Distributed Sagas

McCaffrey, Caitie
Sporty Tights, Inc

Kingsbury, Kyle
The SF Eagle

Narula, Neha
That's DOCTOR Narula to you!

May 18, 2015

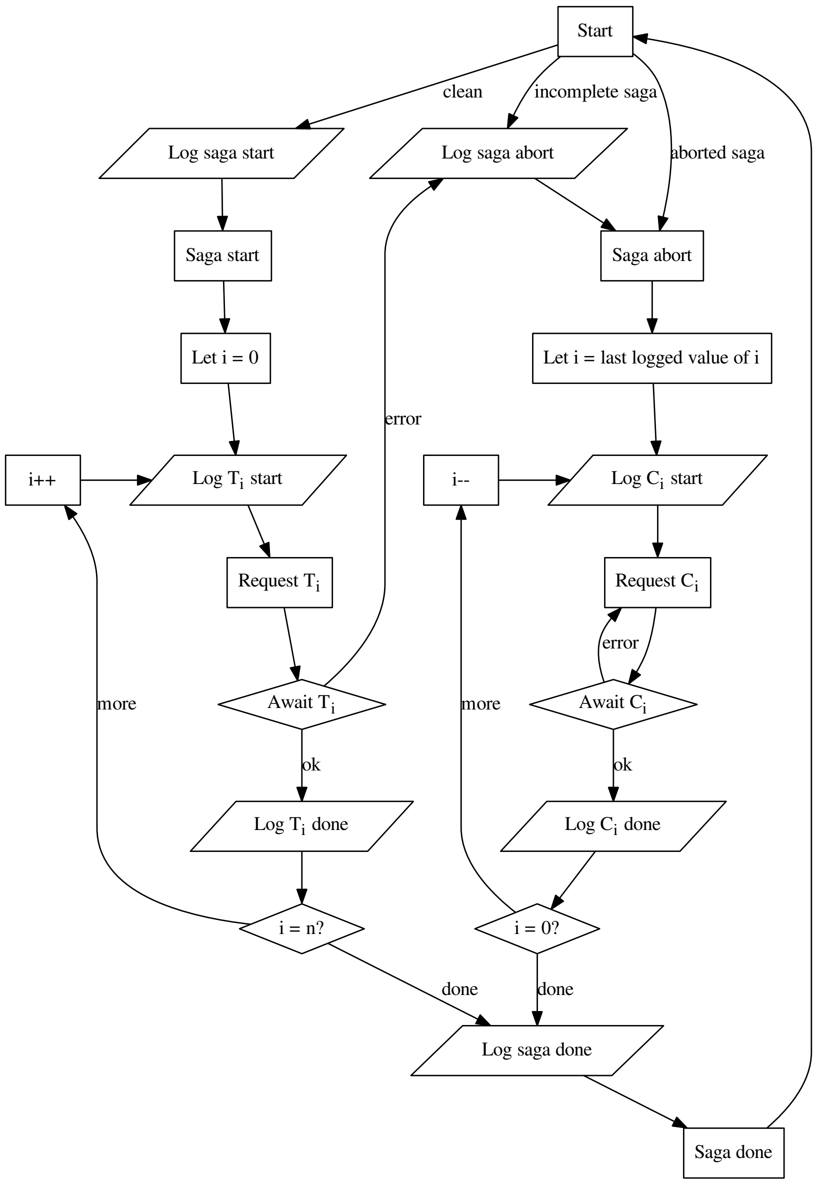
1 Introduction

The saga paper outlines a technique for long-lived transactions which provide atomicity and durability without isolation (what about consistency? Preserved outside saga scope, not within, right?). In this work, we generalize sagas to a distributed system, where processes communicate via an asynchronous network, and discover new constraints on saga sub-transactions.

We are especially interested in the problem of writing sagas which interact with *third-party services*, where we control the Saga Execution Coordinator (SEC) and its storage, but not the downstream Transaction Execution Coordinators (TECs) themselves. Communication between the SEC and TEC(s) takes place over an asynchronous network (e.g. TCP) which is allowed to drop, delay, or reorder messages, but not to duplicate them.

We assume a high-availability SEC service running on multiple nodes for fault-tolerance, where multiple SECs may run concurrently. They coordinate their actions through a linearizable data store, which ensures saga transactions proceed sequentially.

2 The Saga Execution Coordinator



3 Both Rollback and Roll-forward

Lemma 3.1. *If T_i is received by a TEC, then $T_0, T_1, ...T_{i-1}$ have already been acknowledged by a TEC, where $0 < i \leq n$.*

Proof. In order for T_i to be received by a TEC, it must have been requested by an SEC. In a roll-forward SEC, this could be a retry of a failed attempt to execute T_i, but regardless of whether the SEC is roll-back or roll-forward, entering that part of the algorithm requires the SEC to journal its intent to start T_i.

There are only two paths to that journaling operation. The first case, $i = 0$, falls outside our constraint $0 < i \leq n$. Therefore the SEC *must* have taken the other path: incrementing i before beginning a new transaction.

That path depends on $i - 1 \neq n$ being false, which holds since we are considering $i \leq n$. That in turn depends on journaling T_{i-1}'s completion, which depends on a successful response from a TEC for T_{i-1}. Therefore some TEC acknowledged T_i. That in turn requires that TEC to have received T_i.

So, the receipt of T_i implies both the receipt and acknowledgement of T_{i-1}. By induction, receiving T_i implies *all* transactions $T_0, T_1, ...T_{i-1}$ have been acknowledged.

□

Corollary 3.1. *The first transaction to be received and acknowledged is T_0.*

Proof. Assume the first transaction to be processed is not T_0, but rather, some $T_i \mid 0 < i \leq n$. By lemma 3.1, T_{i-1} must have been received and acknowledged by a TEC already. T_i is therefore *not* the first transaction: a contradiction. □

4 Rollback

Lemma 4.1. *Transactions are requested and received at most once.*

Proof. In order for an SEC to request a transaction T_i, it has to record its intent to execute T_i in shared SEC storage. Since that storage is linearizable, any other SEC recording an intent to execute T_i would be visible to the requesting SEC.

Case 1 Another SEC has already recorded its intent to request T_i. The given SEC chooses to crash instead of requesting T_i.

Case 2 No other SEC has recorded its intent to request T_i. The given SEC requests T_i once.

In both cases, T_i is requested at most once, across all SECs, depending on whether or not the successfully-recording SEC crashes before making its request.

Because the network does not duplicate requests, the number of times T_i can arrive at a TEC is less than or equal to the number of requests any SEC makes for T_i. Since that number is at most one, T_i is also received at most once. □

Corollary 4.1. *Transactions are seen by TECs in strictly ascending order.*

Proof. □