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 require 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.**

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 in 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-TX$_{NI}$, where operations by other clients are dynamic and unknown to the coordinator. We describe algorithms realizing these approaches and experimentally evaluate them by comparing to a baseline mechanism, which simulates these guarantees partially with manual waits between operations.

Our results show that the uncertainty introduced by the dynamic behavior renders D-TX/D-TX$_{NI}$ 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.

I. 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 [14]. Moreover, application systems are dynamic and require adaptations or elastic provisioning [12]. Hence, components must be added, removed, or adjusted ad-hoc without disrupting the service. Middleware services, like message queues and pub/-sub, are used in this context as a coordination mechanism for individual components [3], [19]. 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 [13]. 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 [10], [12], [14]. Some WFMSs, for instance, consist of multiple components for data access and control flow computation [9]. Workflow execution requires these components to coordinate w.r.t. a protocol that depends on the atomicity and consistent order of a set of operations [10], [14].

Often, a single workflow instance involves significant communication with its environment and, typically, WFMS handle thousands of instances at a time. To provide an elastically scaling service, industry-strength WFMSs assign individual instances to workflow agents [12]. Each agent is a replica of the WFMS to handle a subset of instances and a load balancer dispatches new instances to available agents for further processing (scale-out); similarly, it consolidates existing instances from agents if the overall load decreases (scale-in). Dispatching and consolidation require a set of operations to be executed atomically and consistent with a protocol-specific order. In general, this load balancing scheme represents a design pattern for elasticity in many PaaS cloud services [12].

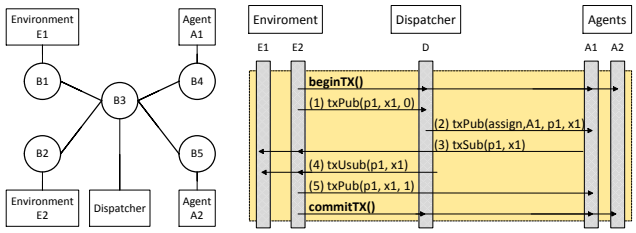
The above scenarios require the transactional execution of a sequence of operations, which is not possible today using common pub/sub systems. We therefore propose a model to define pub/sub transactions and a design to support them.

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 design an execution model for pub/sub transactions.

Also, distributed pub/sub systems introduce a high degree of concurrency in managing the state of various brokers. Modeling consistency and isolation is non-trivial as 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 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 express a working order of operations within the context of a single (multi-client) transaction. This is usually not an issue for database systems where transactions are single-client, or involve clients which directly communicate with each other. In some cases, the order of operations of a multi-client transaction is determined ad-hoc, 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.

In this paper, we study the following use cases stemming from the above industry-inspired application [12]. They can be realized in a pub/sub network connecting all clients involved



(a) Pub/sub overlay. (b) Pub/sub operation sequence.

Fig. 1. Use case 1: instance dispatching in WFMS.

(cf. Figure 1(a)). 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 – Here, 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` 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 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 event delivery can occur when the un-subscription by D has not fully been propagated.

Use case 2: Consolidation – In contrast, instance consolidation refers to a consolidator client acquiring the execution of existing instances from a set of agents (more details in [11]).

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., 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 III). 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$_{\text{NI}}$, our first set of solutions for supporting transactions in the context of a distributed content-based pub/sub system (Sections IV, V). D-TX$_{\text{NI}}$ allows a set of operations to be defined at run-time, provides sequential consistency, and atomicity. In addition, D-TX also provides strong isolation (serializability).
3) We propose S-TX, our third distributed solution (Section VI). 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$_{\text{NI}}$, and S-TX in a distributed pub/sub system, and an evaluation comparing the strengths of our solutions with a baseline and analyzing event traffic disruption (Section VII).

The paper continues with background information about distributed pub/sub (Section II). Related works are described in Section VIII, after the core sections outlined above.

II. Background: 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 $\mathcal{E}_d$, where each dimension is representing an attribute A_i with domain $\texttt{dom}(A_i)$.

$$\mathcal{E}_d = A_1 \times A_2 \times A_3 \times \cdots \times A_d$$

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

A filter F on $\mathcal{E}_d$ is defined as a set of predicates $p_1, \ldots, p_d$, $s.t.$ the i^{th} predicate represents an interval of values $[p_i^l, p_i^u]$ for A_i within $\texttt{dom}(A_i)$, where $p_i^l \geq l_i$ and $p_i^u \leq u_i$. With F_p, we refer to a special filter called *point filter*, if and only if, every predicate $p_i \in F_p$ represents a single value from $\texttt{dom}(A_i)$, i.e., $\forall p_i \in F : p_i^l = p_i^u$. Otherwise, we refer to a filter as a *range filter*, denoted by F_r, if at least a single predicate $p_i \in F_r$ defines an interval with more than a single value, i.e., $\exists p_i \in F : p_i^l < p_i^u$. Two filter relations enable the expression of a matching and a conflict relation among operations: `overlap` (ω: Definition 1) and `subsume` (σ: Definition 2).

Definition 1. ***Filter overlap:*** *The overlap of filters F_g and F_h is defined as a Boolean function $\omega(F_g, F_h)$ that returns true, if and only if $\exists A_i \in \mathcal{E}_d$ 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. ***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 $\forall p_i \in F_g, F_h : F_h(p_i^l) \geq F_g(p_i^l) \wedge F_h(p_i^u) \leq F_g(p_i^u)$; 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 $\mathcal{E}_d$ and the filter concept F, we now formalize the set of elementary pub/sub operations. We assume that an operation is issued by a client $c_i \in C = \{c_1, \ldots, c_n\}$.

$$\mathcal{O}_{PS} \quad = \quad \{\texttt{pub}(F_p), \texttt{adv}(F), \texttt{uadv}(F), \texttt{sub}(F), \texttt{usub}(F)\}$$

where F and F_d represent filters on $\mathcal{E}_d$. To refer to a particular client, c_i, issuing operation $\texttt{op} \in \mathcal{O}_{PS}$, we write $\texttt{op}[c_i](F)$; operations have the following semantics:

- $\texttt{pub}(F_p)$ – publishes an event represented by a point filter F_p on $\mathcal{E}_d$, i.e., a single value for each attribute $A_i \in \mathcal{E}_d$.
- $\texttt{adv}(c)(F)$ – advertises events c will publish in the future. The advertisement set, $\mathcal{A}_c$, forms the client's publication space: $F_{pub}^c = \bigcup_{F_i \in \mathcal{A}} F_i$. Every publication, $\texttt{pub}[c](F_p)$, must be matched by F_{pub}^c, $s.t.$ $\sigma(F_{pub}^c, F_p) = \top$.
- $\texttt{uadv}(F)$ – unadvertises a prior advertisement $\texttt{adv}(F)$ and reduces the publication space $F_{pub}^c = \bigcup_{F_i \in \mathcal{A}} F_i \setminus F$.

