**Introduction**

Scalability

Transaction and consistency models and theory behind them

Message based architectures

Tribe

**Performance and Scalability**

Performance vs scalability

Vertical vs horizontal

*Vertical scalability graph – response time vs concurrent users*

*Horizontal scalability graph – resources / cost vs concurrent users*

**ACID Transaction Model**

Most common transaction model, all major relational databases use ACID.

Atomic

Consistent

Isolated

Durable

Atomic

Atomicity requires that each transaction is "all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

Consistent

The consistency property ensures that any transaction will bring the database from one valid state to another.

Isolated

Concurrent transactions should behave as if each were the only transaction running in the system.

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other.

Durable

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

Locking

Isolation levels

* Serializable
* Repeatable reads
* Read committed (SQL Server default)
* Read uncommitted

|  |  |
| --- | --- |
| **Transaction 1** | **Transaction 2** |
| */\* Query 1 \*/*  **SELECT** \* **FROM** users **WHERE** id = 1; |  |
|  | */\* Query 2 \*/*  **UPDATE** users **SET** age = 21 **WHERE** id = 1;  COMMIT; |
| */\* Query 1 \*/*  **SELECT** \* **FROM** users **WHERE** id = 1;  COMMIT; |  |

Let’s look at read committed and serializable as examples.

**Read committed** takes out “**read locks**” (data can be read by other transactions, but not written to) on data that is selected in a transaction, but only for the **duration of that select statement**. The data returned in the second select statement can be different from the first. This is known as a “**non-repeatable read**”.

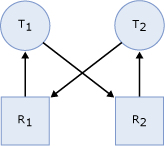
**Serializable** acquires read locks for the **duration of the transaction**, as well as “**range locks**” that prevent inserted data from being read. This ensures that the selected data is consistent for the duration of transaction 1, but the update from **transaction 2 is not committed until after transaction 1 commits**.

This is an oversimplified example, of course, the queries that form your stored procedures, or the batch statements generated by your ORM, or whatever, will be significantly more complex.

Lock Escalation

“Lock escalation is the process of converting many fine-grain locks into fewer coarse-grain locks, reducing system overhead while increasing the probability of concurrency contention.” – technet.microsoft.com

Deadlocks



A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock.

* Task T1 has a lock on resource R1 (indicated by the arrow from R1 to T1) and has requested a lock on resource R2 (indicated by the arrow from T1 to R2).
* Task T2 has a lock on resource R2 (indicated by the arrow from R2 to T2) and has requested a lock on resource R1 (indicated by the arrow from T2 to R1).
* Because neither task can continue until a resource is available and neither resource can be released until a task continues, a deadlock state exists.

Modern database engines have deadlock detection and will fail one or the other transaction. Lower isolation levels will reduce the occurrence of deadlocks.

Vertical Scalability

In terms of vertical scalability, the number of concurrent users we can support is significantly affected by the isolation level. Using serializable transactions where the same set of data is being read and written concurrently will hit a performance wall quickly.

Horizontal Scalability

Replication is generally reserved for disaster recovery scenarios. Distributed transactions can facilitate multiple consistent copies of a database, but introducing network latency into transactions is a recipe for disaster.

Sharding is where we split data into more than one physical database. We can do this in any arbitrary manner, such as by geographical region (Azure supports sharding like this), first letter of customer’s last name, user category, whatever.

Any data that is shared across more than one shard must be kept synchronised through some form of replication, product prices, user profiles, etc. We are making a conscious decision to accept the risk that the data we are working with might not be “consistent” at any given point in time.

**CAP Theorem**

The theorem began as a [conjecture](http://en.wikipedia.org/wiki/Conjecture) made by computer scientist [Eric Brewer](http://en.wikipedia.org/wiki/Eric_Brewer_(scientist)) from the [University of California, Berkeley](http://en.wikipedia.org/wiki/University_of_California,_Berkeley) at the 2000 [Symposium on Principles of Distributed Computing](http://en.wikipedia.org/wiki/Symposium_on_Principles_of_Distributed_Computing) (PODC).[[5]](http://en.wikipedia.org/wiki/CAP_theorem#cite_note-Brewer-5) In 2002, [Seth Gilbert](http://en.wikipedia.org/w/index.php?title=Seth_Gilbert&action=edit&redlink=1) and [Nancy Lynch](http://en.wikipedia.org/wiki/Nancy_Lynch) of [MIT](http://en.wikipedia.org/wiki/MIT) published a formal proof of Brewer's conjecture, rendering it a [theorem](http://en.wikipedia.org/wiki/Theorem).[[1]](http://en.wikipedia.org/wiki/CAP_theorem#cite_note-Lynch-1) This last claim has been criticized however.[[6]](http://en.wikipedia.org/wiki/CAP_theorem#cite_note-Burgess-6)

