MARCH 25, 2015 BY TYLER TREAT

You Cannot Have Exactly-Once Delivery

I'm often surprised that people continually have fundamental misconceptions about how distributed systems behave. I myself shared many of these misconceptions, so I try not to demean or dismiss but rather educate and enlighten, hopefully while sounding less preachy than that just did. I continue to learn only by following in the footsteps of others. In retrospect, it shouldn't be surprising that folks buy into these fallacies as I once did, but it can be frustrating when trying to communicate certain design decisions and constraints.

Within the context of a distributed system, **you cannot have exactly-once message delivery**. Web browser and server? Distributed. Server and database? Distributed. Server and message queue? Distributed. You cannot have exactly-once delivery semantics in any of these situations.

As I've described in the past, **distributed systems are all about trade-offs**. This is one of them. There are essentially three types of delivery semantics: at-most-once, at-least-once, and exactly-once. Of the three, the first two are feasible and widely used. If you want to be super anal, you might say at-least-once delivery is also impossible because, technically speaking, network partitions are not strictly time-bound. If the connection from you to the server is interrupted indefinitely, you can't deliver *anything*. Practically speaking, you have bigger fish to fry at that point—like calling your ISP—so we consider at-least-once delivery, for all intents and purposes, possible. With this model of thinking, network partitions are finitely bounded in time, however arbitrary this may be.

So where does the trade-off come into play, and why is exactly-once delivery impossible? The answer lies in the Two Generals thought experiment or the more generalized Byzantine Generals Problem, which I've looked at extensively. We must also consider the FLP result, which basically says, given the possibility of a faulty process, it's *impossible* for a system of processes to agree on a decision.

In the letter I mail you, I ask you to call me once you receive it. You never do. Either you really didn't care for my letter or it got lost in the mail. *That's the cost of doing business.* I can send the one letter and hope you get it, or I can send 10 letters and assume you'll get at least one of them. The trade-off here is quite clear (postage is expensive!), but sending 10 letters doesn't really provide any additional guarantees. In a distributed system, we try to guarantee the delivery of a message by waiting for an acknowledgement that it was received, but all sorts of things can go wrong. Did the message get dropped? Did the ack get dropped? Did the receiver crash? Are they just slow? Is the network slow? Am *I* slow? **FLP and the Two Generals Problem are not design complexities, they are *impossibility results*.**

People often bend the meaning of "delivery" in order to make their system fit the semantics of exactly-once, or in other cases, the term is overloaded to mean something entirely different. State-machine replication is a good example of this. Atomic broadcast protocols ensure messages are delivered reliably and in order. The truth is, we *can't* deliver messages reliably and in order in the face of network partitions and crashes without a high degree of coordination. This coordination, of course, comes at a cost (latency and availability), while still relying on at-least-once semantics. Zab, the atomic broadcast protocol which lays the foundation for ZooKeeper, enforces idempotent operations.

> *State changes are idempotent and applying the same state change multiple times does not lead to inconsistencies as long as the application order is consistent with the delivery order. Consequently, guaranteeing at-least once semantics is sufficient and simplifies the implementation.*

"Simplifies the implementation" is the authors' attempt at subtlety. State-machine replication is just that, replicating state. If our messages have side effects, all of this goes out the window.

We're left with a few options, all equally tenuous. When a message is delivered, it's acknowledged immediately before processing. The sender receives the ack and calls it a day. However, if the receiver crashes before or during its processing, that data is lost forever. Customer transaction? Sorry, looks like you're not getting your order. This is the worldview of at-most-once delivery. To be honest, implementing at-most-once semantics is more complicated than this depending on the situation. If there are multiple workers processing tasks or the work queues are replicated, the broker must be strongly consistent (or CP in CAP theorem parlance) so as to ensure a task is not delivered to any other workers once it's been acked. Apache Kafka uses ZooKeeper to handle this coordination.

On the other hand, we can acknowledge messages after they are processed. If the process crashes after handling a message but before acking (or the ack isn't delivered), the sender will redeliver. Hello, at-least-once delivery. Furthermore, if you want to deliver messages in order to more than one site, you need an atomic broadcast which is a *huge* burden on throughput. Fast or consistent. Welcome to the world of distributed systems.

Every major message queue in existence which provides any guarantees will market itself as at-least-once delivery. If it claims exactly-once, it's because they are lying to your face in hopes that you will buy it or they themselves do not understand distributed systems. Either way, it's not a good indicator.

