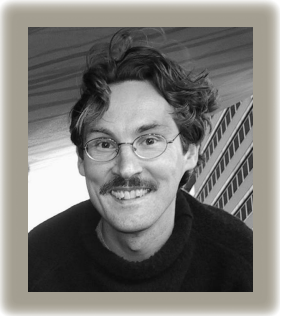
Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Your Coffee Shop Doesn't Use Two-Phase Commit

Gregor Hohpe

You know you're a geek when going to the coffee shop gets you thinking about interaction patterns between loosely coupled systems. This happened to me on a recent trip to Japan. One of the more familiar sights in Tokyo is the numerous Starbucks coffee shops, especially around Shinjuku and Roppongi. While waiting for my "Hotto Cocoa," I started thinking about how a coffee shop processes customer orders. As a business, the coffee shop is naturally interested in maximizing order throughput, because more fulfilled orders mean more revenue.



Interestingly, the optimization for throughput results in a concurrent and asynchronous processing model: when you place your order, the cashier marks a coffee cup with your order and places it into a queue. This queue is literally a line of coffee cups on top of the espresso machine. The queue decouples the cashier and barista, letting the cashier continue to take orders even when the barista is backed up. It also allows multiple baristas to start servicing the queue if the store gets busy, without impacting the cashier.

Asynchronous processing models can be highly efficient but are not without challenges. If the real world writes the best stories, then maybe we can learn something from Starbucks about designing successful asynchronous messaging solutions.

Correlation

For example, the asynchronous processing model means that drink orders aren't necessarily completed in the same sequence in which they were placed. This can happen for two different reasons. First, multiple baristas might be processing orders using different equipment. Blended drinks usually take longer to make than drip coffee, so a drip coffee ordered last might be delivered first. Second, baristas can make multiple drinks in one batch to optimize processing time.

As a result, Starbucks has a correlation problem. Drinks are delivered out of sequence and must be matched up with the correct customer. Starbucks solves the problem with the same "pattern" we use in messaging architectures—they use a *correlation identifier*.[1] In the US, most Starbucks use an explicit correlation identifier by writing your name on the cup and calling it out when the drink is ready. In other countries, they often correlate by drink type. The correlation issue became very apparent in Japan, where I had difficulties understanding the baristas calling out the drinks. My approach was to order extra large "venti" drinks because they're uncommon and therefore easily identifiable—that is, "correlatable."

Exception handling

Exception handling in asynchronous-messaging scenarios is naturally difficult. For example, if the receiver of a message is truly decoupled from the sender, what's the receiver to do when something goes wrong? What does Starbucks do if they've already placed your drink order into

 0740-7459/05/$20.00 © 2005 IEEE

the queue and then it turns out you can't pay? They either pull your cup from the queue or toss the drink if it has already been made. Likewise, if they deliver a drink that's incorrect or unsatisfactory, they remake it. If the machine breaks down and they can't make your drink, they refund your money. Each of these scenarios describes a different but common error-handling strategy for loosely coupled systems (see Figure 1).

Write-off

This is the simplest strategy: do nothing, or discard what you've done (see Figure 1a). This might seem like a poor plan, but in the reality of business, it might be acceptable. If the loss is small, building an error-correction solution might be more expensive. Furthermore, it might slow down the flow of messages, which reduces system throughput.

For example, I've worked for several ISPs who used this approach for errors in the billing and provisioning cycle. Occasionally, they failed to bill a customer with active service, but the revenue loss was small enough that it didn't significantly hurt the business (and customers rarely complained about getting a free account!). Periodically, they'd run reconciliation reports to detect and close such accounts.

Retry

When one operation out of a group of operations (or "transactions") fails, you have essentially two choices: undo the ones that completed successfully or retry the one that failed. Retry is a plausible option if there's a realistic chance that the retry will succeed (see Figure 1b). For example, if an external system is temporarily unavailable or an item is out of stock, a retry might be worthwhile. However, if a business rule was violated, it's unlikely a retry will succeed.

A special case is "retry all," where we simply ask all receivers to retry the operation. This requires all receivers to be *idempotent receivers*[1]—that is, the successful receivers must have built-in intelligence to ignore duplicate commands. This approach requires a little more cooperation from the receivers, but it simplifies the error-handling strategy.