* Consistency (all nodes see the same data at the same time)
* [Availability](http://en.wikipedia.org/wiki/Availability) (a guarantee that every request receives a response about whether it was successful or failed)
* [Partition tolerance](http://en.wikipedia.org/w/index.php?title=Network_partitioning&action=edit&redlink=1) (the system continues to operate despite arbitrary message loss or failure of part of the system)

*Image of replicated service*

The theorem sets up a scenario in which a replicated service is presented with two conflicting requests arriving at distinct locations on a time when a link between them is failed. The obligation to provide availability despite partitioning failures leads the services to respond; at least one of these responses shall necessarily be inconsistent with what a service implementing a true one-copy replication semantic would have done.

In a general sense, partition tolerance refers to our ability to replicate our services over multiple nodes. In fact, the couchdb guide on consistency defines partition tolerance to mean “The database can be split over multiple servers”.

Sacrificing availability is clearly not an option, so to achieve partition tolerance, we need to look at an alternate consistency model, the most popular of which is “eventual consistency”.

**Eventual Consistency**

Eventually consistent services generally operate under what are called “BASE” semantics.

Basically Available, Soft state, Eventually consistent

Basically Available

The system guarantees availability, where availability has the same meaning as that of the CAP theorem.

Soft State

From Wikipedia: “In computer science, soft state is [state](http://en.wikipedia.org/wiki/State_(computer_science)) which is useful for efficiency, but not essential, as it can be regenerated or replaced if needed.”

When we’re talking about services, other than cached data, the service is stateless.

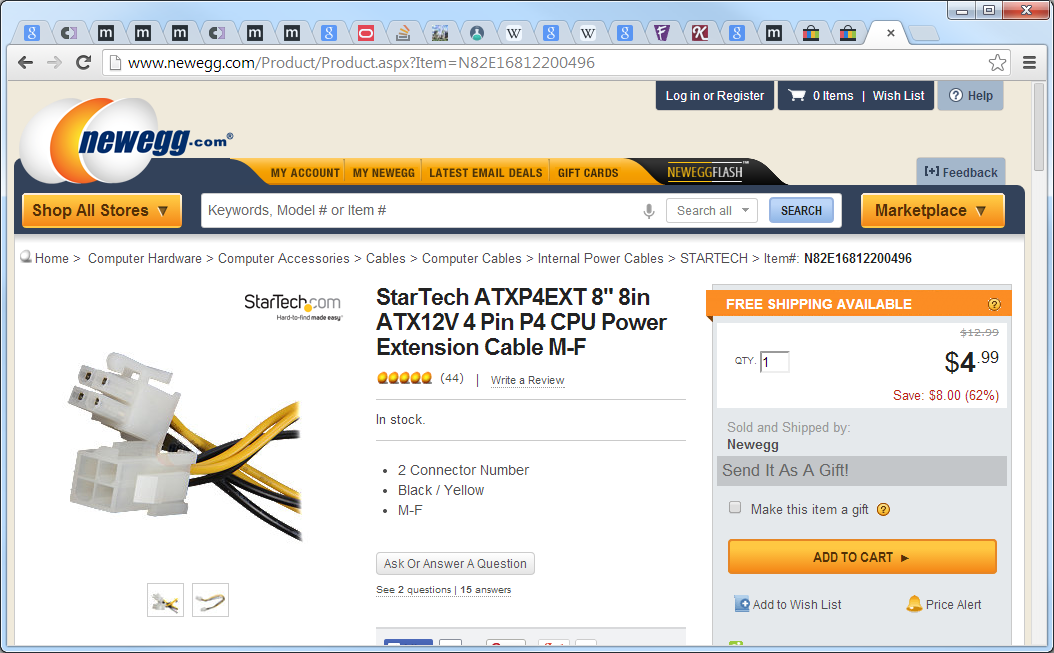
Eventually Consistent

We’ve already talked about ways in which we can partition our data or services at the cost of accepting the risk that some of the data might not be consistent. In fact, whenever we are talking about a distributed system that is not directly connected to a data source, we are accepting that the data we are displaying on the screen has some degree of “staleness”. Cached data has an additional degree of staleness.

We use concurrency mechanisms to deal with stale data. Optimistic concurrency rejects writes that do not target the most recent version of data.

So… if we’re already accepting that our data might not be an accurate representation of the world, let’s look at things a bit closer.