- $\texttt{sub}(F)$ – subscribes to events, $\texttt{pub}(F_p)$, that match the subscription, i.e., $\sigma(F, F_p) = \top$. All subscriptions, $\mathcal{S}_c$, form the clients subscription space: $F^c_{sub} = \bigcup_{F_i \in \mathcal{S}} F_i$.
- $\texttt{usub}(F)$ – unsubscribes a prior subscription $\texttt{sub}(F)$ and reduces the subscription space to $F^c_{sub} = \bigcup_{F_i \in \mathcal{S}} F_i \setminus F$.

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

Broker network – A network of brokers, $B = \{b_1, \ldots, b_n\}$ stores, processes, and forwards pub/sub operations. Processing mainly includes matching different operations, which is expressed by the functions overlap $\omega(F_g, F_h)$ and subsumption $\sigma(F_g, F_h)$. Advertisements are broadcast to all brokers and stored in a Subscription Routing Table (SRT), i.e., a list of $[\texttt{adv}, \texttt{lastHop}]$-pairs, where lastHop points to the broker/-client that sent it. A subscription $\texttt{sub}(F_s)$ is matched against all $\texttt{adv}(F_a) \in \texttt{SRT}$ at a broker, where a match is defined by $\omega(F_s, F_a) = \top$. Matching subscriptions are stored in a Publication Routing Table (PRT), i.e., a list of $[\texttt{sub}, \texttt{lastHop}]$-pairs, and forwarded to the lastHops of the matching advertisements. Non-matching subscriptions are buffered until they match a later advertisement and are forwarded (i.e., an advertisement attracted the subscription); a publication $\texttt{pub}(\texttt{F}_\texttt{p})$ is matched against all $\texttt{sub}(F_s) \in \texttt{PRT}$, where a match is defined by $\sigma(F_s, F_p) = \top$. Non-matching publications are dropped; matching publications are forwarded to the lastHops of the matching subscriptions and eventually clients are notified.

III. Transactional Pub/Sub Model

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

A. Definition and Properties of a Transaction

A pub/sub transaction, $\mathcal{T}_{PS}$, consists of a sequence of elementary pub/sub operations $o_1, \ldots, o_n$, where $o_i \in \mathcal{O}_{PS}$. Each operation originates from any client in the system. In this way, a transaction can involve *multiple clients*, with operations originating from different sources. Also, each transaction is *distributed*, since the operations must be applied at various brokers, which communicate only using overlay links.

In Definition 3, we define the ACID semantics for a transaction $\mathcal{T}_{PS} = \{o_1, \ldots, o_n\}$, $o_i \in \mathcal{O}_{PS}$, executed on a pub/sub system $\Pi = (\mathbb{B}, \mathbb{C})$, where a set of clients $\mathbb{C} = \{c_1, \ldots, c_m\}$ is connected to a broker network $\mathbb{B} = \{b_1, \ldots, b_k\}$.

Definition 3. ***ACID semantics in pub/sub****. We define the ACID semantics for a transaction $\mathcal{T}_{PS}$ as follows:*

- *atomicity* – either all operations $\in \mathcal{T}_{PS}$ are successfully processed by Π or none of them. In particular, the SRTs of brokers are updated with the adv/uadv operations, the PRTs are updated with the sub/usub operations, and clients are notified with publications $\texttt{pub} \in \mathcal{T}_{PS}$.
- *consistency* – $\mathcal{T}_{PS}$ transitions a correct pub/sub system state Σ into another correct state Σ^* (cf. Definition 4). A correct state transition by $\mathcal{T}_{PS}$ is defined through *internal* and *external* consistency:
  1) *internal consistency:* every operation $\texttt{op} \in \mathcal{T}_{PS}$ is executed on a consistent state Σ of Π. In particular, the states of SRTs and PRTs are consistent across all brokers $b \in \mathbb{B}$ while processing op[1].
  2) *external consistency:* the order in which the operations of $\mathcal{T}_{PS}$ are processed by Π is prescribed by the order expressed in the application.
- *isolation* – $\mathcal{T}_{PS}$ only reads and writes to the latest committed state. In particular, a sub/usub only reads the SRTs and a pub only reads the PRTs of brokers created by the latest committed transaction (*serializability*).
- *durability* – a committed transaction, i.e., all routing state changes in $\mathbb{B}$ and all event notifications in $\mathbb{C}$ are durable and survive node and network failures.

Definition 4. ***Consistent pub/sub system state:*** *A consistent state Σ for $\Pi = (\mathbb{B}, \mathbb{C})$ is defined as the state reached at all brokers $b_i \in B$, i.e.,* SRTs *and* PRTs*, and all clients $c_i \in \mathbb{C}$, i.e., F^c_{pub}, F^c_{sub}, and $\mathcal{N}_c$, after completely applying a single operation* $\texttt{op} \in \mathcal{O}_{PS}$*.*

Atomicity is taken into consideration in our designs (see Section IV, V, and VI). 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 external 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.

B. Multi-Client Consistency

Important for reasoning about consistency in multi-client transactions is a definition of a working order between operations from different clients. Thus, we first introduce a special operation $\texttt{tcm}(F_p)$ (transaction control message) and add it to the set of elementary pub/sub operations, $\texttt{tcm}(F_p) \in \mathcal{O}_{PS}$. tcm is a special type of publication used to trigger operations at a receiving client, e.g., a client receiving a tcm issues a subscription. Our model provides three different ways to express an order between operations, as specified in Definition 5.

Definition 5. ***Transaction construction:*** *A $\mathcal{T}_{PS}$ has a coordinator TXC $\in \mathbb{C}$ and identifier* txID*. The TXC issues a first operation with* txID*; the complete sequence is constructed by:*

1) *TXC-OP – the TXC issues another operation $\in \mathcal{O}_{PS}$.*
2) *TCM-triggered-OP – a client that received a* tcm *for* txID*, issues an own operation with* txID*.*
3) *Internally-triggered-OP – a broker that received an* adv *(or* uadv*) operation matching a buffered subscription forwards the* sub *(or* usub*) using* txID*.*

[1]To comprehend internal consistency, consider a network with two brokers b_1 and b_2. A publication, p_1, first reaches b_1, is processed based on b_1's PRT, and is forwarded to b_2. If b_2's PRT represents a different state compared to that of b_1 consistency is violated. A different state might be reached if a subscription was concurrently processed by b_2 and modified its PRT.

