c. In other words, every triggered operation is ordered prior to any subsequent operation from the same client.

4. Transitivity—For any three pub/sub operations a, b, and c, if a ≤SO b and b ≤SO c, then a ≤SO c.

For two operations a ≤SO b ∈ TS1, the pub/sub system Γ = (B, C) must completely process a before b. In particular, every broker b ∈ B must process a before b.

An example for consistent ordering of operations in a transaction is depicted in Figure 2. In this example, a subscription s1 by client C1 is buffered at B1 before the actual transaction starts. The subscription is buffered due to a lack of matching advertisements at B2. In the transaction, client C1 acts as TXC and issues three operations: first, advertisement a1 = adv(x), followed by TCM t1 = tcm(x), and publication pub(y). According to rule Thread, the TXC operations are ordered a1 ≤SO t1 ≤SO p1. According to rule Trigger, a1 ≤SO s1, i.e., the advertisement a1 attracts the subscription s1, and t1 ≤SO s2, i.e., the TCM triggers subscription s2; and according to rule Trigger’, s1 ≤SO t1 and s2 ≤SO p1, which results in the following processing sequence of operations: a1, s1, t1, s2, p1.

3.3 Distributed Isolation

We define isolation in our pub/sub model by specifying, for two concurrent transactions, which updates made by one transaction should be visible to the other transaction.

In pub/sub, updates refer to changes in the routing state of brokers, i.e., SRT and PRT. (Un-)advertisements update the SRT: Every adv() writes an entry and every advd() removes an entry from the SRT. (Un-)subscriptions read from the SRT and update the PRT: Every sub() writes an entry and every unsub() removes an entry from the PRT. Similarly, publications pub read the PRT.

A major problem of not properly isolating transactions in distributed pub/sub is that publications might either not be delivered, or are delivered to unintended recipients. Consider, for example, a scenario in which two transactions t1 = (sub[c2](x), unsub[c3](x)) and t2 = {pub[c1](x)} concurrently execute. Transaction t1 can be seen as the intent to move a subscription sub(x) from client C3 to client C2, so that any of the following publications is no longer notified to C3 but to C2. In general, there are three ways to schedule the operations of t1 and t2:

1. pub[c1](x), sub[c2](x), unsub[c3](x)
2. sub[c2](x), pub[c1](x), unsub[c3](x)
3. sub[c2](x), unsub[c3](x), pub[c1](x)

Only the first and the third schedule describe the intended system behavior. In the second schedule, pub[c1](x) is routed to both C2 and C3 since {1 σ(sub[c2](x), pub[c1](x)) and σ(unsub[c3](x), pub[c1](x)), i.e., the operations are conflicting w.r.t their filters, and (2) the un-subscription unsub[c3](x) was not processed yet. The PRT state which was read for x in order to process pub[c1](x) was consistent but uncommitted and thus subject to further changes by t1.

To avoid this, every transaction must always read the latest committed state from SRTs and PRTs, defined as serializability. For two transactions t1 and t2, if t1 commits before t2 (denoted t1 < t2), then all operations ∈ t2 must be processed based on state written by t1; in particular, if t1 changed the routing state for a filter F, then t2 must read the updated state.

4 Dynamic Transaction Service

In this section, we describe the design and implementation of our D-TX approach. D-TX supports tree-based broker overlays and dynamic pub/sub transactions, where a TXC initiates the transaction with an arbitrary operation (e.g., a tcm), unaware of operations by other clients (e.g., a sub triggered by the tcm). Atomicity is achieved by adapting the 2-phase commit protocol to distributed pub/sub systems. Isolation is realized through a snapshot isolation algorithm and optimistic concurrency control. For sequential consistency, we propose an acknowledgment-based approach to guarantee the working order of pub/sub operations.

Frequently used acronyms, components, and message formats are summarized and briefly explained in Table 1.

|TxID| transaction identifier, tuple (clientID + TS1Commit| |
|RS| stable routing state of broker, i.e., SRT and PRT| |
|SSID| version of the stable routing state RS at broker| |
|TXContext| context for particular transaction, based on the snapshot of the latest committed transaction (basedSSID)| |
|TXRS| isolated routing state for a particular transaction| |
|txMap| index structure to map txIDs to TXContexts| |
|txDependencies| structure to maintain the transaction commit order| |
|ackMap| map to manage pending acknowledgments| |

Table 1. D-TX notation.

Approach overview – Processing a transaction in D-TX is staged into three phases and invoked by the TXC. The purpose of the initialization phase is two-fold: First, all brokers in the overlay are informed about the new transaction and agree on a common snapshot (i.e., the latest committed transaction) to create the transaction context. Second, a commit order among concurrent transactions is established. After initializing the transaction, the actual pub/sub operations are processed in the operation phase. Sequential operation processing is achieved through a nested acknowledgment mechanism that notifies the TXC once all operations have successfully
been applied. The final termination phase adopts a 2-PC protocol,
tailored to distributed pub/sub systems, to commit the transaction:
First, in a prepare round, all brokers are informed about the com-
mit intent. Then, every broker verifies if all transactions ordered
prior to the committing one have terminated and whether conflicts
exist or not. If conflicts exist, the transaction aborts, otherwise, the
transaction commits (commit round).

4.1 D-TX Client Design

Now, we present the client design of D-TX: We describe the client
interface and the concepts relevant to managing transactions. Every
client can take on of two roles in the context of a transaction: as
the coordinator of the transaction (TXC), or as a participant in the
transaction (TXP).

Client API – D-TX extends the standard pub/sub interface with
a callback handler notify() to deliver incoming publications to an
application (cf. Table 2).

Table 2. Client interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginTX() : String</td>
<td>initialize transaction, returns txID</td>
</tr>
<tr>
<td>commitTX(txID): status</td>
<td>terminate tx: returns committed or aborted</td>
</tr>
<tr>
<td>txTCM(txID, Fp): void</td>
<td>send tcm using txID</td>
</tr>
<tr>
<td>txPub(txID, Fp): void</td>
<td>publish using txID</td>
</tr>
<tr>
<td>txAdv(txID, Fp): void</td>
<td>advertise using txID</td>
</tr>
<tr>
<td>txUnadv(txID, Fp): void</td>
<td>un-advertise using txID</td>
</tr>
<tr>
<td>txSub(txID, Fp): void</td>
<td>subscribe using txID</td>
</tr>
<tr>
<td>txUnsub(txID, Fp): void</td>
<td>un-subscribe using txID</td>
</tr>
<tr>
<td>notify():publication</td>
<td>callback: returns publication received</td>
</tr>
</tbody>
</table>

In addition, the method txTCM() enables the construction of
distributed transactions. Like a publication, txTCM() specifies a
single point from an event space, while the remaining operations
take a range filter as input. Every operation is labeled with the txID
of the associated transaction. beginTX() triggers the initialization
phase through an initMsg and returns a txID. commitTX(txID)
terminates the transaction.