**Example: stock levels on online shopping site**

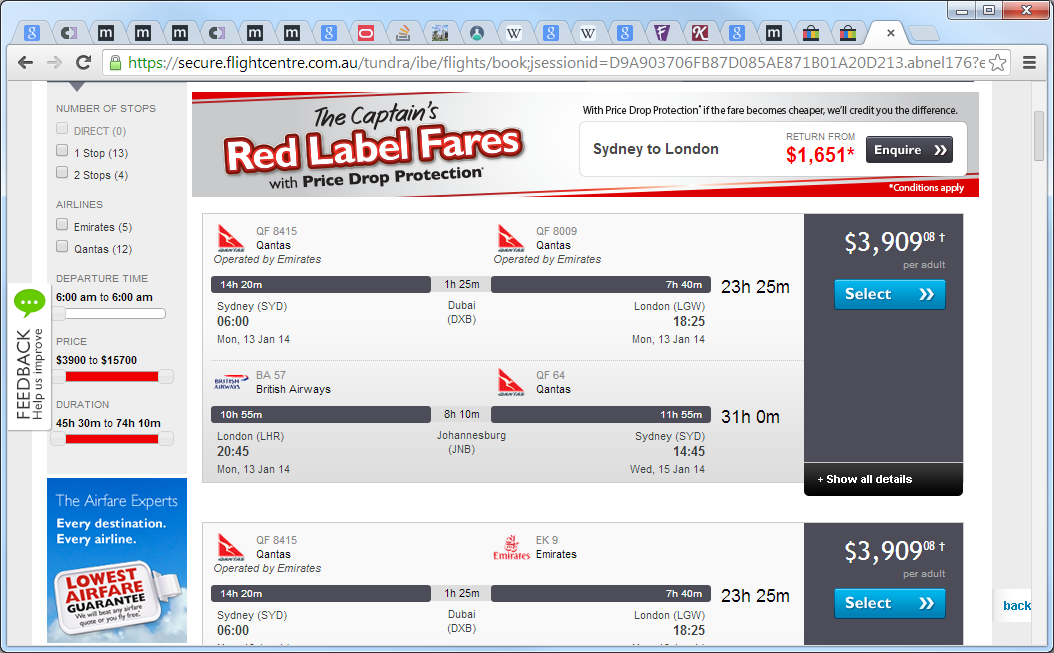


Do you really want to make your customers wait an extra few hundred milliseconds (a second? two?) before they see your hottest product while all these transactions resolve, just so that you can tell them there are exactly 4,972 units in stock?

Let’s make an educated decision about this. How often are we getting orders for a product at peak times? Let’s show an approximate number instead, or even . So if we’ve got a product with 10 units left in stock and we get an order on average every second, we better make sure our product servers are updated at least every 10 seconds.

OK, this is a massive and unrealistic over-generalisation, but you get the idea.

**Example: airline booking**



Booking sites like flightcenter.com – these sites aren’t allocated their own seats. Clearly the data they display on sites like this is not going to be guaranteed accurate. We merely want to ensure the data is accurate enough that it’s unlikely the seat has been taken and start the booking process. It’s not until the actual reservation occurs that conflicts are resolved.

Resolving Conflicts

Let’s go back to our stock level example. In the unlikely event that we get 11 orders in 10 seconds, all is not lost. Again, let’s make an educated decision. Let’s email the customer and tell them the item is on back order, maybe suggest a substitute item.

We can even make a decision about which customers win out in this scenario. Loyal, returning customers, first time customers, invest the effort that will give you maximum ROI.

We’re talking here about resolving conflicts of business rules. If we were talking about infrastructure concerns such as resolving conflicts with data replicated across multiple shards, you can take a different approach. First write wins, last write wins, etc., whatever is most appropriate.

**Message Based Architectures**

What are we talking about? Also known as event driven architectures, I prefer the term message based as it is useful to draw a distinction between “events” and “commands”. NServiceBus actually provides two distinct interfaces to represent events and commands. We’ll talk about some of the reasons why in the next few sections.

Events

An event can be defined as "a significant change in [state](http://en.wikipedia.org/wiki/State_(computer_science))".

- <http://en.wikipedia.org/wiki/Event-driven_architecture>

“For example, when a consumer purchases a car, the car's state changes from "for sale" to "sold". A car dealer's system architecture may treat this state change as an event whose occurrence can be made known to other applications within the architecture.”

A publisher of an event has no awareness of any subscribers to the event.

Commands

An asynchronous request for a specific action to be performed by a receiver.