RabbitMQ attempts to provide guarantees along these lines:

When using confirms, producers recovering from a channel or connection failure should retransmit any messages for which an acknowledgement has not been received from the broker. There is a possibility of message duplication here, because the broker might have sent a confirmation that never reached the producer (due to network failures, etc). Therefore consumer applications will need to perform deduplication or handle incoming messages in an idempotent manner.

The way we achieve exactly-once delivery in practice is by faking it. Either the messages themselves should be idempotent, meaning they can be applied more than once without adverse effects, or we remove the need for idempotency through deduplication. Ideally, our messages don't require strict ordering and are commutative instead. There are design implications and trade-offs involved with whichever route you take, but this is the reality in which we must live.

Rethinking operations as idempotent actions might be easier said than done, but it mostly requires a change in the way we think about state. This is best described by revisiting the replicated state machine. Rather than distributing operations to apply at various nodes, what if we just distribute the state changes themselves? Rather than mutating state, let's just report *facts* at various points in time. This is effectively how Zab works.

Imagine we want to tell a friend to come pick us up. We send him a series of text messages with turn-by-turn directions, but one of the messages is delivered twice! Our friend isn't too happy when he finds himself in the bad part of town. Instead, let's just tell him *where* we are and let him figure it out. If the message gets delivered more than once, it won't matter. The implications are wider reaching than this, since we're still concerned with the *ordering* of messages, which is why solutions like commutative and convergent replicated data types are becoming more popular. That said, we can typically solve this problem through extrinsic means like sequencing, vector clocks, or other partial-ordering mechanisms. **It's usually causal ordering that we're after anyway. People who say otherwise don't quite realize that *there is no now* in a distributed system.**

To reiterate, there is no such thing as exactly-once delivery. We must choose between the lesser of two evils, which is at-least-once delivery in most cases. This can be used to simulate exactly-once semantics by ensuring idempotency or otherwise eliminating side effects from operations. Once again, it's important to understand the trade-offs involved when designing distributed systems. There is asynchrony abound, which means you *cannot* expect synchronous, guaranteed behavior. Design for failure and resiliency against this asynchronous nature.

Follow @tyler_treat

DISTRIBUTED SYSTEMS

CAP THEOREM, CONSENSUS, CRDTS, DISTRIBUTED SYSTEMS, FLP RESULT, MESSAGE QUEUES, MESSAGING, ZAB, ZOOKEEPER

59 Replies to "You Cannot Have Exactly-Once Delivery"

Confused

MARCH 25, 2015 AT 6:57 PM

"I can send 10 letters and assume you'll get at least one of them (at-least-once)."

How is it you can assume 1 of 10 gets through? Isn't this really "at most 10"?

If you can assume 1-of-N will get through, then why not just take N=1 and call it exactly-once?

Tyler Treat

MARCH 25, 2015 AT 7:27 PM

Yes, you're correct. It's not really an example of at-least-once delivery which requires an acknowledgement. Unintentionally misleading :)

Updated the wording to hopefully make it clear (and correct).

Jason Dusek

MARCH 26, 2015 AT 2:32 AM

The FLP result comes with a caveat — it applies to a "completely asynchronous" protocol.

> In this paper, we show the surprising result that no completely asynchronous consensus protocol can tolerate even a single unannounced process death.

With tightly bounded clock drift (hard to bound in practice), it seems reasonable that we can guarantee once-and-only-once delivery, because we can perform consensus.

Joubin Houshyar

MARCH 26, 2015 AT 1:59 PM

It seems the conceptual root cause is subscription to the illusion of continuums and a stubborn belief in the fairy dust of (meaningful) instantaneity. We need to accept the reality of a discreet view of the world– which naturally promotes to first-class design concerns the notions of 'precision' & (time) 'granularities'.

The fact of the matter is that all of our computational systems are operating on data from 'the past'.

(re. clock bounds: Google has done it with Spanner — someone needs to commoditize the necessary h/w.)

Nicolas Correard

FEBRUARY 24, 2016 AT 4:30 PM

CockroachDB do the equivalent of spanner without the hardware. Look for the blog post containing "CockroachDB was designed to work without atomic clocks or GPS clocks. It's an open source database intended to be run on arbitrary collections of nodes: from physical servers in a corp development cluster to public cloud infrastructure using the flavor-of-the-month virtualization layer. It'd be a showstopper to require an external dependency on specialized hardware for clock synchronization."

Imm

MARCH 26, 2015 AT 3:43 AM

Doesn't the existence of three-phase commit contradict this? If I make a change and commit it with 3PC, retrying until it does, how is that not exactly-once delivery?

