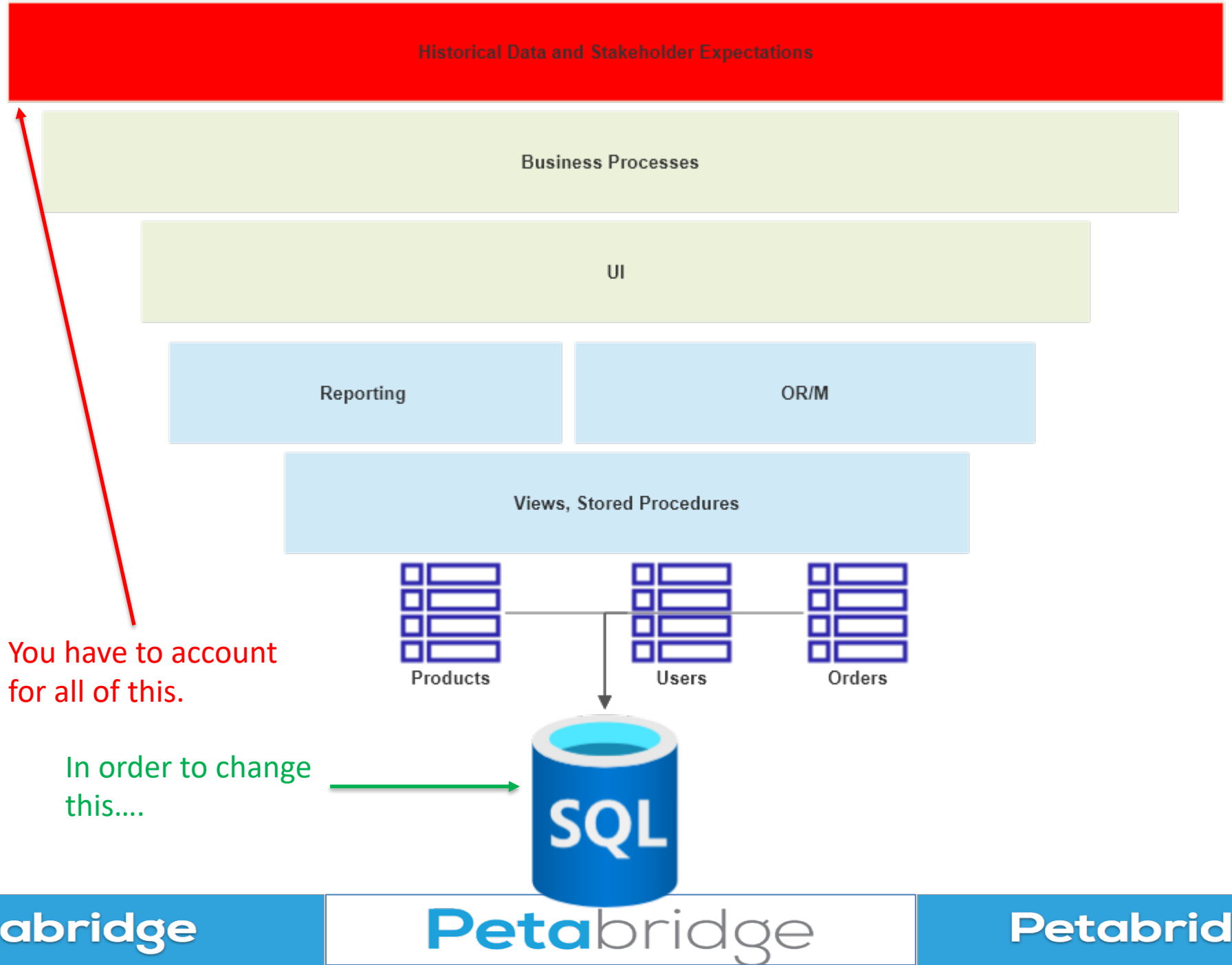
High Optionality Programming

Software Architectures that Reduce Technical Debt

Aaron Stannard, Founder of
Akka.NET and Petabridge

Source: https://vincentdnl.com/drawings/


Historical Data and Stakeholder Expectations
Business Processes
UI
Reporting
OR/M
Views, Stored Procedures
Products
Users
Orders
You have to account for all of this.
In order to change this….
SQL

