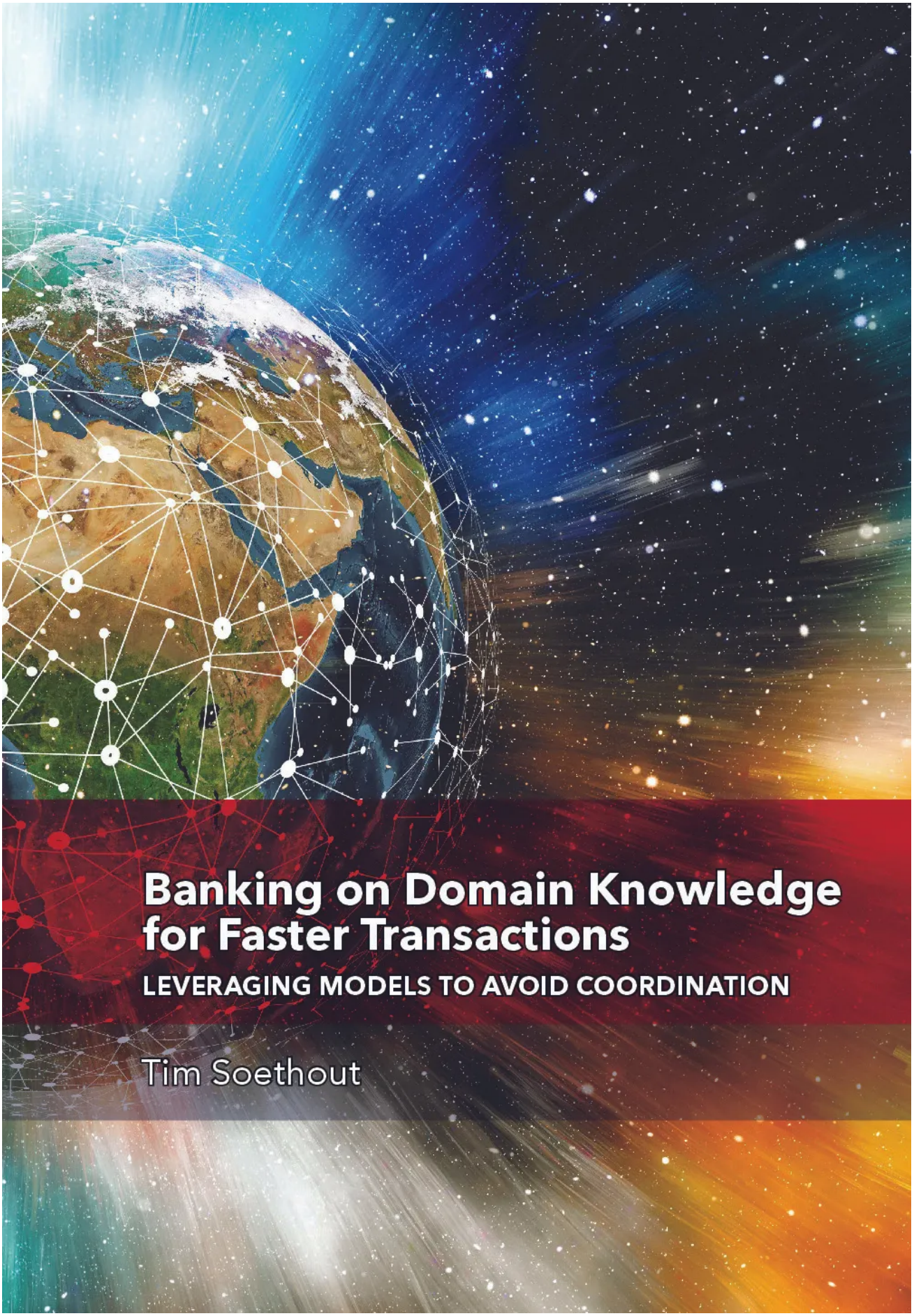
Published in ING Blog

Tim Soethout Follow

Mar 13 · 8 min read · Listen

Banking on Domain Knowledge for Faster Distributed Transactions

On Monday June 27th 2022, I defended my PhD thesis at the Technical University of Eindhoven (TU/e) and received the doctor title. This research project for ING is part of an ongoing collaboration with Centrum voor Wiskunde & Informatica (CWI), the Dutch National Research centre for Math and Computer Science.



Computer servers need to synchronize on data in order to stay correct and this is expensive. The research focusses on avoiding local coordination, by leveraging

domain specifics which general purpose approaches cannot.