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., sequential order, in which clients issued them (cf. Definition 6).

Definition 6. ***Sequential consistency:*** *External consistency in $\mathcal{T}_{PS}$ is defined as a sequential order relation among operations, denoted $<^{SO}$, by the following rules:*

1) *Thread – for any two operations* $\mathtt{a}, \mathtt{b} \in \mathcal{T}_{PS}$ *issued by the same client: If* a *happened before* b*, then* $\mathtt{a} <^{SO} \mathtt{b}$.
2) *Trigger – for* $\mathtt{a}, \mathtt{b} \in \mathcal{T}_{PS}$*: If* a *triggers* b*, where* b *is either an internally-triggered-OP at a broker or a TCM-triggered-OP at a different client, then* $\mathtt{a} <^{SO} \mathtt{b}$.
3) *Trigger* – for* a, b, $\mathtt{c} \in \mathcal{T}_{PS}$*: If* $\mathtt{a} <^{SO} \mathtt{b}$ *according to rule Trigger, and if* $\mathtt{a} <^{SO} \mathtt{c}$ *according to rule Thread, then* $\mathtt{b} <^{SO} \mathtt{c}$*. I.e., 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* $\mathtt{a} <^{SO} \mathtt{b}$ *and* $\mathtt{b} <^{SO} \mathtt{c}$*, then* $\mathtt{a} <^{SO} \mathtt{c}$.

For two operations $\mathtt{a} <^{SO} \mathtt{b} \in \mathcal{T}_{PS}$, the pub/sub system $\Gamma = (\mathbb{B}, \mathbb{C})$ must completely process a before b. In particular, every broker $b \in \mathbb{B}$ must process a before b.

An example for consistent ordering of operations in a transaction is depicted in Figure 2. A subscription s_1 by client C_2 is buffered at B_2 before the transaction starts (due to a lack of matching advertisements at B_2). In the transaction, C_1 acts as TXC and issues three operations: First, $a_1 = \mathtt{adv(x)}$, followed by $t_1 = \mathtt{tcm(x)}$, and, last, $\mathtt{pub(y)}$. According to rule *Thread*, the TXC operations are ordered $a_1 <^{SO} t_1 <^{SO} p_1$. According to rule *Trigger*, $a_1 <^{SO} s_1$, i.e., advertisement a_1 attracts subscription s_1, and $t_1 <^{SO} s_2$, i.e., the TCM triggers subscription s_2; and according to *Trigger**, $s_1 <^{SO} t_1$ and $s_2 <^{SO} p_1$, which results in the following sequence of operations: a_1, s_1, t_1, s_2, p_1.

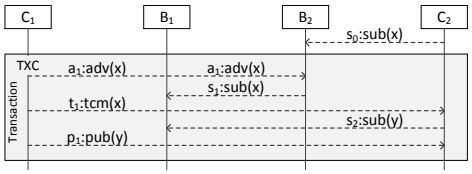


Fig. 2. Example for sequential consistency.

C. 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 and every uadv() removes an entry from the SRT. (Un-)subscriptions read from the SRT and update the PRT: every sub() writes an entry and every usub() removes an entry from the PRT. 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 be delivered to unintended recipients. Consider, for example, a scenario in which two transactions $t_1 = \{\mathtt{sub}[c_2](x), \mathtt{usub}[c_3](x)\}$ and $t_2 = \{\mathtt{pub[c_1](x)}\}$ concurrently execute. Transaction t_1 can be seen as the intent to move a subscription $\mathtt{sub}(x)$ from client c_3 to client c_2, so that any of the following publications is no longer notified to c_3 but to c_2. In general, there are three ways to schedule the operations of t_1 and t_2:

1. $\mathtt{pub}[c_1](x), \mathtt{sub}[c_2](x), \mathtt{usub}[c_3](x)$
2. $\mathtt{sub}[c_2](x), \mathtt{pub}[c_1](x), \mathtt{usub}[c_3](x)$
3. $\mathtt{sub}[c_2](x), \mathtt{usub}[c_3](x), \mathtt{pub}[c_1](x)$

Only the 1^{st} and 3^{rd} schedule describe the intended behavior. In the 2^{nd} schedule, $\mathtt{pub}[c_1](x)$ is routed to both c_2 and c_3 since (1) $\sigma(\mathtt{sub}[c_2](x), \mathtt{pub}[c_1](x)) = \top$ and $\sigma(\mathtt{usub}[c_3](x), \mathtt{pub}[c_1](x)) = \top$, i.e., the operations are conflicting w.r.t their filters, and (2) the un-subscription $\mathtt{usub}[c_3](x)$ was not processed yet. The PRT state read for x to process $\mathtt{pub}[c_1](x)$ was consistent but uncommitted and thus subject to further changes by t_1.

To avoid this, every transaction must always read the latest committed state from SRTs and PRTs, defined as *serializability*. For two transactions t_1 and t_2, if t_1 commits before t_2 (denoted $t_1 < t_2$), then all operations $\in t_2$ must be processed based on state written by t_1; in particular, if t_1 changed the routing state for a filter F, then t_2 must read the updated state.

IV. Dynamic Transaction Service

In this section, we describe the D-TX approach to support dynamic pub/sub transactions in tree-based overlays; the TXC initiates a transaction but is unaware of operations by other clients (i.e., which operation is triggered by its tcms). Atomicity is achieved by adapting the 2-PC protocol and isolation is realized by a snapshot isolation algorithm. For sequential consistency, we propose an acknowledgment-based approach to guarantee the working order of operations.

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: (1) all brokers agree on a common snapshot to create the transaction context, and (2) a commit order among concurrent transactions is established. Next, in the *operation phase*, acknowledgments ensure sequential processing of the actual pub/sub operations. Finally, the *termination phase* adopts a 2-PC protocol, tailored to distributed pub/sub systems, to identify conflicts and to commit the transaction.

A. *D-TX Client Design*

Client API – D-TX extends the standard pub/sub interface with *txTCM()*, a special publication enabling a receiving client to participate in the transaction by issuing any operation(s). In addition, beginTX() and commitTX(txID) are added to initialize and terminate a transaction, respectively.

