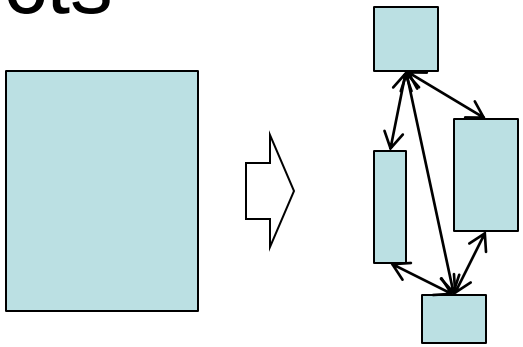
SOA and Cloud Symposium
Berlin 2010-10-05 - 06

All-or-Nothing Transactions – So Good, but cannot be Used in SOA. So What do We do?

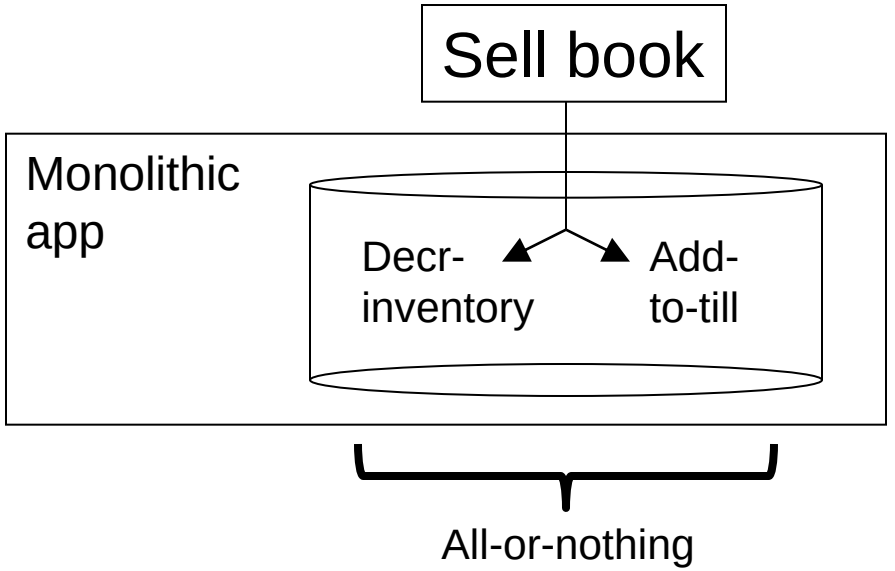
Sven-Håkan Olsson

Breaking up monoliths – some aspects



- Applications used to be large, monolithic entities:
  - Con:
    - Unmanageable when too big
    - Often difficult to integrate with
    - Difficult to analyze consequences when needing to change
    - Not flexible enough to keep up with changing business demands
  - Pro:
    - One single part could manage the app architecture in a cohesive way
    - One cohesive information model could be used
    - **Information often fresh, truly ”online”**
    - **Transactions could be used to keep the information quality high**
- SOA (and/or partial outsourcing in the Cloud) instead:
  - Pro:
    - Much more flexible
    - Needed because of demands for more IT interaction with other businesses
    - Loose coupling gives many advantages
  - Con:
    - **Information often not so fresh, nor truly ”online” because of async patterns**
    - **Transactions usually cannot be used to keep the information quality high**

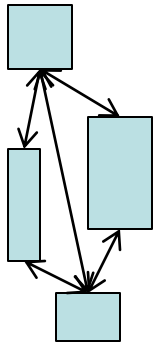
ACID: All-or-nothing



- Selling a book in a bookstore:
  - The old monolithic app would reduce the inventory by 1 and add 10 Euro to the till-ackumulator
  - Should the second update go wrong because of a software bug (now, that neeeever happens, eh?), disk fault, network problem, database timeout, deadlock etc:
    - An automatic rollback takes place, the information is still consistent
    - (By the way, the clerk has got to react on the hopefully understandable error message and keep the book, or retry the sale!)
  - All information is fresh. You can check the inventory online when a customer want to know that a certain book is available. The clerk can imediately tally the coins and bills relative to the till-ackumulator.

ACID = Atomicity – Consistency – Independency – Durability
Other names: All-or-nothing transactions, unit-of-work, commit-rollback, unbreakable-update…

