Side Effects, Front and Center!

**ONE SYSTEM'S
SIDE EFFECT
IS ANOTHER'S
MEAT AND
POTATOES.**

PAT HELLAND

We think of computation in terms of its consequences. The big MapReduce job returns a large result. Web interactions display information. Enterprise applications update the database and return an answer. These are the reasons we do our work.

What we rarely discuss are the *side effects* of doing the work we intend. Side effects may be unwanted, or they may actually cause desired behavior at different layers of the system. This column points out some fun patterns to keep in mind as we build and use our systems.

LAYERS OF ABSTRACTIONS

As we build systems, we come across a bunch of layers of abstractions. The data center provides power, networking, cooling, and protection from rain. The server provides DRAM (dynamic random-access memory), SSD (solid-state drive), network, computation, HDD (hard-disk drive), and more. The operating system provides processes, virtual memory, file systems, and more.

Application and *platform* are subjective terms: application is the stuff that runs on top of me; platform is the stuff I run on top of.

*It's Not Your Grandmother's Database Anymore

As an example, memory management resides in a layer of abstraction below most application code. When memory is allocated from a heap, the application worries about *malloc* and *free* or some equivalent. It doesn't give a darn how the memory is managed or even where it resides. The application certainly doesn't care about fragmentation of the heap.

TMI

In the past few decades, the phrase *TMI*, meaning *too much information*, has entered the lexicon. It generally refers to knowledge about someone's personal life or hygiene that you have heard and wish you could un-hear. When your great uncle tells you about his digestive problems, that's TMI!

TMI can also refer to stuff you really don't want to know about that other subsystem you call from your application.

Side effect is a fancy computer science term for TMI.

SIDE EFFECTS IN LOTS OF PLACES

We see side effects in many, many places at many, many levels of abstraction. We even see side effects in life outside of computers. Here are a few to contemplate:

- ➡ Messages into and out of a microservice are typically logged for monitoring purposes.
- ➡ Competition for any resource may cause congestion and delay for other competing work. This is very much like the bad luck you experience as you try to drive on the freeway just as the ballgame is getting out.
- ➡ Traffic into a microservice may cause heap

fragmentation, impacting the responsiveness of the next request as the garbage in the heap is collected.

➡ Writing to the disk may cause the file system to get full. The next request is impacted.

➡ I may reserve a seat on an airplane, causing the next request by someone else to fail. It doesn't matter if I later cancel and don't use the seat. The other flyer still loses and won't be on that flight.

Each of these examples can be driven by work that is subsequently undone or aborted at the higher layer of abstraction. Logically, the work is undone *from the perspective of the higher layer*. Still, there are persistent changes visible at the lower layer. These are TMI from the perspective of the upper layers.

TRANSACTIONS ARE IN THE EYE OF THE BEHOLDER

The word *transaction* is used to describe some changes that are all or nothing. *ACID transactions*^{1,2} refer to those that are atomic, consistent, isolated, and durable. These attributes ensure a reliable sense of one change at a time and are most commonly associated with databases and database transactions. Transactions are a fascinating tool—and one I've spent a large part of my 38-year career working on.

It turns out transactions are frequently composed of other transactions at different layers of abstraction. This is called an *open nested transaction*.⁴ In an open nested transaction, a higher-layer transaction consists of multiple lower-layer transactions. To abort the higher-layer transaction, the system may need to issue compensated

lower-level transactions that undo the effect of the upper one.

Example 1: The trip to Europe

Now, let's consider some side effects that may result from a simple business trip to Europe.

- ➔ I make a reservation for a Wednesday night at a hotel in Paris. This is part of a set of reservations for airplanes, cars, and hotels for my weeklong trip to Europe.
- ➔ The reservation causes the occupancy of the hotel to cross a threshold so more staff and food for the restaurant are needed.
- ➔ The hotel restaurant orders a new shipment from the grocer for Tuesday.
- ➔ The grocer calls the shipping company for a delivery on Tuesday.
- ➔ The shipping company notices a projected shortfall in its petrol fuel supplies and orders more fuel for Monday.
- ➔ Then I cancel my trip to Europe.

My reservation caused a cascading set of effects that I don't see. Indeed, telling me about them would truly be TMI, causing me a great deal of confusion. Furthermore, these side effects persist even if my initial work is cancelled.

Side effects persist even if the stimulating activity is cancelled or aborted.