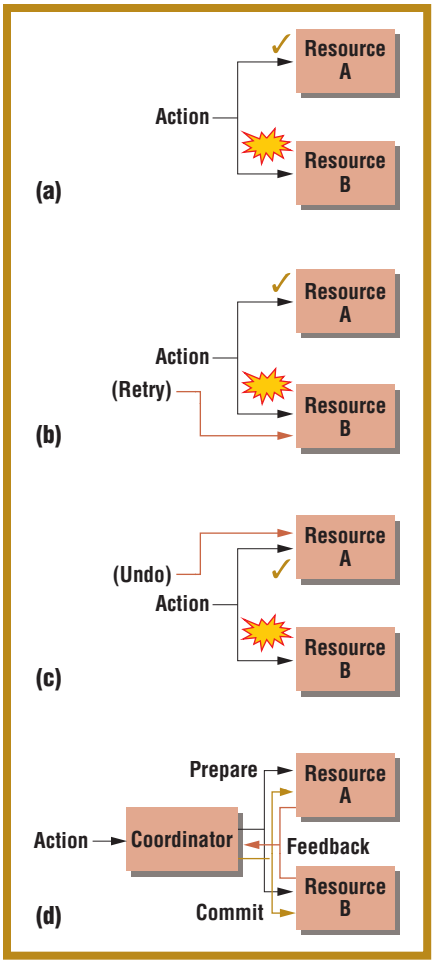


Figure 1. Error-handling strategies for loosely coupled systems: (a) write-off; (b) retry; (c) compensating action; (d) transaction coordinator.

Compensating action

The obvious alternative to retrying failed operations is to undo already completed operations to return the system to a consistent state (see Figure 1c). Such compensating actions work best with monetary systems, where we can credit money already debited. But real life uses many other compensating actions. For example, we might call and ask a customer to ignore a letter that has been sent or to return a package that was sent in error.

Transaction coordinator

The strategies discussed thus far differ from a traditional two-phase commit that relies on separate prepare and execute steps for each participant. In the Starbucks example, a two-phase commit would equate to having the customer wait at the cashier's counter with money and the receipt until the drink is ready. Then, the money, receipt, and drink would change hands in one swoop. Neither the cashier nor the customer could leave until the transaction was complete.

Using such a two-phase-commit approach would certainly kill a coffee shop's business because it would dramatically decrease the number of customers the shop could serve in a certain time interval. Although a two-phase commit can make life a lot simpler, it can also hurt the free flow of messages (and therefore scalability) because it requires the allocation of a stateful transaction resource across the flow of multiple, asynchronous actions. Starbucks follows an optimistic approach to optimize throughput in the "happy day" scenario, even though this results in a small loss when errors occur.

When amounts and stakes are larger, a pessimistic two-phase-commit approach is more appropriate (see Figure 1d). For example, when you purchase a home, an escrow company essentially acts as a transaction coordinator, offering a two-phase-commit service. The escrow company ensures that all required resources from all parties, such as funds, documents, and permits, are available before committing the transaction—that is, the purchase of the property. Once the transaction closes, all resources are consistently committed across all parties.

Yet even this case employs some compensation strategies. Because your bank account doesn't support a traditional *prepare and then commit or rollback* protocol, the escrow company takes money from your account first and then refunds it if the transaction has to be rolled back. Essentially, the *prepare* step translates into *debit money*, *rollback* translates into *credit money*, and *commit* translates into *do nothing*. This translation is legal because we assume we can credit an account—this action won't fail. Debiting money first implements a pessimistic resource allocation strategy that can be reliably reversed. So this scenario exemplifies how to integrate a compensating-action scheme into a coordinated two-phase-commit transaction. How-

