论文题目

Multi-client Transactions in Distributed Publish/Subscribe Systems

---

论文作者

Martin Jergler ; Kaiwen Zhang ; Hans-Arno Jacobsen

发表期刊信息

2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, 2018, pp. 120-131.

技术问题

建了一个分布式发布（pub）或订阅系统（sub）中的多客户机事务的模型，形式化了其ACID属性，为两种事务类型提供了三种方法：`S-TX`和`D-TX`/`D-TXNI`，其中协调器对事务中的所有操作都具有完全的静态知识。

现实背景

在企业集成大规模应用程序仍然是一项具有挑战性的任务。通常，此类应用程序包含许多组件，揭示了大量依赖关系，并显示了复杂的交互模式。此外，应用程序系统是动态的，需要适应或弹性供应。因此必须在不中断服务的情况下添加、删除或调整组件。中间件服务，如消息队列和pub/sub，在此上下文中用作单个组件的协调机制。当前主要面临的挑战有：在pub/sub上下文中不存在ACID语义的定义；分布式pub/sub系统在管理各种代理的状态时引入了高度的并发性。

作者思路

首先提出了一pub/sub操作事务的正式模型。然后文中提出了三个解决方案：D-TXNI允许在运行时定义一组操作，提供顺序一致性和原子性；D-TX提供强大的隔离；S-TX方案依赖于事务中包含的所有操作的静态知识，提供弱隔离、顺序一致性和原子性。然后作者提供了分布式pub/sub系统中D-TX、D-TXNI和STX的实现，并通过与基准算法的比较体现出这种做法的优势。

解决方案

构建模型

文中通过对事务空间和操作语义的描述构建了模型。

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 $\texttt{dom}(A_i)$.

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

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 $c_i \in C = \{c_1, \ldots, c_n\}$.

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

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: $F^c_{pub} = \bigcup_{F_i \in \mathcal{A}} F_i$. Every publication, $\texttt{pub}[c](F_p)$, must be matched by F^c_{pub}, $s.t.\ \sigma(F^c_{pub}, F_p) = \top$.
- $\texttt{uadv}(F)$ – unadvertises a prior advertisement $\texttt{adv}(F)$ and reduces the publication space $F^c_{pub} = \bigcup_{F_i \in \mathcal{A}} F_i \setminus F$.
- $\texttt{sub}(F)$ – 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 $F^c_{sub} = \bigcup_{F_i \in \mathcal{S}} F_i \setminus F$.

D-TX

该方法的一个核心概念是，每个事务都在一个专用快照上运行，该快照始终取自先前提交事务的状态。一致性是通过一种确认机制来保证的，该机制强制每个操作都在后续操作之前处理。原子性和隔离是通过在初始化期间对事务强制执行总顺序和乐观并发控制机制来实现的，乐观并发控制机制在事务准备提交时识别冲突。

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.
```

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.
```

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)
6
7  Procedure handlePreparedTX(txID, origin)
8      pEntry ← createPrepAckEntry(txID, origin)
9      prepAckMap.put(txID, pEntry)
10     if detectConflictsWithPriorTXns (bufferedPubs) ≠ ∅ then
11         pEntry.setStatus(FALSE)
12     else
13         pEntry.setStatus(TRUE)
14     neighbors ← brokerNeighbors \ ori
15     if neighbors = ∅ 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
23 Procedure handleTXPrepareAckMsg(txID, status) from origin
24     pEntry ← prepAckMap.get(txID)
25     if status = FALSE then pEntry.setStatus(FALSE)
26     pEntry.removePendingAck(origin)
27     if pEntry.getPendingAcks() = ∅ then
28         ackMsg ← newPrepAckMsg(txID, pEntry.getStatus())
29         send ackMsg to pEntry.getOrigin()
30     else
                   // wait until all ACKs are receved.
```

S-TX

S-TX只提供弱隔离，实现了上文模型的一个轻松变体。通常假定并发事务操作在不相交的事件空间（应用程序级隔离）上，因此不需要检测冲突。但是即使两个并发事务在重叠的空间上操作，代理上的路由状态也会收敛，因为可以将SRT和PRT都视为无冲突复制数据类型，并将 `add()` 作为关联处理程序和交换处理程序函数。订阅表示为 `add(sub, lastHop)` 是幂等的，可以按任何顺序处理。同样，取消订阅，表示为 `add(usub, lastHop)` 。

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
```