Our use case scenario can be implemented as sketched in Figure
1(b), where operations (1) and (2) are realized with txTCM().

Transaction management – Internally, a client maintains a transac-
tion manager, TXManager, for every transaction it is involved in.
TXManager invokes the initialization and termination protocol and
manages the sequential execution of operations. To this end, it
maintains a data structure operations, which stores all operations
received from client API calls. Every operation is described by a
tuple (seqNo, opID, op, status), where seqNo is a sequence number
for local operations, opID is a globally unique operation identifier,
op is the actual operation, and status describes whether the oper-
a tion has already been processed (acknowledged), is currently being
processed (active), or is buffered due to pending prior operations.

Before any operation is processed, TXManager must complete the
initialization phase, i.e., wait for the corresponding acknowledgment
(initAckMsg). Similarly, in order to terminate a transaction, the
TXManager waits for all operations being processed before in-
voking the termination phase through a prepMsg. Depending on
the outcome, it returns committed or aborted.

4.2 D-TX Broker Design & Protocol

An overview of the broker architecture is depicted in Figure 3.
A broker has a single input queue for incoming messages and a
dedicated output queue for each connected client or broker. The
main component of a broker is the transactional router (TXRouter)
handling protocol messages for all three phases in FIFO order within
a single thread.

TXRouter maintains a stable version of the routing state (RS),
which reflects the SRT and PRT generated by the latest commit-
ted transaction. SSID refers to the version, i.e., the txID of the
transaction that produced RS. On initialization, an isolated base
snapshot, i.e., a copy of RS, is taken to form a new transaction context
(TXContext). This context is used to process all operations of the
transaction using its transactional routing state TXRS; every
operation is either processed by TXRS or updates TXRS. On commit,
TXRS is checked for conflicts with concurrent transactions ordered
prior to the current one; if no conflicts are detected, it is merged
with RS, else, it is discarded and the transaction aborts.

To keep track of transactions and dependencies, txMap provides
access to each TXContext and txDependencies captures the commit
order among concurrent transactions. For each txID, it stores
references to all transactions whose base snapshot is taken from
txID. The order among concurrent transactions in this list is im-
plicitly given through the lexicographical order of their txIDs. In
Figure 3, a transaction tx 1 serves as a base snapshot for tx 2 and tx 3,
where tx 2 < tx 3. A transaction can be in various states: activeTX,txs
indexes all active transactions, i.e., transactions that still process
operations, preparedTXs indexes all transactions that are currently
in the prepare phase, and committedTXs and abortedTXs maintain a
history of committed and aborted transactions, respectively.

Initialization phase – The purpose of this phase is to inform all
brokers about a new transaction, agree on a common base snapshot,
and establish a consistent order among concurrent transactions.
The initialization protocol is shown in Algorithm 1.

A broker receiving an initialization message (initMsg) from the
TXC, determines the base snapshot version for the transaction
(ssID) based on its stable state (SSID) and forwards initMsg(ssID).
Every broker without further neighboring brokers generates a new
transaction context, adds the transaction to the set of active transac-
tions, and responds with an acknowledgment initAckMsg(txID, ssID).
To keep track of whether all neighboring brokers have initialized
the transaction properly, every broker maintains an initAckMap with entries for every broker the initMsg has been forwarded to. After receiving all pending acknowledgments, a broker
initializes the transaction itself and sends an acknowledgment to the
origin of initMsg. Due to the tree structure of the broker
overlay, message propagation is acyclic and the TXC eventually
receives an initAckMsg indicating that all brokers have initialized
Procedure handleInitMsg(txID, ssID) from origin
1. if origin = CLIENT then ssID = SSID
2. oEntry ← createTransactionEntry(txID, ssID, origin)
3. if oEntry ∈ TransactionMap(origin, Entry) then
4. neighbors ← brokerNeighbors, origin
5. if neighbors = ∅ then
6. txCtx ← newTXContext(txID, ssID)
7. txMap.put(txID, txCtx)
8. activeTX.addtxIDID)
9. ackMsg ← createAckMsg(txID, ssID)
10. oEntry.removePendingAck(origin)
11. for all broker ∈ neighbors do
12. iEntry = createInitAckEntry(ssID, origin)
13. txInitMsg = NewTXContext(txID, ssID)
14. iEntry.addPendingAck(origin)
15. forward txInitMsg to broker
16. // wait until all ACKs are received.

Procedure handleTickMsg(ssID, opID) from origin
1. ifEntry ← initAckMap.get(opID)
2. txCtx = newTXContext(txID, ssID)
3. txMap.put(txID, txCtx)
4. activeTX.addtxIDID)
5. if oEntry.getOrigin() ∈ neighbors then
6. tOp.getDestinations() ∈ neighbors
7. for all tDest ∈ oEntry.getDestinations() do
8. tDest = createPendingAckEntry(opID, txID, ssID)
9. txCtx = newTXContext(txID, ssID)
10. txMap.put(txID, txCtx)
11. activeTX.addtxIDID)
12. if oEntry.getDestinations() ∈ neighbors then
13. oEntry.addPendingAck(opID)
14. forward txCtx to broker
15. // check if all ACKs are received.

Procedure handleTickMsg(ssID, opID) from origin
1. ifEntry ← initAckMap.get(opID)
2. txCtx = newTXContext(txID, ssID)
3. txMap.put(txID, txCtx)
4. activeTX.addtxIDID)
5. if oEntry.getOrigin() ∈ neighbors then
6. tOp.getDestinations() ∈ neighbors
7. for all tDest ∈ oEntry.getDestinations() do
8. tDest = createPendingAckEntry(opID, txID, ssID)
9. txCtx = newTXContext(txID, ssID)
10. txMap.put(txID, txCtx)
11. activeTX.addtxIDID)
12. if oEntry.getDestinations() ∈ neighbors then
13. oEntry.addPendingAck(opID)
14. forward txCtx to broker
15. // check if all ACKs are received.

Algorithm 1: Initialization phase in D-TX.

Algorithm 2: Pub/Sub operation processing in D-TX.

The transaction in general, it is possible that during the initialization phase other transactions concurrently commit. However, the protocol is still safe because a base snapshot was selected that has been committed at every broker in the network.