Technical Debt

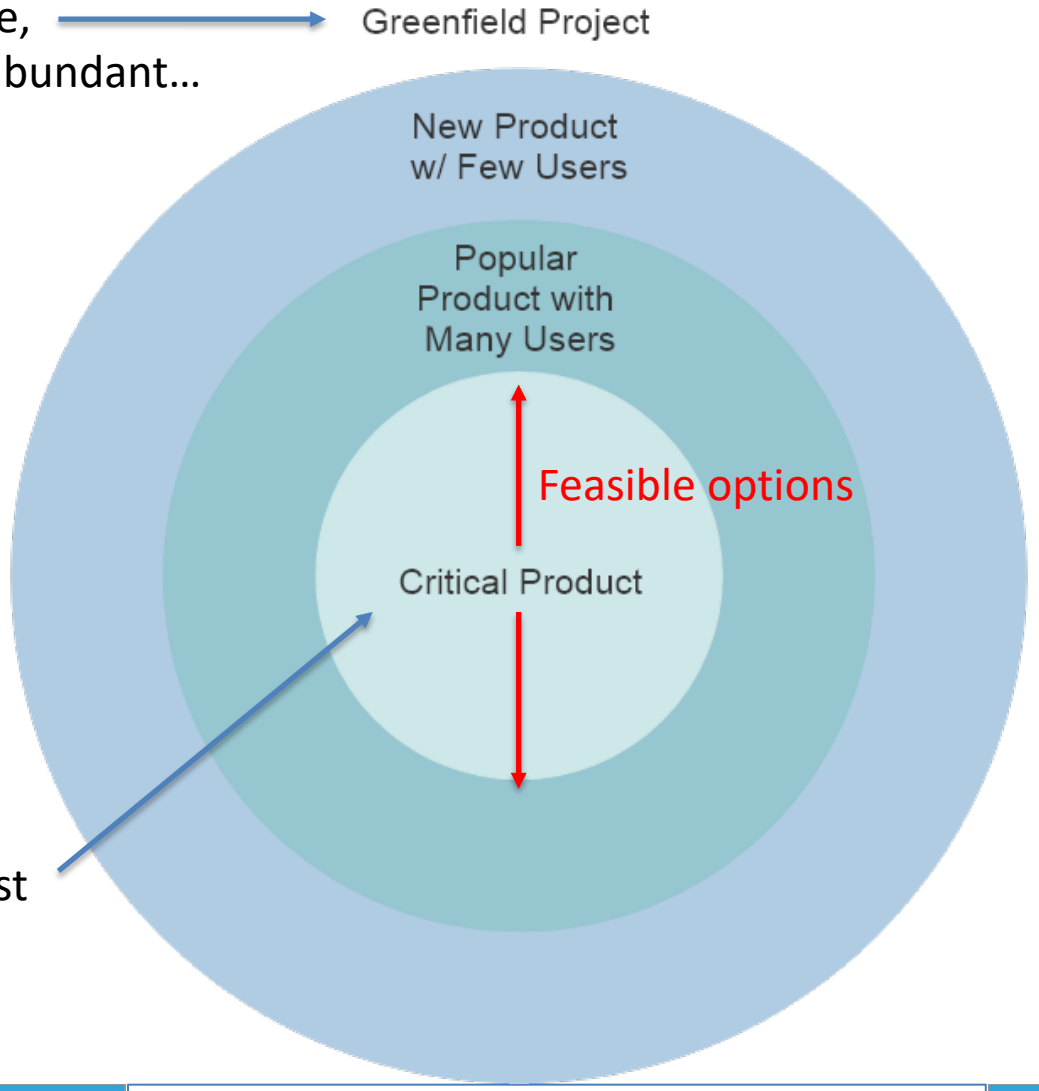
- Initial technical debt == cost incurred from previous software design and implementation choices;
- Interest compounds in the form of layering;
- Full cost of technical debt is not known until a future need to modify the system arrives; and
- Cost to make that modification == actual technical debt.

"To have options."

OPTIONALITY

Options in Technology

Pay a premium here, when options are abundant…



To avoid having to pay a very high cost to expand options here later.

