Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications

Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh
{nviennot, mathias, jbell, roxana, nieh}@cs.columbia.edu
Columbia University

Abstract

The growing demand for data-driven features in today's Web applications – such as targeting, recommendations, or predictions – has transformed those applications into complex conglomerates of services operating on each others' data without a coherent, manageable architecture. We present *Synapse*, an easy-to-use, strong-semantic system for large-scale, data-driven Web service integration. Synapse lets independent services cleanly share data with each other in an isolated and scalable way. The services run on top of their own databases, whose layouts and engines can be completely different, and incorporate read-only views of each others' shared data. Synapse synchronizes these views in real-time using a new scalable, consistent replication mechanism that leverages the high-level data models in popular MVC-based Web applications to replicate data across heterogeneous databases. We have developed Synapse on top of the popular Web framework Ruby-on-Rails. It supports data replication among a wide variety of SQL and NoSQL databases, including MySQL, Oracle, PostgreSQL, MongoDB, Cassandra, Neo4j, and Elasticsearch. We and others have built over a dozen microservices using Synapse with great ease, some of which are running in production with over 450,000 users.

1. Introduction

We live in a data-driven world. Web applications today – from the simplest smartphone game to the Web's most complex application – strive to acquire and integrate data into new, value-added services. Data – such as user actions, social information, or locations – can improve business revenues by enabling effective product placement and targeted advertisements. It can enhance user experience by letting applications predict and seamlessly adapt to changing user needs and preferences. It can enable a host of useful value-added features, such as recommendations for what to search for, what to buy, or where to eat. Agility and ease of integration of new data types and data-driven services are key requirements in this emerging data-driven Web world.

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741975

Unfortunately, creating and evolving data-driven Web applications is difficult due to the lack of a coherent data architecture for Web applications. A typical Web application starts out as some core application logic backed by a single database (DB) backend. As data-driven features are added, the application's architecture becomes increasingly complex. The user base grows to a point where performance metrics drive developers to denormalize data. As features are removed and added, the DB schema becomes bloated, making it increasingly difficult to manage and evolve. To make matters worse, different data-driven features often require different data layouts on disk and even different DB engines. For example, graph-oriented DBs, such as Neo4j and Titan, optimize for traversal of graph data and are often used to implement recommendation systems [7]; search-oriented DBs, such as Elasticsearch and Solr, offer great performance for textual searches; and column-oriented DBs, such as Cassandra and HBase, are optimized for high write throughput.

Because of the need to effectively manage, evolve, and specialize DBs to support new data-driven features, Web applications are increasingly built using a service-oriented architecture that integrates composable services using a variety and multiplicity of DBs, including SQL and NoSQL. These loosely coupled services with bounded contexts are called *microservices* [29]. Often, the same data, needed by different services, must be replicated across their respective DBs. Replicating the data in real time with good semantics across different DB engines is a difficult problem for which no good, general approach currently exists.

We present *Synapse*, the first easy-to-use and scalable cross-DB replication system for simplifying the development and evolution of data-driven Web applications. With Synapse, different services that operate on the same data but demand different structures can be developed independently and with their own DBs. These DBs may differ in schema, indexes, layouts, and engines, but each can seamlessly integrate subsets of their data from the others. Synapse transparently synchronizes these data subsets in real-time with little to no programmer effort. To use Synapse, developers generally need only specify declaratively what data to share

Type	**Supported Vendors**	**Example use cases**
Relational	PostgreSQL, MySQL, Oracle	Highly structured content
Document	MongoDB,TokuMX, RethinkDB	General purpose
Columnar	Cassandra	Write-intensive workloads
Search	Elasticsearch	Aggregations and analytics
Graph	Neo4j	Social network modeling

Table 1: **DB types and vendors supported by Synapse.**

with or incorporate from other services in a simple publish/-subscribe model. The data is then delivered to the DBs in real-time, at scale, and with delivery semantic guarantees.

Synapse makes this possible by leveraging the same abstractions that Web programmers already use in widely-used Model-View-Controller (MVC) Web frameworks, such as Ruby-on-Rails, Python Django, or PHP Symfony. Using an MVC paradigm, programmers logically separate an application into *models*, which describe the data persisted and manipulated, and *controllers*, which are units of work that implement business logic and act on the models. Developers specify what data to share among services within model declarations through Synapse's intuitive API. Models are expressed in terms of high-level objects that are defined and automatically mapped to a DB via *Object/Relational Mappers* (ORMs) [5]. Although different DBs may need different ORMs, most ORMs expose a common high-level object API to developers that includes create, read, update, and delete operations. Synapse leverages this common object API and lets ORMs do the heavy lifting to provide a cross-DB translation layer among Web services.

We have built Synapse on Ruby-on-Rails and released it as open source software on GitHub [43]. We demonstrate three key benefits. First, Synapse supports the needs of modern Web applications by enabling them to use many combinations of *heterogeneous DBs*, both SQL and NoSQL. Table 1 shows the DBs we support, many of which are very popular. We show that adding support for new DBs incurs limited effort. The translation between different DBs is often automatic through Synapse and its use of ORMs.

Second, Synapse provides *programmer simplicity* through a simple programming abstraction based on a publish/subscribe data sharing model that allows programmers to choose their own data update semantics to match the needs of their MVC Web applications. We have demonstrated Synapse's ease of use in a case study that integrates several large and widely-used Web components such as the e-commerce platform, Spree. Synapse is already field-tested in production: a startup, Crowdtap, has been using it to support its microservices architecture serving over 450,000 users for the past two years. Our integration and operation experience indicates that Synapse vastly simplifies the construction and evolution of complex data-driven Web applications, providing a level of agility that is crucial in this burgeoning big-data world.

Finally, Synapse can provide excellent *scalability* with low publisher overheads and modest update propagation delays; we present some experimental data showing that Synapse scales well up to 60,000 updates/second for various workloads. To achieve these goals, it lets subscribers parallelize their processing of updates as much as the workload and their semantic needs permit.

2. Background

MVC (Model View Controller) is a widely-used Web application architecture supported by many frameworks including Struts (Java), Django (Python), Rails (Ruby), Symfony (PHP), Enterprise Java Beans (Java), and ASP.NET MVC (.NET). For example, GitHub, Twitter, and YellowPages are built with Rails, DailyMotion and Yahoo! Answers are built with Symfony, and Pinterest and Instragram are built with Django. In the MVC pattern, applications define data models that describe the data, which are typically persisted to DBs. Because the data is persisted to a DB, the model is expressed in terms of constructs that can be manipulated by the DBs. Since MVC applications interact with DBs via ORMs (Object Relational Mappers), ORMs provide the model definition constructs [5].

ORMs abstract many DB-related details and let programmers code in terms of high-level objects, which generally correspond one-to-one to individual rows, or documents in the underlying DB. Although ORMs were initially developed for relational DBs, the concept has recently been applied to many other types of NoSQL DBs. Although different ORMs may offer different APIs, at a minimum they must provide a way to *create*, *update*, and *delete* the objects in the DB. For example, an application would typically instantiate an object, set its attributes in memory, and then invoke the ORM's save function to persist it. Many ORMs and MVC frameworks also support a notion of *active models* [18], which allow developers to specify *callbacks* that are invoked before or after any ORM-based update operation.

MVC applications define controllers to implement business logic and act on the data. Controllers define basic units of work in which data is read, manipulated, and then written back to DBs. Applications are otherwise stateless outside of controllers. Since Web applications are typically designed to respond to and interact with users, controllers typically operate within the context of a *user session*, which means their logic is applied on a per user basis.

3. Synapse API

Synapse extends the MVC pattern to create an easy-to-use platform for integrating Web services that use heterogeneous DBs. Because MVC frameworks and ORMs provide common abstractions that are often used in practice for Web development, Synapse leverages them to provide a transparent and mostly automatic data propagation layer.

Using the Synapse API shown in Table 2, developers make simple modifications to their existing model definitions to share their data across services. At a high level, an application that uses Synapse consists of one or more

Abstraction	Description
Publisher	Service publishing attributes of a model.
Subscriber	Service subscribing to attributes of a model.
Decorator	Service subscribing and publishing a model.
Ephemeral	DB-less publisher.
Observer	DB-less subscriber.
Virtual attribute	Deterministic functions (can be published).

API	Description
`publish`, `subscribe`	Annotations to denote which attributes to publish or subscribe.
`before_create`, `before_update`, `before_destroy`	Re-purposed active model callbacks for subscriber update notification. Similar callbacks for after_create/update/destroy.
`add_read_deps`, `add_write_deps`	Specify explicit dependencies for read and write DB queries.
`delivery_mode`	Parameter for selecting delivery semantic.
`bootstrap?`	Predicate method denoting bootstrap mode.

Table 2: **Synapse Abstractions and API.**

publishers, and one or more subscribers. Publishers are services that make attributes of their data models available to subscribers, which maintain their own local, read-only copies of these attributes. Synapse transparently synchronizes changes to published models (creations, updates, or deletions of model instances) from the publisher to the subscriber. It serializes updated objects on the publisher, transmits them to the subscriber, deserializes them, and persists them through the ORM.