ING Context

The project came forth from an ongoing collaboration between ING and the national Dutch research institute for Mathematics and Computer Science, CWI. CWI is a government-funded research institute and according to the last official evaluation on par with institutes like MIT.
I had the honor of being able to do a PhD project with them for ING.
The ongoing project is on managing IT complexity from the ING Tech R&D department. It is about extracting domain knowledge out of the heads of experts and model this into explicit code models by using a domain specific language designed for financial use.

One of the open research challenges was: How do we get from these domain models to running executable code? Early in the project I prototyped, together with other colleagues, a code generator based on actors, which can run massively parallel and be scaled easily over multiple servers. This code generator was used to pitch and gather starting capital for ING's incubator scale-up project XLinQ, which is a low-code/no-code platform based on model driven engineering. Of course there were also lots of open questions, some of which required research. One of them was how to make sure that the generated implementation would perform well in as many cases as possible. One of the main benefits here is that we have these domain models available where generic solutions do not. That's where my research spun off.

Research is a lot different than my previous role of software engineer. The research mindset of setting the bigger context, using research methods and questions take some time to get used to, but is a great way to objectively and rigorously approach a subject and solution. This scientific approach will definitely help ING forward.

Research Contributions

Post & Bank

A while back, when you wanted to transfer money to other people, you would use a giro "overschrijvingskaart", a payment order. You would fill in the details of the receiver, fill the amount, sign it and put it in the mail via the post to your bank.

Your bank would process it semi-automatically and send the money to the receiver's bank. The whole process took a couple of days. This was one of the first massive technology usages for consumers in the Netherlands and abroad. All communications would be done via messages, both in post and partly digital.

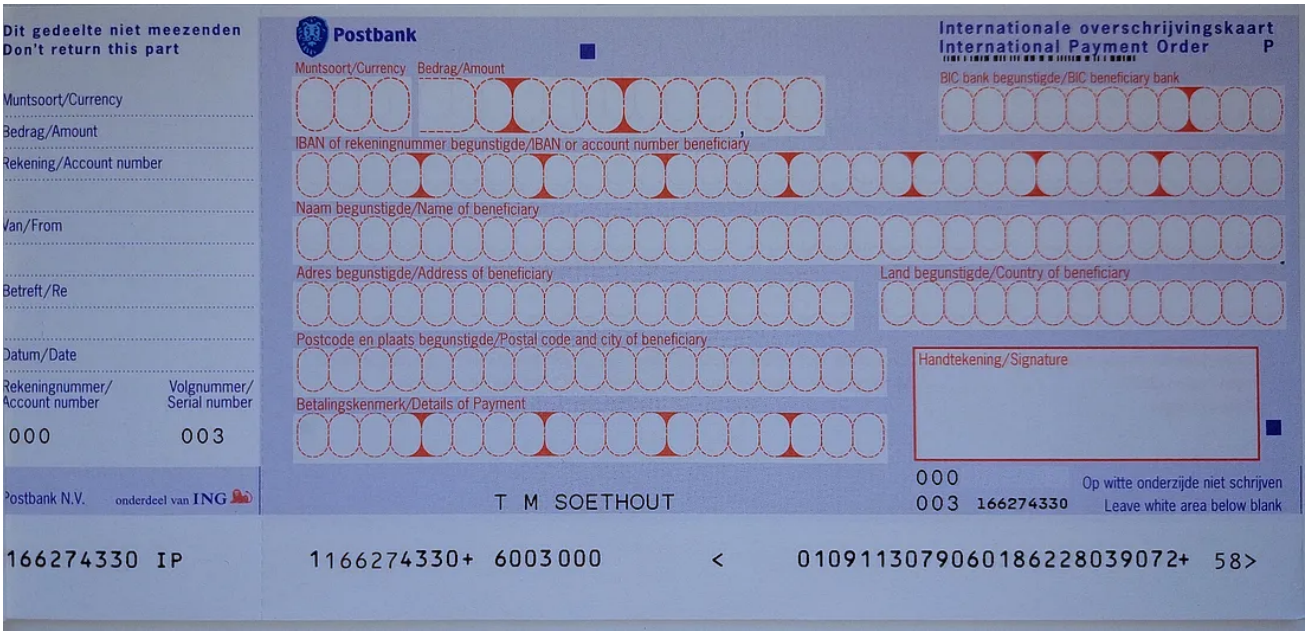


Image living about a century ago, and wanting to send money to someone. You would tell your own bank to transfer money. If the money cannot be transferred within the same bank, it has to be sent by postal carriage. After a couple of days the receiver can collect the money. Of course things can go wrong during transit of the money, such as lost carriages and lost money.

100 years into the future the process is similar. Except instead of physically going to the bank office, you connect to the online bank application and gives the order to transfer money. Via a digital message over the internet this is communicated with the computer servers of the receiver's bank and they can collect the money.