Technical Debt == Destruction of Options

- Technical debt occurs when you fail to anticipate the evolution of software
- i.e. Database-Driven Development
  - "Who ever needs to switch databases?"
  - "Repositories are unnecessary abstractions"
  - "Why bother with an OR/M layer? Just use stored procedures"
- Future technical debt can be reduced by planning for possible future changes today

Case Study: MarkedUp (2012)

Low Optionality

MARKED UP

- Real-time analytics startup
- Used RavenDb + MapReduce indices
- Went from 10,000 events per day to 5-8m per day in ~3 days
- RavenDb couldn't keep up and went stale
- Decided not to use repositories – Db logic was spread out everywhere
- Thus: had to change everything all at once (high risk)
- Highly dependent on Db-specific constructs

Case Study: MarkedUp (2013-14)

High Optionality

MARKED UP

- Created event-driven processing model
- Created middleware that could translate events into analytic deltas
- Separated read & write models into discrete services
- Added unit tests back to suite (had to be 100% integration tests before)
- Created DSL that allowed dynamic per-user filtering of events
- Greatly improved developer throughput

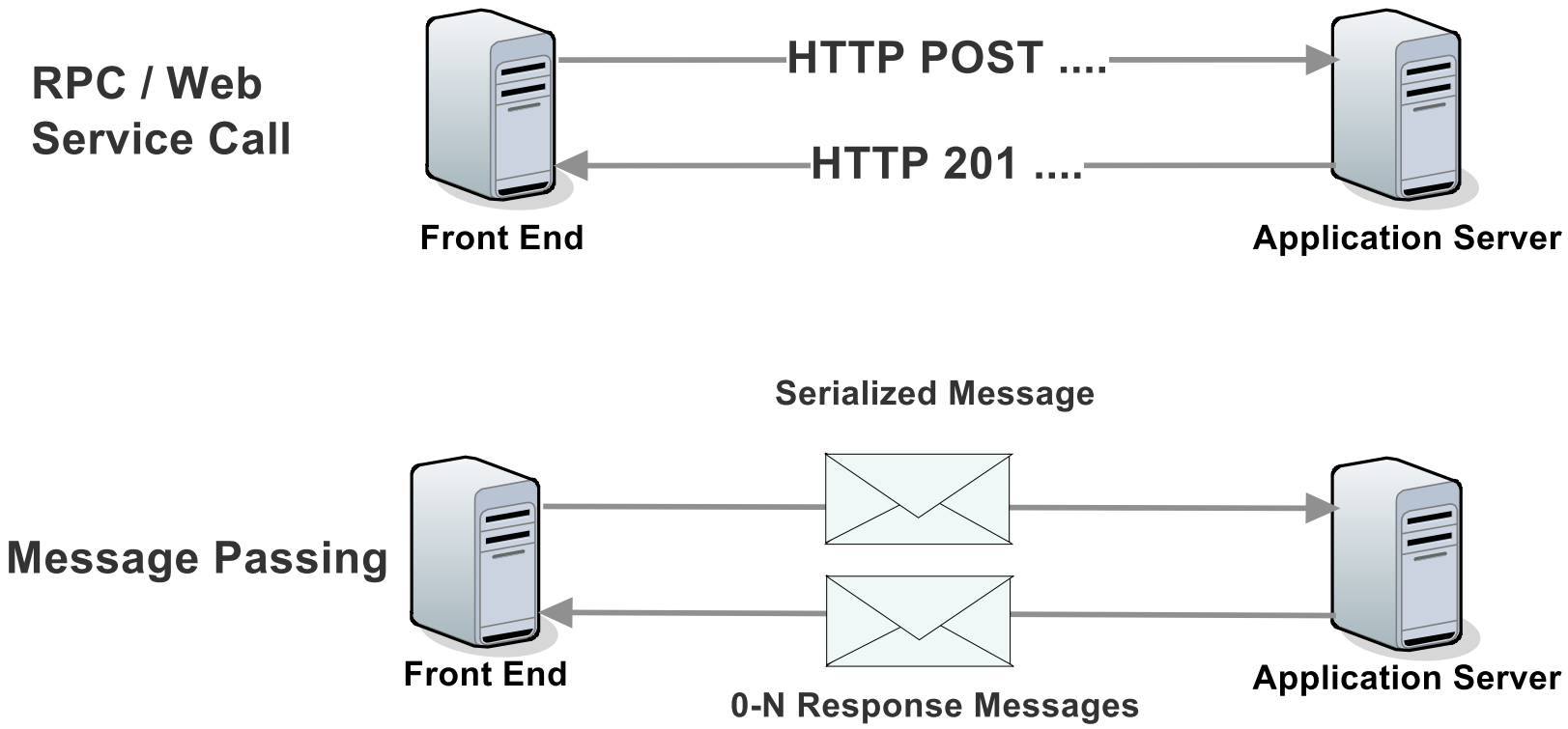
Architecture models that preserve future choices.