A service can subscribe to a model, *decorate* that model by adding new attributes to it, and publish these attributes. By cascading subscribers into publishers developers can create complex ecosystems of Web services that subscribe to data from each other, enhance it with new attributes, and publish it further. This programming model is easy to use and supports powerful use cases as shown in §5. We discuss the Synapse API using Ruby-on-Rails, but similar APIs can be built for other frameworks.

3.1 Synapse Abstractions

Publishers. To publish a model, the developer simply specifies which attributes within that model should be shared. The code at the top of Fig.1 shows how to publish in Ruby using the `publish` keyword, with Synapse-specific code underlined. Each published model has a globally unique URI, given by `app_name/model_name`. Synapse generates a publisher file for each publisher listing the various objects and fields being published and is made available to developers who want to create subscribers for the published data. A factory file is also made available for each publisher that provides sample data for writing integration tests (§4.5). Other API calls that can be used by publishers are `add_read_deps`, `add_write_deps`, and `delivery_mode`, discussed in §3.2 and §4.2.

```
# Publisher side (Pub1).
class User
  publish do
    field :name
  end
end
```

```
# Subscriber side (Sub1).
class User
  subscribe from: :Pub1 do
    field :name
  end
end
```

Figure 1: **Publisher (top), subscriber (bottom).**

Subscribers. To subscribe to a published model, the developer simply marks the attributes of interest accordingly. In Fig.1, the code at the bottom shows how to subscribe in Ruby to the publisher at the top. Since the model name is the same in the subscriber as the publisher, it does not need to be explicitly identified in conjunction with the `subscribe` keyword. A subscriber application can subscribe to some or all of a publisher's models, and can subscribe to models from multiple publishers. While there may be many subscribers for a given model, there can only be one publisher (the *owner* of that model). The owner is the only service who can create or delete new instances of the model (i.e., *objects*). Moreover, subscribers cannot update attributes that they import from other services, although they can update their own decoration attributes on these models. We enforce this *read-only* subscription model to avoid difficult issues related to concurrent update conflicts from distinct services. That said, Synapse handles concurrent updates made from different servers from the same service. Subscribers often need to perform application-specific processing of updates before applying them to their DBs. For example, a subscriber may need to compute new fields, denormalize data, or send a notification. Synapse supports this by piggybacking upon active model callbacks often supported by MVC frameworks, including `before/after_create`, `before/after_update`, or `before/after_destroy`. The code on the right shows an example of an *after* subscriber callback that sends a welcome email for each newly created User. These callbacks are particularly useful to adapt schemas between publishers and subscribers, as discussed in §3.3. Other API calls that can be used by subscribers are `bootstrap?` and `delivery_mode`, discussed in §3.2.

```
# Notification Subscriber
class User
  subscribe from: :PubApp do
    field :name
    field :email
  end
  after_create do
    unless Synapse.bootstrap?
      self.send_welcome_email
    end
  end
end
```

Figure 2: **Callback.**

Decorators. Decorators are services that subscribe to a model and publish new attributes for it. Conceptually, decorators mix the publisher and subscriber abstractions, although several subtle restrictions apply to them. First, decorators cannot create or delete instances of a model, because they are not its originators. Second, decorators cannot update the attributes of the model that they subscribe to. Third, decorators cannot publish attributes that they subscribe to. Our decorator

```
# Decorator side (Dec2).
class User
  subscribe from: :Pub1 do
    field :name
  end
  publish do
    field :interests
  end
end
```

```
# Subscriber side (Sub2).
class User
  subscribe from: :Pub1 do
    field :name
  end
  subscribe from: :Dec2 do
    field :interests
  end
end
```

Figure 3: **Decorator.**

abstraction encapsulates and enforces these restrictions. As an example, the code at the top of Fig.3 shows a decorator service, which decorates the User model from Pub1 with the user's interests. The data used to compute those interests comes from other sources, such as social activity, but is omitted here. Other services can then subscribe to any subset of the model's attributes or decorations by specifying the originators of those attributes, as shown in the code at the bottom of Fig.3. Using decorators, one can construct complex ecosystems of services that enhance the data in various ways, as shown in the examples in §5.

Ephemerals and Observers. Synapse aims to support as many use cases for data-driven integration as possible. Often times we find it useful to also support integration of non-persisted models. For example, one could define a mailer application that observes user registrations and sends a welcome message, but does not need to store the data. Similarly, although user-facing services may receive user actions (such as clicks, searches, mouse hovering, etc.), it is backend analytics services that truly use that information. Having the front-end application just pass on (publish) the data onto persisting subscribers is useful in such cases. Synapse hence lets programmers mix persisted models with *ephemerals* (non-persisted published models) and/or *observers* (non-persisted subscribed models). Aside from supporting application-driven needs, non-persisted models are often used to adapt mismatching data models across heterogeneous DBs, as shown in §3.3.

Virtual Attributes. To perform data translation between ORMs, Synapse simply calls field getter methods on the publisher side, and then calls the corresponding field setters on the subscriber side. Synapse additionally lets programmers introduce getters and setters for attributes that are not in the DB schema. We call these programmer-provided attributes *virtual attributes*. Virtual attributes are valuable for schema mappings, as shown in §3.3.

3.2 Synapse Delivery Semantics

Update delivery semantics define the ordering of updates as viewed by subscribers and are an important part of Synapse's programming model. Different applications may require different levels of semantics: while some may be able to handle overwritten histories, others may prefer to see every single update. Similarly, while some applications may be able to handle updates in any order, others may expect them in an order that respects application logic. In support of applications with different needs, and inspired by well-established prior art [6], Synapse allows publishers and subscribers to use the `delivery_mode` configuration directive to select among three delivery semantics: *global*, *causal*, and *weak*.

Global Ordering. On the publisher side, global order delivery mode means that all object updates will be sequentially ordered by the publisher. On subscribers, it means that the sequential order from a global order publisher will be provided to subscribers. This provides the strongest semantics, but in practice limits horizontal scaling and is rarely if ever used in production systems.

Causal Ordering. Causal ordering identifies for each update U the prior update that must be applied before U to avoid negative effects, such as sending a notification for a new post to an out-of-date friends set. On the publisher, causal order delivery mode means that (1) all updates to the same object are serialized, (2) all updates performed within a controller are serialized to match developer expectations of sequential controller code, and (3) controllers within the same user session are serialized so that all updates performed within the same user session are serialized to match user expectations of Web applications. On the subscriber, causal ordering provides the same three semantics as on the publisher, but also ensures causality between reads and writes across controllers. Specifically, when a subscriber processes an update U, Synapse guarantees that if the subscriber reads objects from its DB that were specified as read dependencies during the publishing, the values of these objects are equal to the ones on the publisher's DB when it performed U. In other words, it's as if the subscriber was given a snapshot of the objects specified as read dependencies along with the update U. These semantics are useful because publisher controllers are stateless, so updates are performed after reading and validating dependent objects (e.g., access control, foreign keys) without relying on caches (otherwise the publisher would be racy). This mode provides sufficient semantics for many Web applications without the performance limitations of global order delivery mode, as shown by Fig. 8 in §4 and our evaluation in §6.4.

Weak Ordering. On the publisher, weak order delivery mode means that all updates to the same object will be sequentially ordered by the publisher, but there is no ordering guarantee regarding updates to different objects. On subscribers, it means that the sequential order of updates for each object is provided, but intermediate updates may be missed or ignored if they are delivered out-of-order. Essentially, weak delivery subscribers always update objects to their latest version. This mode is suitable for applications that have low semantic requirements and provides good scaling properties, but its most important benefit is high availability due to its tolerance of message loss. For example, causal order delivery mode requires delivery of every single update for all objects, so loss of an update would result in failure. In production, unfortunately, situations occur where messages may get lost despite the use of reliable components (see §6.5). Weak order delivery mode can ignore causal dependencies and only update to the latest version.

Selecting Delivery Modes. Publishers select the modes that deliver the strongest semantics that they wish to support for their subscribers, subject to the performance overheads they can afford. The more flexible the delivery order semantics,

```
# Publisher 1 (Pub1).
# Runs on MongoDB.
class User
  include Mongoid::Document
  publish do
    field :name
  end
end
```

```
# Subscriber 1a (Sub1a).
# Runs on any SQL DB.
class User < ActiveRecord::Base
  subscribe from: :Pub1 do
    field :name
  end
end
```

```
# Subscriber 1b (Sub1b).
# Runs on Elasticsearch.
class User < Stretcher::Model
  subscribe from: :Pub1 do
    property :name, analyzer: :simple
  end
end
```

```
# Subscriber 1c (Sub1c).
# Runs on MongoDB.
class User
  include Mongoid::Document
  subscribe from: :Pub1 do
    field :name
  end
end
```

Figure 4: **Example 1: Basic Integration.** Shows publishing/subscribing examples with actual ORMs. Synapse code is trivial. This is the common case in practice.

the more Synapse can enable subscribers to process updates with as much parallelism as possible to keep up with a high throughput publisher. Subscribers can only select delivery semantics that are at most as strong as the publishers support. Subscribers can select different delivery modes for data coming from different publishers. In the common case, a publisher would select to support causal delivery, while the subscriber may configure either causal or weak delivery. For example, given a causal mode publisher, a mailer subscriber that sends emails on the state transitions of a shopping cart would not tolerate overwritten histories without additional code, although it is generally tolerable for the mailer service to be unavailable for short periods of time. The causal semantic would be well fit for such a subscriber. In contrast, a real-time analytics service that aggregates million of rows at once may not care about orders, while being unavailable, even for short period of time, may damage the business. The weak semantic would be sufficient for this subscriber.