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

Transaction management – For every transaction a client maintains a transaction manager, TXManager, to control the initialization and termination protocol, and manage the sequential execution of operations. Before any operation is processed, the TXManager must complete the *initialization phase*. Similarly, in order to terminate a transaction, the TXManager waits for all operations being acknowledged.

Algorithm 1: Initialization phase in D-TX.

```
1  Procedure handleInitMsg(txID, ssID) from origin
2      if origin = CLIENT then ssID = SSID
3      iEntry ← createInitAckEntry(ssID, origin)
4      initAckMap.put(txID, iEntry)
5      neighbors = brokerNeighbors \ origin
6      if neighbors = ∅ then
7          txCtx ← newTXContext(txID, ssID)
8          txMap.put(txID, txCtx)
9          ackMsg ← createInitAckMsg(txID, ssID)
10         send ackMsg to origin
11     else
12         forall broker ∈ neighbors do
13             iEntry.addPendingAck(broker)
14             forward txInitMsg to broker
15 Procedure handleInitAckMsg(txID, ssID) from origin
16     iEntry ← initAckMap.get(txID)
17     iEntry.removePendingAck(origin)
18     if iEntry.getPendingAcks() = ∅ then
19         txCtx ← newTXContext(txID, ssID)
20         txMap.put(txID, txCtx)
21         forward initAckMsg to iEntry.getOrigin()
22     else
           // wait until all ACKs are receved.
```

B. D-TX Broker Design & Protocol

The architecture of the D-TX broker is depicted in Figure 3. Its main component is the transactional router (`TXRouter`), which handles messages for all three phases in FIFO order.

`TXRouter` maintains a stable version of the routing state (`RS`) generated by the latest committed transaction. On initialization, an isolated snapshot, i.e., a copy of `RS`, is taken to form a new context (`TXContext` with transaction state `TXRS`) to process all operations of the transaction. On commit, `TXRS` is checked for conflicts with concurrent prior-ordered transactions; if no conflicts are detected, it is merged with `RS`, else, it is discarded and the transaction aborts.

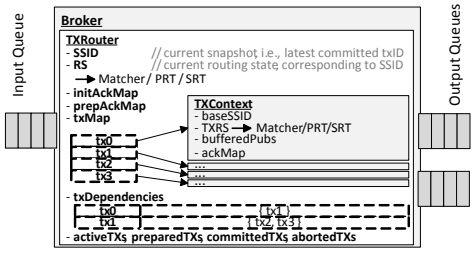


Fig. 3. D-TX: Internal broker architecture.

To keep track of transaction dependencies, `txMap` provides access to each `TXContext` and `txDependencies` captures the commit order among concurrent transactions. For each `txID`, it stores a list with all transactions whose base snapshot is taken from `txID`. Concurrent transactions in this list are ordered lexicographically by their `txIDs` (e.g., $tx2 < tx3$).

Initialization phase – For initialization (cf.Algorithm 1), the first broker receiving an `initMsg` from the TXC, determines the base snapshot version for the transaction (`ssID`) based on its own state (`SSID`) and forwards `initMsg(ssID)`. Edge brokers generate a new transaction context based on `ssID` and respond with an `initAckMsg(txID,ssID)`. Intermediary brokers ensure that all downstream brokers have initialized before initializing themselves. Due to the tree structure of the overlay, message propagation is acyclic and the TXC eventually receives an `initAckMsg` indicating that all brokers have initialized the transaction.

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

```
1  Procedure handleOpMsg(txID, opID, op) from origin
2      ctx ← txMap.get(txID)
3      ctx.add(opID, op)                          // add op to TXRS
4      oEntry ← createOpAckEntry(opID, origin)
5      ctx.getAckMap.put(opID, oEntry)
6      triggeredOps ← ctx.getMatchingBufferedOps(op)
7      forall tOp ∈ triggeredOps do
8          tEntry ← createOpAckEntry(tOp.opID, tOp.origin)
9          ctx.getAckMap.put(tOp.opID, tEntry)
10         oEntry.addPendingOp(tOp.opID)
11         tEntry.addDependency(opID)
12         forall tDest ∈ tOp.getDestinations() do
13             tEntry.addPendingAck(tDest)
14             forward tOp to tDest
15     oDestinations ← ctx.getRoutingMatches(op)
16     forall oDest ∈ oDestinations() do
17         oEntry.addPendingAck(oDest)
18         forward op to oDest
19     if oDestinations ≠ ∅ ∧ triggeredOps ≠ ∅ then
20         send ackMsg(txID, opID) to origin
21 Procedure handleOpAckMsg(txID, opID) from origin
22     ctx ← txMap.get(txID)
23     oEntry ← ctx.getAckMap.get(opID)
24     oEntry.removePendingAck(origin)
25     if oEntry.getPendingAck = ∅ then
26         send ackMsg(txID, opID) to oEntry.getOrigin()
27         forall dOpID ∈ oEntry.getDependencies() do
28             dEntry ← ctx.getAckMap.get(dopID)
29             dEntry.removePendingOp(opID)
                   // check if dOp is ack'ed, forward ack.
```

Operation phase – The protocol for operation processing in D-TX according to Definition 5 is shown in Algorithm 2.

Handle operation message: When a broker receives a pub/-sub operation, `opMsg=(txID,opID,op)`, it first generates an entry in the acknowledgment data structure (`ackMap`). Then, it checks if the operation triggers any internal operations, `tOp`, e.g, subscriptions (Line 6). Such operations are prioritized; the processing of `op` is delayed and a dependency is added to `op`. If no internal operations are triggered, `op` is forwarded based on the matching result (Lines 15-18). Publications are also buffered in `TXContext`, to enable conflict detection during termination. If `op` neither triggers an internal operation, nor can be forwarded, an acknowledgment message, `opAckMsg=(txID, opID)`, is returned to `origin`.

Handle acknowledgment message: When a broker receives an `ackMsg` it removes the corresponding entry from `ackMap`. If all acknowledgments are received, `ackMsg` is forwarded to the origin of `op` (Lines 23–26). 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 27–29).

The same mechanism for sequential operation processing is applied at clients. If a client, for instance, receives a `tcm` and issues a `sub` in turn, it waits for an `ackMsg` for the subscription before acknowledging the `tcm`. The operation phase is complete once the TXC's last operation got acknowledged.

Termination phase – This phase is comprised of two rounds: In the *prepare* round, conflicts are identified, and 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)`, which is flooded through the overlay. Opposite edge brokers respond with acknowledgments, which are collected at brokers to eventually notify the TXC about the outcome of this round.