HIGH OPTIONALITY PROGRAMMING

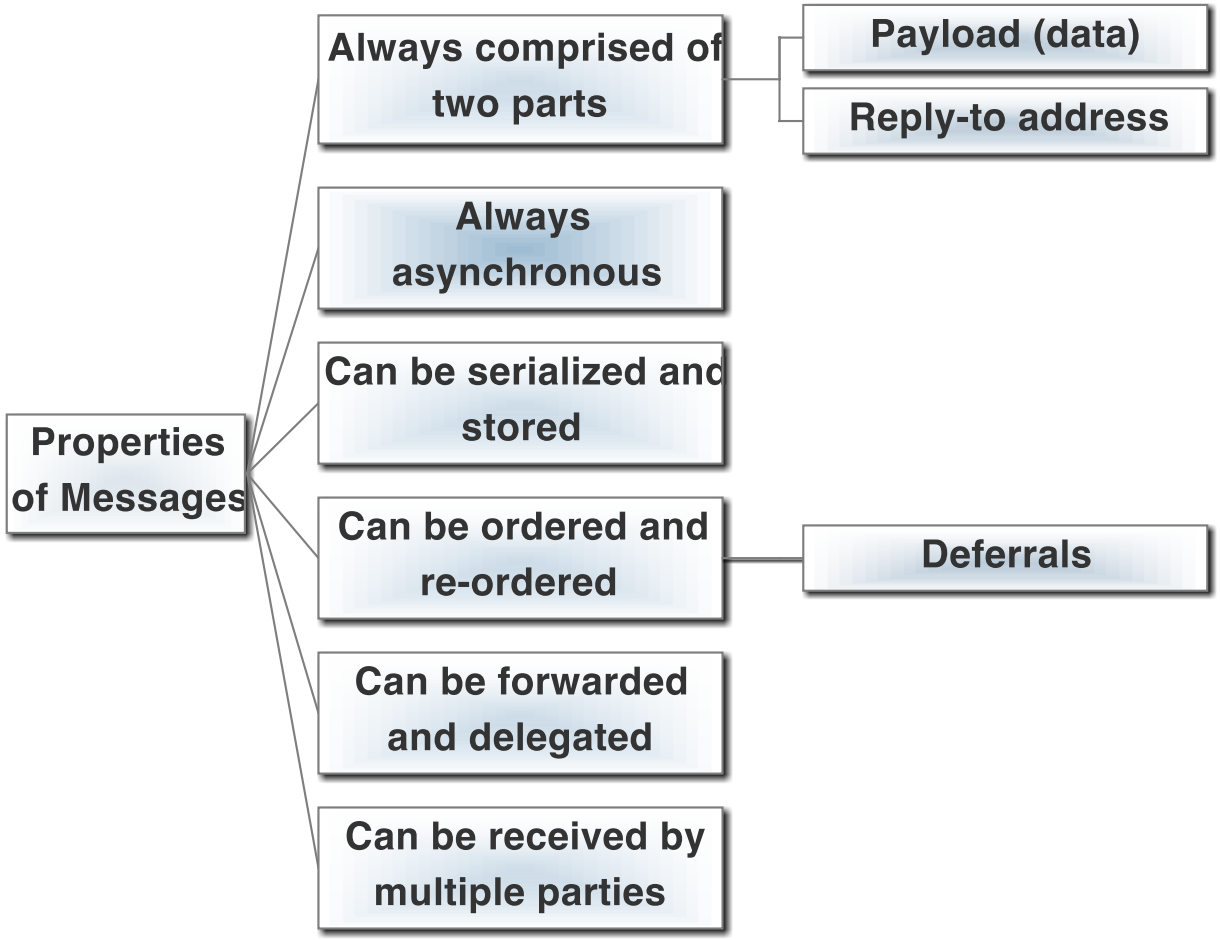
The Shortlist

- Event-Driven Programming
- Event-Sourcing
- Command + Query Responsibility Segregation
- Actors
- Extend-Only Design