Commands are asynchronous – we do not expect a response from them. We make an assumption that the command is valid based on the information the client has at the time the request is made. If the command fails, we need to perform conflict resolution, or take compensating action, as we discussed before.

Messages

To transfer these two distinct types of communication between clients and server-side components, they are encapsulated within messages. A message envelope typically contains the type or “topic” of the message, the message payload itself and other pieces of metadata such as a unique identifier, sequence number, the originator, etc.

Guaranteed delivery

When a client publishes a message to a message bus, any resulting operations are performed asynchronously; we do not expect an immediate response. Instead, the message bus provides an acknowledgement that the message has been received, and along with that, a guarantee that at some point, the message will be delivered to any recipients.

This generally involves some sort of persistent storage and transport mechanism for messages, such as MSMQ, Azure queues, RabbitMQ, etc, etc. NServiceBus uses MSMQ out of the box, but can be configured to use other transport mechanisms through the use of plug-ins.

Error handling

Error handling is different from a failed command. It occurs in genuinely exceptional circumstances, for example, a coding error or bug, persistent network failure, etc.

Messages that cause exceptions are placed into a holding area so that the issue can be fixed and the message reprocessed at a later time.

Transactions / idempotence

So what happens if an error occurs half way through processing a message involving more than one atomic operation?

Distributed transactions

MSMQ supports transactional queues through MSDTC. NServiceBus allows you to seamlessly wrap your operations in distributed transactions with minimal configuration. This is clearly going to have a significant impact on throughput, though this may be acceptable as the operation is occurring asynchronously.

Idempotence

“Idempotence is the property of certain operations in [mathematics](http://en.wikipedia.org/wiki/Mathematics) and [computer science](http://en.wikipedia.org/wiki/Computer_science), that can be applied multiple times without changing the result beyond the initial application” - <http://en.wikipedia.org/wiki/Idempotence>

Having message handlers that can be rerun more than once with the same outcome allows us to reprocess messages that caused exceptions without the need for transactions. While it is a highly desirable characteristic of message handlers, it is not always possible to achieve.

Loose coupling

* Kind of “inversion of control”
* compare to “traditional” architectures
* components depend explicitly on other components and need to call out to them – coupled to the API
* **Example / diagram**
* Real advantage is to be able to add and remove subscribers without to publisher knowing / caring
* as subscribers grow, remember SRP

User interface design

A core element of designing message based systems is rethinking the design of the user interface to enable us to capture our users’ intent, for example, that making a customer preferred is a different unit of work for the user than indicating that the customer has moved or that they’ve gotten married. Using an Excel-like UI for data changes doesn’t capture intent.

**CQRS**

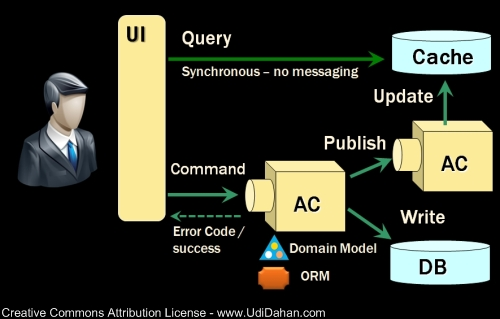
While a simple message based architecture can offer benefits over more traditional architectures, we can continue to build on the abstractions we’ve talked about.

Before describing the details of CQRS we need to understand the two main driving forces behind it: collaboration and staleness.

Collaboration refers to circumstances under which multiple actors will be using/modifying the same set of data – whether or not the intention of the actors is actually to collaborate with each other. There are often rules which indicate which user can perform which kind of modification and modifications that may have been acceptable in one case may not be acceptable in others. We’ll give some examples shortly. Actors can be human like normal users, or automated like software.

Staleness refers to the fact that in a collaborative environment, once data has been shown to a user, that same data may have been changed by another actor – it is stale. Almost any system which makes use of a cache is serving stale data – often for performance reasons. What this means is that we cannot entirely trust our users decisions, as they could have been made based on out-of-date information.

Standard layered architectures don’t explicitly deal with either of these issues. While putting everything in the same database may be one step in the direction of handling collaboration, staleness is usually exacerbated in those architectures by the use of caches as a performance-improving afterthought.



<http://www.udidahan.com/wp-content/uploads/cqrs.png>

Separate read / write models

Cache can be highly denormalised, even match our view models 1-1. This allows us to remove relationships from the structures and use other non-relational, high performance data stores like Azure Storage, mongodb, etc.

Since we’re accepting that our cache is stale, we can also scale out our cache to multiple instances and use the same update mechanism to update all instances.