A broker that receives a `prepMsg` verifies if all prior-

Algorithm 3: Prepare phase in D-TX.

```
1  Procedure handleTXPrepareMsg(txID) 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
8  Procedure handlePreparedTX(txID, origin)
9      pEntry ← createPrepAckEntry(txID, origin)
10     prepAckMap.put(txID, pEntry)
11     if detectConflictsWithPriorTXns(bufferedPubs) ≠ ∅ then
12         pEntry.setStatus(FALSE)
13     else
14         pEntry.setStatus(TRUE)
15     neighbors ← brokerNeighbors \ ori
16     if neighbors = ∅ then
17         ackMsg ← newPrepAckMsg(txID, pEntry.getStatus())
18         send ackMsg to origin
19     else
20         forall broker ∈ neighbors do
21             pEntry.addPendingAck(broker)
22             forward txPrepMsg to broker
24
25 Procedure handleTXPrepareAckMsg(txID, status) from origin
26     pEntry ← prepAckMap.get(txID)
27     if status = FALSE then pEntry.setStatus(FALSE)
28     pEntry.removePendingAck(origin)
29     if pEntry.getPendingAcks() = ∅ then
30         ackMsg ← newPrepAckMsg(txID, pEntry.getStatus())
31         send ackMsg to pEntry.getOrigin()
32     else
           // wait until all ACKs are receved.
```

ordered concurrent transactions are terminated; if not, the *prepare* round is paused (Lines 2–3). Next, conflicts are identified: For all buffered publications, the routing behavior under TXRS is compared with the behavior under the stable state RS. The corresponding *status* is set (Lines 12–14) and ultimately reported to the TXC as part of the prepAckMsg.

A TXC receiving a prepAckMsg starts the *commit* round: If the *status* flag is set to TRUE, the TXC sends commit, cmtMsg(txID), else it sends abort, abtMsg(txID); both message types are flooded through the overlay. If a broker receives a cmtMsg, it commits the transaction by merging TXRS with the stable routing state RS. If a broker receives an abtMsg, it aborts the transaction by discarding TXContext. Clients deliver all buffered publications to the application (cmtMsg), or discard them (abtMsg).

V. Dynamic transactions without isolation

In this section, we briefly sketch our D-TX$_{\text{NI}}$ approach. Just like D-TX, D-TX$_{\text{NI}}$ supports dynamic transactions in tree-based overlays, where a TXC initiates a transaction unaware of any other TXPs and their operations; also, the client interface remains unchanged. Again, atomicity is achieved by a 2-PC protocol and sequential operation processing is realized using nested acknowledgments. In contrast to D-TX, however, D-TX$_{\text{NI}}$ does not provide serializability (i.e., strong isolation). Thus, in the remainder of this section, we emphasize on this difference and describe changes in protocol design.

Approach overview – Processing a transaction in D-TX$_{\text{NI}}$ is still staged into three phases: *initialization phase*, *operation phase*, and *termination phase*. Differences mainly occur in the initialization and termination phase. As D-TX$_{\text{NI}}$ does not provide serializability, brokers do not maintain a 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-TX$_{\text{NI}}$ discussion – D-TX$_{\text{NI}}$ is very similar to D-TX and the broker designs do not fundamentally differ. We decided to implement this approach for two reasons: (1) Some transactional scenarios might not require the strong isolation properties given by D-TX because isolation is naturally provided by the application. (2) Implementing D-TX$_{\text{NI}}$ allows for a more specific analysis of the transaction costs (cf. Section VII).

VI. Static Transaction Service

Our S-TX approach supports static pub/sub transactions in tree-based overlays, where the TXC has full a priori knowledge about the transaction. For S-TX, we assume weak isolation: 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. The proof of correctness for S-TX can be found in [11].

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. tcm operations are used to send a subset of these operations to 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 current operation. In the *termination* phase, the transaction either commits, i.e., all publications are delivered to the application at TXPs, or aborts, i.e., compensating operations are sent and publications are discarded.

A. *S-TX Client Design*

S-TX exposes the same API as D-TX but uses a slightly different message format. A tcm has a payload field, tcmOps, that contains all operations a receiving TXP should issue. In addition to txID and opID, every operation also has a field dependencies with information about prior operations.

Again, our use case can be implemented as sketched in Figure 1(b), where operations (1, 2) are realized as *txTCM()* containing all other operations and dependencies in tcmOps.

For each transaction, a client has a TXManager to buffer publications until commit and to manage API calls. Unlike in D-TX, operations are not buffered until a preceding operation has been acknowledged; instead, they are issued directly and the ordering is done at brokers based on the dependencies.

B. S-TX Broker Design & Protocol

The core component of the S-TX broker is `TXRouter`, which processes all protocol messages and maintains a single version of the routing state `RS` (i.e., `SRT` and `PRT`). Also, it maintains a list of processed operations (`processedOps`) and a map, `opBuffer`, buffering operations that are not processed yet because some dependencies have not been satisfied. Dependencies are represented with two data structures: The `precedeMap` maps an operation `opID` to all other operations, which `opID` depends on, i.e., prior-ordered operations. The `succeedMap` maps an operation `opID` to all operations that depend on `opID`, i.e., 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, a broker validates if it is on such a path by (1) checking whether the `SRT` of the broker contains a matching advertisement, and (2), whether the `lastHop` of this advertisement is different from the `lastHop` of the (subscription) dependency[2]. Thus, dependencies do not only comprise the `opID` but also information about the issuing client (`senderID`) and about its filter predicates (`filter`)

$$\texttt{dependency} = (\texttt{opID},\ \texttt{senderID},\ \texttt{filter})$$

Client Routing Table – To identify the origin of a depending operation, `TXRouter` maintains a *Client Routing Table* (`CRT`), i.e., a list of [`clientID`, `nextHop`]-pairs. The `CRT` contains routing information for all clients connected to the overlay and enables the broker to determine from which neighbor a dependency subscription will arrive.

After connecting, a new client issues an `overlayJoin` message, which is broadcast to update the `CRT`s of all brokers.

Operation phase – Algorithm 4 describes the processing of a message `opMsg = (txID, opID, op, `$\mathcal{D}$`)` at brokers.

First, the set of *satisfiable* dependencies for operation `opID` is calculated (Lines 2, 32–39); satisfiable dependencies that are not processed yet are stored in `precedeMap` and `succeedMap`. If not all dependencies, i.e., 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 (Lines 14–17). After processing `op`, it is marked (Line 18); procedure `markProcessed` removes the corresponding `opID` from the dependency sets of all post-ordered operations (`sID`) in its `succeedMap`. When an operation `sID` has no more dependencies, i.e., its `precedeMap` is empty, it is added to a set of operations that is processed next (Lines 21–29).