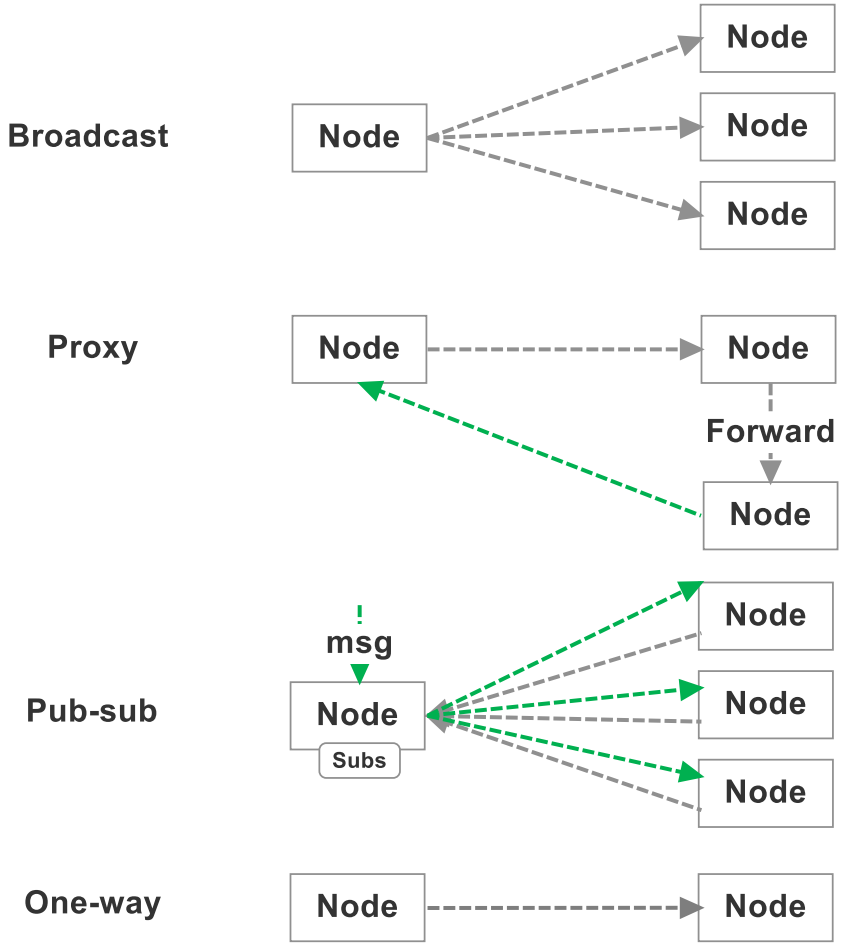
Event-Driven Programming



Events: More Options



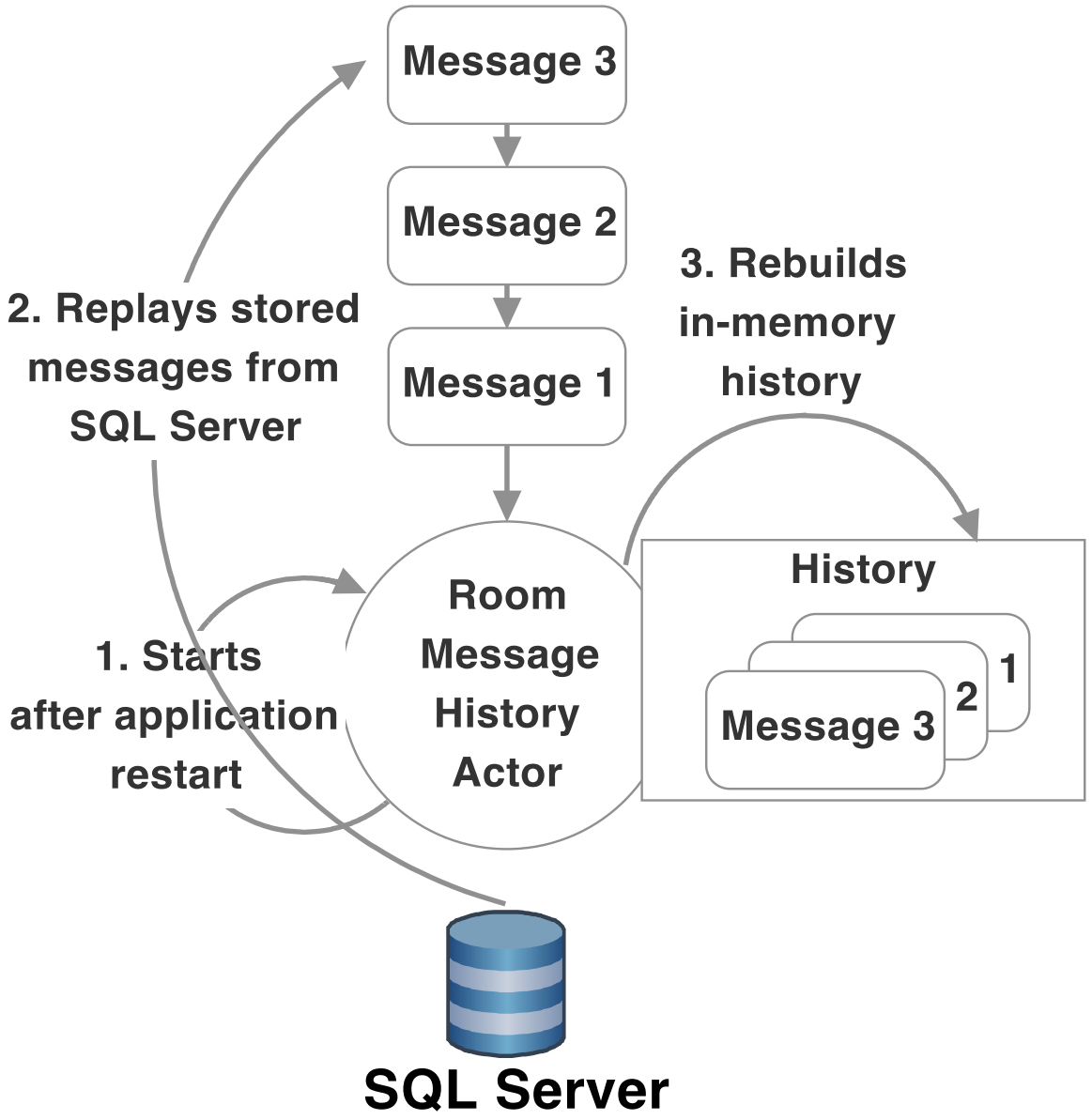
Events: More Interaction Patterns



Event Sourcing w/ Akka.Persistence

DIAGRAMS

Message 3
Message 2
Message 1
2. Request history
4. Present to user
SignalR Chat History for User X
Room Message History Actor
History
Message 3
2
1
3. Deliver history as materialized view
1. Build history, persist to store
SQL Server

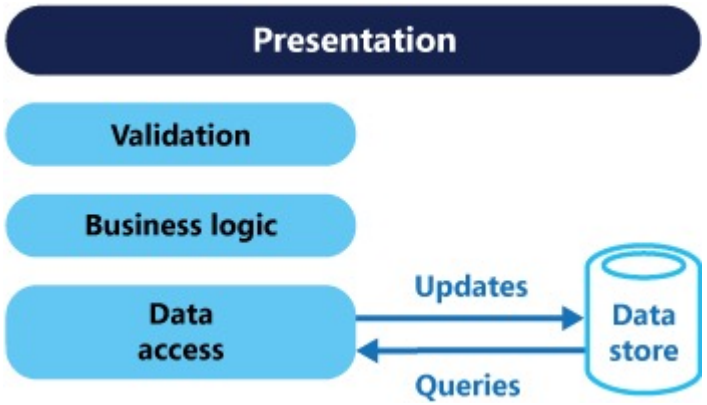

Message 3
Message 2
Message 1
2. Replays stored messages from SQL Server
3. Rebuilds in-memory history