The transfer time goes from days to milliseconds, but the process stays similar. Messages and orders are sent between persons, computers and banks. In practice computer systems require multiple messages go back and forth, to make sure that accounts actually exist and no money is lost in between. Everyone expects it to be fast and immediate. Delay or latency should be as low as possible.

Messages & Computers

All interactions with banks or other companies need to be fast. All extra delay is counterproductive, especially when there are a lot of transactions. This table

shows relative waiting times for different operations on computers. To give some more human intuition on the time scale, one computer clock tick is framed relatively to one second.

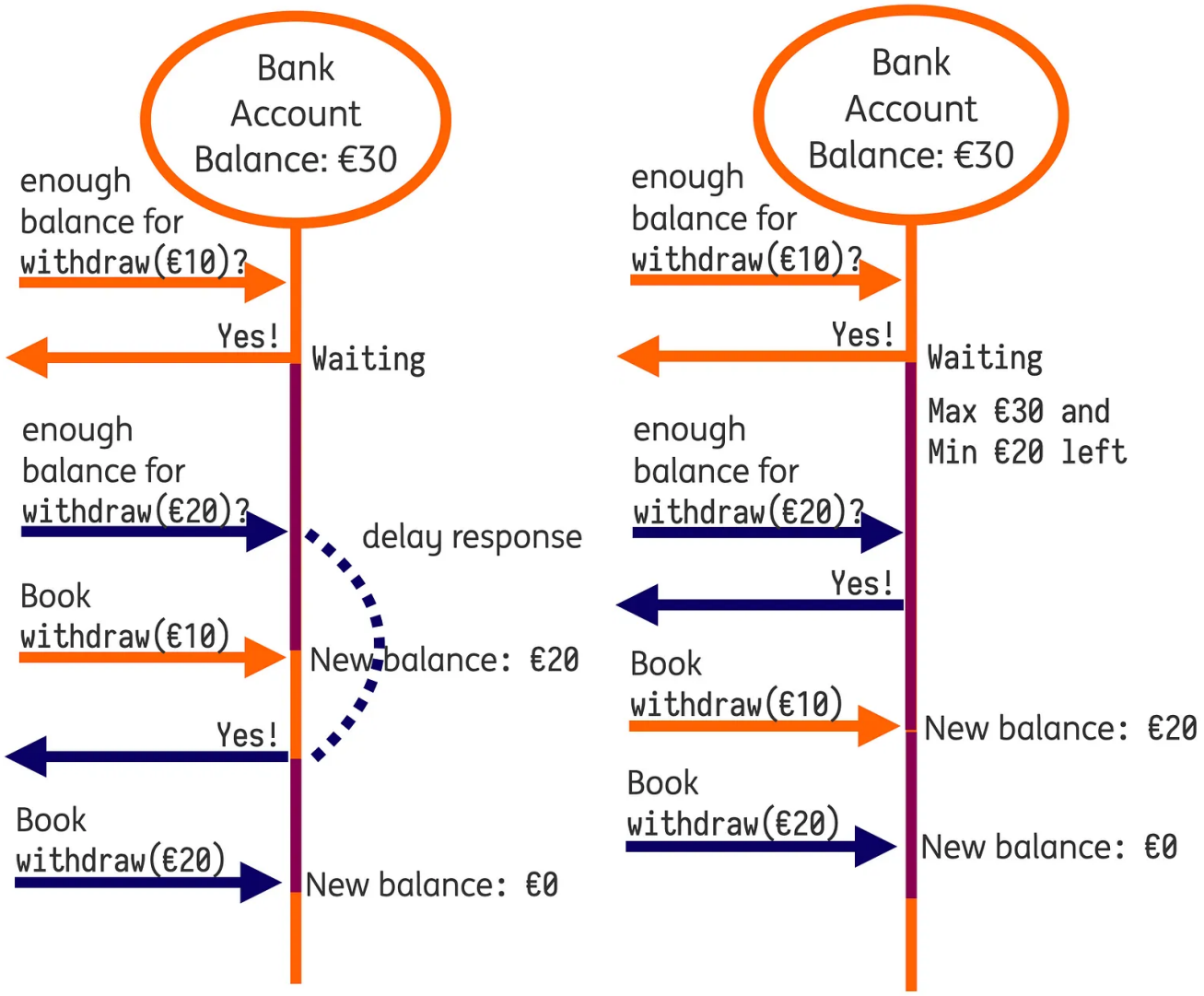
Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 second
Level 1 cache access	0.9 ns	3 seconds
Level 2 cache access	2.8 ns	9 seconds
Level 3 cache access	12.9 ns	43 seconds
Main memory access (DRAM, from CPU)	120 ns	6 minutes
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–0 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware virtualization system reboot	40 s	4 millennia

Computers wait a lot of time

Sending a message over the internet from San Francisco to the United Kingdom takes 81 milliseconds. This does not seem like much. In the human time scale, sending a message would take 8 years, in which you can do nothing else, except standing around waiting. The computer has to wait on response before it can continue processing.