Termination phase – In the *termination* phase, a transaction is either committed and all publications buffered at clients are delivered to the app, or aborted and publications are discarded and all changes applied to the `RS` are compensated. An abort in S-TX only occurs due to an explicit command by the TXC and not due to conflicts among concurrent transactions.

[2]It is impossible that a subscription and a matching advertisement arrive at a broker from the same neighbor as subscriptions are routed the reverse path.

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

```
1  Procedure handleOpMsg(txID, opID, op, D)
2      D_s ← getSatisfiableDependencies(D)
3      forall d ∈ D_s do
4          if d ∉ processedOps then
5              precedeMap.put(opID, d)
6              succeedMap.put(d, opID)
7      if precedeMap.get(opID) = ∅ then
8          process(txID, opID, op)
9      else
10         opBuffer.add(opID, opMsg)
12
13 Procedure process(opMsg = (txID, opID, op))
14     RS.add(op)          // (u)adv/(u)sub update routing state
15     destinations ← RS.getRoutingMatches(op)
16     forall dest ∈ destinations do
17         forward op to dest
18     markProcessed(opID)
20
21 Procedure markProcessed(opID)
22     processedOps.add(opID)
23     next ← {}
24     forall sID ∈ succeedMap do
25         precedeMap.get(sID).remove(opID)
26         if precedeMap.get(sID) = ∅ then
27             next ∪ sID
28     forall nID ∈ next do
29         process(nID, opBuffer(nID))
31
32 Procedure getSatisfiableDependencies(D)
33     D_s ← {}
34     forall d ∈ D | d ∈ {sub, usub} do
35         mAdvs ← RS.getMatchingAdvertisements(d.filter)
36         forall a ∈ mAdvs do
37             if a.lastHop = CRT.get(d.senderID) then
38                 D_s ∪ d
39     return D_s
```

Commit: To commit a transaction, the TXC sends commit, `cmtMsg = (txID,` $\mathcal{D}$`)`, to the system, where $\mathcal{D}$ contains dependencies to all operations of the transaction. `cmtMsg` is broadcast through the system; similar to the operation messages, brokers ensure that all dependencies are processed before forwarding `cmtMsg`. Clients and edge brokers receiving `cmtMsg` respond with a `cmtAckMsg`, which is aggregated in the overlay to notify the TXC about the successful commit.

Abort: To abort, the TXC generates a sequence of compensating operations, $\mathcal{C}$, and issues them to the system followed by an abort message. For each operation changing the routing state in the transaction, $\mathcal{C}$ contains the inverse operation, e.g., for `sub`($\mathcal{F}$), the operation `usub`($\mathcal{F}$) is added to $\mathcal{C}$. To issue compensation operations, the same `tcm` constructions are used as in $\mathcal{T}_{PS}$ and compensating operations are processed in the same order as the original operations. Their dependency lists contain references to the original and to prior compensating operations. The abort message, `abtMsg = (txID,` $\mathcal{D}$`)`, which contains dependencies to all prior operations, is broadcast through the system. Clients receiving texttttabtMsg discard publications and respond with an `abtAckMsg`, which ultimately notify the TXC about the successful abort.

C. S-TX Discussion

S-TX implements a relaxed variant of the model in Section III by only providing weak isolation. In general, the assumption 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, since both `SRT` and `PRT` can

be considered conflict-free replicated data types (CRDT) [2] with *add()* as associative and commutative handler function. Subscriptions, represented as `add(sub, lastHop)`, are idempotent and can be processed in any order. Likewise un-subscriptions, represented as `add(usub, lastHop)`. Note that the order between a subscription and an unsubscription does matter if they came from the same last hop. However, as this is true, a previous node can explicitly enforce the order between both operations and thus resolve the conflict.

In S-TX, the TXC is assumed to have complete knowledge about the transaction. This simplifies the implementation of consistency. Operation ordering is realized using dependencies attached to every operation, which can be calculated statically before the transaction starts.

VII. Evaluation

We implemented D-TX, D-TX$_{NI}$, and S-TX as extension to the PADRES enterprise event bus [7], a content-based pub/sub system using a broker network to disseminate pub/sub operations as described in Section II. We experimentally evaluated our implementations in an OpenStack 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 I, where a single transaction is comprised of five operations by four clients (cf. Figure 1(b)).

We varied broker topologies (cf. Figure 4), number of scenario instantiations (i.e., 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., time to complete a single transaction, and *throughput*, i.e., how many transactions a system can 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. 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 with acknowledgments 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 *wait* time between Op. 1 and Op. 5. To minimize *wait*, we approximate its value up to a delta $\Delta = 50ms$. We start by running the scenario with a high value for $wait = 2000ms$; then, we gradually reduce/increase *wait* until we can execute a 1000 transactions correctly (i.e., Op. 5 is correctly routed) and, in addition, some prior run with a $wait_p = wait - \Delta$ has failed. Once, we fixed *wait*, we execute the scenario and measure latency and throughput.

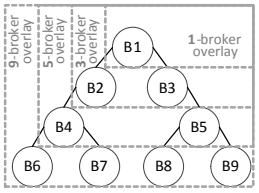


Fig. 4. Overlays.