1. Starts after application restart
Room Message History Actor
History
Message 3
2
1
SQL Server

Event Sourcing & Optionality

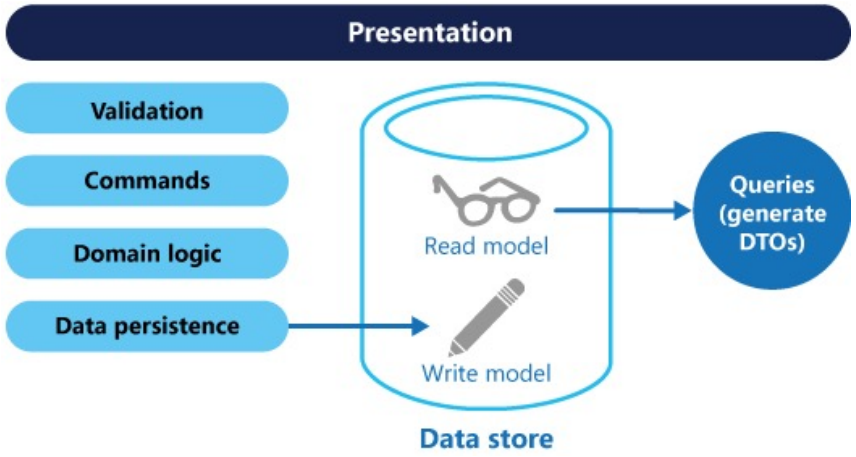
- Current state is the sum of past events.
- Option: state can always be rebuilt by replaying previous events.
- Option: the way state is built can be changed without changing events themselves.
- Option: events can be replayed for simulations, predictions, and regression tests.
- Option: new event types can be safely introduced without modifying prior types.
- Doesn’t rely on DB-specific features. Can be meaningfully abstracted.

