Enterprise Integration Patterns

Gregor's Ramblings

Starbucks Does Not Use Two-Phase Commit

November 19, 2004

My blog posts related to IT strategy, enterprise architecture, digital transformation, and cloud have moved to a new home: ArchitectElevator.com.

Hotto Cocoa o Kudasai

I just returned from a 2 week trip to Japan. One of the more familiar sights was the ridiculous number of Starbucks (スターバックス) coffee shops, especially around Shinjuku and Roppongi. While waiting for my "Hotto Cocoa" I started to think about how Starbucks processes drink orders. Starbucks, like most other businesses is primarily interested in maximizing throughput of orders. More orders equals more revenue. As a result they use asynchronous processing. When you place your order the cashier marks a coffee cup with your order and places it into the queue. The queue is quite literally a queue of coffee cups lined up on top of the espresso machine. This queue decouples cashier and barista and allows the cashier to keep taking orders even if the barista is backed up for a moment. It allows them to deploy multiple baristas in a Competing Consumer scenario if the store gets busy.

Correlation

By taking advantage of an asynchronous approach Starbucks also has to deal with the same challenges that asynchrony inherently brings. Take for example, correlation. Drink orders are not necessarily completed in the order they were placed. This can happen for two reasons. First, multiple baristas may

Hi, I am Gregor Hohpe, co-author of the book Enterprise Integration Patterns. I like to work on and write about asynchronous messaging systems, service-oriented architectures, and all sorts of enterprise computing and architecture topics. I am also an Enterprise Strategist at AWS.

TOPICS

ALL RAMBLINGS
Architecture (12) Cloud (10)
Conversations (8)
Design (26) Events (27)
Gregor (4) Integration (19)
Messaging (12) Modeling (5)
Patterns (8) Visualization (3)
WebServices (5) Writing (12)

POPULAR RAMBLINGS

Starbucks does not use 2-phase commit

be processing orders using different equipment. Blended drinks may take longer than a drip coffee. Second, baristas may 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 need to be matched up to the correct customer. Starbucks solves the problem with the same "pattern" we use in messaging architectures -- they use a Correlation Identifier. 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 complete. In other countries, you have to correlate by the type of drink.

Exception Handling

Exception handling in asynchronous messaging scenarios can be difficult. If the real world writes the best stories maybe we can learn something by watching how Starbucks deals with exceptions. What do they do if you can't pay? They will toss the drink if it has already been made or otherwise pull your cup from the "queue". If they deliver you a drink that is incorrect or nonsatisfactory they will remake it. If the machine breaks down and they cannot make your drink they will refund your money. Each of these scenarios describes a different, but common error handling strategy:

- **Write-off** - This error handling strategy is the simplest of all: do nothing. Or discard what you have done. This might seem like a bad plan but in the reality of business this option might be acceptable. If the loss is small it might be more expensive to build an error correction solution than to just let things be. For example, I worked for a number of ISP providers who would chose this approach when there was an error in the billing / provisioning cycle. As a result, a customer might end up with active service but would not get billed. The revenue loss was small enough to allow the business to operate in this way. Periodically, they would run reconciliation reports to detect the "free" accounts and close them.
- **Retry** - When some operations of a larger group (i.e.

Is this architecture?
RESTful Conversations
Hub and Spoke
Correlation and Conversations
Diagram-driven Design

RECENT

Serverless Loan Broker @ GCP
Serverless Loan Broker @ AWS, Part 5: Integration Patterns with CDK
Serverless Loan Broker @ AWS, Part 4: Automation
Serverless Loan Broker @AWS, Part 3: Publish-Subscribe with SNS
Loan Broker @AWS, Part 2: Recipient List
Loan Broker Implementation with AWS Step Functions
Serverless Integration Patterns on Google Cloud Functions
Modern Examples for Enterprise Integration Patterns
37 Things or "Where have all my ramblings gone?"