Authoritative database can also be on the technology of choice. Relational databases are a reasonable choice here due to their strong consistency (ACID).

**Event Sourcing**

“Capture all changes to an application state as a sequence of events.”

The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application state itself.

Consider a shipping application that tracks the location of ships as they travel from port to port. The ship sends a notification each time it arrives or departs a port and we can update the state of a ship object to represent the new state.

Additionally we store the events that were sent by each ship. The obvious benefit we have is that we now have a log of each ship movement, but this is a relatively small gain. We could have quite simply kept a log by other means.

The key to event sourcing is that we guarantee that all changes to the domain objects are initiated by the event objects. This leads to a number of facilities that can be built on top of the event log:

* Complete Rebuild: We can discard the application state completely and rebuild it by re-running the events from the event log on an empty application.
* Temporal Query: We can determine the application state at any point in time. Notionally we do this by starting with a blank state and rerunning the events up to a particular time or event. We can take this further by considering multiple time-lines (analogous to branching in a version control system).
* Event Replay: If we find a past event was incorrect, we can compute the consequences by reversing it and later events and then replaying the new event and later events. (Or indeed by throwing away the application state and replaying all events with the correct event in sequence.) The same technique can handle events received in the wrong sequence - a common problem with systems that communicate with asynchronous messaging.

Reversing events

As well as events playing themselves forwards, it's also often useful for them to be able to reverse themselves. We can do this with any arbitrary event sourcing system by create a new instance of an object and rebuilding it from our event store to the desired point in time, but this can be expensive.

We could also implement a reversal method for each type of event that the target consumes. This is often a valid approach but obviously requires significant additional effort to implement the reverse of each event handler.

Epochs

“A point in time beginning a new or distinctive period” – Collins English Dictionary

So instead of recreating our object from scratch each time we want to rebuild it, let’s take a snapshot of the target object at certain intervals (epoch) and rebuild our object from the epoch plus events that occurred since that epoch.

**Epoch image**

**Composite UIs**

Composite pattern

The composite pattern describes that a group of objects are to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

Composite View

The Composite View design pattern describes a recursive UI structure of views containing children that are themselves views. The views are then composed by a mechanism—usually at run time, in contrast to being statically composed at design time.

**Image of yoursports.net broken into panes**

MVVM

Model View ViewModel is an [architectural pattern](http://en.wikipedia.org/wiki/Architectural_pattern) used in software engineering that originated from [Microsoft](http://en.wikipedia.org/wiki/Microsoft) as a specialization of the Presentation Model design pattern introduced by [Martin Fowler](http://en.wikipedia.org/wiki/Martin_Fowler).

Presentation model

“Represent the state and behaviour of the presentation independently of the GUI controls used in the interface” - http://martinfowler.com/eaaDev/PresentationModel.html

Presentation Model may interact with several domain objects, but Presentation Model is not a GUI friendly facade to a specific domain object. Instead it is easier to consider Presentation Model as an abstract of the view that is not dependent on a specific GUI framework.

“The view model of MVVM is a value converter, meaning that the view model is responsible for exposing the data objects from the model in such a way that those objects are easily managed and consumed.” - <http://en.wikipedia.org/wiki/Model_View_ViewModel>

Data binding



Communicating between components

We have two options for communicating between individual components in a composite UI.

Composite UI frameworks such as Microsoft Prism and Tribe provide “event aggregation” mechanisms in the form of a publish / subscribe message bus.

Alternatively, we can share “observables” between components by passing them to child components when constructing our UI. This is less desirable as it introduces couplings between components, sometimes of a subtle nature.

**SOA**

A discussion of SOA is beyond the scope of this presentation, but it is worth noting that the principles discussed above are ideal for construction of service oriented architectures.

It is simple to construct autonomous components, including user interface components, that are owned by an individual service that communicate with published services, or even publish events directly to an enterprise service bus. The user interface components can be stitched together into “mashups”.

**Tribe**

Intro

* why?
* focused around reducing friction
* Pure JavaScript
* run the same code on both the client and server
* such code needs to be constrained and given a common environment
* highly testable!
* Ultimate debugging experience
* Simple and powerful deployment
* Purely static resources
* CDN
* Caching

Components

* PubSub
* Composite
* Dependencies
* Forms
* Components
* Server side
* Previously SignalR, now nodejs based
* Pure JavaScript
* Dependencies
* PackScript

**Learn More**

<http://www.udidahan.com/wp-content/uploads/Clarified_CQRS.pdf>

<http://martinfowler.com/eaaDev/EventSourcing.html>