Adaptive

Application-Level Consensus

March 2017

Application-level consensus

This article explores the benefits of using a consensus algorithm, such as Raft, to build clustered services. The core of this type of system is deterministic execution, replicated consensus log, and snapshotting of state to avoid replay from the beginning of time. Such a consensus approach offers simplicity, debug-ability, fault tolerance and scalability.

Years ago I came across a video in which Martin Thompson and Michael Barker discuss the architecture they devised at LMAX to build their core FX matching engine. You might have heard about this project or about the Disruptor they open-sourced some time later.

The idea of an “application-level consensus” (for want of a better phrase) that I explore in this article really resonated with me - and at Adaptive we’ve enjoyed huge benefits since we first started exploring its potential in 2015. In 2016, we were able to deploy it in the design and implementation of two trading systems, including a financial exchange.

What were the problems for LMAX?

LMAX is a financial exchange for FX (foreign-exchange), commodities and indices that processes buy and sell orders from clients in real time. FX is a very volatile market so the exchange needs to process a large quantity of orders at very low latencies (sub-millisecond). The core system in the exchange is the matching engine, responsible for organizing client orders in orders books and matching them. A matching engine is by definition stateful: it holds client orders until they get matched or cancelled.

The challenge with such systems is that they are highly contended and don’t suit sharding for parallelism or throughput: for instance, EUR/USD (euro vs US dollar) is in itself very volatile and all the orders for this currency pair need to be managed in a single order book: you can’t really shard this problem so how do you solve it, especially when you have to consider high availability as well?

Traditionally, most systems (other than exchanges) try to be stateless at the compute level and hold the state in some cache or database, but this design doesn’t cut it when throughput is that high and latency requirements so low.

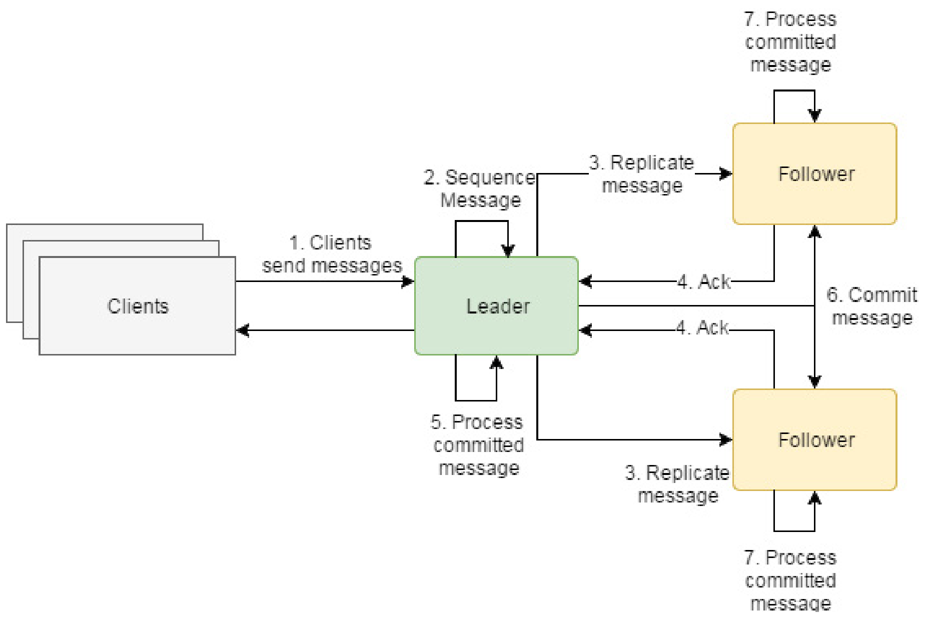
What model solved it?

The team at LMAX tried the highly contended database approach and various others, including SEDA, the actor model and others. Then they tried an idea dating back to the seventies and eighties: state machine replication: apply the same sequence of messages to state machines distributed on different nodes of a network and they will stay in sync, in the same state - and that's guaranteed, because state machines are deterministic.

