Thank you for the intro.
My name is Nico, and I’m going to talk about DB replication for Web application.
Let me start with a bit of background.

Nowadays, web applications are increasingly built using a service oriented architecture with loosely coupled components that can be deployed and scaled independently. Each component is responsible for a specific business feature.
Let me illustrate this with a simple example featuring an e-commerce application. So we have our store frontend that we’d like to augment with some feature. We’d like to add a product recommendation feature and do some analytics to the platform. We instantiate two services that are responsible for these features.

Each component uses some common subset of the data. For example, both the frontend and the recommendation service would use pieces of user data and product data. So this data is stored in different databases. The database is picked depending on what the service needs to accomplish. We might want to use a graph DB to power our recommendation service, like neo4j. Similarly, we might want to use elasticsearch to power our analytics service, because elasticsearch is extremely performant when it comes to doing aggregation and analytics.

To generalize, each component in the eco-system of the web application represents different facets of some common subset of data. This data is stored in the most appropriate database for the task at hand.

This leads us directly to the problem we are solving. How do we synchronize the data across all these different DBs properly?
We want to do DB replication with the following requirements:
1. Our replication system should be compatible with a vast amount of DBs.
2. It should be easy to use. Orchestrating the data flows inside the internal eco-system should be seamless for developers.
3. It should provide good consistency guarantees at scale.
4. The replication mechanism should be failure tolerant. Specifically, network partitions should not result in having half of the DBs missing some data.
So what are people doing right now to address the problem of DB replication?
We've looked at the latest work in academia and the industry. They fall into the following categories:
1. Same DB replication. There's ton of work in this area, mainly to address availability and scalability concerns. These systems are not useful for us, as we want to do cross-DB replication.
2. Then there are cross-DB replication systems that tail the transaction log of the source DB. Parsing the proprietary binary format of the transaction log is brittle and hardly generalizable. An example of such system is LinkedIn's Databus that parses their Oracle DB's commit log.
3. Then there are systems that do DB federation. These systems aim at making a bunch of DBs appear as a single DB though a single API. Depending on the query that needs to be performed, the system should pick the right DB to execute the query. It's hard to leverage full feature set of each DB through a single abstraction.
4. Then there are manual solutions leveraging publisher/subcriber infrastructures. Sadly, this is hard for developers to get right. Specifically, fault tolerance is often missed and consistency semantics are poor.
We present Synapse, the first system that solves the DB replication problem with all the requirements that we need, namely, compatible with many DBs, easy to use, good consistency at scale, and failure tolerant.

We solve this problem specifically in the context of web applications.
Our key insight is simple: instead of operating at the DB level, like most of the other replication systems, we operate at the application level. By operating at a much higher level of abstraction, we are able to fulfill our desired guarantees with little effort by leveraging the semantics of traditional web applications.
Typically, web applications are structured following the MVC pattern. MVC means model-view-controller. It's a way of separating concerns. The browser sends an HTTP request to the controller, and the controller accesses data from the models and then render a response back to the browser with views.

Each language has its set of popular MVC frameworks. With Ruby, you can use the Ruby on Rails framework, if you like Python, you can use Django, for C#, there's ASP.NET MVC.
Let's look at models
Models are built on top of object-relational-mappers (ORM).
The ORM does the heavy lifting of interacting with the DB so developers don't have to write DB queries.
An example of model is the User model. The User class would be mapped to the database table users. user objects would correspond to database rows.
The application stack actually looks like this: at the bottom we have the DB, then above it we have the ORM, then the models, then the controllers, and the views. Over the years, many ORMs have been developed, each one targeting a different DB.
These ORMs have similar APIs. For example, regardless of the combination ORM/DB, you can expect to just call User.create, and it would work with any DB. Here I'm showing what a SQL ORM would do when receiving a User.create(). It generates the appropriate SQL code to insert a new user in the users table.
Synapse interposes on the ORM to monitor accesses to data objects. This allows Synapse to replicate data from one DB to another without developer intervention.

With this unified data layer, Synapse is compatible with many DBs, including postgresql, mysql, oracle, mongodb, tokumx, cassandra, elasticsearch, neo4j, and rethinkdb. In most cases, Synapse seamlessly translates data models between DBs, but in some cases, it might not be so straightforward. Synapse provides lightweight abstractions to developers to specify translation layers easily. You can find more details in the paper.