Operation phase – The purpose of this phase is to process all operations of a transaction constructed according to Definition 3.3, i.e., in the order specified by participating clients. The protocol is shown in Algorithm 2.

Handle operation message – The TXManager of a client generates an operation message, opMsg = (txID, opID, op) and sends it to the broker it is connected to. Brokers receiving an opMsg from some broker/client origin, access the corresponding TXContext and generate an entry, oEntry, for op in the ackMap. oEntry keeps track of whether the broker can acknowledge processing of the operation to origin.

First, every broker checks if op triggers further internal operations, tOp, such as buffered subscriptions (Line 6). If yes, another entry, tEntry, with a dependency to op is created, and tOp is added as pending operation to oEntry (Lines 8–11). The broker must then wait for the completion, i.e., acknowledgment, of tOp before acknowledging op itself. The triggered operation is now forwarded to the newly matching destinations and pending ack references are added to tEntry (Lines 12–14).

Second, the forward destinations for the actual operations are determined by matching op with the transaction’s routing state TXRS. Then, op is forwarded and corresponding pending ack references are added to oEntry (Lines 15–18). In addition to forwarding, publications are also buffered in the TXContext of every broker on their path to subscribers. This is necessary to determine potential conflicts to prior-ordered transactions later on.

If op does not trigger any internal operation, nor is forwarded to anyone else (e.g., at an edge broker for a subscription), an acknowledgment message, opAckMsg = (txID, opID), is generated and returned to origin.

Handle acknowledgment message – A broker that receives an ackMsg from a client/broker origin removes the corresponding entry from its ackMap. If all acknowledgments are received, ackMsg is forwarded to the origin of op (Lines 24–28). For all operations, dOp, that depend on op, the corresponding reference is removed from ackMap. If dOp has no more pending acknowledgments or operations, it can also forward an ackMsg (Lines 29–31).

The same mechanism for sequential operation processing is applied at clients. If a client, for instance, receives a tcm and issues a subscription, it waits for an ackMsg for the subscription before acknowledging the tcm. For a TXC, the operation phase is completed once it receives the acknowledgment for its last operation.

Termination phase – This phase is realized in two rounds: In the prepare round, potential conflicts are identified. Based on the outcome, in the commit round, the transaction either commits, or aborts. The algorithm for the prepare round is shown in Algorithm 3.

The TXC generates a prepare message, prepMsg(txID, CXID), and sends it to its edge broker. Similar to the initialization, the prepMsg is flooded and opposite edge brokers respond with acknowledgments, which are collected at intermediary brokers to eventually notify the TXC about the completion of this round and its outcome. First, a broker receiving a prepMsg verifies if all prior-ordered concurrent transactions are terminated; if not, the prepare round is paused until this is the case (Lines 2–3). Next, potential conflicts are identified by comparing, for all buffered publications, the routing behavior based on the transaction’s state, TXRS, with the routing behavior based on the consistent routing state RS. In other words, a conflict occurs if a concurrent transaction ordered prior to the commiting one has changed the routing state matching one of the publications. If a conflict is detected, a status flag in the ackMap entry for the prepare phase is set to FALSE (Line 12); otherwise, it is set to TRUE, i.e., no conflict. Ultimately, this status is reported to the TXC as part of the prepAckMsg.

Once the TXC receives a prepAckMsg, it starts the commit round: If the status flag is set to TRUE, the TXC sends a commit message, commitMsg(txID), else it sends an abort message, abortMsg(txID). Both message types are flooded through the overlay and acknowledged in
Algorithm 3: Prepare phase in D-TX.

```java
1 Procedure handleTXPrepareMsg(txID, origin) from origin
2   if not all prior-ordered transactions terminated then
3       preparedTX.put(txID, origin) // buffer and wait.
4   else
5       handlePreparedTX(txID, origin)

7 Procedure handlePreparedTX(txID, origin)
8   pEntry ← createPrepAckEntry(txID, origin)
9   if detectConflicts(txID) then
10      preparedTX.put(txID, origin)
11      pEntry.setStatus(FALSE)
12     else
13        pEntry.setStatus(TRUE)
14        neighbors ← brokerNeighbors \ ori
15     if neighbors ≠ 0 then
16        ackMsg ← newPrepAckMsg(txID, pEntry.getStatus())
17        send ackMsg to origin
18     else
19        forall broker ∈ neighbors do
20           pEntry.addPendingAck(broker)
21        forward txPrepMsg to broker
22    // wait until all Acks are received.
```

reverse direction. If a broker receives a cmtMsg, it merges TXRS with the stable routing state RS, truncates TXContext, and marks the transaction committed. If a broker receives an abtMsg, it discards TXContext and marks the transaction as aborted. Clients receiving a cmtMsg, deliver all buffered publications to the application, or discard them when receiving an abtMsg.

4.3 D-TX Discussion

D-TX implements the distributed pub/sub transaction model with the strong ACID semantics defined in Section 3. A core concept of the approach is that every transaction operates on a dedicated snapshot, which is consistently taken from the state of a prior committed transaction. Consistency is achieved by ensuring that all operations are executed in the order specified by clients. This is guaranteed through an acknowledgment mechanism enforcing that every operation is completely processed before its subsequent operation. Atomicity and isolation are achieved by imposing a total order on transactions during initialization and an optimistic concurrency control mechanism, which identifies conflicts when the transaction is preparing to commit.

5 Dynamic transactions without isolation

In this section, we sketch our D-TXNI approach. D-TXNI supports the same type of transactions as D-TX, i.e., dynamic transactions in tree-based overlays, where the TXC initiates a transaction unaware of any other TXPs and their operations. Essentially, both approaches are pretty similar: The client interface remains unchanged. Atomicity is achieved by a variant of the 2-PC protocol and consistency is realized by using nested operation acknowledgments. However, in contrast to D-TX, D-TXNI does not provide serializability (i.e., strong isolation). Thus, in the remainder of this section we emphasize on this difference, and describe changes in the broker and protocol design (we continue to use the notation summarized in Table 1).

Approach overview – Processing a transaction in D-TXNI is still staged into three phases: **initialization phase**, **operation phase**, and **termination phase**. Differences mainly occur in the initialization phase and the termination phase. As D-TXNI does not provide serializability, brokers do not maintain an separate transaction context including an isolated copy of the routing state for each transaction. Instead, all transactions operate on a shared routing state RS. Hence, initializing a transaction does neither require brokers to agree on a common snapshot nor on a commit order. The actual operation processing for a transaction works similar to D-TX and provides sequential consistency. Since pub/sub operations from different transactions are executed on RS potential conflicts cannot be resolved. Hence, the termination phase, which implements a 2-PC protocol, does not require the identification of conflicts.