So this is the overall idea: the matching engine logic (business logic) is written following some simple rules, which ensures determinism. Then this code is deployed on several nodes in a network and the same sequence of messages (buy, sell, cancel orders, etc) is applied to all nodes. At this stage you might be wondering how you can build this sequence: orders are coming from different places (UIs, APIs, etc) so there isn't really a "single sequence" available out of the box.

This is how you do it:

1. Place an algorithm to elect one of the nodes leader.
2. The leader processes all the incoming messages and sequences them (clients of the cluster always talk to the leader).
3. The leader replicates the sequence to follower nodes.
4. Followers acknowledge once they have received the message.
5. Once the leader has received acks from a quorum of the nodes in the cluster, the message is marked committed and is ready to be processed by the business logic.
6. The leader also notifies the followers that the message has been committed.
7. Followers can now apply it to their own state machine (business logic).



That's what the happy path looks like.

For the system to be highly available, things are more complex and there are quite a few rules to follow to guarantee that the sequence applied to all the nodes is the same.

Consensus algorithms

If you've heard about consensus algorithms such as Paxos or Raft, what you've just read about LMAX must sound quite familiar. There's a good reason for that: we're talking about the same thing. LMAX didn't use Raft: the paper wasn't yet published at the time.

Consensus algorithms are traditionally used to build distributed databases or distributed coordination systems (Chubby, Zookeeper, Etcd, Consul, etc). What's interesting in the LMAX case is that they **did not use the consensus algorithm at the database level but at the application level.** The engineering effort and the R&D involved in designing a distributed database and an application are generally fundamentally different and I don't think many have tried to build an application using a consensus algo directly in its service layer.

Consensus algorithms solve a hard problem in distributed systems: they guarantee that a set of nodes will agree and replicate the same state, even in the event of a fault: a node failure or network partition. Algorithms such as Raft guarantee linearizability to the clients of the cluster (the C of the CAP theorem). They are also fault-tolerant but a majority of nodes need to be available for the cluster to be available: those systems favour consistency (the C of CAP) over availability (the A of CAP).

Benefits for developing applications

What are the benefits, when developing applications, of using the consensus algo at the database level?

Simplicity

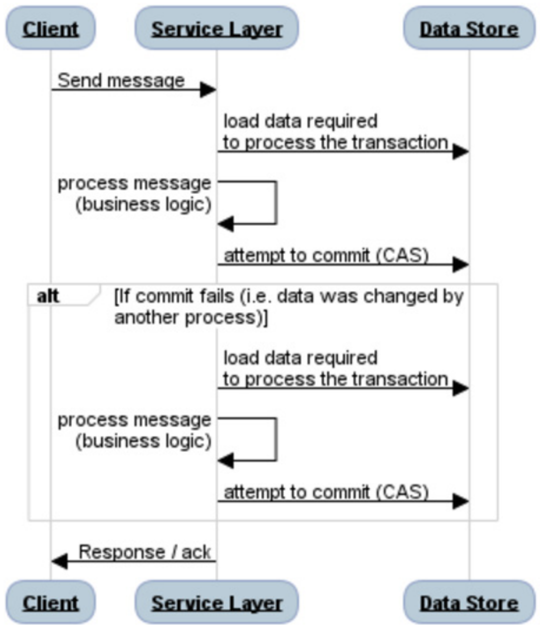
I think that what surprised me the most when I implemented the architecture for the first time was how simple it is. Once the Raft clustering infrastructure is in place, implementing the state machine (deterministic business logic) is quite straightforward and certainly much simpler than what I've seen with other approaches. I hadn't seen another design before with such a clean separation of concerns between infrastructure, the Raft consensus module, and the business logic.

It's a great environment to apply DDD. Our exchange business logic code is free of any framework or technical infrastructure: simple plain old objects, data structures and algorithms, all running on one thread. The exchange we've built implements some quite advanced credit management logic and different complex market models. And, to be honest, I couldn't think of any other design which would have allowed us to meet the client functional requirements and the high-availability targets: certainly not with such a quick time to market.