"transaction") fail, we have essentially two choices: undo the ones that are already done or retry the ones that failed. Retry is a plausible option if there is a realistic chance that the retry will actually succeed. For example, if a business rule is violated it is unlikely a retry will succeed. However, if an external system is not available a retry might well be successful. A special case is a retry with Idempotent Receiver. In this case we can simply retry all operations since the successful receivers will ignore duplicate messages.

- **Compensating Action** - The last option is to undo operations that were already completed to put the system back into a consistent state. Such "compensating actions" work well for example if we deal with monetary systems where we can recredit money that has been debited.

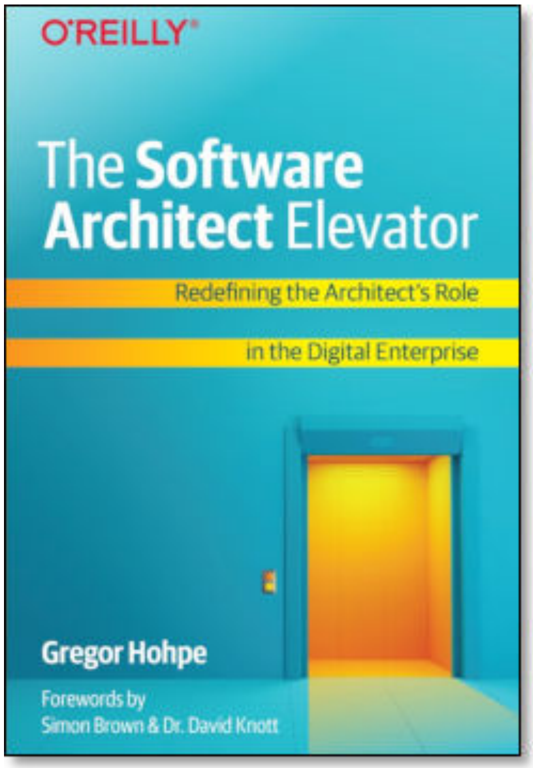
All of these strategies are different than a two-phase commit that relies on separate prepare and execute steps. In the Starbucks example, a two-phase commit would equate to waiting at the cashier with the receipt and the money on the table until the drink is finished. Then, the drink would be added to the mix. Finally the money, receipt and drink would change hands in one swoop. Neither the cashier nor the customer would be able to leave until the "transaction" is completed. Using such a two-phase-commit approach would certainly kill Starbucks' business because the number of customers they can serve within a certain time interval would decrease dramatically. This is a good reminder that a two-phase-commit can make life a lot simpler but it can also hurt the free flow of messages (and therefore the scalability) because it has to maintain stateful transaction resources across the flow of multiple, asynchronous actions.

Conversations

The coffee shop interaction is also a good example of a simple, but common Conversation pattern. 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 is quite common in purchasing scenarios. For example, when placing an order on Amazon the short synchronous interaction assigns an order number and all subsequent steps (charging credit card, packaging, shipping) are done asynchronously. You are notified via e-mail (asynchronous) when the additional steps complete. If anything goes wrong, Amazon usually compensates (refund to credit card) or retries (resend lost goods).

In summary we can see that the real world is often asynchronous. Our daily lives consists of many coordinated, but asynchronous interactions (reading and replying to e-mail, buying coffee etc). This means that an asynchronous messaging architecture can often be a natural way to model these types of interactions. It also means that often we can look at daily life to help design successful messaging solutions. Domo arigato gozaimasu!

The Software Architect Elevator


O'REILLY
The Software Architect Elevator
Redefining the Architect's Role in the Digital Enterprise
Gregor Hohpe
Forewords by Simon Brown & Dr. David Knott

Share: Tweet Share

Follow: Follow @ghohpe SUBSCRIBE TO FEED

More On: DESIGN CONVERSATIONS ALL RAMBLINGS

Gregor is an Enterprise Strategist with Amazon Web Services (AWS). He is a frequent speaker on asynchronous messaging, IT strategy, and cloud. He (co-)authored several books on architecture and architects.