So that's how Synapse translates data from one DB to another without having to deal with the intricacy of each DBs.
So how does this work under the cover?
Suppose we have two MVC applications, a publisher and a subscriber.
We want the publisher to export some data, which the subscriber imports.
1) During runtime, we monitor object accesses on the publisher side. Any modifications made to a published objects triggers the replication mechanism.
2) To generate the message payload to be sent to subscribers, We call the getter methods on the object that just changed to get all the published attribute values. For example, with a user object, we would call the name and email getters and put these values in the payload.
3) Once prepared, the payload is pushed to the message broker. The message broker persists these payloads in message queues and distribute the payloads to the appropriate subscribers. Synapse relies on existing publish/subscribe systems such as RabbitMQ, or Kafka.
4) Once the subscriber receives the payload, it processes it by instantiating the appropriate model, settings the attributes through the setter methods, and finally acks the payload to the message broker.

To summarize, we replicate objects from publishers to subscribers, and the data translation is done by calling getters and setters on each side. All of this is done transparently without the intervention of the developer aside from specifying what gets to be published and subscribed.

So how does the developer specify what gets to be published and subscribed? Let me introduce you to the Synapse API.
Suppose you have a User model written in the Ruby language.

Here we are using the Mongoid ORM, a MongoDB ORM for Ruby. There are two fields declared, email and name.

To publish the User model all we need to do is wrap the fields that we want to export with the publish keyword as such
By the way, this is not pseudo-code, this is real code. This is exactly what a developer would do to export the User model.

Now let's say that we want to import that data in a separate service running on a SQL database.
Here we are using the ActiveRecord ORM, which is a standard SQL ORM for Ruby.

Similarly to the publish API, we use the subscribe API and wrap the attributes that we want to import. That’s it.
So Synapse is really super easy to use. Developers use the `publish` and `subscribe` keywords directly in their models, and Synapse handles the rest.
Even though the Synapse API is very simple, just with the two keywords publish and subscribe, one can construct complex eco-systems of services. I will show two examples.
The first example is how to do data model mashups.
Suppose you have the main application publishing a user model.
Say you have a moderation service that subscribe to the user model and detects suspicious users by doing some machine learning. This service can publish a new attribute "is_suspicious" on the subscribed User model.
Finally, a third service can then subscribe to both sources of the user model to get an aggregate view of the user model. This way, developers can use the publish and subscribe keywords to do mashups of models in an intuitive manner.
The second example is how to implement asynchronous triggers. For example, suppose you want to implement a service that sends a welcome email whenever a new user registers. One can simply subscribe to the User model, and add an ORM provided callback, `before_create`, that triggers the email. This is intuitive to developers as they are already familiar with this ORM-provided callback API.

So far we've seen that synapse provides a foundation to share data among independent services in a clean manner with a very simple API that can be used for various use cases, namely DB replication, data model mashups, and asynchronous triggers.
But doing it at scale brings another problem.
Consistency.
I'm going to illustrate the consistency issue with an example.
Suppose you have two services. A publisher implementing a document sharing platform, and a subscriber sending emails. When a user uploads a new document on the platform, we want to email that document to his friends.
Consider the follow scenario: a C-level executive comes back from his company's quarterly meeting, and proceeds to make some radical changes. He is going to use the document sharing platform to share his new plan. But before doing so, he carefully removes an employee from his friends list, and only then uploads a very confidential document.
So what happens under the cover?
The publisher is going to broadcast two different data updates to the subscriber. The first one being the removal of the soon to be fired employee, and then the creation of the confidential document.
At scale, without proper synchronization, the two data updates may get processed in the wrong order. This will result in sending the confidential document to the wrongfully friended employee. This bug is vicious, because the developers won't catch it. No exceptions are thrown, no database constraints are violated. The bug instead manifests into a lawsuit, which is undesirable.
So, what are we supposed to do? Intuitively, we want to parallelize the workload as much as possible, while serializing what needs to be serialized. How do we know what needs to be serialized? We cannot rely on developers to tell us, because they will make mistakes. We cannot rely on static data relationships, like foreign keys declarations on models, it's not good enough. Instead, Synapse dynamically discovers data dependencies at runtime.

It does so by leveraging the controller abstraction of the MVC framework. The controller is the entity that process HTTP requests and interacts with models. Synapse monitors model accesses within controllers.
Causal Delivery Semantics

- Serializes all writes within a controller
- Serializes all writes within a user session
- Causality between reads and writes across controllers are preserved

**Causal Semantics**: The subscriber should function as if it was running on the publisher's DB.

Synapse provides three useful guarantees:
1) all the writes within a controller are serialized. This ordering is enforced on subscribers. This matches developers expectations.
2) all writes within a user session are serialized. This matches users' expectations.
3) any given write is marked to have causal dependencies on the previously read objects within the controller. The reason why this is useful is because controllers are stateless, any data that is important for a write will be read through the DB during the controller execution.
And so when a subscriber processes an object update, it can read other related objects, and their state will all be coherent.

To summarize, we try to give the subscriber the illusion that it's running on the publisher DB.