Command and Query Responsibility Segregation

Before



After



Source: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs

CQRS

- Separate models for reads and writes
- Often used with Event Sourcing
- Option: read models can be updated independently from each other and write models
- Option: performance characteristics for read / write models can be managed separately
- Option: read / write models can use totally different DB instances or technology

Actors

- Dynamic, partitioned, concurrent event processors
- Provides a thread-safe unit-of-work to process events in real-time
- Can be distributed over a network or run in a single process

Basic Akka.NET Actor

```
public class PingActor : Akka.Actor.ReceiveActor
{
    private readonly ILoggingAdapter _log = Context.GetLogger();

    public PingActor()
    {
        Receive<Ping>(handler: p =>
        {
            _log.Info(format: "Received {0}", p);

            // reply back at a random, short interval
            var replyTime = TimeSpan.FromSeconds(
                ThreadLocalRandom.Current.Next(1, 5));

            Context.System.Scheduler.ScheduleTellOnce(
                replyTime, // delay
                Sender, // target
                message: p.Next(), // message
                Self); // sender (optional)
        });
    }
}
```

Actor base type

Handle to built-in logging system (automatically thread-safe)

Message handler for messages of type Ping.

Sender = reference to actor who sent us the Ping message.

Creating Actors

```
// create ActorSystem (allows actors to talk in-memory)
var actorSystem = ActorSystem.Create("PingPong");

// Props == formula used to start an actor.
var pingActorProps = Props.Create(factory: () => new PingActor());

// start pingActor and get actor reference (IActorRef)
IActorRef pingActor = actorSystem.ActorOf(pingActorProps, name: "ping");

// tell pingActor a message
pingActor.Tell(new Ping(count: 0));
```

Actor will run its Receive<Ping> code when it receives this message (it's asynchronous.)

Actors & Optionality

- Actors are a common way to implement event-driven programming.
- Option: makes live application state query-able at run-time.
- Option: makes stateful server-side applications viable.
- Option: can make event processing dynamic.
- Option: can be partitioned, parallelized, and distributed dynamically with no code changes.

Extend-Only Design

- Schema, wire formats, and APIs are frozen for updates or deletes.
- New properties, event types, or schema can always be added, but old properties can never be removed or changed.
- Old schema is gradually made obsolete and goes unused.

Example: Protobuf Messages

```
message Ask{
        string orderId = 1;
        string stockId = 2;
        double quantity = 3;
        double price = 4; /* normally a decimal in C# - might have loss of precision here */
        int64 timeIssued = 5;
}
```

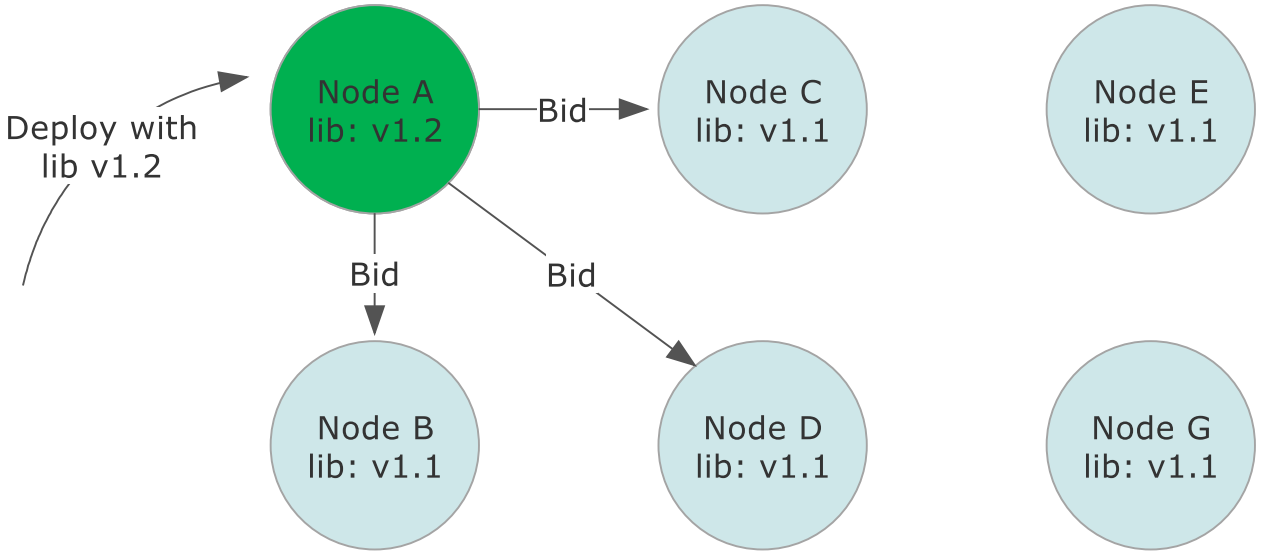
Extending Protobuf Messages…

```
message Ask{
        string orderId = 1;
        string stockId = 2;
        double quantity = 3;
        double price = 4; /* normally a decimal in C# - might have loss of precision here */
        int64 timeIssued = 5;
        bool onMargin = 6;
}
```