D-TXNI Discussion – D-TXNI is very similar to D-TX, and both client API and broker design do not fundamentally differ. Nonetheless, we decided to implement this approach for two reasons: (1) Some transactional scenarios might not require the strong isolation properties given by D-TX. (2) Implementing D-TXNI allows for a more specific analysis of the transaction costs (cf. Section 7).

6 Static Transaction Service

We now describe the design of our S-TX approach to support static pub/sub transactions in tree-based overlays. In contrast to D-TX, here, the TXC has full knowledge about the transaction. For S-TX, we assume weak isolation: This means if two concurrent transactions do not operate on disjoint event spaces, still, routing states of brokers converge, but publications might be routed to concurrent subscribers. Again, we guarantee atomicity by adapting the 2-PC algorithm to ensure that publications are only delivered to the application when the transaction committed. Sequential consistency is realized by attaching a list of dependencies, referencing prior-ordered operations, to every operation, and evaluating at brokers whether dependencies have already been processed. Frequently used terminology in addition to the one for D-TX (cf. Table 1) is summarized in Table 3. The proof of correctness for S-TX can be found in Appendix A.

<table>
<thead>
<tr>
<th>S-TX notation.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opMsg(txID,opID,op, D)</td>
<td>operation message for a client transaction, contains the set of operations to trig-</td>
</tr>
<tr>
<td>cmtMsg(txID, D)</td>
<td>commit message depending on D</td>
</tr>
<tr>
<td>overlayJoinMsg(clientID)</td>
<td>overlay join message for a client operation message for op ∈ OP depending on prior</td>
</tr>
<tr>
<td>tcn(Fp, txOPs)</td>
<td>TCM operation, contains the set of operations to trig-</td>
</tr>
<tr>
<td>processedOps</td>
<td>maintains all processed operations</td>
</tr>
<tr>
<td>processedOp</td>
<td>maintains dependencies for each operation</td>
</tr>
<tr>
<td>processedMap</td>
<td>references all operations depending on an operation</td>
</tr>
<tr>
<td>opBuffer</td>
<td>maintains operations with pending dependencies</td>
</tr>
<tr>
<td>txOPs</td>
<td>stable routing state of broker, i.e., SRT and PRT</td>
</tr>
<tr>
<td>client routing table</td>
<td>maintains all processed operations</td>
</tr>
<tr>
<td>SRT</td>
<td>stable routing table of broker, i.e., SRT and PRT</td>
</tr>
<tr>
<td>PRT</td>
<td>TXC operation, contains the set of operations to trig-</td>
</tr>
</tbody>
</table>

S-TX overview – Transaction processing in S-TX is realized in two phases: In the **operation phase**, the TXC generates all operations for the transaction and issues them to the system. tcn operations are used to send a subset of these operations to particular TXPs, which are intended to issue them as their own operations. Every operation message contains a list of dependencies. This list describes all preceding operations and can also be statically computed by the TXC. Brokers evaluate whether all dependencies have been processed before processing the actual operation. In the **termination phase**,
6.1 S-TX Client Design

S-TX exposes the same API as D-TX (cf. Table 2) but uses a slightly different message format and transaction management. A txCM message has a payload field, txCMOps, that contains all operations a receiving TXP should issue for this transaction. In addition to txID and opID, every operation also has a payload field, dependencies, with information about prior operations.

Again, our use case can be implemented as sketched in Figure 1(b), where operations (1) is realized as txTCM() containing all other operations and dependencies in txCMOps.

A client still maintains a TXManager for every transaction it is involved in. The purpose of TXManager is to buffer incoming publications until the transaction finally commits, and to manage operations received from API calls that should be sent to the broker network, which includes attaching opID and dependencies. However, compared to D-TX, operations are no longer required to be buffered until the preceding operation has been acknowledged; instead, they are issued directly and the ordering is done at brokers based on the dependencies.

6.2 S-TX Broker Design & Protocol

The core component of the S-TX broker is the transactional router, TXRouter, which processes all protocol messages in FIFO order. In S-TX, TXRouter maintains a single version of the routing state RS representing SRT and PRT. In addition, it maintains a list of all processed operations (processedOps) and a map, opBuffer, buffering operations that are not processed because the corresponding dependencies have not been satisfied. Dependencies are represented using two data structures: The precedeMap maps an operation opID to IDs of all other operations, which opID depends on, i.e., all prior-ordered operations. The succeedMap maps an operation opID to all operations that depend on opID, i.e., all post-ordered operations.

Satisfiable dependencies – Before processing an operation, a broker verifies if all dependencies have been processed by checking processedOps. However, not every dependency is satisfiable at all brokers. (Un-)subscriptions are processed only by brokers on a path between the subscriber and clients with matching advertisements. To determine if a dependency is satisfiable and, thus, whether the operation must wait for the dependency to be resolved, a broker validates if it is on such path; by (1) checking whether the PRT of the broker contains a matching advertisement, and (2), whether the lastHop of this advertisement is different from the lastHop of the (subscription) dependency.

For this reason, dependencies do not only comprise the opID but also information about the issuing client (sendeID) and about its filter predicates (filter)

\[
\text{dependency} = (\text{opID}, \text{sendeID}, \text{filter})
\]

Client Routing Table – In addition, to reason about the origin of a depending operation, TXRouter maintains a client routing table (CRT), which is a list of [clientID, nextHop]-pairs. The CRT contains routing information for all clients connected to the broker network and thereby enables the broker to determine from which neighbor a dependency subscription will arrive.

After connecting to a broker, a new client issues an overlayJoin message. Similar to advertisements, this message is broadcast to all brokers in order to update the CRT.

Operation phase – Algorithm 4 describes the processing of a message opMsg = (txID, opID, op, D) at brokers.

First, the set of satisfiable dependencies for operation opID is calculated (Lines 2, 31–38); satisfiable dependencies that are not processed yet are stored in precedeMap and succeedMap. If not all dependencies are processed, i.e., not all corresponding operations have been processed, opMsg is buffered. Otherwise, the operation first updates the routing state RS and is then forwarded to neighboring brokers or clients according to RS (Lines 14–16).

After processing operation op, it is marked as processed (Line 17). Procedure markProcessed includes removing the corresponding opID from the dependency sets of all post-ordered operations, sID, in its succeedMap. Whenever, an operation sID has no more unprocessed dependencies, i.e., its precedeMap is empty, sID is added to a set of operations that is processed next (Lines 20–28).