There is only one constraint on delivery mode choice in Synapse. During the bootstrapping period, which occurs when a subscriber must catch up after a period of unavailability, Synapse forces the weak semantic (i.e., the subscriber may witness overwritten histories and out-of-order deliveries). We signal such periods clearly to programmers in our API using the `bootstrap?` predicate. Fig. 2 shows a usage example and §4.4 describes this situation and explains how subscribers demanding higher semantics can deal with semantics degradation.

3.3 Synapse Programming by Example

Synapse addresses many of the challenges of heterogeneous-DB applications automatically, often in a completely plug-and-play manner thanks to its use of ORM abstractions. In other cases, the programmer may need to perform explicit translations on the subscriber to align the data models. Our experience suggests that Synapse's abstractions facilitate these translations, and we illustrate our experience using examples showcasing Synapse's usability with each major class of DB: SQL, document, analytic, and graph.

Example 1: Basic Integrations. Our experience suggests that most integrations with Synapse are entirely automatic

```
# Publisher 2 (Pub2).
# Runs on any SQL DB.
class User < ActiveRecord::Base
  publish do
    field :name
    field :likes
  end
  has_many :friendships
end
class Friendship < ActiveRecord::Base
  publish do
    belongs_to :user1, class: User
    belongs_to :user2, class: User
  end
end
```

```
# Subscriber 2 (Sub2).
# Runs on Neo4j.
class User # persisted model
    include Neo4j::ActiveNode
    subscribe from: :Pub2 do
      property :name
      property :likes
    end
    has_many :both, :friends, \
      class: User
end
class Friendship # not persisted
  include Synapse::Observer
  subscribe from: :Pub2 do
    belongs_to :user1, class: User
    belongs_to :user2, class: User
  end
  after_create do
    user1.friends << user2
  end
  after_destroy do
    user1.friends.delete(user2)
  end
end
```

Figure 5: **Example 2: SQL/Neo4j.** Pub2 (SQL) stores friendships in their own table; Sub2 (Neo4j) stores them as edges between Users. Edges are added through an Observer.

and require only simple annotations of what should be published or subscribed to, similar to the ones shown in Fig.1. For example, Fig.4 shows the integration of a MongoDB publisher (Pub1) with three subscribers: SQL (Sub1a), Elasticsearch (Sub1b), and MongoDB (Sub1c). The programmers write their models using the specific syntax that the underlying ORM provides. Barring the `publish/subscribe` keywords, the models are exactly how each programmer would write them if they were not using Synapse (i.e., the data were local to their service). In our experience deploying Synapse, this is by far the most frequent case of integration.

That said, there are at times more complex situations, where programmers must intervene to address mismatches between schemas, supported data types, or optimal layouts. We find that even in these cases, Synapse provides just the right abstractions to help the programmer address them easily and elegantly. We describe next complex examples, which illustrate Synapse's flexibility and great added value. We stress that not all integrations between a given DB pair will face such difficulties, and vice versa, the same difficulty might be faced between other pairs than those we illustrate.

Example 2: Mapping Data Models with Observers. Different DBs model data in different ways so as to optimize different modes of accessing it. This example shows how to map the data models between a SQL and Neo4j DB to best leverage the DBs' functions. Neo4j, a graph-oriented DB, is optimized for graph-structured data and queries. It stores relationships between data items – such as users in a social network or products in an e-commerce app – as edges in a graph and is optimized for queries that must traverse the graph such as those of recommendation engines. In contrast, SQL stores relationships in separate tables. When integrating these two DBs, model mismatches may occur. Fig.5 illustrates this use case with an example.

Pub2, the main application, stores Users and their friends in a SQL DB. Sub2, an add-on recommendation engine, integrates the user and friendship information into Neo4j to

```
# Publisher 3 (Pub3).
# Runs on MongoDB.
class User
  include Mongoid::Document
  publish do
    field :interests
  end
end
```

```
# Subscriber 3a (Sub3a).
# Runs on any SQL DB.
# Searching for users based on
# interest is not supported.
class User < ActiveRecord::Base
  subscribe, from: :Pub3 do
    field :interests
  end
  serialize :interests
end
```

```
# Subscriber 3b (Sub3b).
# Runs on any SQL DB.
# Supports searching for users by interest.
class User < ActiveRecord::Base
  has_many :interests
  subscribe from: :Pub3 do
    field :interests, as: :interests_virt
  end
  def interests_virt=(tags)
    Interest.add_or_remove(self, tags)
  end
end
class Interest < ActiveRecord::Base
  belongs_to :user
  field :tag
  def self.add_or_remove(user, tags)
    # create/remove interests from DB.
  end
end
```

Figure 7: **Example 3: MongoDB/SQL.** Shows one publisher running on MongoDB (Pub3) and two SQL subscribers (Sub3a,b). Default translations work, but may be suboptimal due to mismatches between DBs. Optimizing translation is easy with Synapse.

provide users with recommendations of what their friends or network of friends liked. Its common type of query thus involves traversing the user's social graph, perhaps several levels deep. As in the previous examples, we see here that the programmer defines his subscriber's User model in the way that she would normally do so for that DB (the top of Sub2). However, in this case, Synapse's default translation (achieved by just annotating data with publish/subscribe) would yield low performance since it would store both the user and the friendship models as nodes just like the publisher's SQL schema does, ignoring the benefits of Neo4j.

To instead store friendships as edges in a graph between users, the programmer leverages our observer abstraction. She defines an observer model to subscribe to the Friendship model, which rather than persisting the data as-is, simply adds or removes edges among User nodes. This solution, which involves minimal and conceptually simple programmer input, lets the subscriber leverage Neo4j's full power.

Example 3: Matching Data Types with Virtual Attributes. At times, DBs may mismatch on data types. As an example, we present a specific case of integration between MongoDB and SQL. MongoDB, a document-oriented database, has recently become popular among startups thanks to its schemaless data model that allows for frequent structural changes. Since the DB imposes so little structure, importing data into or exporting data from MongoDB is typically similar to Fig.4. We choose here a more corner case example to show Synapse's applicability to complex situations.

Fig. 7 shows a MongoDB publisher (Pub3), which leverages a special MongoDB feature that is not generally available in SQL, Array types, to store user interests. Fig.7 shows two options for integrating the interests in a SQL subscriber, both of which work with all SQL DBs. The first option (Sub3a) is to automatically flatten the array and stores it as text, but this would not support efficient queries on interests.

The typical solution to translate this array type to a generic SQL DB is to create an additional model, Interest, and a one-to-many relationship to it from User. Sub3b shows how Synapse's virtual attribute abstraction easily accomplishes this task, creating the Interest model and a virtual attribute (interests_virt) to insert the new interests received into the separate table.

4. Synapse Architecture

Fig. 6(a) shows the Synapse architecture applied to an application with a single publisher and subscriber. The publisher and subscriber may be backed by different DBs with distinct engines, data models, and disk layouts. In our example, the publisher runs on PostgreSQL, a relational DB, while the subscriber runs on MongoDB, a document DB. At a high level, Synapse marshals the publisher's model instances (i.e., objects) and publishes them to subscribers, which unmarshal the objects and persist them through the subscribers' ORMs.

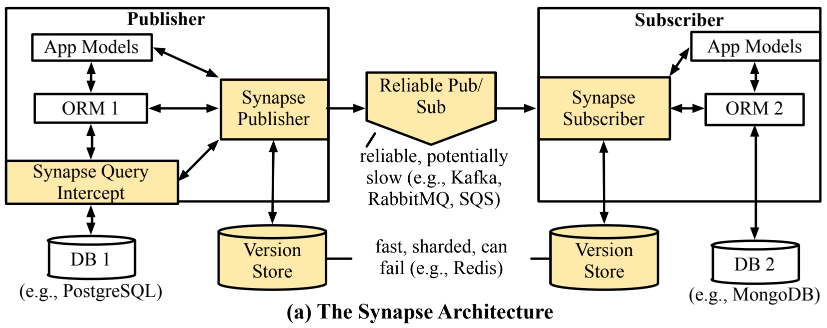
Synapse consists of two DB- and ORM-agnostic modules (*Synapse Publisher* and *Synapse Subscriber*), which encapsulate most of the publishing and subscribing logic, and one DB-specific module (*Synapse Query Intercept*), which intercepts queries and relates them to the objects they access. At the publisher, Synapse interposes between the ORM and the DB driver to intercept updates of all published models, such as creations, updates, or deletions of instances – collectively called *writes* – before they are committed to the DB. The interposition layer identifies exactly which objects are being written and passes them onto the *Synapse Publisher*, Synapse's DB-independent core. The Publisher then marshals all published attributes of any created or updated objects, attaches the IDs of any deleted objects, and constructs a *write message*. Synapse sends the message to a reliable, persistent, and scalable message broker system, which distributes the message to the subscribers. All writes within a single transaction are combined into a single message.