All-or-nothing problems

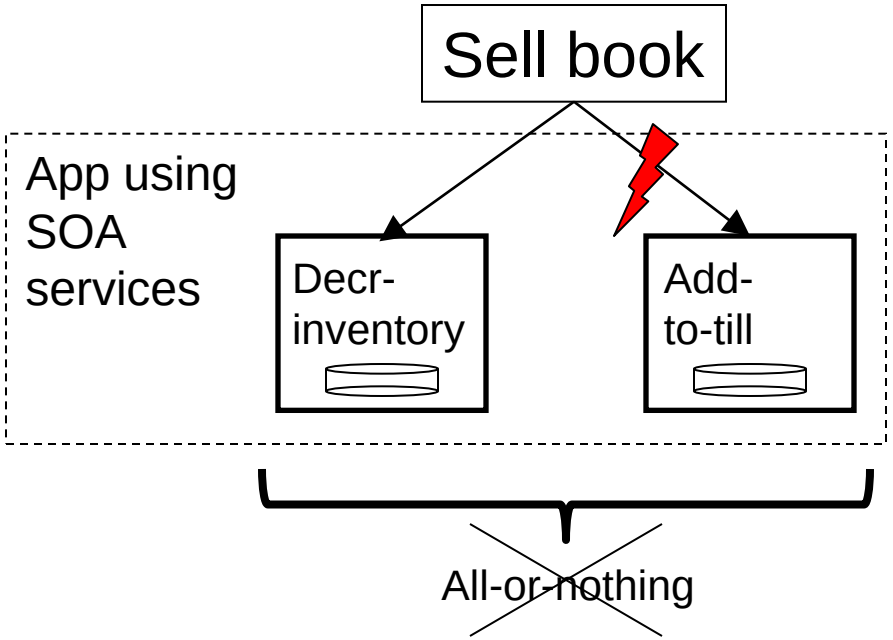


- The ACID thing has worked really well for decades

- Now SOA: We break up the monolith into separate, interoperable parts – so we **would need distributed ACID** between the SOA parts, but usually that's **not a good idea**:
  - The distributed ACID protocol (two-phase-commit, 2PC) is really complicated and very slow
  - 2PC is a synchronous operation as a whole, dependant on distributed parts to be alive during the entire 2PC operation – gives substantially lower uptime expectancy
  - 2PC gives much longer internal database locks, reduces scalability
  - 2PC is a stateful protocol which violates SOA guidelines
  - Difficult dependencies on versions and details in the software inside the SOA parts for support of an interoperabable 2PC (and WS-AtomicTransaction is typically not the solution…)
  - So, 2PC: Not loose coupling

ACID = Atomicity – Consistency – Independency – Durability
Other names: All-or-nothing transactions, unit-of-work, commit-rollback, unbreakable-update…

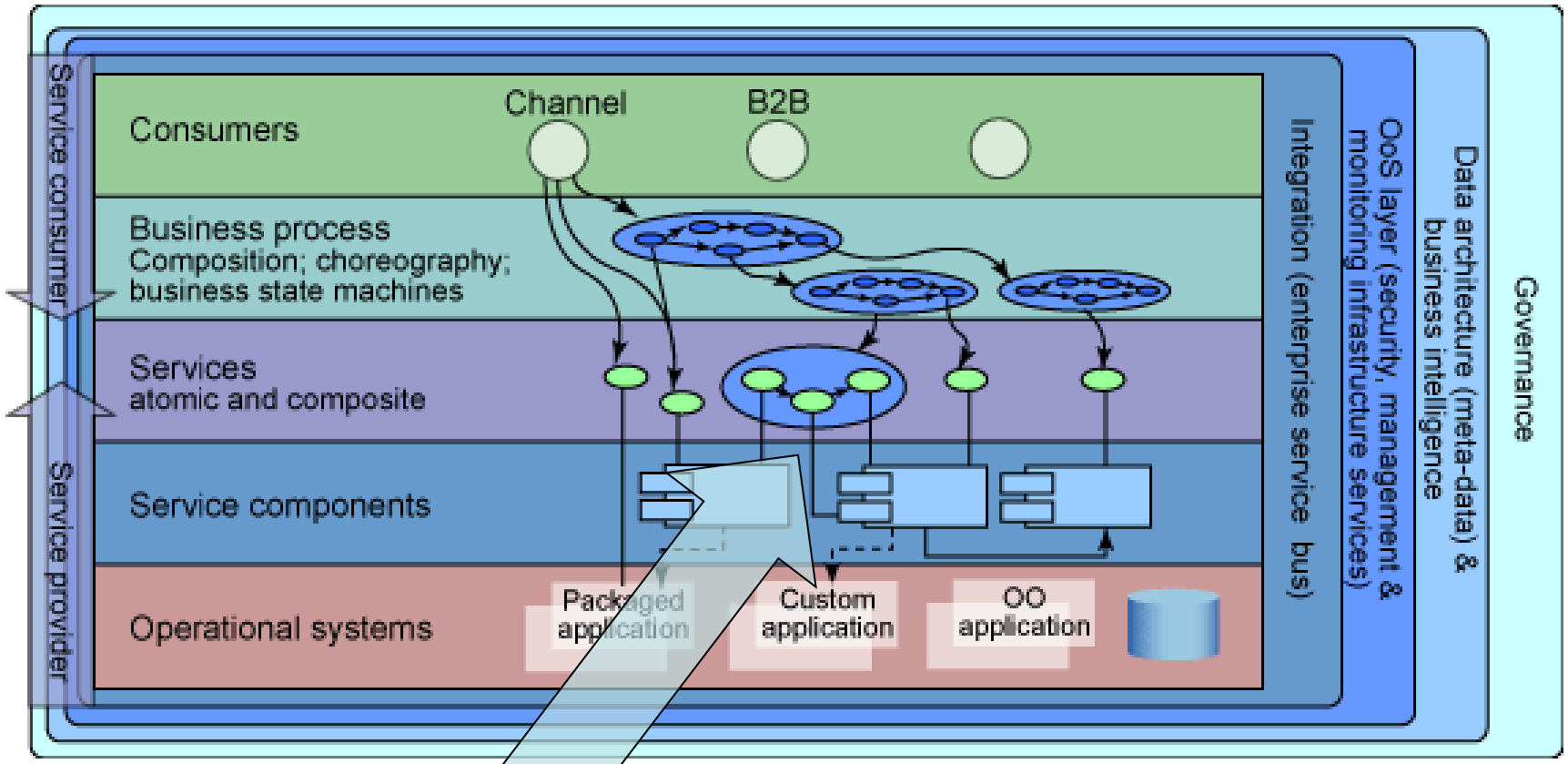
SOA book selling…



- Two separate SOA services are used instead
  - Many SOA advantages
  - Usually we cannot use ACID, all-or-nothing
  - So, if there is a problem executing *Add-to-till* after *Decr-inventory*, the **information will be inconsistent**
  - Often SOA implementations are naïve regarding this, there is no logic in place to deal with it

Another example, the Composite Service Pattern

- Very often Composite Service layers are included in architectures
- This is a picture from an IBM whitepaper, but many others use the same principles:



How about ACID, all-or-nothing?