Termination phase – The intention of the termination phase is to either commit the transaction and deliver all publications, buffered at clients, to the application, or abort the transaction, discard buffered publications, and undo all changes applied to the routing state through compensating operations. Because S-TX assumes isolation to be managed at the application-level, an abort in S-TX does not occur due to conflicts among concurrent transactions but only due to an explicit command by the TXC.

---

Algorithm 4: Pub/Sub operation processing in S-TX.

```plaintext
1 Procedure handleOpMsg(txID, opID, op, D)
2 \( D_{\text{Sat}} \leftarrow \text{getSatisfiableDependencies}(D) \)
3forall d \in D_{\text{Sat}} do
4    if d.processedOps then
5        precedentMap.put(opID, d)
6        succeedMap.put(sID, op)
7    if precedentMap.get(opID) = \emptyset then
8        process(txID, opID, op)
9    else
10       opBuffer.add(opID, opMsg)
11 Procedure process(opMsg = (txID, opID, op))
12    RX.getMatchingAdvertisements(d.filter)
13    if precedentMap.get(txID) == \emptyset then
14        RS.add(op)
15    else
16        succeedMap.put(d.senderID, op)
17    if precedentMap.get(txID) == \emptyset then
18        process(txID, opID, op)
19 Procedure getSatisfiableDependencies(D)
20 forall d \in D | d \in (\text{sub, unsub}) do
21    precedentMap.put(d.senderID, d)
22    succeedMap.put(d.senderID, d)
23    if precedentMap.get(d.senderID) = \emptyset then
24        process(txID, opID, op)
25    else
26        succeedMap.put(d.senderID, d)
27    if precedentMap.get(d.senderID) = \emptyset then
28        if a lastHop \neq CRT.get(sID) then
29            return D_{\text{Sat}}
30 Procedure markProcessed(sID)
31 forall opID, opMsg = (txID, opID, op)
32    if precedentMap.get(opID) == \emptyset then
33        succeedMap.put(sID, opMsg)
34        if precedentMap.get(opID) == \emptyset then
35            if precedentMap.get(opID) == \emptyset then
36                if precedentMap.get(opID) == \emptyset then
37                    if precedentMap.get(opID) == \emptyset then
38                        return D_{\text{Sat}}
39        return D_{\text{Sat}}
40```

---

It is not possible that a subscription and a matching advertisement arrive at a broker from the same neighbor because subscriptions are routed along the reverse path.
Commit—to commit a transaction, the TXC issues a commit message, \( \text{cmtMsg} = (\text{txID, } D) \), to the system, where \( D \) contains dependencies to all operations of the transaction. \( \text{cmtMsg} \) is broadcast through the system; similar to the regular operation messages, brokers ensure that all dependencies are processed before forwarding \( \text{cmtMsg} \). Clients and edge brokers receiving \( \text{cmtMsg} \) respond with a \( \text{cmtAckMsg} \), which is aggregated at intermediary brokers to notify the TXC about the successful commit.

Abort—to abort a transaction, the TXC generates a sequence of compensating operations. \( C \) and issues them to the system followed by an abort message. For each operation changing the routing state in the transaction, \( C \) contains the inverse operation, e.g., for \( \text{sub}(F) \in T_{DS} \), the operation \( \text{usub}(F) \) is added to \( C \). To distribute the compensation operations to the TXPs that issued the original operation, the same \( \text{tcm} \) constructions are used as in \( T_{DS} \). Compensating operations are issued and processed in the same order as the original operations; their dependency lists contain both references to the original operations and to prior-ordered compensating operations. The abort message, \( \text{abtMsg} = (\text{txID, } D) \), containing dependencies to all prior operations, is broadcast through the system. Clients receiving the \( \text{abtMsg} \) discard buffered publications and respond, just like edge brokers, with an \( \text{abtAckMsg} \). These messages are again aggregated in the broker network to notify the TXC about the successful abort.

### 6.3 S-TX Discussion

S-TX implements a relaxed variant of the pub/sub transaction model presented in Section 3 by only providing weak isolation. In general, the assumption in S-TX is that concurrent transactions operate on disjoint event spaces (application-level isolation), and, hence, no conflict detection is required. However, even if two concurrent transactions operate on overlapping spaces, routing states at brokers will converge, because both SRT and PRT can be considered conflict-free replicated data types (CRDT) [2] as associative and commutative handler function. Subscriptions, which are represented as \( \text{add}(\text{sub}, \text{lastHop}) \), are idempotent and can be processed in any order. Likewise for un-subscriptions, represented as \( \text{add}(\text{usub}, \text{lastHop}) \). Note that the order between a subscription and an unsubscription does matter if they came from the same last hop. However, since both operations came from the same origin, this previous node can explicitly enforce the order between the two operations and thus resolve the conflict.

Also, different to D-TX is that the TXC is assumed to have complete knowledge about the transaction simplifying the implementation of consistency. Operation ordering is realized using dependencies attached to every operation, which can be calculated statically and before the transaction starts. Atomicity is realized similar to D-TX but does not require conflict detection.

### 7 Evaluation

We implemented D-TX, D-TX\(_{NI} \), and S-TX as extension to the PADRES enterprise event bus [7]. PADRES\(^3\) is a content-based pub/sub implementation using a network of brokers to disseminate pub/sub operations as described in Section 2.2. We experimentally evaluated our implementations in an OpenStack cloud infrastructure, where every virtual machine (VM) was equipped with 4GB RAM, 2 vCPUs @ 2GHz, and Ubuntu 14.04.1 LTS. For our experiments, we implemented the instance dispatching scenario described in Section 1, where a single transaction is comprised of 4 clients issuing five operations in total (cf. Figure 1(b)).

We varied broker topologies (cf. Figure 4), number of scenario instantiations (i.e., varying number of clients), and degree of concurrency (i.e., transactions a TXC manages concurrently).

Every broker and every client was deployed on a separate VM and clients were uniformly distributed among brokers. We also implemented a baseline approach BL-WAIT, which does not provide isolation and uses a best-effort attempt to achieve consistent operation ordering using manually introduced delays. It is important to note that BL-WAIT is not a viable, competing solution since it does not support ACID semantics. We evaluated all four approaches w.r.t. latency, i.e., the time it takes to complete a single transaction, and throughput, i.e., how many transactions a system is able to process in a certain time.

**Baseline implementation (BL-WAIT)** – In the instance dispatching scenario (cf. Figure 1), operation ordering is critical to achieve the desired system behavior. In particular the subscription (Op. 3) and the un-subscription (Op. 4) must be completely processed at all brokers before the publication (Op. 5) is routed. D-TX and S-TX manage ordering by an acknowledgment mechanism and dependency checking, respectively. However, BL-WAIT does not provide such mechanisms; if the Environment (E) issues the second publication (Op. 5) right after the first publication (Op. 1), it is likely that it is routed to an unintended subscriber, i.e., the Dispatcher (D), and not an Agent (A). To prevent this and to estimate the cost of ordering, we manually introduce a \( \text{wait} \) time between Op. 1 and Op. 5. To minimize \( \text{wait} \), we approximate its value up to a delta \( \Delta = 50\text{ms} \).

We start by running the scenario with a high value for \( \text{wait} = 2000\text{ms} \); then, we gradually reduce/increase \( \text{wait} \) until we find a value that allows to execute a 1000 transactions correctly (i.e., the publication (Op. 5) is correctly routed) and, in addition, some prior run with a \( \text{wait}_p = \text{wait} – \Delta \) has failed. Once, we fixed \( \text{wait} \), we execute the scenario accordingly and measure latency and throughput. To take accurate measurements we use a timer at client E, which is started before sending Op. 1; when an Agent A receives the second publication (Op. 5), it sends an acknowledgment to E, which stops the timer.