The message broker reliably disseminates the write message across subscribers. Of the many existing message brokers [21, 22, 35], we use RabbitMQ [35] in our implementation, using it to provide a dedicated queue for each subscriber app. Messages in the queue are processed in parallel by multiple subscriber workers per application, which can be threads, processes, or machines.

When a new message is available in the message broker, a Synapse subscriber worker picks it up and unmarshals all received objects by invoking relevant constructors and attribute setters (using the language's reflection interface). The worker then persists the update to the underlying DB and then acks the message to the broker.

4.1 Model-Driven Replication

To synchronize distinct DBs, Synapse needs to (1) identify the objects being written on the publisher, (2) marshal them for shipping to the subscribers, (3) unmarshal back to objects at the subscriber, (4) and persist them. Although steps 1



(a) The Synapse Architecture

```
{ app: "pub3",
  operations: [ {
    operation: "update",
    type: ["User"],
    id: 100,
    attributes: {
      interests: ["cats", "dogs"]
    } } ],
  dependencies: {
    "pub3/users/id/100": 42 },
  published_at: "10/11/14 07:59:00",
  generation: 1 }
```

(b) Sample Write Message

Figure 6: **The Synapse Architecture.** (a) Synapse components are shaded. To replicate data between heterogeneous DBs, Synapse marshals the publisher's objects and sends them to subscribers, which unmarshal and save them into their DBs. (b) Published message format (JSON).

and 4 seem completely DB specific, we leverage ORMs to abstract most DB specific logic.

To intercept writes, Synapse uses a DB-engine specific query interceptor. Collecting information about objects written is generally straightforward, as many DBs can easily output the rows affected by each query. For example, in SQL, an `INSERT`, or `DELETE` query ending with `RETURNING *` will return the contents of the written rows. Many DBs support this feature, including: Oracle, PostgreSQL, SQL Server, MongoDB, TokuMX, and RethinkDB. For DBs without this feature (e.g., MySQL, Cassandra), we develop a protocol that involves performing an additional query to identify data being written; it is safe but somewhat more expensive.

After intercepting a write, Synapse uses the ORM to map from the raw data written back to application objects (e.g., an instance of the `User` model). The published attributes of these written object(s) are marshaled to JSON, and published along with object dependencies (described in §4.2) and a generation number (for recovery, described in §4.4). When marshalling objects, Synapse also includes each object's complete inheritance tree, allowing subscribers to consume polymorphic models. Fig. 6(b) shows a write message produced upon a `User` creation; the post object's marshalling is in the message's `attributes` field.

On the subscriber, Synapse unmarshals a new or updated object by (1) instantiating a new instance of that type, or finding it in the DB based on its primary key with the ORM's `find` method, (2) recursively assigning its subscribed attributes from those included in the message by calling the object setter methods, and (3) calling the `save` or `destroy` method on the object. For a delete operation, step (2) is skipped. Different ORMs may have different names for these methods (e.g., `find` vs `find_by`) but their translation is trivial. Any callbacks specified by the programmer are automatically called by the ORM.

4.2 Enforcing Delivery Semantics

Synapse enforces update-message ordering from publishers to subscribers to be able to provide subscribers with a view of object updates that is consistent with what would be perceived if the subscribers had direct access to the publisher's DB. Specific consistency semantics are determined based on the choice of publisher and subscriber delivery order modes, global, causal, and weak. In all cases, Synapse uses the same general update delivery mechanism, which is inspired by previous work on deterministic execution record-replay [24, 44].

The update delivery mechanism identifies dependencies on objects during persistence operations and tracks their version numbers to send to subscribers. Synapse defines an operation as having a dependency on an object if the operation, or the construction of the operation, may reference the object. On the publisher, Synapse tracks two kinds of dependencies: read and write dependencies. An operation has a read dependency on an object if the object was read, but not written, and used to construct the given operation. An operation has a write dependency on an object if the operation modifies the object. An operation may have a combination of both read and write dependencies on different objects. Since an object may have many versions due to updates, Synapse uses version numbers to track object versions and expresses dependencies in terms of specific object versions. For each object, Synapse maintains two counters at the publisher, ops and $version$, which represent the number of operations that have referenced the object so far and the version number of the object, respectively. For each operation at the publisher, Synapse identifies its dependencies and uses this information to publish a message to the subscriber.

At the publisher, Synapse performs the following steps. First, locks are acquired on the write dependencies. Then, for each dependency, a) Synapse increments ops, b) sets $version$ to ops in the case of a write dependency, and c) use $version$ for read dependencies and $version - 1$ for write dependencies as the version to be included in the final message. Next, the operation is performed, written objects are read back, and locks are released. Finally, the message is prepared and sent to subscribers. In our implementation, the publishing algorithm is slightly more complex due to 2PC protocols at every step of the algorithm to allow recovery at any point in case of failures. At the subscriber, Synapse maintains a version store that keeps track of the latest ops counter for each dependency. In contrast, the publisher maintains two counters per dependency. When the subscriber receives a message, it waits until all specified dependencies' versions in its version store are greater than or equal to those in the message, then processes the message and updates its version store by incrementing the *ops* counter for each dependency in the message. When subscribers publish messages (e.g., when decorating models), published messages include dependencies from reading other apps objects allowing cross-application dependencies. These *external* dependencies behave similarly to read dependencies, except they are not incremented at the publisher nor the subscriber, relaxing semantics to a level similar to traditional causal replication systems [25, 26].

With this mechanism in place, Synapse can enforce various update-message ordering semantics. To support global order delivery from the publisher, Synapse simply adds a write dependency on a global object for every operation which serializes all writes across all objects because all operations are serialized on the global object. To support causal order delivery from the publisher, Synapse serializes all updates within a controller by adding the previously performed update's first write dependency as a read dependency to the next update operation. To serialize all writes within a user context, it is sufficient to add the current user object as a write dependency to each write operation (shown in Fig. 8(b)). To support weak order delivery from the publisher, Synapse only tracks the write dependency for each object being updated. To support the same delivery mode at the subscriber as provided by the publisher, Synapse respects all the dependency information provided in the messages from the publisher. In the case of weak order delivery, the subscriber also discards any messages with a $version$ lower than what is stored in its version store. To support weaker delivery semantics at the subscriber, Synapse ignores some of the dependency information provided in the messages from the publisher. For example, a causal order delivery subscriber will ignore global object dependencies in the messages from a global order delivery publisher. Similarly, a weak order delivery subscriber will respect the dependency information in the message for the respective object being updated by the message, but ignore other dependency information from a global or causal order delivery publisher.

Fig. 8 shows an example of how the read and write dependencies translate into dependencies in messages. Both publisher and subscriber use causal delivery mode. Fig. 8(a) shows four code snippets processing four user requests. User1 creates a post, User2 comments on it, User1 comments back on it, and then User1 updates the post. Synapse automatically detects four different writes, automatically detects their dependencies, updates the version store, and generates messages as shown in Fig. 8(b). A subscriber processing these messages would follow the dependency graph shown in Fig. 8(c) by following the subscriber algorithm.

Tracking Dependencies. To enforce causal ordering semantics, Synapse discovers and tracks dependencies between

```
# Users are issuing requests
# against the application.
# '—' separates different
# controller executions.
current_user = User.find(1)
Post.create(
  author: current_user,
  body: "helo")
—
current_user = User.find(2)
post = Post.find(1)
Comment.create(post: post,
  author: current_user,
  body: "you have a typo")
—
current_user = User.find(1)
post = Post.find(1)
Comment.create(post: post,
  author: current_user,
  body: "thanks for noticing")
—
current_user = User.find(1)
post = Post.find(1)
post.update(body: "hello")
```
(a)

```
# Assume "post/id/1" hashes to "p1",
# "user/id/2" hashes to "u2", etc.
W1: write(...,
  read_deps: [],
  write_deps: ["user/id/1", "post/id/1"])
# u1.ops = 1, u1.version = 1
# p1.ops = 1, p1.version = 1
M1: {..., dependencies: ["u1": 0, "p1": 0]}
—
W2: write(...,
  read_deps: ["post/id/1"],
  write_deps: ["user/id/2", "comment/id/1"])
# u2.ops = 1, u2.version = 1
# c1.ops = 1, c1.version = 1
# p1.ops = 2, p1.version = 1
M2: {..., dependencies:
        ["u2": 0, "c1": 0, "p1": 1]}
—
W3: write(...,
  read_deps: ["post/id/1"],
  write_deps: ["user/id/1", "comment/id/2"])
# u1.ops = 2, u1.version = 2
# c2.ops = 1, c2.version = 1
# p1.ops = 3, p1.version = 1
M3: {..., dependencies:
        ["u1": 1, "c2": 0, "p1": 1]}
—
W4: write(...,
  read_deps: [],
  write_deps: ["user/id/1", "post/id/1"])
# u1.ops = 3, u1.version = 3
# p1.ops = 4, p1.version = 4
M4: {..., dependencies: ["u1": 2, "p1": 3]}
```
(b)

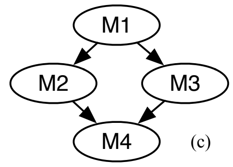

$$\begin{tikzcd} & M1 \arrow[dl] \arrow[dr] & \\ M2 \arrow[dr] & & M3 \arrow[dl] \\ & M4 & \end{tikzcd}$$
(c)

Figure 8: **Dependencies and Message Generation.** (a) shows controller code being executed at the publisher. (b) shows the writes Synapse instruments with their detected dependencies, along with the publisher's version store state updates in comments, and the resulting generated messages. (c) shows a dependency graph resulting from applying the subscriber algorithm. M2 and M3 are processed when the typo is present in the post.

operations. Synapse implicitly tracks data dependencies within the scope of individual controllers (serving HTTP requests), and the scope of individual background jobs (e.g., with Sidekiq [32]). Within these scopes, Synapse intercepts read and write queries to transparently detect corresponding dependencies. In contrast, prior work relies on explicit dependencies and requires their use with *all writes* [2, 3], which is onerous for developers and error-prone.