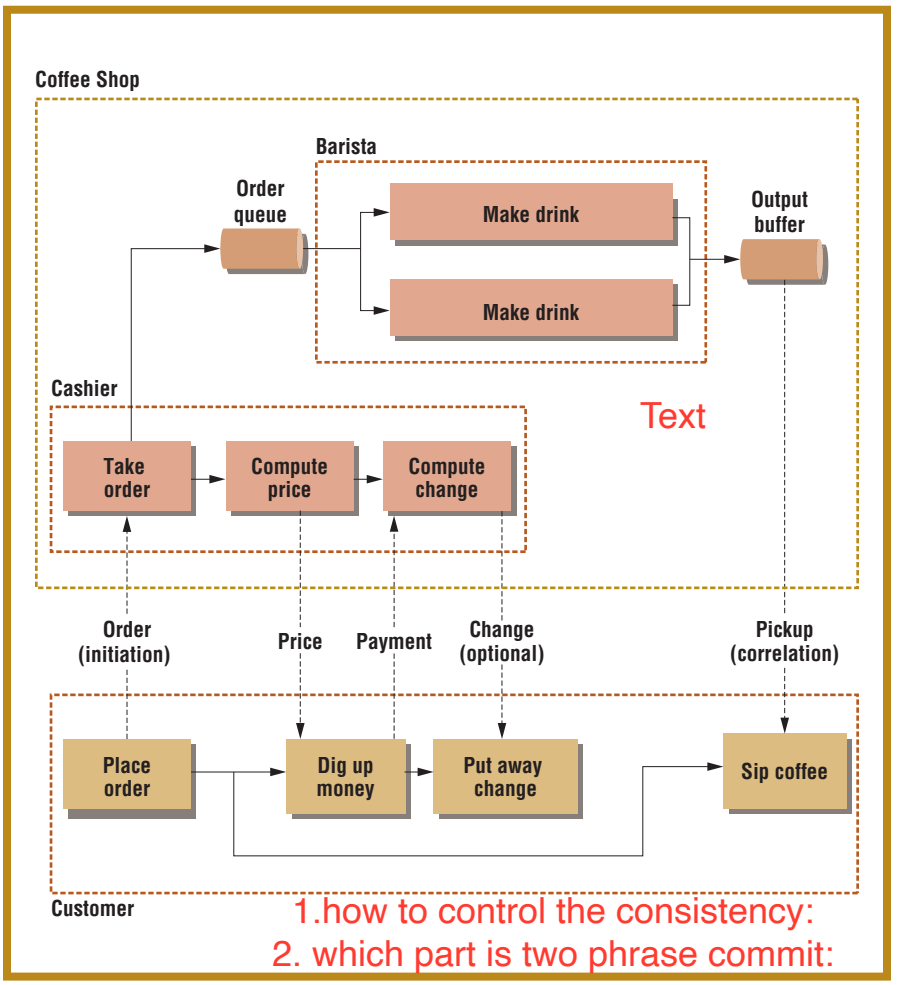


Figure 2. A conversation pattern.

ever, even this two phase approach doesn't guarantee that all the ACID properties often associated with transactions are supported. For example, the "I," which stands for *isolation*, isn't guaranteed. Although the escrow "transaction" is in progress, the money is actually taken out of the buyer's bank account, causing this transaction to not be isolated from other purchasing transactions.

Conversations

The coffee shop interaction is also a good example of a simple but common *conversation pattern* (see Figure 2). The interaction between two parties (customer and coffee shop) consists of a short synchronous interaction (ordering and paying) and a longer, asynchronous interaction (making and receiving the drink). This type of conversation pattern is quite common in purchasing scenarios. For example, when you place an order on Amazon.com, a short synchronous interaction assigns a unique order number first. All subsequent steps (charging your credit card and packaging and shipping the product) are performed asynchronously: you're notified via (asynchronous) email as additional processing steps complete. If anything goes wrong, Amazon usually uses similar compensation strategies—refunding your credit card or retrying the action by resending the goods.

Figure 2 illustrates the sequence of messages interchanged (the conversation) in a coffee shop and the internal process that each party follows. For example, a customer can pay only after he or she knows how much the coffee costs. Similarly, although the customer is ready to sip the coffee at any time, he or she must wait for the drink to arrive.

Asynchrony: A matter of perspective

The Starbucks scenario also demonstrates that being asynchronous often applies only to part of the interaction. For example, the baristas prefer an asynchronous model because it increases throughput—they can immediately move on to the next order without having to wait for a customer to retrieve his or her drink. The customer, on the other hand, expects a synchronous interaction; he or she came to the store to buy a drink and will have to stay until the drink is delivered. This type of synchronization is fairly common and has been documented as the *Half-sync, Half-async* pattern.[2] The main ingredient to solving both parties' requirements is a message buffer that lets the asynchronous participant (the barista) deliver messages (drinks) asynchronously without having to wait for the synchronous participant (the customer). In a coffee shop, the pickup counter acts as this message buffer.

The real world is often asynchronous. Our daily lives consist of many coordinated but asynchronous interactions (such as reading and replying to email or buying coffee). This means that an asynchronous messaging architecture can often be a natural way to model these types of interactions. It also means that looking at daily life can help us solve our messaging problems. For another great example of a high throughput system, I recommend spending rush hour at Shinjuku Station—over one million people pass through there every weekday. And yes, a Starbucks is nearby. Domo arigato gozaimasu!

References

1. G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley, 2003.
2. D. Schmidt et al., *Pattern-Oriented Software Architecture*, vol. 2, John Wiley, 2000.

Gregor Hohpe is an integration architect at ThoughtWorks and maintains the Web site www.eaipatterns.com. Contact him at ghohpe@thoughtworks.com.