**Base comparison** – First, we compared the base performance of all approaches using an overlay with three brokers.

We used a single instantiation of the scenario, i.e., one Environment client (E) serving as TXC and one Dispatcher client (D), that dispatches workflow instances in an alternating fashion to two Agent clients (A1, A2). All clients are randomly but uniformly assigned and connected to brokers. In every experiment, we executed...
1000 transactions (i.e., dispatched 1000 workflow instances). All transactions were executed sequentially by the TXC. The results obtained for the four approaches are depicted in Figure 5. Since transactions were executed sequentially, i.e., the TXC waited for the completion of one transaction before starting the next, throughput and latency are directly related by throughput \( \times \) latency = 1. As expected, D-TX (570 ms averaged latency) performs worse than BL-WAIT (190 ms); the reason for the difference is the increased message overhead for ordering, which requires acknowledgments for every operation, and for isolation, which requires three broadcast rounds in total (one for initialization and two for termination). D-TX and D-TXNI show similar performance as both approaches require the same amount of messages and snapshotting has a minor impact. More surprising is the fact that S-TX (\( \sim 20 \) txns/s) beats the performance of BL-WAIT (\( \sim 5 \) txns/s). To some extent the speedup (\( \sim 4X \)) can be explained by the imprecision introduced with the wait accuracy through \( \Delta \). Another factor is the reduced transmission time for the second publication (Op. 5), which was buffered in the network in S-TX and not at client E as in BL-WAIT. Most important, however, is that in BL-WAIT, we had to fix parameter wait conservatively so that the desired correct output was achieved in every single transaction. The graph in Figure 5 shows the average latencies: In the experiments for S-TX, we observed that some transactions terminated much faster than the average. Similar, in our parametrization runs for BL-WAIT, we found that for some of the transactions, correct processing was achieved with less wait time. In these cases, wait introduced unnecessary latency, which, however, is important to ensure that all transactions terminate correctly. In contrast for S-TX, the wait time was minimal in every transaction as the publication could be directly processed once the dependencies were fulfilled.

Impact of overlay size on performance – We analyzed the impact of the broker overlay on performance. We used a single instantiation of the instance dispatching scenario (i.e., four clients), executed 1000 transactions sequentially, and compared four different overlay networks (shown in Figure 4). Again the clients have been distributed randomly and uniformly among brokers: For the single broker overlay, all clients were connected to the single broker, for the three-broker overlay, one broker was connected with two clients, and for the remaining overlays some brokers had no direct client connection. Throughout our experiments, we varied the allocation of clients to brokers. The results represent the averages of all experiment runs and are depicted in Figure 6.

Essentially, the base performance relation among all four approaches is confirmed (cf. also Figure 5): BL-WAIT is faster than D-TX and D-TXNI in all overlay settings, and S-TX always beats BL-WAIT. However, with increasing overlay sizes, the performance of all four approaches declines. For instance, the average latency of D-TX for a single broker, \( \sim 410\) ms, is more than doubled with nine brokers, \( \sim 911\) ms, resulting in \( \sim 2.5X \) performance degradation. This decline can also be observed in BL-WAIT (\( \sim 2.6X \)). In S-TX, the degradation is less (\( \sim 1.9X \)). In addition, with increasing overlays, D-TXNI performs better than D-TX, which originates from the reduced state kept at brokers in D-TXNI. In general, the performance drop with increasing overlays can be explained by the increased communication effort resulting from additional hops every message must take. D-TX and D-TXNI are particularly affected for two reasons: First, the acknowledgments must also traverse more hops, and second, the broadcast rounds in the initialization and termination phases require more hops. S-TX, in contrast, requires fewer broadcast messages and no acknowledgments; hence, it is less sensitive to the network size. Thus, the latency of \( \sim 60\) ms with nine brokers is relatively good.

Impact of concurrency on performance – In this experiment, we investigated the impact of concurrency on performance. Concurrency refers to the number of parallel transactions (threads) executed by the TXC. We varied the number of threads between 1, 2, 5, and 10. Again, we used a single instantiation of the scenario (i.e., four clients), fixed the overlay to three brokers, and executed 1000 transactions. Note, that there are no conflicts among transactions, as the TXC dispatches different workflow instances in every transaction, i.e., the event spaces do not overlap.

The results are depicted in Figure 7. Again, S-TX performs better than BL-WAIT, which beats D-TX and D-TXNI. With all four approaches, latencies are increasing as the number of parallel transactions rises at brokers and clients. While BL-WAIT is comparatively stable (latency increases by 50% for 10 concurrent threads compared to sequential execution), the latencies for S-TX (3X) and D-TX/D-TXNI (\( \sim 4X \)) grow faster. However, concurrency also has a positive effect on throughput as brokers spend less time idling: For BL-WAIT throughput increases by \( 7\times \), for S-TX by \( 3.7\times \) and for D-TX by \( 2.3\times \). While in absolute numbers, S-TX reveals the best performance, its concurrency behavior is worse compared to BL-WAIT because of the management overhead for the termination phase. For similar reasons, the concurrency behavior of D-TX is also worse compared to BL-WAIT. In conclusion, concurrency improves the throughput but trades off with increased latency.

Impact of client quantity on performance – In this experiment, we scaled the number of clients that concurrently execute transactions and analyze the performance impact. Using the instance dispatching scenario, we varied the number of concurrent executions from a single instance (i.e., 4 clients) to two, five, and ten instances (i.e., 40 clients). Each instantiation dispatches instances for a different workflow, so the event spaces of individual transactions do not overlap. We used a fixed overlay of three brokers and at every TXC, transactions are sequentially executed. The results are depicted in Figure 8.

Again, S-TX performs better than BL-WAIT and D-TX/D-TXNI for all configurations. In addition, the scaling behavior for S-TX and BL-WAIT is pretty similar: In both approaches, the latencies increase by only 50% but the throughput increases by \( 7\times \) when scaling up to 40 clients. Compared to scaling at a single client (cf.