Synapse always infers the correct set of dependencies when encountering read queries that return objects, including joins. For example, when running a query of the form `"SELECT id,... FROM table WHERE ..."`, Synapse registers an object dependency on each returned row. Note that Synapse automatically injects primary key selectors in read queries if these are missing. In our experience, read queries returning objects constitute the vast majority of true dependency queries. The other types of read queries are aggregations (e.g., count) and their results are not true dependencies in practice. However, in the hypothetical case where one would need to track dependencies on such queries, Synapse lets developers express explicit dependencies with `add_read_deps` and `add_write_deps` to synchronize arbitrary read queries with any write queries. In over a dozen applications we integrated with Synapse, we have not encountered one single query that could be considered as a dependency but is not marked as such by Synapse automatically.

Synapse always infers the correct set of dependencies when encountering write queries. When encountering write queries issued to transactional DBs, Synapse infers the updated objects from the result of the write query (with `RETURNING *`), or by performing an additional read query. It registers these objects as write dependencies, which are later used during the commit operation. With non-transactional DBs, Synapse supports write queries that update at most one, well identified object (so the corresponding lock can be acquired before performing the write query). Multi-object updates queries are seldom used as their usage prevents model-defined callbacks to be triggered. However, when encountering such query, Synapse unrolls the multi-object update into single-object updates.

Scaling the Version Store. We implemented the version stores with Redis [39], an in-memory datastore. All Synapse operations are performed by atomically executing LUA scripts on Redis. This technique avoids costly round-trips, and simplifies the 2PC implementation in the algorithm. Scaling can be problematic in two ways. First, the version store can become a throughput bottleneck due to network or CPU, so Synapse shards the version store using a hash ring similar to Dynamo [15] and incorporates mechanisms to avoid deadlocks on subscribers as atomicity of the LUA scripts across shards can not be assumed. Second, the version store memory can be limiting, so Synapse hashes dependency names with a stable hash function at the publisher. This way, all version stores consume $O(1)$ memory. When a hash collision occurs between two dependencies, serialization happens between two unrelated objects, reducing parallelism. The number of effective dependencies that Synapse uses is the cardinal of the hashing function output space. Each dependency consumes around 100 bytes of memory, so a 1GB server can host 10M effective dependencies, which is more than enough in practice. As an interesting property, using a 1-entry dependency hash space is equivalent to using global ordering for both the publisher and its subscribers.

Transactions. When publishers support transactions, we enforce the same atomicity in messages delivery, allowing these properties to hold for subscribers. All writes in a transaction are included in the same message. Subscribers process messages in a transaction with the highest level of isolation and atomicity the underlying DB permits (e.g., logged batched updates with Cassandra). At the publisher, we also hijack the DB driver's transaction commit functions and execute the transaction as a two-phase commit (2PC) transaction instead. The 2PC lets us ensure that either the following operations all happen or that none do: (1) commit the transaction locally, (2) increment version dependencies and (3) publish the message to the reliable message broker. As an optimization, the publisher algorithm does not attempt to lock write dependencies as the underlying DB retains locks on the written objects until the commit is persisted.

4.3 Live Schema Migrations

When deploying new features or refactoring code, it may happen that the local DB schema must be changed, or new data must be published or subscribed. A few rules must be respected: 1) Updating a publisher DB schema must be done in isolation such that subscribers are not able to observe the internal changes done to the publisher. For example, before removing a published attribute from the DB, a virtual attribute of the same name must be added. 2) The semantics of a published attribute must not change; e.g., its type must not change. Instead of changing the semantics of a published attribute, one can publish a new attribute, and eventually stop publishing the old one. 3) Publishing a new attribute is often motivated by the need of a subscriber. When adding the same attribute in a publisher and subscriber, the publisher must be deployed first. Finally, once the new code is in place, a partial data bootstrap may be performed to allow subscribers to digest newly subscribed data.

4.4 Bootstrapping and Reliability

When a new subscriber comes online, it must synchronize with the publisher in a three-step bootstrapping process. First, all current publisher versions are sent in bulk and saved in the subscriber's version store. Second, all objects in the subscribed model are sent and persisted to the subscriber's DB. Third, all messages published during the previous steps are processed to finish synchronizing the objects and versions. Once all messages are processed, the subscriber is now *in sync* and operates with the configured delivery semantics. Subscriber code may call `Synapse.bootstrap?` to determine whether Synapse is still bootstrapping or in sync. Fig. 2 shows an example of how the mailer subscriber checks for bootstrapping completion before sending emails.

Should the subscriber fail, its queue may grow to an arbitrary size. To alleviate this issue, Synapse decommissions the subscriber from the Synapse ecosystem and kills its queue once the queue size reaches a configurable limit. If the subscriber comes back, Synapse initiates a *partial bootstrap* to get the application back in sync.

Failures may also happen when the version store dies on either the publisher or subscriber side. When the subscriber's version store dies, a partial bootstrap is initiated. When the publisher's version store dies, a generation number reliably stored (e.g., Chubby [8], or ZooKeeper [47]) is incremented and publishing resumes. Messages embed this generation number as shown on Fig. 6(b). When subscribers see this new generation number in messages, they wait until all the previous generation messages are processed, flush their version store, and process the new generation messages. This generation change incurs a global synchronization barrier and temporarily slows subscribers.

4.5 Testing Framework

Synapse provides a solid testing framework to help with development and maintenance of apps. For instance, Synapse

DB	ORM	Pub?	Sub?	ORM LoC	DB LoC
PostgreSQL	ActiveRecord	Y	Y	474	44
MySQL	ActiveRecord	Y	Y	”	52
Oracle	ActiveRecord	Y	Y	”	47
MongoDB	Mongoid	Y	Y	399	0
TokuMX	Mongoid	Y	Y	”	0
Cassandra	Cequel	Y	Y	219	0
Elasticsearch	Stretcher	N/A	Y	0	0
Neo4j	Neo4j	N	Y	0	0
RethinkDB	NoBrainer	N	Y	0	0
Ephemerals	N/A	Y	N/A	N/A	N/A
Observers	N/A	N/A	Y	N/A	N/A

Table 3: **Support for Various DBs.** Shows ORM- and DB-specific lines of code (LoC) to support varied DBs. For ORMs supporting many DBs (e.g., ActiveRecord), adding a new DB comes for free.

statically checks that subscribers don’t attempt to subscribe to models and attributes that are unpublished, providing warnings immediately. Synapse also simplifies integration testing by reusing model factories from publishers on subscribers. If a publisher provides a model factory [17] (i.e., data samples), then developers can use them to write integration tests on the subscribers. Synapse will emulate the payloads that would be received by the subscriber in a production environment. This way, developers are confident that the integration of their ecosystem of applications is well tested before deploying into the production environment.

4.6 Supporting New DBs and ORMs

Adding subscriber support for a new ORM is trivial: a developer need only map the CRUD operations (create/read-/update/delete) to Synapse’s engine. To add publisher support for a new ORM, a developer needs to first plug into the ORM’s interfaces to intercept queries on their way to the DB (all queries for causality and writes for replication). Then, the developer needs to add two phase commit hooks to the DB driver for transactional DBs (as discussed in §4.2).

To illustrate the effort of supporting new DBs and ORMs, we report our development experience on the nine DBs listed in Table 3. A single developer implemented support for our first DB, PostgreSQL in approximately one week, writing 474 lines of code specific to the ORM (ActiveRecord) and 44 lines specific to the DB for two phase commit. After building support for this DB, supporting other SQL DBs, such as MySQL and Oracle, was trivial: about 50 lines of DB-specific, each implemented in only several hours. Supporting subsequent DBs (e.g., MongoDB, TokuMX, and Cassandra) was equally easy and took only a few days and 200-300 lines of code per ORM. We find that supporting various DBs is a reasonable task for an experienced programmer.

5. Applications

We and others have built or modified 14 web applications to share data with one another via Synapse. Those built by others have been deployed in production by a startup, Crowdtap. The applications we built extend popular open-source apps to integrate them into data-driven ecosystems. Overall, our development and deployment experience has been positive: we made *no logical changes* to application code, only adding on average a single line of configuration per attribute of each model published.