JW

NOVEMBER 5, 2021 AT 11:32 PM

FLP literally does not apply to message passing lol. It important for writers to actually understand the proof of the theorem they're talking about before they make wild claims like those in this article. In fact, FLP assumes reliable links! You read that right–*it literally assumes the existence of exactly-once delivery between correct nodes*, but proceeds to show that this doesn't affect its main result. If you are claiming that one of the core system assumptions of FLP (the existence of reliable links) is disproved by FLP, you *might* just not understand distributed systems as well as you think you do.

But what about in practice? Well, in practice it's really easy to get exactly-once delivery between nodes, as long as you're in a system strong enough to eventually solve consensus. In most contexts in which people want exactly-once delivery (such as between managed nodes in datacenters) this is a completely reasonable assumption. So this is a dumb post in practice, too.

Webhiker

MARCH 26, 2015 AT 4:01 AM

All your objections to Exactly-once also apply to AtLeastOnce.
And ExactlyOnce is possible…all your objections are based on the current design flay of messaing systems which decouple delivery into a message queue.

If you don't decouple, the act of the recipient "reading" the message can be easily detected, including it's failure. But then all the investment in expensive message queues looks stupid, so no-one will be able sell their superior knowledge of straw man arguments on why message delivery cannot be guaranteed. :)

MC

APRIL 18, 2022 AT 10:37 PM

You have absolutely no clue what you're babbling about.

Michael Chermside

MARCH 26, 2015 AT 8:58 AM

Although it is impossible to create a system that guarantees "exactly once" delivery, it *is* possible to create a system that guarantees that EITHER (1) it will deliver exactly once, OR (2) it will report an error to a human being. This is also the technique best used for attempts at "at least once" delivery which fail over an extended period of time.

None of this invalidates anything you said about the usefulness of idempotency, but I like to point it out because it emphasizes two things: the impossibility of perfect message delivery (like "exactly once" being impossible) and the need to have someone monitor the error queues of your messaging system.

sumit

MARCH 27, 2015 AT 2:43 AM

Hey Michael,

It would be really helpful if you could point out some simple and relevant article(s) that supports the guarantee you mentioned.

Michael Ho

JUNE 5, 2015 AT 11:05 AM

Fancy meeting you here, MCherm!

Stu

MARCH 26, 2015 AT 9:16 AM

You're right, but I'm not sure about the melodrama. Accusing folks like IBM or BEA/Oracle of lying for 20 years is a reach, it's more like you weren't there when they coined the term.

"Exactly once" has *always* meant "at least once but dupe-detected". Mainly because we couldn't convince customers to send idempotent and communitative state changes.

Sebastien Lorber

APRIL 5, 2015 AT 7:10 PM

+1

With event-sourcing/stream processing for example you would version every event/message so that it's easy to dedup. If your friend receive 2 messages with id=456 telling him to turn left them it is easy for him to ignore one of them.

Another problem is about message ordering. If you have multiple datacenters and want to keep allow local writes during a network partition it seems impossible to guarantee global event ordering.
Kafka does only guarantee ordering across a single Kafka partition for example.
See how Eventuate is trying to solve this with causal consistency:
https://github.com/RBMHTechnology/eventuate

Pingback: Process Focus vs. System Architecture | Thinking Matters

Pingback: Endnu en god artikel om udvikling | Hennings blog

Pingback: Service-Disoriented Architecture | Brave New Geek

Mike Spooner

JUNE 8, 2015 AT 3:11 AM

Well said, nice article. But this has been well-understood since at least 1986… sigh

John B

SEPTEMBER 9, 2015 AT 9:06 AM

yes, it has been well understood by certain people for a long time, but there's been an entire new generation of software developers since 1986.

While it may be tedious for those of us who have been around for a long time, re-introducing key concepts to young developers is incredibly valuable work.

Abhinav Singh

SEPTEMBER 6, 2015 AT 2:05 AM

Great post as always. Wanted to leave my 2 cents here.

Let's forget engineering and take a real world example like you did. Assume a distributed system of 2 nodes, me and my wife sitting in next room. If I want to communicate with her, I shout out her name and wait for response. Well, if I don't hear back from her, we can assume:

– probably she didn't hear me (partitioned by walls)
– simply ignored my message coz she is busy
– received my message but it wasn't clear to her what to do with it
– received and she did shout back, but I just couldn't hear her due to partition caused by walls and due to her soft voice
– may be I did likely heard her, but I am not sure
– may be I was too busy when she called out to me
– ….. We can go on here

Now, if I seriously want her attention and mean business, I will have to move past this "exactly-once" melodrama and shout out to her again.