experiments on concurrency and Figure 7), the two approaches perform better, which reveals that concurrency at clients impacts latency. The scaling behavior of D-TX however, is not as good. While throughput increases up to 20 clients, the performance begins to drop again for 40 clients. The main reason for this is the isolation management of D-TX, which requires agreement on a consistent snapshot during initialization and, more importantly, maintenance of the commit order during termination. Adherence to the commit order may delay some ready-to-commit transactions while waiting for a prior-ordered transaction to terminate. This, however, is not a problem in D-TXNI, which does not provide the isolation property and, thus, scales way better compared to D-TX.

Impact on regular event routing – In practice, pub/sub systems operate largely without the need for transactional behavior by routing publications in a best-effort strategy (e.g., workflow events are routed to agents). Transactions are required only in specific situations, i.e., when scaling in or out. In the following experiment, we measured the impact of transaction processing on the throughput rate of regular publications. The setup consists of a single instance dispatching scenario with one Environment, one Dispatcher, and two Agents connected by three brokers. After every 20k regular workflow events, which are uniformly send to both agents, 4 new instances are dispatched to the agents, and the inbound rate at agents is measured in events/second.

The results are depicted in Figure 9. In general, the rates for all three approaches are rather unstable, which is likely a result of batch processing introduced by multi-threading in PADRES. But there is a clear distinction between D-TX/D-TXNI and S-TX: For the former two approaches a significant drop in the event rate can be observed when dispatching new instances, i.e., using transaction processing (cf. green squares for D-TX and blue circles for D-TXNI in the graph). For S-TX, to the contrary, this drop cannot be precisely identified and the whole course of the rate shows less variance. In conclusion, transactional processing with D-TX/D-TXNI disrupts regular event routing while S-TX stays stable.

Interpretation – We evaluated D-TX, D-TXNI and S-TX by comparing them to a baseline solution BL-WAIT, which we used to obtain an estimate of the performance of the underlying pub/sub system and the overhead of transaction management. Note that BL-WAIT is by no means a competing solution, as the parameter wait must be carefully chosen, depends on the current system, and requires a set of prior configuration rounds, which is impractical in real scenarios. In our experiments, we showed that S-TX efficiently guarantees consistency according to an application specific order without disrupting regular event routing and even beats this baseline in all experiments. Dynamic transaction construction, as provided by D-TXNI and especially D-TX (which guarantees isolation), is costly and practical only for smaller scenarios with infrequent transactional occurrences where the content of transactions cannot be determined prior to their execution.

8 Related Work

Related works are classified in three categories: (1) centralized message queues with transactional support [3, 4, 18], (2) middleware-mediated transactions [14, 15, 24], and (3) transactional messaging in distributed broker architectures [5, 6, 22, 26].

Transactional message queues include proprietary systems like TIBCO’s Enterprise Message Service [4], ActiveMQ [3] based on the Java Message Service (JMS), or RabbitMQ [18] based on the Advanced Message Queuing Protocol (AMQP). All systems rely on point-to-point communication or on a single message broker to deliver messages to subscribers. A transaction defines a context, which is used to group a set of messages that need to be atomically sent and received. This set of operation is issued by a single publisher and buffered. On commit of the transaction, messages are delivered to subscribers; otherwise, a rollback is performed and messages are discarded. For instance, TIBCO [4] employs a subject-based pub/sub model and uses 2-phase commit (2-PC) to atomically publish or consume messages on a set of subjects. There are also some database systems providing a pub/sub interface and similar transactional mechanisms. In Redis [1], for instance, a transaction groups a set of operations and executes it atomically and isolated.

---

Footnote: Some approaches allow clustering like RabbitMQ.
Although, the systems bear similarities to our work, i.e., atomic delivery, they significantly differ as they neither support distributed brokers, nor do transactions encompass a mixture of publications and subscriptions by different clients.

Middleware-mediated transactions integrate message queues and distributed object transactions [15]. X²TS [14] is based on topic-based pub/sub and integrates CORBA’s Object Transaction Service and Notification Service to provide transactional guarantees for multicasting. Similar to message queues, an implicit transaction context is propagated with messages and 2-PC is used for atomic commitment but without compensation. X²TS contains a caching mechanism at the broker to provide different levels of visibility, which enables a recipient to process a message before the transaction is committed and only be notified about an abort afterwards.

D-Spheres focuses on operationally grouping distributed object transactions [24]. It uses point-to-point communication and allows the descriptive specification of producer/consumer dependencies. Atomicity is provided by 2-PC and a compensation mechanism cancels enqueued messages of aborted transactions.

Compared to our work, both above approaches do not support distributed message routing and transactional combinations of publications and subscriptions by different clients.

There are several approaches dealing with transactional guarantees in distributed broker architectures. In the approach by Hill et al. [5], a publisher can request a reply to its publication from subscribers. Receiving subscribers then decide if, and which type of reply they want to send (e.g., acknowledgment or result). Replies are routed on the reverse paths of publications and are presented to the publisher using a reply view. The work is motivated by combining the decoupling and scaling features of pub/sub with request/response requirements of certain application and is similar to our acknowledgment mechanism in D-TX.

The Hermes Transaction Service (HTS) supports transactions in content-based pub/sub [26]. HTS is built on top of Hermes, a topic-based system using rendezvous-based routing in broker network [17]. A transaction demarcates the process of generating one or more events at a publisher, the set of events, and a set of processes that are executed at subscribers on consuming these events. HTS supports compensatable transactions; a transaction service creates a transaction context, delivers events together with the context, and provides atomicity through 2-PC. If the transaction aborts, HTS ensures that the operations performed by subscribers in reaction to receiving an event are compensated.

One system built on top of HTS is TOPS [22]. It features support for distributed transactions, i.e., TOPS allows multiple clients to publish as part of the same transaction.

Although, both works share similarities with our approach, neither HTS nor TOPS support subscriptions or routing state modifications within a transaction. Isolation is also not considered.

An approach to transactional client mobility in content-based pub/sub is presented in [6]. A transaction encompasses the migration of a client from one broker to another to enable dynamic system adaptation. The protocol is based on 3-phase-commit and compensation but fundamentally differs from our approach. The focus lies on transferring the client state and adapting the routing state according to the new edge broker. No general-purpose transaction model is defined and supported.