5.1 Synapse at Crowdtap

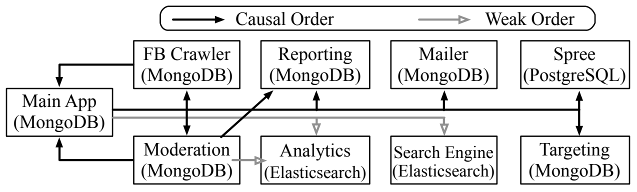
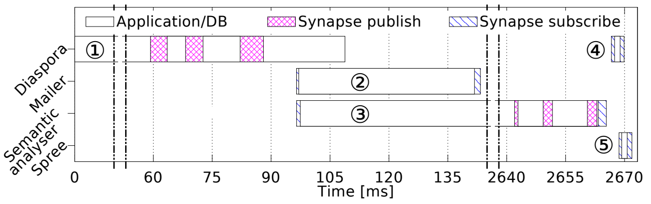


Figure 10: **Crowdtap’s services.** Arrows show Synapse connections.

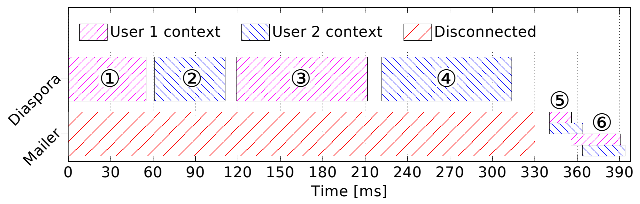
Crowdtap is an online marketing-services company contracted by major brands such as Verizon, AT&T, Sony and MasterCard. Crowdtap has grown rapidly since its founding, and by October 2014 has seen over 450,000 users. As Crowdtap grew and gained more clients, both their application offering and development team evolved. At first, engineers attempted to enhance their core application by building new features directly into the same codebase and DB. However, as new features were added, the data was used in different ways, requiring different indexes and denormalization, bloating the DB. When features were canceled, traces of their schema changes were often left orphaned. Moreover, it was difficult to bring newly hired engineers up to speed with the complex and rapidly evolving codebase and DB. To alleviate this issue, engineers factored out some features and used synchronous APIs to connect them to the core DB, but found this technique difficult to get right. Specifically, a bug in an e-commerce service, which accessed user data from Crowdtap’s core app via a synchronous API, was able to bring down the entire app due to the lack of performance isolation. Installing rate limits between the components of their app was not good option for Crowdtap, and they preferred the de-coupling that a replication-based solution cross-service would provide. Synchronization of these DBs then became a challenge.

To address these challenges, Crowdtap began to experiment with Synapse, first to provide synchronization between their original core app and a separate targeting service. These services had previously communicated over a synchronous API; this API was factored out, and replaced with Synapse, reducing the app from 1500 LoC to 500 LoC. While previous attempts to integrate this feature required deep knowledge of the core application held by senior engineers, this integration was performed by a newly-hired engineer thanks to the abstractions provided by Synapse.

After this initial integration, two other Crowdtap engineers extracted two other business features, the mailer and the analytics engine from the main application, using Synapse. The email service subscribes to 24 of the core service’s models to isolate all notification related aspects within



(a) **Execution Sample:** ① a user posts on Diaspora. The mailer ② and semantic analyzer ③ receive the post in parallel. Diaspora ④ and Spree ⑤ each receive the decorated model with in parallel.



(b) **Execution with Subscriber Disconnection:** ① and ③ User 1 posts. ② and ④ User 2 posts. Mailer comes online and processes the each users first request ⑤, then each user's second request ⑥ in parallel.

Figure 9: **Execution Samples in Social Ecosystem.** Time grows from left to right on x axis. (a) Execution in the open-source ecosystem when a user posts a new message. (b) Execution when two users post messages with the mailer disconnected.

their application, while the analytics engine subscribes to a similar number of models.

Crowdtap's engineering management was so pleased with the ease of development, ease of maintenance, and performance of Synapse, that after this experience, all major features have been built with it. Fig. 10 shows the high-level architecture of the Synapse ecosystem at Crowdtap with the main app supported by eight microservices. Synapse-related code is trivial in size and logic. The Crowdtap main app consists of approximately 17,000 lines of ruby code, and publishes 257 attributes of 53 different models. However, Synapse-related configuration lines are minimal: only 360 (less than 7 lines of configuration per model, on average).

Crowdtap has chosen different delivery semantics for their various subscribers (shown with different arrows in Fig.10). While all publishers are configured to support causal delivery mode, subscribers are configured with either causal or weak delivery modes, depending on their semantics requirements. The mailer service registers for data from the main app in causal mode so as to avoid sending inconsistent emails. In contrast, the analytics engine lacks stringent order requirements, hence selects a weak consistency mode.

5.2 Integrating Open-Source Apps with Synapse

We used Synapse to build a new feature for *Spree*, a popular open source e-commerce application that powers over 45,000 e-commerce websites world wide [40]. By integrating *Diaspora*, a Facebook-like open source social networking application and *Discourse*, an open source discussion board, with Spree, we were able to create a social-based product recommender. Fig.11 shows the architecture of the ecosystem. We started by configuring Diaspora and Discourse to publish the models for posts, friends, and access control lists. We needed to add only several lines of declarative configuration each app: 23 for Diaspora (compared to its 30k lines of code), 5 for Discourse (compared to its 21k lines), and 7 for Spree (compared to its 37k lines).

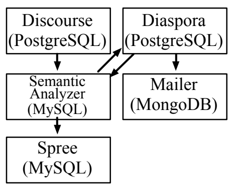


Figure 11: **Social Product Recommender.** Arrows show Synapse connections.

Next, we built a semantic analyzer that subscribes to these posts and extracts topics of interest, decorating Users with apparent topics of interest (using an out-of-the-box semantic analyzer, Textalytics [42]). The analyzer publishes its decorated `User` model (with user interests) to Spree.

Finally, since Spree did not have *any* recommendation mechanism in place, we added several lines of code to it to implement generic targeted searching. With this code in place, one can construct as complex of a recommendation engine as desired, although our prototype uses a very simple keyword-based matching between the users' interests and product descriptions. Such code need not be concerned with where the user's interests come from as they automatically exist as part of the data model (thanks to Synapse).

6. Evaluation

We leverage both our deployment and the applications we built to answer three core evaluation questions about Synapse: (Q1) How expensive is it at the publisher? (Q2) How well does it scale? (Q3) How do its various delivery modes compare? and (Q4) How useful is it in practice?

To answer these questions, we ran experiments on Amazon AWS with up to 1,000 c3.large instances (2-core, 4GB) running simultaneously to saturate Synapse. We use the in-memory datastore Redis [39] for version stores. As workloads, we used a mix of Crowdtap production traffic and microbenchmarks that stress the system in ways that production workload cannot. Unless otherwise noted, our evaluation focuses on the causal delivery mode for both publishers and subscribers, which is the default setting in our prototype. After providing some sample executions, we next discuss each evaluation question in turn.

6.1 Sample Executions

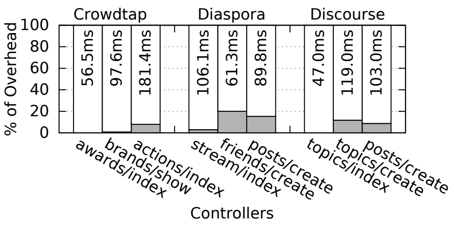
To build intuition into how Synapse behaves and the kinds of overheads it brings, we show two sample executions of our open-source ecosystem applications (see §5.2). All applications are configured with a causal delivery mode, hence the examples reflect this mode's functioning.

Fig. 9(a) shows a timeline of the applications' execution starting with a user's post to Diaspora and ending with Spree's receipt of the semantically-enhanced User model. We observe that Synapse delivers messages shortly after publication (within 5ms), in parallel to both the mailer and the semantic analyzer. Fig. 9(b) illustrates visually Synapse's causal engine in action. It shows two users posting mes-

Most Popular Controllers	% Calls (of 170k)	Published Messages		Dependencies per Message		Controller Time (ms)		Synapse Time (ms)	
		mean	99^{th}	mean	99^{th}	mean	99^{th}	mean	99^{th}
awards/index	17.0%	0.00	0	0.0	0	56.5	574.1	0.0 (0.0%)	0
brands/show	16.0%	0.03	2	1.0	2	97.6	333.4	0.8 (0.8%)	44.9
actions/index	15.0%	0.67	3	17.8	339	181.4	1676.8	14.4 (8.6%)	114.7
me/show	12.0%	0.00	0	0.0	0	14.7	39.3	0.0 (0.0%)	0
actions/update	11.5%	3.46	6	1.8	4	305.9	759.0	84.1 (37.9%)	207.9

Overhead across all 55 controllers: mean=8%

(a) **Synapse Overheads at Crowdtap**



(b) **Synapse Overheads in Real Applications**

Figure 12: **Application Publishing Overheads.** (a) Crowdtap dependencies and overheads, sampled from production data. For each of the five most frequently invoked controllers in Crowdtap, shows the percent of calls to it, average number of published messages, average number of dependencies between messages, the average controller execution time, and the average overhead from Synapse. (b) Synapse overhead (gray parts) for 3 controllers in 3 different applications. Labels give the total controller times. *Synapse publisher overheads are small.*

sages on different Diaspora profiles app while a Mailer subscriber is deployed to notify a user's friends whenever the user makes a new post. Initially, the mailer is disconnected. When the mailer comes back online, it processes messages from the two users in parallel, but processes each user's posts in serial order, thereby enforcing causality.