Every message back and forth takes a lot of time waiting compared to the number of computations a computer can do, in which nothing else productive can be done. A single bank transaction would take multiple messages, say 5, which would be 40 years. So the goal is to send as few messages as possible, but to still give the same required result and guarantees.

On the left below is an example of an algorithm that makes sure that rules for bank accounts are never violated, based on two-phase locking. To be on the safe side, it waits until a money transfer is completely finalized before handling another.



On the left side, we see a bank account, and a message arriving for a withdrawal at this bank account. The account checks if enough balance is available, €30 is enough to deduct €10, without going below €0, so it responds yes. It cannot immediately deduct the €10 because the other side of the transaction might fail or abort, so it waits. In the meantime it cannot accept any other operations. This is what we call locking in computer science. The account is locked for other operations while waiting.
Later the book operation arrives and the balance is updated. Now the lock can be released. The locking is needed to be sure no money is lost and first checks if all involved accounts actually allow the withdraw and deposit, and not only some of them, in which case money would disappear, and we do not want that. This is a generic algorithm or approach that is often used to make sure nothing bad happens for all kinds of transactions with multiple participants, potentially spread around the globe.

Now another withdraw (blue messages) can come in the meantime, but it has to

wait. It's response it delayed. Only after the first is final, it can also vote yes, and the transaction can continue and deduct the money on the next book message.

Something that we already see happening here is that there is actually enough balance available for both withdraws, even if the first withdraw would not go through. The operations do not conflict with each other, but still have to wait, to be on the safe side. While this may be clear for us to see, the computer systems doing these steps do not know this without our help, because it depends on the specifics of the accounts and allowed operations. So it stays on the safe side by locking and waiting.

The PhD research focused on automagically detecting these kinds of scenarios and allowing multiple operations to continue in parallel, and with that avoiding local coordination. It can do this, by using the available domain knowledge, in this case the requirements on the account and withdraw operations, to see if there are no conflicts.

Local Coordination Avoidance

A big part of the research is about the Local Coordination Avoidance algorithm, or LoCA for short. Instead of the locking we just saw on the left side, which takes a lot of time, and effectively lets the computer run idle, it runs multiple operations in parallel on for example the account. But only when it is safe to do so and does not violate any requirements there might be, such as not over-drafting an account, or exposing invalid data to clients. When it can parallelize, this results in higher transaction throughput and less waiting.
To do this, it leverages a simple contract on objects and their operations, by simulating possible outcomes of operations and detecting if already starting the next one influences the result in ways that would not be valid.

The image on the right illustrates this. The same withdraw to €10 arrives, but instead of totally locking, the algorithm takes into account all possible outcomes, which would be either €30 or €20 left, depending on if the withdraw actually aborts or not.

A second incoming withdraw (blue message) can already be checked and in both cases enough balance is available, so it can already vote yes without waiting. This results in less delay and more transactions being handled in the same amount of

time.

So by allowing allow these non-conflicting operations to already vote and continue processing, overall throughput is increased, and responses are faster. You can imagine that less waiting here also positively effects all the other accounts, because everyone has to wait less on each other. This improves performance of the whole application and other connected applications.

This only works for specific operations: withdraws are only non-conflicting if enough balance is available for all. Another conflicting example is when an account becomes blocked for whatever reason, it should not continue withdrawing. This is also exactly the wanted behavior, since, as a bank, we do not want to book money from blocked accounts. So operations are only parallelized when the business rules or requirements on the account actually allow it.

Backing and Grounding

The LoCA algorithm is one of the main contributions of the PhD, but there are of course all kinds of underlying and related parts as well. One part is statically detecting non-conflicting operations: some non-conflicting operations are always non-conflicting, independent of the run-time state — the account balance in our examples. For example a *deposit* w s added to an account, can always go through, no matter the actual balance.

The dissertation also discusses formalizing the guarantees of the algorithm, and how to make sure these guarantees are upheld using model checking.

Conclusion

We saw that all distributed computer systems are based on messages, where iv rove performance and response times, waiting should be reduced, without violating application constraints. The Local-Coordination Avoidance waits less by running non-conflicting operations in parallel, but only when safe, speeding everything up.

The thesis is available here.