These semantics allow developers to drastically simplify their codebase because they no longer have to take in account these weird corner cases of out-of-order updates. Developers think in a single threaded mindset, while their application is massively parallelized.
Synapse implements these consistency semantics by transparently intercepting all read and write queries to the DB, inferring accessed objects, and versioning them. The object versions are stored in a separate version store per service. In our implementation, we use Redis.

Finally, Synapse is fault tolerant. In a nutshell, synapse does two phase commits all the way from the publisher to the subscriber. On the publisher, the tricky part is to do these 3 things atomically:

1) Perform the Write on the DB
2) Update the object versions
3) Publish the message

We do all this with 2PC.
Synapse

1. Cross-DB replication with a vast number of DBs
2. Easy to use
3. Good consistency at scale
4. Failure tolerant

To summarize, we have a framework to share data in a clean manner to do cross-DB replication, and asynchronous triggers, with good consistency at scale in with a dead simple API.
We implemented Synapse on Ruby on Rails in 4000 lines of ruby. Synapse currently supports 9 DBs. We implemented a more features besides what I talked about. For example, we provide a comprehensive testing framework so developers can write integration tests of their publisher and subscribers.
Crowdtap

- Online marketing-services company
- Contracted by major brands (Verizon, AT&T, Sony)
- 450,000 users Q3 2014
- Using Synapse for the past two years

Let me show what can be done with Synapse by showcasing a company that have been using Synapse for the past two years.

Crowdtap is an online marketing-services company contracted by major brands such as Verizon, AT&T, Sony and MasterCard. Crowdtap has grown rapidly since its inception, and by now, had seen over 450,000 users.

The platform allow brands to target a specific demographic among crowdtap’s users, and give them stuff to do, like answering polls, or trying out products. Users would be rewarded points, usable on an ecommerce store.
In its early days, because of their rapid growth, they had to extract some features into separate services to keep things under control. They had two services aside from their core application. A targeting service responsible to match users and brand actions, and an ecommerce platform where users could redeem their points. They were synchronizing data across services via a synchronous API which was problematic as it suffered from performance issues and data inconsistencies due to lack of failure isolation and fault tolerance.
To address these issues, they rolled out Synapse in production.
Not only the performance and data inconsistencies issues were addressed, but the targeting engine went from 1500LOC to 500LOC due to code simplification.
Quickly after, a junior engineer extracted the mailer functionality from the core application. This required the sharing of 24 models. This would have been very difficult to do manually.
Because they were so pleased with Synapse, they went on and extracted other major features into separate services. Synapse coordinates 53 different models within their eco system. Let me tell you two stories around the reporting and analytics service that show how Synapse allowed the crowdtap development team to be more agile.
Because they were so pleased with Synapse, they went on and extracted other major features into separate services. Synapse coordinates 53 different models within their ecosystem.

Let me tell you two stories around the reporting and analytics service that show how Synapse allowed the Crowdtap development team to be more agile.
Synapse allows the team to try out new ideas very easily.
An employee developed a prototype of the reporting service during a hackathon using real time production data on his laptop. Because of the isolation guarantees Synapse provides, he was able to consume fresh production data without impacting the main application.
The business team was able to use it right away and the reporting service has been deployed in production ever since.
The second story is about a live DB switch done the analytics service. Normally, to switch DB engine on a running service, you would have to wait off hours, take it offline, migrate the data from the old DB to the new one, deploy the new code, and hope for the best. The development team actually used Synapse to do a live DB switch, a usecase we never imagined.

The way it works is that you deploy two versions of the same service. One running with the old DB, and one running with the new DB. Once everything looks good, you redirect all the traffic to the new version of the service, and kill the old one. And so this is how Synapse allowed the team to migrate the analytics service from MongoDB to elasticsearch without taking any downtime.
To generalize, Synapse allows development teams to be more agile, it lets developers try new ideas easily with no commitment. It also let the team manage their services in a very efficient manner. These use cases are common among many startups and medium scale company.

Let's look at the numbers.
To evaluate the overhead of Synapse, we instrumented the production application of Crowdtap during a 24h window. On average, Synapse adds 8% of additional latency on HTTP endpoints. This overhead is not perceivable by end users.
We were also interested in looking at what would the dependencies look like in practice. So we instrumented the production application to render a dependency graph.
It looks like a hot mess. Each node represents a DB update, with edges as dependencies. Synapse is responsible to transparently discover these dependencies with minimal overhead on the publishers, and apply these updates in the correct order on the subscribers in real time.

This image is actually a zoomed in portion of the bigger picture
When we zoom out, it looks like this. And this is only 2 minutes of real traffic, going at roughly 50 updates/s. But with real traffic, we could not push Synapse in its limits.
So to evaluate the scaling properties of Synapse, we deployed Synapse on a thousand servers on Amazon.