Yields inconsistent data – what do we do?

- Need to find the best optimisation and a working compromise

- Can we design the service interfaces differently?
- Can we have a dialogue with the business experts to change the SOA service demands slightly?
- Can we use an extended process to complement the SOA service?

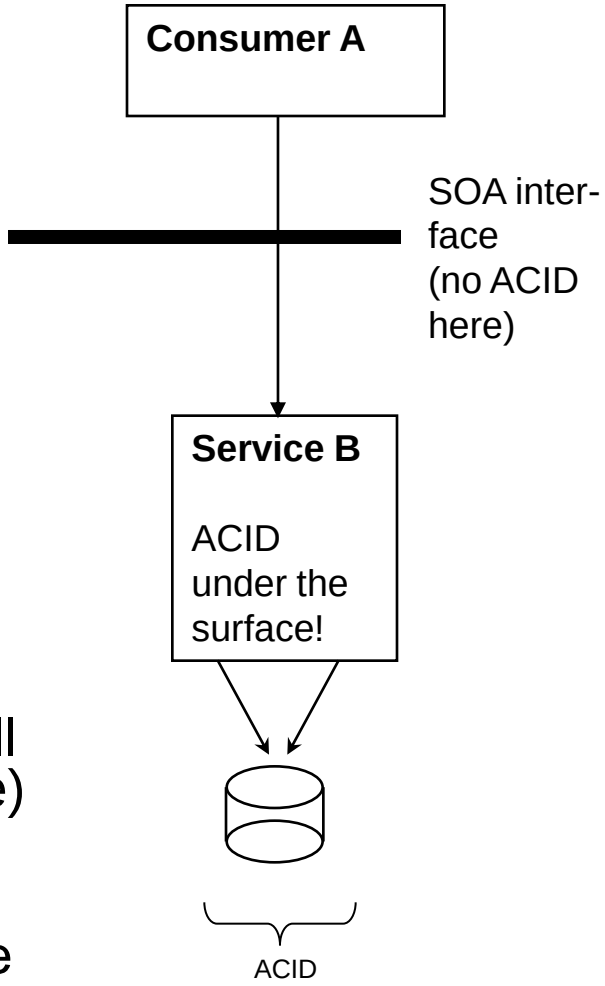
- In the following slides I propose a couple of solutions that can be used, when applicable.

Solution 1: ACID under the surface

- Design the service interface so that only one service invocation is needed, not several to fulfill a business process step
- Do use ACID under the surface, **inside** the service, where it works fine (usually same programming platform, same database etc)

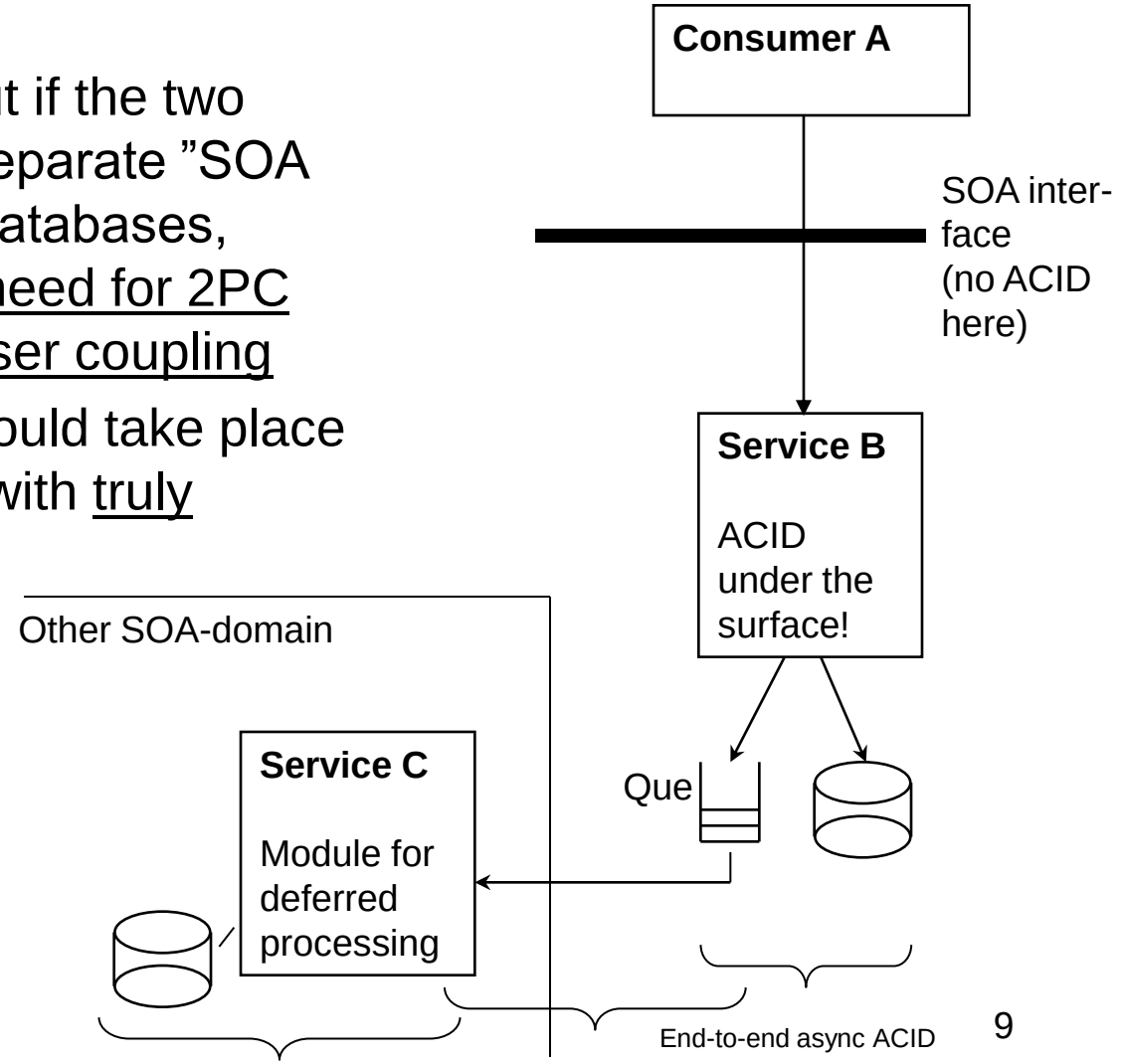
- Ex 1: Design *Bank_account_transfer* (do **not** use a separate withdrawal and deposit services)
- Ex 2: Design *Store_whole_invoice* including invoce head and invoice detail rows in one call (the hierarchical style of XML is excellent here)