New field

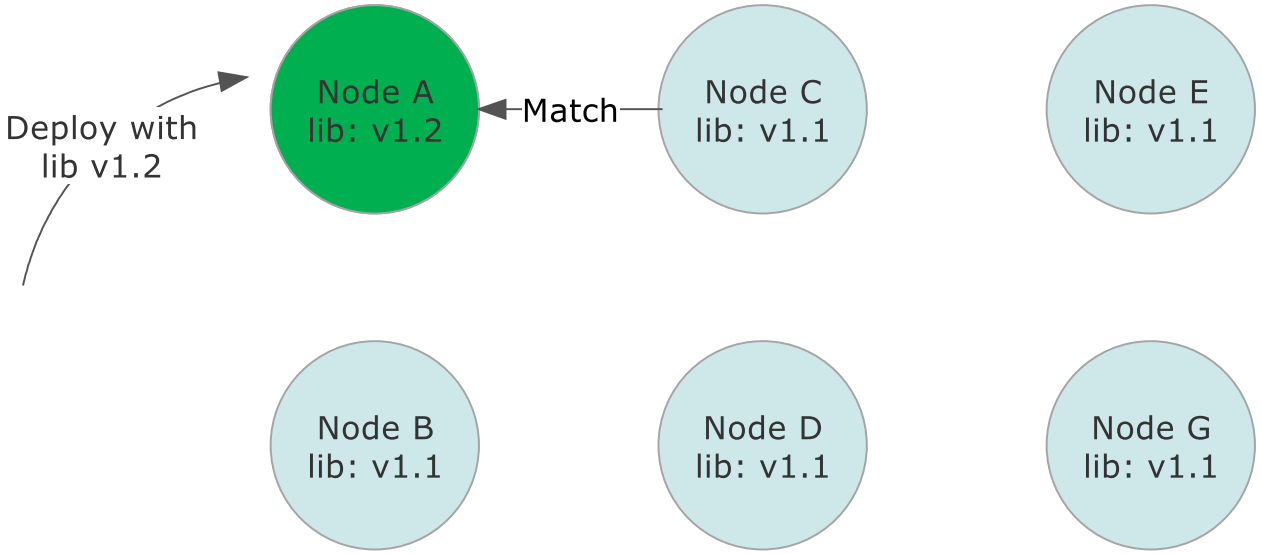
Doesn’t Break Wire Compatibility

Ignores new property if not defined.
Handles 1.2 messages as though they were 1.1



In Either Direction

If 1.1 message is missing property 1.2 message has, provide `null` or default value.



Extend-Only Design & Optionality

- Preserves old schema, but allows new modifications to be introduced
- Option: no more schema migrations; schema changes can be introduced well-ahead of the code that uses it.
- Option: no schema rollbacks. Old schema is still viable.
- Option: zero-downtime deployments. Both versions of schema still supported.

High Optionality Programming

- What do these patterns have in common?
  - Immutability: don't destroy or change the meaning of data
  - Conservation: errs on the side of preserving the past in perpetuity for future reuse
  - Dynamism: can dynamically route, process, react, or update state with ease
  - Separated Concerns: each pattern addresses different facets of modern software

Recap

- Technical debt is caused by the destruction of future, viable options
- High optionality architectures cost more to develop upfront, but pay for themselves quickly when software systems evolve
- High optionality is something you should protect unless you're absolutely certain your requirements won't change

Thank you!

https://petabridge.com/

@Aarononthewebb