Consistency

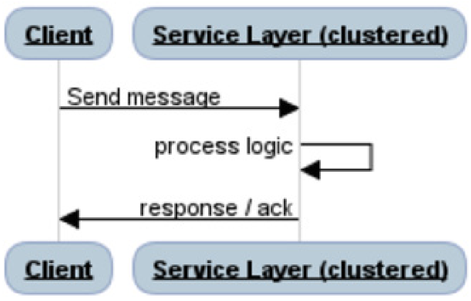
A very significant advantage of this architecture is that you have the guarantee that the state in your business logic is consistent between nodes. In a "traditional" system, this is what processing a transaction looks like:

1. Receive a message from a client.
2. Load the corresponding data from some store (database, cache, etc).
3. Process the message, applying some logic on the data and take a decision.
4. Try to commit the new data to the consistent store - but since we're in a distributed system and somebody else (another node, thread, etc) might have changed the data in the meantime, we need some form of optimistic locking or CAS (compare and swap) operation on the database to make sure we don't overwrite the data of the other user.
5. If the CAS fails, we have to load the data again and retry.


Client
Service Layer
Data Store
Send message
load data required to process the transaction
process message (business logic)
attempt to commit (CAS)
alt
[If commit fails (i.e. data was changed by another process)]
load data required to process the transaction
process message (business logic)
attempt to commit (CAS)
Response / ack
Client
Service Layer
Data Store

Contrast the above with what we have to do when we are "inside" a consistent system:

1. Receive a client message.
2. Apply the logic on the data we already have in memory (in this style of system we tend to load upfront most of the data we need at run time - but that's not a necessity, it just makes life easier).
3. Since the system is consistent, there is nothing to commit back anywhere - the state is the same in all nodes already.



This replaces reams of code that needs to handle multiple possible failure scenarios (failure to read from the store, to write to the store, etc) with simple and straightforward logic that you apply to state you already have in memory.

Much simpler.

Compute and data in one place

Another consequence of have strong consistency in your service layer is that you can aggressively load all the data you need in your business logic without being afraid that it becomes stale or inconsistent between your nodes, as long as you manage this data within the cluster. This means that processing an inbound message rarely requires a call to be made to an external system or datastore, which simplifies the code significantly and yields much better performance than an approach where compute and data live in different layers of the architecture.

No need for a database

"No need for a database" might sound controversial but ultimately you don't really need a database with such a design. Raft, like most consensus algorithms, stores locally on each node of the cluster all the messages the system has received: this is called the Raft log.

The log is used in several scenarios:

- If the cluster is restarted, the log can be applied to all nodes to put the system back in the same state - remember that the state machine is deterministic, so reapplying the same sequence of events will produce the same state.
- The log can also be used if one of the nodes of the cluster fails or is restarted: when it joins it can replay its local log then query the leader to retrieve any message it might have missed while it was offline.
- This property is also invaluable to troubleshoot bugs in your code: if your systems fails, you just need to retrieve the Raft log and replay locally, with the same version of the code and with a debugger attached. Do this and you'll reproduce the same issue. Anybody who has experience of diagnosing highly concurrent systems will understand the significant advantage of the approach I'm describing here.

Snapshotting

Since the Raft log could grow indefinitely it is generally combined with snapshotting: the system takes a snapshot of the application state at a given sequence number in the Raft log and stores the snapshot on disk. It is then possible to restart the system, or heal a failed node, by loading the most recent snapshot and then applying all subsequent messages in the Raft log.

No "ad hoc" resilience

Another radical difference with other architectures is how you approach resilience. Since the app is built on top of the consensus algorithm, you don't have to think about resilience - it's built in. This removes a huge amount of complexity from the code. I've designed many systems before where I had to carefully consider possible failures for lots of different flows and how they should be handled. I have no doubt that I missed quite a few in the process. You don't have to think about those when your application is built on top of this kind of consensus algorithm; you can focus on the business logic instead.

Deterministic business logic

I mentioned earlier that the business logic (state machine) needs to be deterministic: when you're applying the same sequence twice to your business logic, the system must end up in the same state. Practically, this means that the business logic should never:

- Use the system time. If you query the system time you will get a different output every time. Time needs to be abstracted away in the infrastructure and be part of the Raft log. Each message is timestamped by the leader and replicated to followers as part of the log entry. The business logic should always use this timestamp instead of the system time. **"Injecting" time in the system** also has a very nice side effect: it makes time- dependent code very easy to test (you can fast forward).