- Thus, coarse granularity is ideal for service interfaces. Also, “SOA domains” should not be too small.



Solution 1a: BASE: ACID under the surface + async

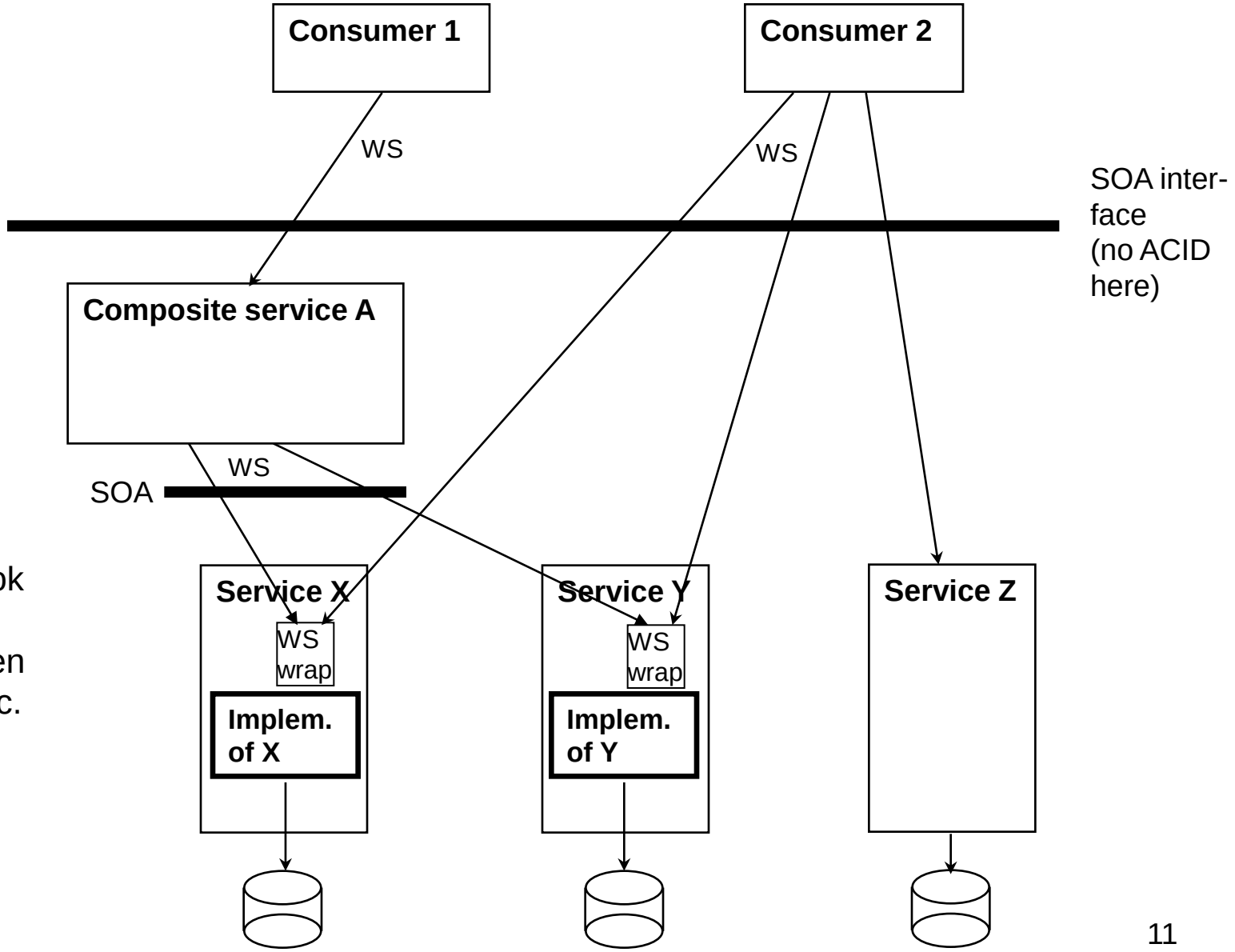
- Similar to solution 1, but if the two updates should be to separate ”SOA domains” or separate databases, solution 1a eliminates need for 2PC and provides much looser coupling
- Update to service C should take place via asynchronous que with truly guaranteed delivery
- Only usable if it is allowed that data may be less fresh in C
- Also called deferred processing or BASE



Balance: If there is ACID there should be BASE…

- BASE
  - Basically Available - Soft-state - Eventual consistency
  - Pattern along the lines that updates may be allowed to happen in a short while rather than exactly now
  - Deferred processing in solution 1a of the previous slide is an example of BASE
  - Also popular in the REST-sphere and when you need massive scalability and high availability
  - See for exemple http://queue.acm.org/detail.cfm?id=1394128