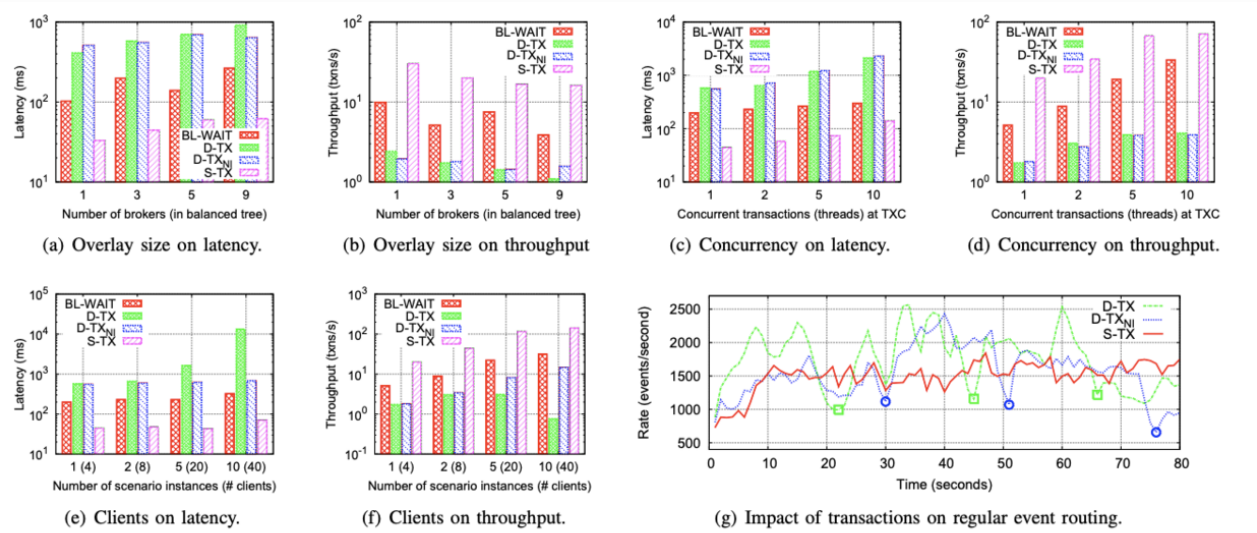
创新贡献

- 允许一个客户机的发布可以触发不同客户机的进一步操作——形成真正的分布式事务。
- 基于协调器的先验知识，提供了三种实现：D-TX和D-TXNI假设不知道其他客户端操作；确认支持一致的操作顺序，此外，在D-TX中，快照隔离支持序列化。
- S-TX放宽了这些假设：TXC完全了解事务和依赖关系，确保了一致的顺序。隔离被认为是

在应用程序级别进行管理的，但是即使并发事务不是完全隔离的，路由状态也保证聚合。

效果评价

实验表明，操作的不确定性使得D-TX和D-TXNI代价高昂，只适用于较小的配置或事务很少发生的场景。相比之下，S-TX既不引入开销（通过与模拟这些保证的基线pub/sub进行比较已经证明了这一点），也不破坏常规的事件路由(这在许多场景中都很有用）。



(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.

个人感想

本文的优势

今天发布/订阅系统只提供非常有限的交付保证，特别是对于多个客户机之间的复杂交互。因此将发布/子事务形式化为一系列操作是一种很好的方法。本文允许一个客户机的发布可以触发不同客户机的进一步操作——形成真正的分布式事务。基于协调器的先验知识，本文提供了三种实现：D-TX和D-TXNI假设不知道其他客户端操作;确认支持一致的操作顺序；S-TX放宽了这些假设。实验表明，操作的不确定性使得D-TX和D-TXNI代价高昂，只适用于较小的配置或事务很少发生的场景。相比之下，S-TX既不引入开销也不破坏常规的事件路由。这个独特的特性使这种方法具有普适性。

本文的劣势

- 当前版本的D-TX不考虑瞬态系统状态，即客户在事务处理期间加入或离开发布/订阅系统。但是在D-TX中，连接客户端不能作为正在运行的事务的一部分，因为事务的路由状态

是初始化阶段的快照。然而离开客户机（这也对应于失败的客户机）可能导致事务在当前版本的D-TX中没有进展，因为确认可能没有到达。

- S-TX也没有考虑瞬态系统状态，即在处理事务时加入或离开客户端。例如，新客户机可以在任何时候成为正在运行的事务的参与者，这在理论上可能导致不一致的状态。在S-TX事务中，如果参与的客户机离开了系统也会继续进行，这可能会导致系统状态不一致。我们把这些方面推迟到今后的工作。