Example 2: The B-tree split

Database management systems typically store records in B-trees. Consider the following scenario:

- ➡ Record X is inserted by the user at the record-oriented layer of abstraction as a part of transaction T1.
- ➡ The database system calls its B-tree manager, which climbs down the B-tree to insert Record X. Upon discovering that the leaf of the B-tree is too full to receive Record X, it splits the leaf into two and stores Record X in one of them.
- ➡ Transaction T1 does some more stuff.
- ➡ Transaction T1 is aborted at the record-oriented layer. As a consequence, the B-tree manager is called to delete Record X from the B-tree, which it does. *The split of the leaf of the B-tree is not undone.*

When transaction T1 is aborted, all effects of T1 are eliminated from the set of records making up the database. Still, the leaf of the B-tree has been split and remains split. Figure 1 shows layered abstractions with database records on top and B-tree implementations below. A database transaction inserts into a B-tree, causing a block split. Later, the database transaction aborts, causing a delete to the B-tree. While Record X is deleted from the B-tree, the block split is not necessarily undone.

The record-oriented database is correct with T1 removed. The B-tree as a B-tree is correct with the proper leaves, indices, and pointers. Still, the B-tree is different because the transaction inserted and later deleted Record X.

The split of the B-tree page is a side effect of the aborted transaction T1. From the perspective of the set of records in the database, that's TMI.

FIGURE 1: LAYERED ABSTRACTIONS

IDEMPOTENCE AND SIDE EFFECTS

Personally, I think all distributed computing depends on timeouts, retries, and idempotence.³ *Idempotence* is the property of certain operations that you can do more than once but get the same result as if you did it once. Timeouts, retries, and idempotence allow the distribution of work with very high probabilities of success.

Now, what does idempotence mean if there are side effects? Is an operation idempotent if it causes monitoring of the call? That yields two monitoring records and is, hence, not an identical result. An operation is idempotent if it is repeatable at the desired layer of abstraction. It's typically considered OK if logging and monitoring record both attempts.

Idempotence is in the eye of the beholder!

Side effects to an idempotent operation are always OK. After all, they're side effects and, hence, not semantically important.

I'LL GET AROUND TO HYSTERESIS

It is quite common for one layer of the system to be slow in undoing stuff it recently did. This avoids the overall system flopping and jittering too aggressively.

For example, when the hotel reservation is cancelled because I chose not to go to Europe, that probably didn't change the order for groceries. Perhaps my reservation pushed the occupancy to 200 rooms and a new level of demand for the restaurant. Most likely, the expected occupancy will need to drop to 180 or so before the hotel will fiddle with the grocery order. Repeatedly calling the grocer to schedule, then cancel, then schedule deliveries is likely to drive the grocer to remove you from its list of customers.

Similarly, most B-tree managers are not anxious to merge two adjacent blocks when they fall below 50 percent each. The cost of rejiggering their contents repeatedly is too high.

Side effects from cancelled work will sometimes leave the system in a different state from what it was before. That may, in turn, impact subsequent requests.

CONCLUSION

Our systems compose in fascinating ways that have interesting interactions. To cope with this, many times we need to ignore the complications inside of the systems we use and just pretend life is simpler than it really is. That's great! We live in a higher level of abstraction and don't sweat the details.

One system's side effect is another's meat and potatoes

Still, the system providing the lower level of abstraction sees its job as its reason for existence. An order of groceries is the main purpose of the restaurant-scheduling application. Similarly, the B-tree manager has to keep records, fit them into the B-tree, and split when necessary. That's not a side effect but rather part of the job.

Side effects are only side effects to busybodies not minding their own business!

Just look past the TMI!

If every system pays attention to its own layer of abstraction and ignores the TMI of other layers of abstraction, all of this composition makes sense. Good design involves knowing when stuff is relevant and when stuff is TMI. After all, your great uncle's digestive problems are relevant to his doctor!

Related articles

- ➡ A Conversation with Erik Meijer and José Blakeley
The Microsoft perspective on ORM
<http://queue.acm.org/detail.cfm?id=1394137>
- ➡ Abstraction in Hardware System Design
Applying lessons from software languages to hardware languages using Bluespec SystemVerilog
Rishiyur S. Nikhil, Bluespec Inc.
<http://queue.acm.org/detail.cfm?id=2020861>
- ➡ Bridging the Object-Relational Divide
ORM technologies can simplify data access, but be aware of the challenges that come with introducing this new layer of abstraction.
Craig Russell, Sun Microsystems
<http://queue.acm.org/detail.cfm?id=1394139>

References

1. Gray, J., Reuter, A. 1993. *Distributed Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.
2. Haerder, T., Reuter, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys* 15(4): 287.
3. Helland, P. 2012. Idempotence is not a medical condition. *acmqueue* 10(4).
4. Weikum, G., Schek, H.-J. 1991. Multi-level transactions and open nested transactions. *Data Engineering* 14(1): 60-64.

Pat Helland has been
implementing transaction systems,

databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.

Copyright © 2017 held by owner/author. Publication rights licensed to ACM.