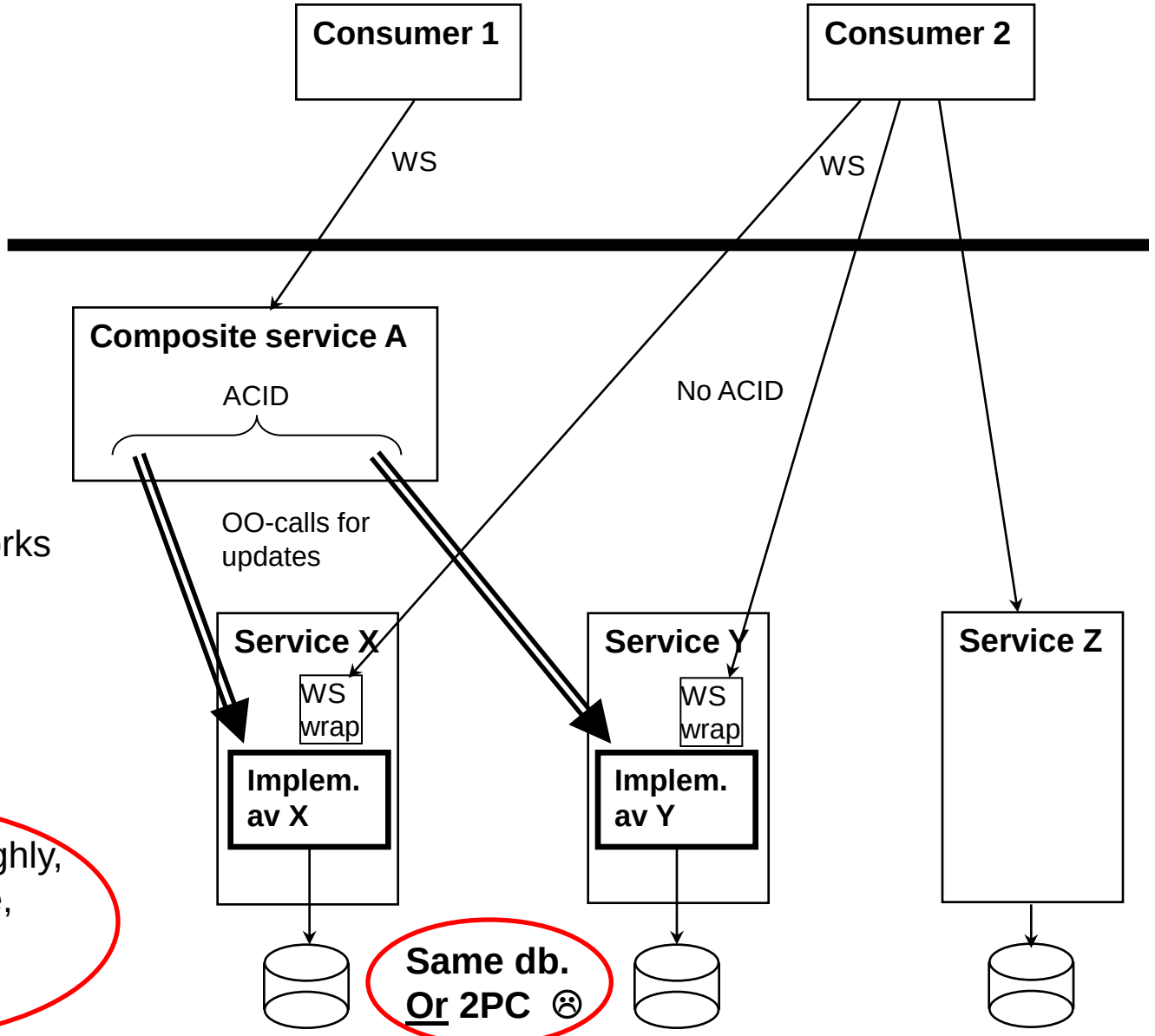
Solution 1b: For read-only, compose w/o ACID needs



Be aware of risk with composite services, they look so great in ppt and Visio but often ACID-problematic.

But composite read-only services are OK!

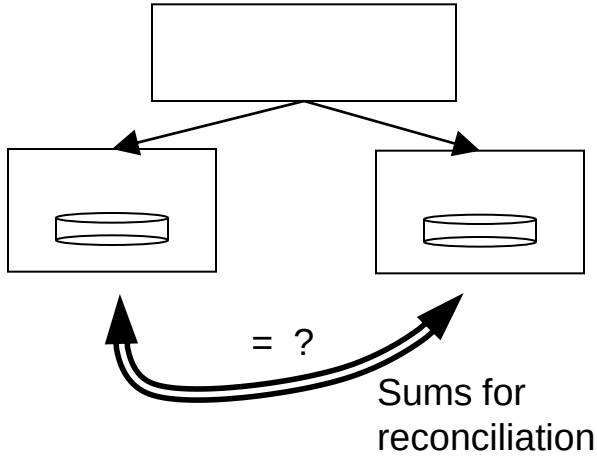
Solution1c: Cheat: Compose with ACID, ”less-SOA”



Typically only works if all the services A, X, and Y are using the same programming platform.

Consider thoroughly, not interoperable, may or may not work well!