9 Conclusions

The seamless integration of transactions is a major challenge for large-scale enterprises. In this context, pub/sub has widely been adopted as it offers strong decoupling properties and scalability. A drawback of existing solutions, however, is the limited event delivery guarantees they expose to application developers. We argue that these guarantees can be summarized as transactional properties analogous to database transactions.

To this end, we formalized pub/sub transactions as a sequence of operations that are to be atomically processed by a distributed pub/sub system and isolated from each other. In our model, we allow that publications by one client can trigger further operations by different clients—forming truly distributed transactions. Based on the a priori knowledge a transaction coordinator (TXC) has, we provide three approaches implementing our model: D-TX and D-TXₙᵢ assume no prior knowledge on operations by other transaction participants (TXP) and acknowledgments enable consistent operation ordering; additionally, in D-TX, snapshot isolation enables serializability. S-TX relaxes the above assumptions: Isolation is considered to be managed at the application level and the TXC has full knowledge about the transaction. However, even if concurrent transaction are not perfectly isolated at the application level, S-TX achieves convergence of the routing states at brokers by leveraging their CRDT-like nature. In S-TX, consistency is achieved through dependency checking among operations at brokers. All approaches adopt 2-PC to provide atomicity.

Our experimental evaluation showed that isolation and the uncertainty about operations renders D-TX and D-TXₙᵢ costly and suitable only for smaller configurations or scenarios with infrequent transactional occurrences. In contrast, S-TX introduces no significant overhead, which has been proven by comparing to a baseline pub/sub mimicking the guarantees, and does not disrupt regular event routing.

The applicability of the approaches heavily depends on the requirements and the design of targeted applications. Our process instance dispatching use case, for instance, could be implemented in either way. In the S-TX implementation, we assumed that a target agent for a process instance is known to the Environment. If we drop this assumption, D-TX or D-TXₙᵢ are suitable.

References

A.1 Atomicity

S-TX employs a variation of the 2PC protocol, which has been proven to preserve atomicity [25]. Publication messages contain the final commit message cmtnsg in their dependency list D. Since S-TX assumes prior knowledge of all messages, it is possible to create publications which will reference a future commit message at the moment they are sent. According to Line 7, operations are not processed until their satisfiable dependency list is empty. This is akin to the locking that occurs during the first phase of 2PC. In the termination phase (which is the same as Phase 2 of 2PC), S-TX sends a commit or abort message which “unlocks” the brokers and allows them to publish or discard the publications.

The major difference between our algorithm and 2PC is that no voting occurs during Phase 1. Instead, the TXC decides the outcome of a transaction based on application-level semantics. Thus, our system assumes the crash-recovery model, where each broker is guaranteed to recover if it crashes, and continue with the protocol.

A.2 Durability

Durability is outside the scope of our paper. As stated above, we assume a crash-recovery failure model where the state of each broker can be recovered using fault-tolerant mechanisms covered in other works [20].

A.3 Consistency

Internal consistency is guaranteed by virtue of the correct implementation of pub/sub operations as described in Section 2.

For external consistency, sequential consistency can be guaranteed by inserting in each operation’s dependency list D all operations preceding it in the transaction order. Given two operations op1 and op2, if op1 < op2 (i.e., op1 precedes op2), each broker receives either (1) none of them, (2) only one of them, or (3) both of them.

**Case 1: None** – In this situation, the broker will not process op1 or op2, therefore it cannot violate sequential consistency.

**Case 2: One** – If only op1 is received, it will be processed immediately, since it has no dependency on op2. If only op2 is received, the procedure getSatisfiableDependencies (Lines 32–39) will determine that op1 will never be delivered to this broker. Therefore, processing op2 without receiving op1 will not cause a violation of sequential consistency.

**Case 3: Both** – In this situation, getSatisfiableDependencies will determine that op1 must be processed before op2. If op1 is received first, it will be buffered at Line 10 until op1 is received and processed. Therefore, op1 will always be processed before op2, which respects sequential consistency.

Note that more general external consistency models are feasible with S-TX, since it can express a partial order for operations. In this case, D should include all predecessors of a given operation according to the poset, which is known prior to the execution of the transaction in a static context.

A.4 Isolation

Since there is no snapshotting of data, procedure process (Lines 13–18) operates on a shared, potentially uncommitted, state. To show that the state of each broker converges, we demonstrate that both SRT and PRT can be considered operation-based, conflict-free replicated data types (CRDT) [2] with add() as associative and commutative handler function. As shown in [21], operation-based
CRDTs are proven to achieve convergence even if operations are sent asynchronously.

We now prove separately that SRT and PRT are CRDTs.

SRT – Two operations are possible on this data structure: \( \text{adv}(F) \) and \( \text{uadv}(F) \). Suppose only advertisements are sent for now. Recall that for the publication space for a client is \( F^c_{\text{pub}} = \bigcup_{F_i \in \mathcal{A}} F_i \). The \( \in \) operation defined over filters is commutative, associative, and idempotent, since filters are sets of predicates. According to set algebra, unions of sets respect the aforementioned properties \( [23] \).

Now suppose \( \text{uadv}(F) \) are also sent. They are processed as the \( \setminus \) operation over the set of predicates of a client. Since \( \setminus \) is not a commutative operation, it appears to violate the properties of the CRDT. However, the pub/sub model has an basic assumption: each client \( c \) is only allowed to advertise and unadvertise its own publication space \( F^c_{\text{pub}} \). In other words, a client is not allowed to advertise for other clients, nor can then unadvertise others. Therefore, a client can clearly create a FIFO ordering of advertisements and unadvertisements that affect its own publication space. Since SRT can be considered a collection of disjoint subsets, each for a single client, every broker can process advertisements and unadvertisements in the relative order specified by the client. Note that the ordering of unadvertisements for one client in relation to advertisements from different clients is not important since they operate on disjoint subsets, and are therefore commutative.

PRT – Similarly to the above, the operation \( \text{sub}(F) \) is commutative, associative, and idempotent. \( \text{usub}(F) \) is not commutative to the subscriptions made by the same client: However, each client can specify a FIFO order on all the operations it is sending out which forces each broker to process unsubscriptions in the correct order, therefore avoiding the problem of commutativity.