6.2 Application Overheads (Q1)

We next evaluate Synapse's publishing overheads in the context of real applications: Crowdtap and the open-source apps we modified. For Crowdtap, we instrumented Crowdtap's main Web application to record performance metrics and recorded accesses to the application over the 24 hour period of April 16, 2014. In total, we recorded one fifth of the traffic totaling 170,000 accesses to application controllers. For each controller, we measured the number of published messages, the number of dependencies per message published, the total execution time of the controller, and the Synapse execution time within these controllers. For each of these quantities, we measured the arithmetic mean and 99^{th} percentile.

Fig. 12(a) shows our results. In total, 55 controllers were invoked. We show average overheads across them, as well as detailed information about the five most frequently accessed controllers, which account for over 70% of the traffic. On average, Synapse overheads are low: 8%. For the most popular two controllers (`awards/index` and `brands/show`), the overheads are even lower: 0.0-0.8%. This is because they exhibit very few published messages (writes). As expected, Synapse overhead is higher in controllers that publish more messages, showing an average overhead of 37.9% for the controller `actions/update`. The number of dependencies stays low enough to not become a bottleneck with `actions/index` showing 17.8 dependencies per message on average. To further illustrate Synapse's impact, we also show the 99^{th} percentile of controller execution time and Synapse's execution time within the controller. Some controller executions are long ($>$ 1 second). These high response times may be attributed to network issues, and Ruby's garbage collector. Indeed, Synapse is unlikely to be the cause of these latency spikes as the 99^{th} percentile of its execution time remains under 0.2s.

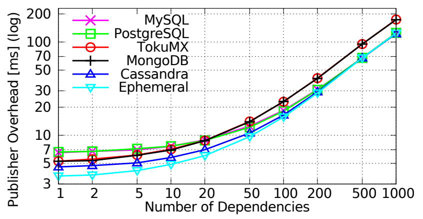
To complement our Crowdtap results, we measured controllers in our open-source applications, as well. Fig. 12(b) shows the Synapse overheads for several controllers within Diaspora and Discourse (plus Crowdtap for consistency). Grey areas are Synapse overheads. Overheads remain low for the two open-source applications when benchmarked with synthetic workloads. Read-only controllers, such as `stream/index` and `topics/index` in Diaspora and Discourse, respectively, exhibit near-zero overheads; write controllers have up to 20% overhead.

These results show that Synapse overheads with real applications are low and likely unnoticeable to users. However, the results are insufficient to assess performance under stress, a topic that we discuss next.

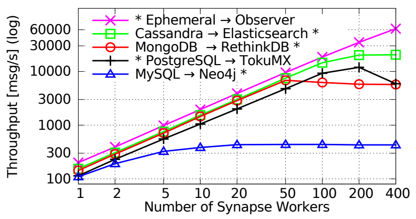
6.3 Scalability (Q2)

To evaluate Synapse throughput and latency under high load, we developed a stress-test microbenchmark, which simulates a social networking site. Users continuously create posts and comments, similar to the code on Fig. 8. Comments are related to posts and create cross-user dependencies. We issue traffic as fast as possible to saturate Synapse, with a uniform distribution of 25% posts and 75% comments. We run this experiment by deploying identical numbers of publishers and subscribers (up to 400 for each) in Amazon AWS. We use several of our supported DBs and combinations as persistence layers. We applied different DBs as publishers and subscribers. We measure a variety of metrics, including the overheads for creating a post, as well as Synapse's throughput.

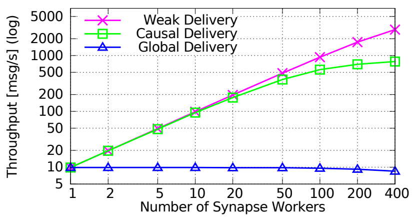
Overheads under Heavy Load. Fig. 13(a) shows the overheads for different DBs with increasing numbers of dependencies. Focusing on the one-dependency case (x=1), Synapse adds overheads ranging from 4.5ms overhead on Cassandra to 6.5ms on PostgreSQL. This is in comparison to the 0.81ms and 1.9ms latencies that PostgreSQL and Cassandra, respectively, exhibit *without* Synapse. However, compared to realistic Web controller latencies of tens of ms, these overheads are barely user-visible. As the number of dependencies increases, the overhead grows slowly at first, remaining below 10ms for up to 20 dependencies. It then shoots up to a high 173ms for 1,000 dependencies. Fortunately, as shown in Fig. 12(a), dependencies in real applications remain low enough to avoid causing a bottleneck.



(a) **Publisher Overhead vs. Dependencies**



(b) **Throughput on Different DBs**



(c) **Throughput on Various Delivery Semantics**

Figure 13: **Microbenchmark Results.** (a) Publisher overhead on different DBs. (b) Throughput vs number of workers end to end benchmark. Each line represents a different DB setup. The slowest end in each pair is annotated with a (*) symbol. (c) Throughput vs number of workers with subscribers running a 100ms callback. Each line represents a different delivery mode. *Causal and weak delivery modes scale well.*

Cross-DB Throughputs. Fig. 13(b) shows how Synapse's end-to-end throughput scales with the number of publisher/-subscriber workers, for various DB combinations, as well as for our DB-less models (observer to ephemeral). We keep the number of dependencies per message constant at 4 and shard the version stores on 80 AWS instances. We have not sharded any of the DBs. For ephemerals, Synapse scales linearly with the number of workers, reaching a throughput of more than 60,000 msg/s. Even at such high rates, Synapse does not become a bottleneck. When DBs are used to back the publishers and subscribers, the throughput grows linearly with the number of workers until one of the DBs saturates. Saturation happens when the slowest of the publisher and subscriber DBs reaches its maximum throughput. For each combination, we label the limiting DB with a *. For instance, PostgreSQL bottlenecks at 12,000 writes/s, and Elasticsearch at 20,000 writes/s.

6.4 Delivery Semantic Comparison (Q3)

Synapse supports three delivery modes – global, causal, and weak – which provide different scaling properties. Fig.13(c) compares subscriber scalability with increased number of subscriber workers available to process writes in parallel. We configure subscribers with a 100-ms callback delay to simulate heavy processing, such as sending emails. Each line shows a different delivery mode, for which we both configure the publisher and the subscriber to operate under that delivery mode. The global delivery mode, which requires the subscriber to commit each write serially, scales poorly. The causal delivery mode, which only requires the subscriber to serialize dependent updates, provides much better scalability. Its peak throughput is limited by the inherent parallelism of the workload. Finally, the weak delivery mode scales perfectly, never reaching its peak up to 400 subscriber workers. In practice, we recommend choosing causal for the publisher and either the causal or weak mode for subscribers.

6.5 Production Notes (Q4)

Crowdtap has given us very positive feedback on Synapse's usability and value, including the following interesting stories from their use of Synapse in production.

Supports Heavy Refactoring: Crowdtap discovered a new use for Synapse that we had not anticipated: implementing live DB migrations. Unhappy with MongoDB's performance, they migrated their Main App to TokuMX, another document-oriented DB. To do so, they bootstrapped a subscriber app implementing the same functionality as the original app but running on TokuMX. The subscriber registered for all the Main App's data. Once it was up to date, developers just switched their load balancer to the new application and the migration was completed with little downtime. They also applied this mechanism to address otherwise difficult schema migration challenges. For example, after performing a heavy refactor on one of their services, instead of updating the deployed service, they deployed the new version as a different service with its own DB to ensure that everything was running as expected. For some period of time, the two different versions of the same service run simultaneously, enabling the team to perform QA on the new version, while keeping the possibility to rollback to the old version if needed. This mechanism allow no downtime procedures.

Supports Agile Development: A key aspect in a startup company is agility. New features must be rolled out quickly and securely evaluated. According to Crowdtap, Synapse helps with that. One developer said: "It allows us to be very agile. We can experiment with new features, with real data coming from production." For example, during a hackathon, one of the developers implemented a new reporting prototype. He was able to subscribe to real time production data without impacting the rest of the system thanks to Synapse's isolation properties. The business team immediately adopted this reporting tool, and has been using it ever since.

Flexible Semantic Matters: Interestingly, Crowdtap initially configured all of its services to run in causal mode. However, during an upgrade of RabbitMQ, the (otherwise reliable) message queuing system that Synapse relies upon, some updates were lost due to an upgrade failure. Two subscribers deadlocked, and their queues were filling up, since they were missing dependencies and could not consume the updates. After timeouts, Synapse's recovery mechanisms, which rebootstrap the subscribers, kicked in and the system was unblocked. However, the subscriber apps were unavailable for a long period of time. Crowdtap now chooses between causal and weak delivery modes for each of its subscribers, taking into account its availability/consistency needs. It is not an easy choice, but it *can* and *must* be done in a production environment where even reliable components can fail. We recommend other engineers implementing causal systems to make message loss recovery an integral part of their system design, specifically to avoid human intervention during failures. Ideally, subscribers would operate in causal mode, with a mechanism to give up on waiting for late (or lost) messages, with a configurable timeout. Given these semantics, Synapse's weak and causal modes are achieved with the timeout set to $0s$ and ∞, respectively.