Pingback: You cannot have at-least-once broadcast - 250bpm

Pingback: » Reportáž z GeeCON Praha 2015 Myšlenky dne otce Fura

Pingback: Use Cases für Apache Kafka: "Viele Data-Probleme sind gar nicht so big" - JAXenter

Pingback: Exactly Once Stream Processing Semantics ? Not Exactly | Mawazo

Pingback: [illegible] – [illegible]

Pingback: [illegible] | [illegible]

Pingback: Monzo[illegible]0[illegible]7*24[illegible] - [illegible]

Pingback: Monzo[illegible]0[illegible]7*24[illegible]-zoues

Pingback: performance tuning kafka – Technology Musings

Kunal

DECEMBER 12, 2016 AT 1:22 PM

You can achieve the intent of "exactly-once" i.e. no duplicates and no data-loss on failures by making the receiver (client) state aware (i.e. offset, IDs); the client de-dupes.

There is business intent to building technology always. Religiously speaking, it is correct that exactly-once is not possible on the network protocol level in a distributed system; I don't think anyone will argue that. When people say we need exactly once, they really are speaking from a business or application intent.

Mike Spooner

DECEMBER 12, 2016 AT 6:12 PM

Although sequence-numbers/IDs does mean that, like the 787 Dreamliner, you have to poweroff or restart the entire system-universe every so often, at least until we get systems that really can count to infinity (at least “A0”, not necessarily as far as Cantors number).

Pingback: Tuning Kafka – Technology Musings

Pingback: Building a modern bank backend – At Monzo – Advance

Pingback: Delivering Billions of Messages Exactly Once · Segment Blog | Artificia Intelligence

Pingback: □□□□“□□□□”□□□□□□□□□ - □□□□

Pingback: □□□□□□□□□□□□□□□□□□□□□□ – □□□□□□□

Pingback: Apache Kafka gets 'exactly-once' message delivery - and that's a big deal - SiliconANGLE

Pingback: Apache Kafka 1.0 Released Exactly Once – IoT up2date

Pingback: Apache Kafka

Homesh Rawat

FEBRUARY 17, 2018 AT 4:12 AM

Great read!

mark

JUNE 18, 2018 AT 5:45 AM

I thinks these works like deduplication and… must execute on consumer side

Pingback: Reliable notifications between two apps or microservices

Pingback: 1 – A victim of its own popularity: Scaling our CloudWatch integration | Traffic.Ventures Social

Pingback: 1 – Nancy Lynch – Distributed Systems Pioneer | Traffic.Ventures Social

Pingback: Reliable notifications between two apps or microservices – NICE CODE

Nicholas

FEBRUARY 11, 2019 AT 12:11 PM

“The way we achieve exactly-once delivery in practice is by faking it. Either the messages themselves should be idempotent, meaning they can be applied more than once without adverse effects, or we remove the need for idempotency through deduplication.”

If the message only writes exactly once then that’s successful exactly-once semantics. Back when this article was written it was a hard problem a lot of people struggled with, but today Kafka has a system for exactly-once, and I work at a different company that does it in a different way. You can call it “fake” but in that case we have stable, well-functioning “fake” exactly-once semantics, and a lot of customers use that “fake” system successfully to solve real problems.

Pingback: A Game of Snakes & Ladders Called Microservices | APIscene

Pingback: Stream Deduplication with Hazelcast Jet | Hazelcast

Pingback: Stream Deduplication with Hazelcast Jet - Hazelcast Hazelcast the Leading In-Memory Data Grid

Mehedi

MAY 22, 2020 AT 7:14 PM

How Gmail ensure only one email?

Pingback: Cassandra: difference between exactly-once and at-least-once guarantees - IZZIDB

Pingback: Handling access token expiry when internet is unstable - Tutorial Guruji

Pingback: Exactly-Once Delivery[illegible] - [illegible]

Pingback: Exactly-Once Delivery[illegible] - [illegible]

temporal user

JANUARY 13, 2022 AT 9:05 PM

temporal does exactly once delivery

Pingback: [illegible]-[illegible]-[illegible] - [illegible]

Pingback: [illegible] – [illegible]

Pingback: Part 3: Processing Payments – Ethereum Payment | Code Capsule

Alexey Stogny

FEBRUARY 23, 2023 AT 10:15 AM

Nice post! Thanks! Wouldn’t post useless comment, but there’s no other way to subscribe to new posts ;)