Solution 2: Reconciliation

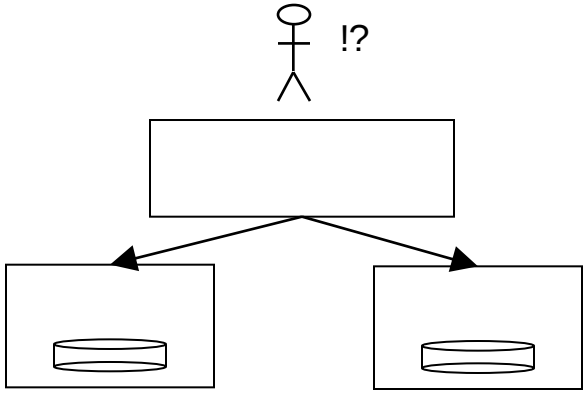


- Well-functioning 60:ies solution (or older…)
- Create a sum per batch, sum per day etc in both consumers and in services. Transfer these sums (preferably via another communication path).

- Check and tally (manually or automatically)
- If wrong, compensate (manually or automatically)

- Manual compensation may very well be optimal – this should happen seldom and troubleshooting may be needed to compensate in the right way. Thus, logs are needed in both consumers and services to make it possible to troubleshoot afterwards.

Solution 3: Show error info in user interface



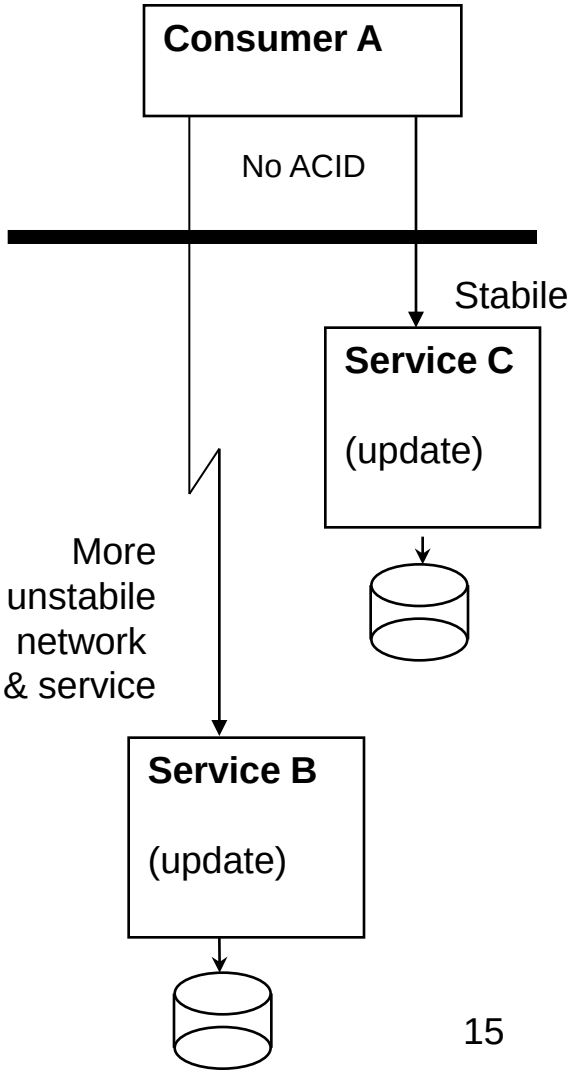
- If there is a user interface directly on top of a module that is consuming several services that contain updates without ACID:
  - Give an error message to the user that e.g. adding to the till ackumulator did not work but that decreasing the inventory did take place.
  - Try to write this event to the log for traceability afterwards. This is a reason why logs technically should be extremely simple so that the log-write doesn't crash as well.
  - Explain in the error text that the user has got to take care of this fault, compensate or act in some other fashion. Has to be understandable!

- Only works with well-motivated and trained users.

Solution 4: Optimized sequence + fault detection

- This pattern reduces the probability of faults, but has to be combined with other patterns.

- Optimized sequence:
  1. First, call the service that is the most probable to fail (B in the figure)
  2. If B fails, the consumer A should retry a number of times.
  3. B must cope with idempotency (eliminating duplicates etc) because of the retrying!
  4. If B has succeeded, call C. The probability should be low that it fails.
  5. Even so, if C faults, write to log, and fallback to one of the other patterns.



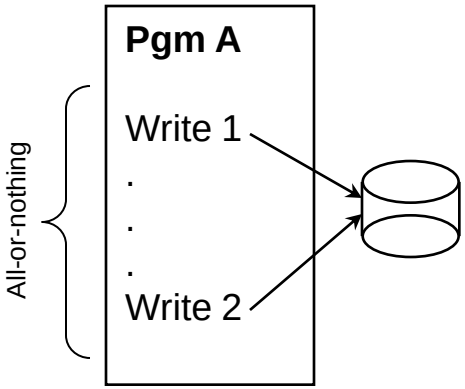
Solution 5: Controlled inconcistency

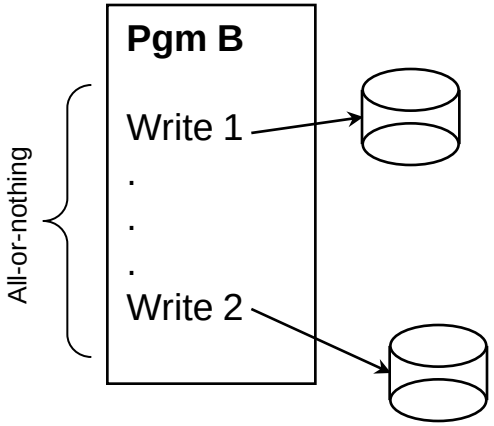
- Allow inconsistency, but only in a controlled fashion
  - Similar to the reconcililation pattern (solution 2) but more specific.
    - Example: If the monthly invoicing routine finds that there are invoice details regarding a customer number, but that this customer does not exist (due to an earlier update service fault), alarm the person reponsible for invoicing.
    - Logs are needed to allow for troubleshooting after such an alarm.
  - The data models inside the services must allow for orphans etc (less ER-constraints).
  - Expensive and complex exception handling – often manual routines