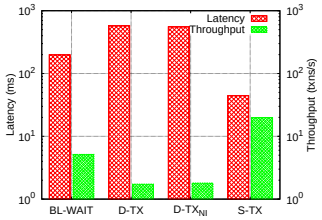


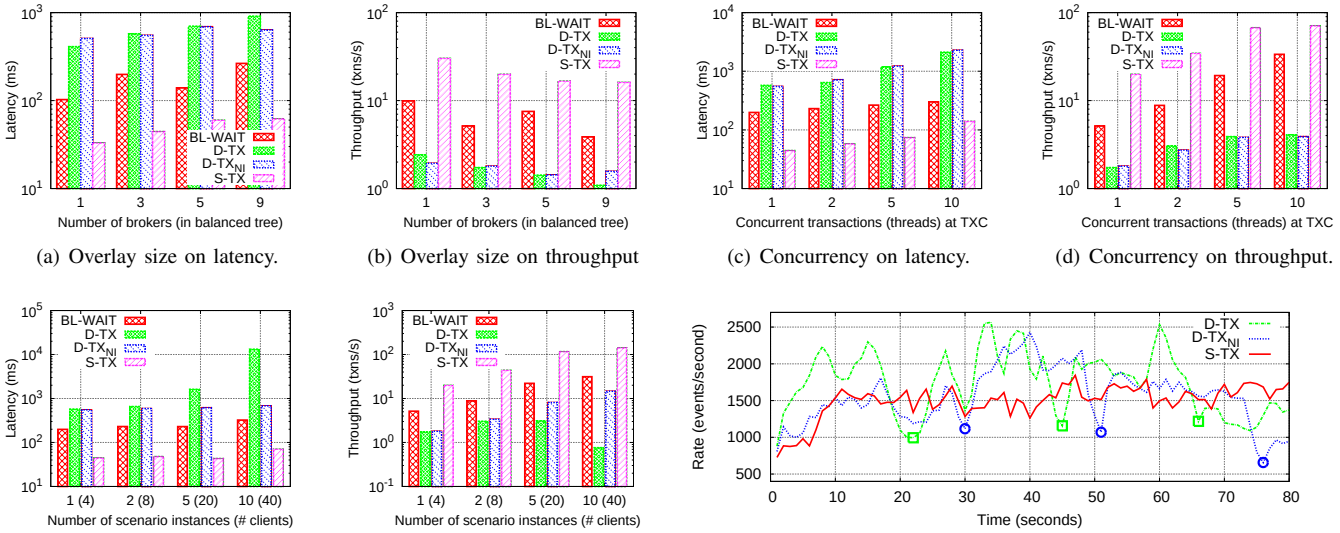
Fig. 5. Base comparison.

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. As expected, D-TX ($570\ ms$ averaged latency) performs worse than BL-WAIT($190\ ms$) due to 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-TX$_{NI}$ 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 ($\sim 20\ txns/s$) outperforms 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 Δ. 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 transaction. The graph in Figure 5 shows the average latencies: In the experiments for S-TX, we observed that some transactions terminated faster than the average. Similar, in our parametrization runs for BL-WAIT, we found that for some transactions, correct processing was achieved with less wait time. In these cases, *wait* introduced unnecessary latency, which, however, is important to ensure that all runs 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 – To analyze the impact of the overlay size 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 (cf. Figure 4). Again, clients have been distributed randomly and uniformly among brokers. Throughout the experiments, we varied the allocation of clients to brokers if possible. The results represent the averages of all runs and are depicted in Figures 6(a) and 6(b).

Essentially, the base performance relation among all four approaches is confirmed: BL-WAIT is faster than D-TX and D-TX$_{NI}$ in all overlay settings, and S-TX always beats BL-WAIT. However, with increasing overlay size, the performance of all four approaches declines. For instance, the average latency of



(a) Overlay size on latency. (b) Overlay size on throughput (c) Concurrency on latency. (d) Concurrency on throughput.

(e) Clients on latency. (f) Clients on throughput. (g) Impact of transactions on regular event routing.

Fig. 6. Results from experimental evaluation: Sensitivity analysis for overlay size, concurrency, client quantity and impact of approach on regular event traffic.

D-TX for a single broker, $\sim 410ms$, has more than doubled with nine brokers, $\sim 911ms$, resulting in ~ 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-TX$_{NI}$ performs better than D-TX, which originates from the reduced state kept at brokers in D-TX$_{NI}$. 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-TX$_{NI}$ 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 much less sensitive to the network size.

Impact of concurrency on performance – We also investigated the impact of concurrency on performance. Concurrency refers to the number of parallel transactions 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, fixed the overlay to three brokers, and executed 1000 transactions.

The results are depicted in Figures 6(c) and 6(d). Again, S-TX performs better than BL-WAIT, which beats D-TX and D-TX$_{NI}$. With all four approaches, latencies are increasing as the number of parallel transactions rises. While BL-WAIT is comparably stable (latency increases by 50% for 10 concurrent threads compared to sequential execution), the latencies for S-TX (3X) and D-TX/D-TX$_{NI}$ ($\sim$ 4X) grow faster. However, concurrency also has a positive effect on throughput as brokers spend less time idling: For BL-WAIT throughput increases by 7X, for S-TX by 3.7X and for D-TX by 2.3X. While in absolute numbers, S-TX reveals the best performance, its concurrency behavior is worse compared to BL-WAIT due to the overhead for the termination phase. In the end, concurrency improves 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 analyzed the impact. Using the instance dispatching scenario, we varied the number of parallel executions from a single instance (i.e., 4 clients) to 2, 5, and 10 instances (i.e., 40 clients). Each instantiation dispatches instances for a different workflow, so their event spaces do not overlap. We used a fixed overlay of three brokers and at every TXC, transactions are executed sequentially.

The results are depicted in Figures 6(e) and 6(f). Again, S-TX outperforms BL-WAIT and D-TX/D-TX$_{NI}$. The scaling behavior for S-TX and BL-WAIT is pretty similar: In both approaches, the latencies increase by only 50% but throughput increases by 7X when scaling up to 40 clients. Compared to scaling at a single client (cf. experiments on concurrency, Figure 6(d)), the two approaches perform better, which confirms that concurrency at clients impacts latency. The scaling behavior of D-TX is not as good: Throughput increases up to 20 clients but begins to drop again for 40 clients. The reason is the isolation management of D-TX, which requires agreement on a consistent snapshot and, more importantly, maintenance of the commit order during termination. Adherence to the commit order delays some ready-to-commit transactions while waiting for a prior-ordered transaction to terminate. This, however, is not an issue in D-TX$_{NI}$, which does not provide isolation 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 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, i.e., four clients 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 6(g). In general, the rates for all approaches are rather instable, which is likely a result of batch processing introduced by multi-threading in PADRES. But there is a clear distinction between D-TX//D-TX$_{NI}$ 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-TX$_{NI}$ 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-TX$_{NI}$ disrupts regular event routing while S-TX remains stable.

Summary – We evaluated D-TX, D-TX$_{NI}$ 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. Our experiments showed that S-TX efficiently guarantees consistency without disrupting regular event routing and even beats this baseline in all experiments. Dynamic transaction construction, as provided by D-TX$_{NI}$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.

VIII. Related Work

Related work can be classified into: (1) Centralized message queues with transaction support [3], [19], [4], (2) middleware-mediated transactions [16], [15], [22], and (3) transactional messaging in distributed brokers [5], [6], [17], [21], [23].