- Use random numbers without carefully defining the seed. If the system needs to generate random numbers you need to do this deterministically by seeding the random generator with the same seed on all nodes. You can, for instance, use the current message time (not system time!) as a seed.

- Use libraries that are not deterministic. This may sound very restrictive but remember that we're talking about the business logic of the system here, and in my experience **plain old objects work great.**

Is it really that easy?

Simple does not mean easy - you should watch this talk from Rich Hickey if you're not sure what I mean.

There is a significant effort to put such architecture together the first time and to adopt or build a consensus implementation; in this article I've only scratched the surface.

But I think this is totally worth the effort. If you have a chance to try this you should, and the chances are high you won't want to go back to something else. Some of the devs who worked at LMAX were saying that Martin broke them with this architecture: once they'd used it, it was very hard to work on any other system because it was too painful.

Now I understand why.

Trade-off

Be aware that strong consistency comes with a price: all nodes in the cluster are participating in the consensus algorithm and are processing every message the system receives.

If your system doesn't require strong consistency guarantees you're better off looking at a "share nothing" architecture, where each node processes requests independently. But even in this case, you can consider implementing some of the ideas we've discussed: sequencing messages received by each node, journaling them and using deterministic business logic so you can replay and easily diagnose issues.

Also be aware that a consensus algorithm requires a minimum of three nodes: you'll need at least three processes, running on three different servers and ideally in three different data centers (or availability zones, if you run on AWS). Three-node clusters are resilient to the failure of one node at most: if two nodes fail, the third node won't be able to become leader and the cluster will be unavailable until at least one other node rejoins. Five-node clusters are resilient to the failure of up to two nodes (three nodes down and the cluster is unavailable).

Also, since messages need to be replicated to a majority of nodes before being committed, the latency between the nodes will directly affect the maximum throughput of the cluster. For this reason, nodes are generally deployed in the same region.

In search of a name

I think this style of architecture is fit for many classes of trading system (real-time workflows, RFQ engines, OMSs, matching engines, credit-check systems, smart order routers, hedging engines, etc) but is also very relevant outside of finance and would yield far better results in some cases than the more traditional application layer sitting on top of some SQL or NoSQL database. But of course, as with any architecture, it suits some systems better than others.

David Farley (co-author of Continuous delivery):

"I think that it is more broadly applicable than that. Although the contention problem at the centre of trading was a driving force in its evolution, the advantages of simplicity, debug-ability, fault tolerance and scalability mean that it has very broad applications. I am currently building development tools using the same basic architecture. Works so far!"

When I talk with others about the architecture we tend to refer to it as the "LMAX architecture" but I think it deserves its own name. I've not found anything better so far than "application- level consensus" - but if you find a better name, please let me know.

Final thoughts

Application-level consensus deserves a lot more attention than it's had so far. It yields significant benefits and will help to solve complex requirements while keeping the system simple to reason about and correct. Unfortunately, at this stage the barrier to entry is still quite high, there is not yet a good place where you can learn about this architecture end to end, and a lot could be done in terms of infrastructure to get up and running on a project more quickly. We will do our best in 2017 to close this gap. In the meantime, if you think this architecture could help for one of your projects but you aren't sure where to start, please get in touch!

Acknowledgements

I would like to thank the following people who kindly accepted to review this article: Dave Farley, Martin Thompson, Julian Maynard-Smith, Thomas Pierrain, Matt Barrett, Shaun Laurens, James Kirkland

Olivier Deheurles

I am a software developer and I've been designing and building real-time trading systems for more than ten years. I worked on several open-source projects including Disruptor, Simple Binary Encoding, and Aeron. I co-founded Adaptive in 2011.

Adaptive

We are a software consultancy specialised in designing and building real-time trading systems for financial and commodity markets with offices in London, Barcelona and Montreal.

Adaptive

25th Floor,
Salesforce Tower
110 Bishopsgate
London EC2N 4AY

+44 203 725 6000
info@weareadaptive.com
weareadaptive.com