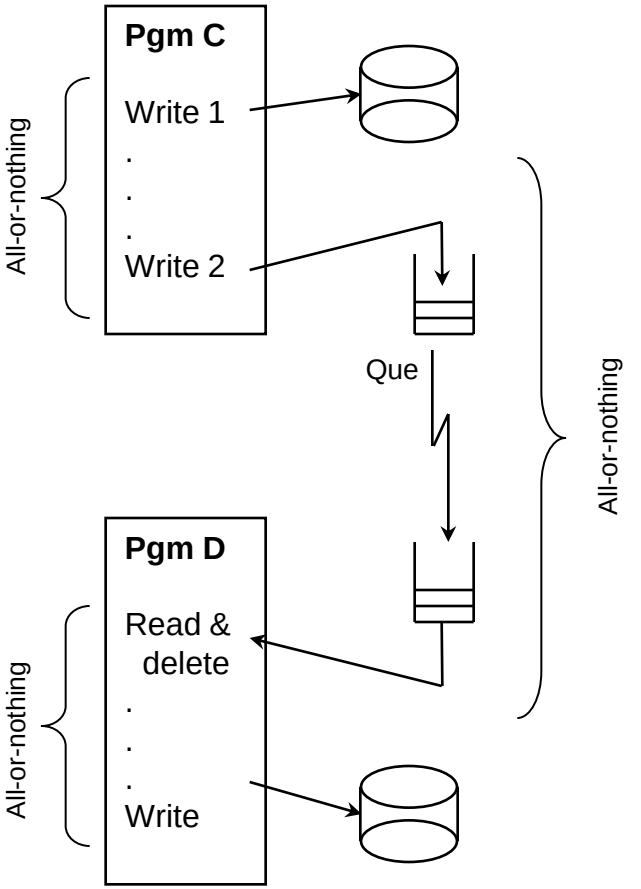
Lösning 6: Long-running transactions

- The pattern of long-running transactions has been used quite a lot in SOA.
- Such a transaction could go on for weeks until the final update or business event happens.
- This needs to be developed as a business process – a long-running transaction is not a technical term and is only distantly related to SQL transactions etc.
- Many of the above solutions (e.g. reconciliation) are actually special cases of long-running transactions
- Compensation schemes, needs to be programmed into the modules. Don't erase data, rather insert compensating data – to get traceability.
- Initiate comensation either manually or automatically
- It may be possible to use tools for workflow or orchestration
- Long-running transactions contain a lot of asynchronous exception handling which always is complicated, error-prone in itself, hard to test and expensive to design and develop. Therefore, always look at long-running transactions as the last resort!

True BASE needs <u>true</u> guaranteed delivery…







Simple

2PC Synchronous

<u>Asynchronous with true guaranteed delivery (end-to-end)</u>

To sum up:

- **Naïve use of SOA leads to information inconsistencies and lost data.** SOA architects sometimes does not take this problem seriously.

- Best solutions (because they are platform solutions to the problem):
  - 1. ”ACID under the surface” (coarse-granular service interfaces)
  - 1a. ”BASE” (deferred processing)
- Solutions 2-6 can be very useful, but they all push exception handling over to the business processes and thus to the users.

- Asynchronous patterns are more stabile but also lead to less fresh data.
- All exception handling that is distributed in time (async) is complicated, error-prone in itself, hard to test and expensive to design and develop. However, sometimes it must be used.
  Also many businesses intrinsically have business rules that need long-running transactions, e.g. credit card paying, hotel booking etc.

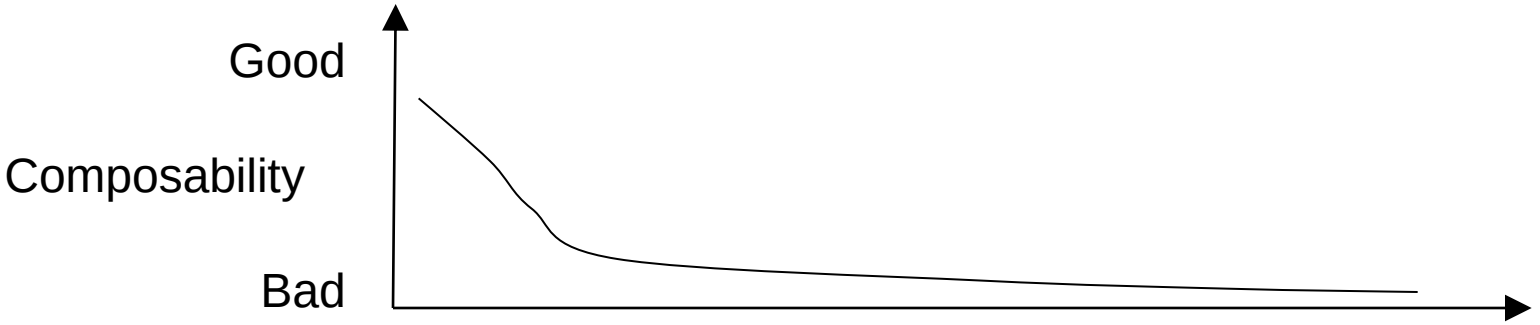
WS-* (WS-AtomicTransaction etc) is typically not the solution since all use of 2PC leads to problems in most scenarios

Composability and ACID

- The following three pages are taken from my session about Composability, SOA Symposium in Amsterdam, 2008
- For more info, and the whole presentation, see www.definitivus.se (click ”In English”)