We deployed 400 publisher machines, 400 subscriber machines. 50 Redis servers on each side to do the dependency tracking, and 100 RabbitMQ servers to move data from the publishers to subscribers.

We measured the throughput of the System. Synapse scales linearly and reaches 60,000 updates per second. As a comparison point, on Twitter, there's on average 10,000 tweets per second. So Synapse scales well even for large workloads.

During our experiment, we also ran the publishers and subscribers on different DB engines, and saw that Synapse was never a bottleneck.
Conclusion

1. Cross-DB replication with a vast number of DBs
2. Easy to use
3. Good consistency at scale
4. Failure tolerant

To conclude, Synapse:

1. Does cross-DB replication with a vast number of DBs.
2. It is easy to use.
3. It provides good consistency guarantees at scale.
4. Fault tolerant.

We've open sourced synapse, you can grab the sources on github.
Synapse
A Microservices Architecture for Heterogeneous-Database Web Application

http://github.com/nviennot/synapse

Reach me on twitter at @nviennot
## Support for Various ORMs and DBs

<table>
<thead>
<tr>
<th>DB</th>
<th>ORM</th>
<th>Pub?</th>
<th>Sub?</th>
<th>ORM LoC</th>
<th>DB LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>ActiveRecord</td>
<td>Y</td>
<td>Y</td>
<td>474</td>
<td>44</td>
</tr>
<tr>
<td>MySQL</td>
<td>ActiveRecord</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>52</td>
</tr>
<tr>
<td>Oracle</td>
<td>ActiveRecord</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>47</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Mongoid</td>
<td>Y</td>
<td>Y</td>
<td>399</td>
<td>0</td>
</tr>
<tr>
<td>TokuMX</td>
<td>Mongoid</td>
<td>Y</td>
<td>Y</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Cequel</td>
<td>Y</td>
<td>Y</td>
<td>219</td>
<td>0</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>Stretcher</td>
<td>N/A</td>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Neo4j</td>
<td>Neo4j</td>
<td>N</td>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RethinkDB</td>
<td>NoBrainer</td>
<td>N</td>
<td>Y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ephemerals</td>
<td>N/A</td>
<td>Y</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Observers</td>
<td>N/A</td>
<td>N/A</td>
<td>Y</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Dependency Example

class PicturesController
  def create
    current_user = User.find(session[:user_id])
    Picture.create(:content => params[:content],
                    :author => current_user)
    render 'pictures/show'
  end
end
Dependency Example

```ruby
1 class PicturesController
2   def create
3     current_user = User.find(session[:user_id])
4     Picture.create(:content => params[:content],
5                       :author => current_user)
6     render 'pictures/show'
7   end
8 end
```

read  User(id=1)
write Picture(id=2)
Dependency Example

```ruby
1 class PicturesController
2   def create
3     current_user = User.find(session[:user_id])
4     Picture.create(:content => params[:content],
5                      :author => current_user)
6     render 'pictures/show'
7   end
8 end
```

```
read  User(id=1)
write Picture(id=2)
```

Synapse guarantees that the subscriber sees the User(id=1) object in the same version as when the publisher issued the Picture.create

... This works because controllers are stateless. We know they’ll have to read the data they need from the DB.
Synapse Consistency API

- **delivery_mode**: Parameter for selecting delivery semantic.

- **add_read_deps / add_write_deps**: Specify explicit dependencies for read and write DB queries.

- **bootstrap?**: Predicate method denoting bootstrap mode.

Now you've seen the synapse API. Time to look at the Architecture to see how Synapse is implemented.
Throughput for Various Delivery Semantics

Note: Subscribers are running a 100ms callback.
## Synapse Overhead at Crowdtap

<table>
<thead>
<tr>
<th>Most Popular Controllers (&gt;70% of total)</th>
<th>Published Messages</th>
<th>Dependencies per Message</th>
<th>Controller Time (ms)</th>
<th>Synapse Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>awards/index</td>
<td>0.0</td>
<td>0.0</td>
<td>56.5</td>
<td>0.0 (0%)</td>
</tr>
<tr>
<td>brands/show</td>
<td>0.03</td>
<td>1.0</td>
<td>97.6</td>
<td>0.8 (0.8%)</td>
</tr>
<tr>
<td>actions/index</td>
<td>0.67</td>
<td>17.8</td>
<td>181.4</td>
<td>14.4 (8.6%)</td>
</tr>
<tr>
<td>me/show</td>
<td>0.0</td>
<td>0.0</td>
<td>14.7</td>
<td>0.0 (0.0%)</td>
</tr>
<tr>
<td>actions/update</td>
<td>3.46</td>
<td>1.8</td>
<td>305.9</td>
<td>84.1 (37.9%)</td>
</tr>
</tbody>
</table>

Overhead across all 55 controllers: 8%