Transactional message queues include proprietary systems like TIBCO's Enterprise Message Service [4], ActiveMQ [3] based on the Java Message Service (JMS), or RabbitMQ [19] based on the Advanced Message Queuing Protocol (AMQP). All systems rely on point-to-point communication or a single message broker to deliver messages to subscribers. A transaction defines a context to group a set of messages that need to be atomically sent and received. This operation set is issued by a single publisher and buffered. On commit, 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-PC to atomically publish or consume messages. Also, Redis [1] groups a set of operations and executes it atomically and isolated.

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 multiple clients.

Middleware-mediated transactions integrate message queues and distributed object transactions [16]. X^2TS [15] 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, a transaction context is propagated with messages and 2-PC is used for atomic commitment. D-Spheres focuses on operationally grouping distributed object transactions [22]. 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 [5], a publisher can request a reply for its publication from subscribers. Replies are routed on the reverse paths of publications. The work aims for combining the decoupling and scaling features of pub/sub with request/response requirements of certain application. It is similar to the D-TX acknowledgment mechanism.

The Hermes Transaction Service (HTS) supports transactions in content-based pub/sub [23]. It is built on top of Hermes, a topic-based system using rendezvous-based routing [18]. A transaction demarcates the process of generating 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 [21], which allows multiple clients to publish as part of the same transaction.

Both works share similarities with our approach, but neither HTS nor TOPS support subscriptions or routing state modifications within a transaction and do not consider isolation.

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 system adaptation. The protocol is based on 3-phase-commit and compensation but the focus lies on transferring the client state and adapting the routing state according to the new edge broker. No general-purpose model is defined and supported.

IX. Conclusions

Today pub/sub systems provide only very limited delivery guarantees, especially for complex interactions between multiple clients. In this work, 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; 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 coordinator (TXC) has, we provide three implementations: D-TX and D-TX$_{NI}$ assume no knowledge on operations by other clients; acknowledgments enable consistent operation ordering, and additionally, in D-TX, snapshot isolation enables serializability. S-TX relaxes these assumptions: TXC has full knowledge about the transaction and dependencies ensure consistent ordering. Isolation is considered to be managed at the application level but routing states are guaranteed to converge even if concurrent transactions are not perfectly isolated. Our experiments showed that the uncertainty about operations renders D-TX and D-TX$_{NI}$ costly and suitable only for smaller configurations or scenarios with infrequent transactional occurrences. In contrast, S-TX does neither introduce overhead, which has been proven by comparing to a baseline pub/sub simulating these guarantees, nor disrupt regular event routing, which can be useful in many scenarios.

REFERENCES

[1] Redis. https://redis.io/, 2017.

[2] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguica, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems*, pages 405–414, 2016.

[3] Apache. ActiveMQ. http://activemq.apache.org/, 2017.

[4] A. Chan. Transactional publish/subscribe: The proactive multicast of database changes (abstract). In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, page 521, 1998.

[5] J. C. Hill, J. C. Knight, A. M. Crickenberger, and R. Honhart. Publish and subscribe with reply. Technical Report TR CS-2002-32, University of Virginia, 2002.

[6] S. Hu, V. Muthusamy, G. Li, and H. Jacobsen. Transactional mobility in distributed content-based publish/subscribe systems. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, pages 101–110, 2009.

[7] H.-A. Jacobsen, A. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The PADRES Publish/Subscribe System. In *Principles and Applications of Distributed Event-Based Systems*, pages 164–205. 2010.

[8] H. Jafarpour, B. Hore, S. Mehrotra, and N. Venkatasubramanian. Subscription subsumption evaluation for content-based publish/subscribe systems. In *Proceedings of the 9th International Middlware Conference*, pages 62–81, 2008.

[9] M. Jergler, M. Sadoghi, and H. Jacobsen. D2WORM: A management infrastructure for distributed data-centric workflows. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1427–1432, 2015.

[10] M. Jergler, M. Sadoghi, and H. Jacobsen. Geo-distribution of flexible business processes over publish/subscribe paradigm. In *Proceedings of the 17th International Middleware Conference*, page 15, 2016.

[11] M. Jergler, K. Zhang, and H.-A. Jacobsen. Multi-client Transactions in Distributed Publish/Subscribe Systems. Technical report, TU München, http://msrg.org/papers/pstx-tr, 2017.

[12] M. Kim, A. Mohindra, V. Muthusamy, R. Ranchal, V. Salapura, A. Slominski, and R. Khalaf. Building scalable, secure, multi-tenant cloud services on IBM bluemix. *IBM Journal of Research and Development*, 60(2-3), 2016.

[13] G. Li and H.-A. Jacobsen. Composite Subscriptions in Content-based publish/Subscribe Systems. In *Proceedings of the 6th International Middleware Conference*, pages 249–269, 2005.

[14] G. Li, V. Muthusamy, and H.-A. Jacobsen. A Distributed Service-oriented Architecture for Business Process Execution. *TWEB*, 4(1):2:1–2:33, 2010.

[15] C. Liebig, M. Malva, and A. P. Buchmann. Integrating notifications and transactions: Concepts and X^2TS prototype. In *Engineering Distributed Objects, Second International Workshop*, pages 194–214, 2000.

[16] C. Liebig and S. Tai. Middleware mediated transactions. In *3rd International Symposium on Distributed Objects and Applications*, page 340, 2001.

[17] A. Michlmayr and P. Fenkam. Integrating distributed object transactions with wide-area content-based publish/subscribe systems. In *In Proceedings of the 25th International Conference on Distributed Computing Systems Workshops*, pages 398–403, 2005.

[18] P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *In Proceedings of the 22nd International Conference on Distributed Computing Systems, Workshops*, pages 611–618, 2002.

[19] Rabbit Technologies. RabbitMQ. https://www.rabbitmq.com/, 2017.

[20] P. Salehi, C. Doblander, and H. Jacobsen. Highly-available content-based publish/subscribe via gossiping. In *Proceedings of the 10th International Conference on Distributed Event-Based Systems (DEBS)*, pages 93–104, 2016.

[21] Y. Shatsky and E. Gudes. TOPS: a new design for transactions in publish/subscribe middleware. In *Proceedings of the Second International Conference on Distributed Event-Based Systems*, pages 201–210, 2008.

[22] S. Tai, T. A. Mikalsen, I. Rouvellou, and S. M. Sutton. Dependency-spheres: A global transaction context for distributed objects and messages. In *Proceddings of the 5th International Enterprise Distributed Object Computing Conference*, page 105, 2001.

[23] L. Vargas, L. I. W. Pesonen, E. Gudes, and J. Bacon. Transactions in content-based publish/subscribe middleware. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, page 68, 2007.