1 The ACID problem for updating services

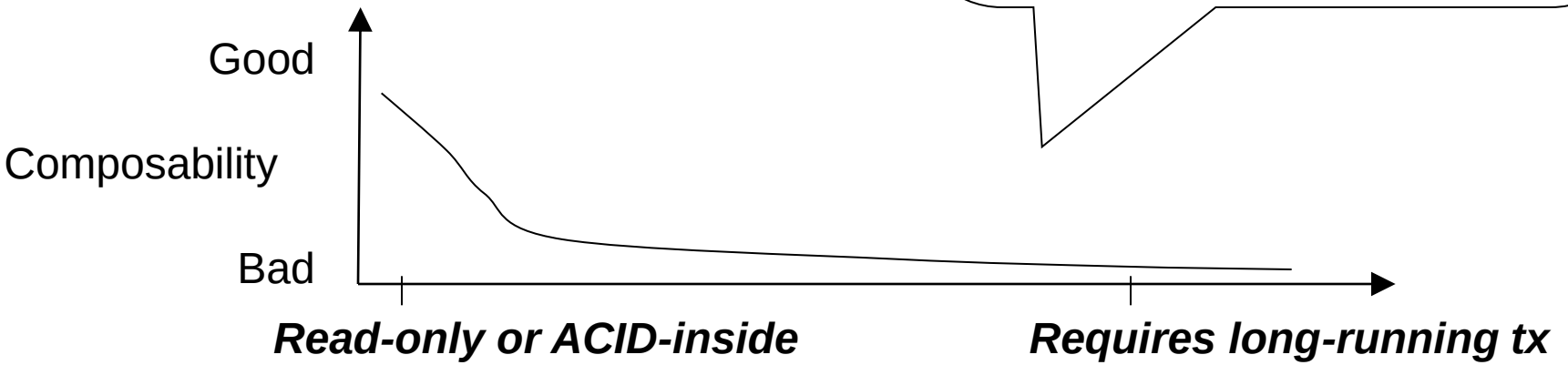
- Information fidelity and consistency cannot be guaranteed if we do not do ACID updates. Since SOA ACID normally would be too tightly coupled, we may have to take care of update errors in a couple of days (via long-running transactions).
  - Either
    - Good composability: Read-only services. Or update services that include the ACID needs inside of them.
  - Or:
    - Updating services that are likely to be used together with other updating services, thus requiring employment of long-running transactions (costly, complex, less user-friendly).



1 The ACID problem for updating services

Better to make updating services that are larger, so that they can contain needed ACID handling inside the service (inside, it is OK to use tight coupling that often wouldn't be OK for SOA).
E.g.: One single SOA interface invocation for updating invoice-head together with all invoice-rows)

Long-running business transactions caused by the absence of ACID should be kept to a minimum, but cannot always be avoided.
Cumbersome because long-running transactions increase the volume of needed business logic and because personell usually have to handle the compensation.
Also, meanwhile the long-running tx is pending, the users must be informed that the info is only preliminary

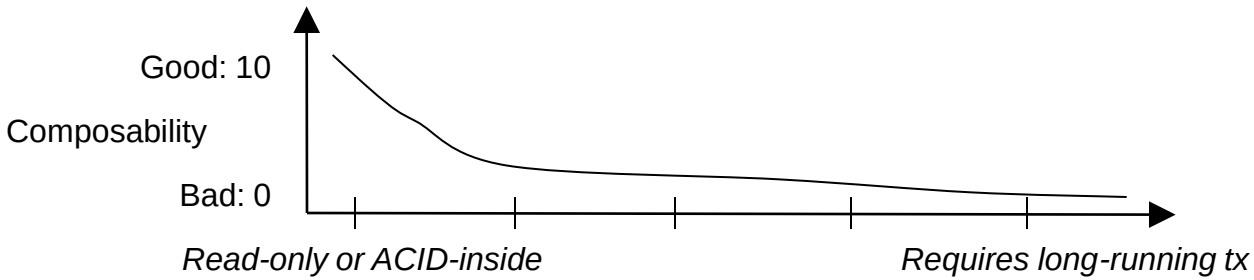


ACID = Atomicity, Consistency, Independecy, Durability. I.e. all-or-nothing updating or Unit-of-work updating..

Questionaire sample for the ACID problem

- Check one of the following:

  - ❑ This is a read-only service interface.
    Or, all conceivable updates that should be kept together, are kept together inside the service, through internal ACID — ”Good-weight”: 10
  - ❑ Internal ACID is used for related updates that should be kept together, but at rare times, related info is expected to have to be updated via another service ”at the same time” — ”Good-weight”: 4
  - ❑ Internal ACID is used for related updates that should be kept together, but sometimes, related info is expected to have to be updated via another service ”at the same time” — ”Good-weight”: 3
  - ❑ Internal ACID is used for related updates that should be kept together, but often, related info is expected to have to be updated via another service ”at the same time” — ”Good-weight”: 1
  - ❑ No internal ACID is used, several service invocations have to be carried out to complete update of related info. — ”Good-weight”: 0



Sven-Håkan Olsson is an independent consultant, course leader and speaker who focuses on application architecture and SOA. Since 1977 he has worked in a large number of IT development projects, ranging from for embedded microcontrollers to main-frames. He has carried out modeling, architecture design and programming in diverse business areas. He has also specialized in reviews of problem projects.

Sven-Håkan Olsson holds an MScEE from the Royal Institute of Technology in Stockholm. In 2008 and 2010 he was appointed one of the 'top developers in Sweden' by the magazine IDG Computer Sweden. He is also a co-founder of the public consultancy company Know IT.

Please feel free to contact me if you have comments or suggestions!

D▪E▪F▪I▪N▪I▪T▪I▪V▪U▪S

Tel +46 708 840134
sven-hakan.olsson[.]definitivus.se
www.definitivus.se