7. Related Work

Synapse builds on prior work in DB replication, data warehousing, federated DBs, publish/subscribe systems, and consistency models. We adopt various techniques from these areas, but instantiate them in unique ways for the domain of modern MVC-based applications. This lets us break through challenges incurred by more general prior approaches, and design the first real-time service integration system that supports heterogeneous DBs with strong delivery semantics.

Same-DB Replication. The vast majority of work in DB replication, as surveyed in [10], involves replicating data across different instances of *the same DB engine* to increase the DB's availability, reliability, or throughput [13]. Traditional DB replication systems plug in at low levels [10], which makes them DB specific: e.g., they intercept updates inside their engines (e.g., Postgres replication [34]), between the DB driver and the DB engine (e.g., MySQL replication [30]), or at the driver level (e.g., Middle-R [31]). Synapse operates at a much higher level – the ORM – keeping it largely independent of the DB.

Data Warehousing and Change Capture Systems. Data warehousing is a traditional approach for replicating data across heterogeneous DBs [12, 16]. While many warehousing techniques [11, 20, 27, 28, 45, 46] are not suitable for real-time integration across SQL and NoSQL engines. Replication is usually implemented either by installing triggers that update data in other DBs upon local updates, or by tailing the transaction log and replaying it on other DBs, as LinkedIn's Databus does [14]. Although transaction logs are often available, it is generally agreed that parsing these logs is extremely fragile since the logs are proprietary and not guaranteed to be stable across version updates. Synapse differs from all of these systems by replicating at the level of ORMs, a much more generic and stable layer, which lets it replicate data between both SQL and NoSQL engines.

In general, existing systems for replicating between SQL and NoSQL DBs, such as MoSQL [28], or MongoRiver [27] work between only specific pairs of DBs, and offer different programming abstractions and semantics. In contrast, Synapse provides a unified framework for integrating heterogeneous DB in realtime.

DB Federation. The DB community has long studied the general topic of integrating data from different DBs into one application, a topic generally known as DB federation [36]. Like Synapse, federation systems establish a translation layer between the different DBs, and typically rely on DB views – materialized or not – to perform translations. Some systems even leverage ORMs to achieve uniform access to heterogeneous DBs [4]. However, these systems are fundamentally different from Synapse: they let the *same* application access data stored in different DBs uniformly, whereas Synapse lets *different* applications (subscribers) replicate data from one DB (the publisher). Such replication, inspired by service-oriented architectures, promotes isolation and lets the subscribers use the best types of DBs, indexes, and layouts that are optimal for each case.

Similar to DB federation is projects that aim to create "universal" ORMs, which definite a common interface to all DBs (SQL or otherwise), such as Hibernate [37], DataMapper [23] and CaminteJS [19]. Such ORMs should in theory ease development of an application that accesses data across different DBs, a problem complementary to that which Synapse solves. However, since they expose a purely generic interface, such an ORM will encourage a design that does not cater to the individual features provided by each DB. In contrast, Synapse encourages developers to use different ORMs for different sorts of DBs, providing a common programming abstraction to replicate the data across them.

Publish/Subscribe Systems. Synapse's API is inspired by publish/subscribe systems [1, 9, 33, 38, 41]. These systems require programmers to specify which messages should be included in which unit of order, while Synapse *transparently* intercepts data updates, compiles their dependencies automatically, and publishes them.

8. Conclusions

Synapse is an easy-to-use framework for structuring complex, heterogeneous-database Web applications into ecosystems of microservices that integrate data from one another through clean APIs. Synapse builds upon commonly used abstractions provided by MVC Web frameworks, such as Ruby-on-Rails, Python Django, or PHP Symfony. It leverages models and ORMs to perform data integration at data object level, which provides a level of compatibility between both SQL and NoSQL DBs. It leverages controllers to support application-specific consistency semantics without sacrificing scalability. We have implemented Synapse for Ruby-on-Rails, shown that it provides good performance and scalability, released it on GitHub [43], and deployed it in production to run the microservices for a company.

9. Acknowledgments

Kareem Kouddous and Michael D'Auria were instrumental in implementing, deploying and maintaining Synapse at Crowdtap. Atul Adya, Magda Balazinska, Ken Ross, and Serdar Tasiran provided helpful comments on the paper. This work was supported by a Google Research Award, NSF CNS-1162447, CNS-1422909, CCF-1162021, CNS-1351089, and DARPA FA8650-11-C-7190.

References

[1] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A Client Notification Service for Internet-Scale Applications. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[2] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The Potential Dangers of Causal Consistency and an Explicit Solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, Oct. 2012.

[3] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2013.

[4] H. Balsters and B. Haarsma. An ORM-Driven Implementation Framework for Database Federations. In R. Meersman, P. Herrero, and T. S. Dillon, editors, *OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 659–670. Springer, Nov. 2009.

[5] R. Barcia, G. Hambrick, K. Brown, R. Peterson, and K. Bhogal. *Persistence in the Enterprise: A Guide to Persistence Technologies*. IBM Press, first edition, May 2008.

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[7] M. Bowers. Database Revolution: Old SQL, New SQL, NoSQL... Huh? `https://www.marklogic.com/resources/database-revolution-old-sql-new-sql-nosql-huh/`, Apr. 2013.

[8] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2006.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Trans. Comput. Syst.*, 19(3), Aug. 2001.

[10] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based Database Replication: The Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD)*, June 2008.

[11] G. K. Y. Chan, Q. Li, and L. Feng. Design and Selection of Materialized Views in a Data Warehousing Environment: A Case Study. In *Proceedings of the 2nd ACM International Workshop on Data Warehousing and OLAP (DOLAP)*, Nov. 1999.

[12] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, Mar. 1997.

[13] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. arno Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, Aug. 2008.

[14] S. Das, C. Botev, and K. Surlaker, et.al. All Aboard the Databus! LinkedIn's Scalable Consistent Change Data Capture Platform. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, Oct. 2012.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[16] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, July 2012.

[17] J. Ferris and J. Clayton. Factory Girl. A library for setting up Ruby objects as test data. `https://github.com/thoughtbot/factory_girl`.

[18] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, Nov. 2002.

[19] A. Gordeyev. Camintejs. `http://www.camintejs.com`.

[20] H. Gupta and I. S. Mumick. Selection of Views to Materialize in a Data Warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):24–43, Jan. 2005.

[21] Java Message Service. A Java message oriented middleware API for sending messages between two or more clients. `http://en.wikipedia.org/wiki/Java_Message_Service`.

[22] Kafka. A High-Throughput Distributed Messaging System. `http://kafka.apache.org`.

[23] D. Kubb. Datamapper. `http://datamapper.org`.

[24] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2010.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[26] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[27] R. Louapre. A MongoDB to Elasticsearch Translator. `https://github.com/richardwilly98/elasticsearch-river-mongodb/`.

[28] MoSQL. A MongoDB to SQL Streaming Translator. `https://github.com/stripe/mosql`.

[29] S. Newman. *Building Microservices – Designing Fine-Grained Systems*. O'Reilly Media, Feb. 2015.

[30] Oracle Corporation. MySQL 5.5 Reference Manual – Replication. `http://dev.mysql.com/doc/refman/5.5/en/replication.html`.

[31] M. Patiño Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst.*, 23(4), Nov. 2005.

[32] M. Perham. Sidekiq. Simple, efficient background processing for Ruby. `http://sidekiq.org`.

[33] P. R. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, July 2002.

[34] Postgres-R. A Database Replication System for PostgreSQL. `http://www.postgres-r.org`.

[35] RabbitMQ. Messaging That Just Works. `http://www.rabbitmq.com`.

[36] R. Ramakrishnan and J. Gehrke. *Database Management Systems, 3rd Edition*. McGraw-Hill Education, Aug. 2002.

[37] Red Hat Community. Hibernate. `http://hibernate.org`.

[38] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Proceedings of the International COST264 Workshop on Networked Group Communication*, Nov. 2001.

[39] S. Sanfilippo. Redis. An Advanced Key-Value Store. `http://redis.io`.

[40] Spree. Ecommerce Platform. `http://spreecommerce.com`.

[41] R. Storm, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 1998.

[42] Textalytics. Web Services for Text Analysis and Mining. `https://textalytics.com/`.

[43] N. Viennot. Synapse Sources. `https://github.com/nviennot/synapse`.

[44] N. Viennot, S. Nair, and J. Nieh. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2013.

[45] K. Wang. A DynamoDB to Elasticsearch Translator. `https://github.com/kzwang/elasticsearch-river-dynamodb`.

[46] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, Aug. 1997.

[47] ZooKeeper. Centralized Service for Maintaining Configuration Information, Naming, and Providing Distributed Synchronization. `http://hadoop.apache.org/zookeeper/`.