# Thinking in Events: From Databases to Distributed Collaboration Software

Keynote at the 15th ACM International Conference on Distributed and Event-Based Systems (DEBS)

Martin Kleppmann
University of Cambridge
Cambridge, UK
mk428@cst.cam.ac.uk

## ABSTRACT

In this keynote I give a subjective but systematic overview of the landscape of distributed event-based systems, with an emphasis on two areas I have worked on over the last decade: large-scale stream processing with Apache Kafka and associated tools, and real-time collaboration software in the style of Google Docs. While these may seem at first glance to be very different topics, there are also important points of overlap. This paper lays out a taxonomy of event-based systems that shows where their commonalities and differences lie. It also highlights some of the key trade-offs that arise in the implementation of event-based systems, drawing both from distributed systems theory and from experience of their practical deployment. Finally, the paper outlines a number of open research problems in this field.

## CCS CONCEPTS

• **Information systems** → **Data streams**; • **Software and its engineering** → **Publish-subscribe / event-based architectures**; • **Applied computing** → *Event-driven architectures*; • **Theory of computation** → *Distributed computing models.*

## KEYWORDS

stream processing, event sourcing, state machine replication, CRDTs, real-time collaboration

**ACM Reference Format:**
Martin Kleppmann. 2021. Thinking in Events: From Databases to Distributed Collaboration Software: Keynote at the 15th ACM International Conference on Distributed and Event-Based Systems (DEBS). In *The 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21), June 28-July 2, 2021, Virtual Event, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3465480.3467835

## 1 INTRODUCTION

The term *event* means many different things in different branches of computing. The goal of this paper is to illuminate, categorise, and differentiate some of the main families of event-based systems, with an emphasis on systems that are also distributed.

Section 2 presents a taxonomy that breaks down the field into categories based on simple technical criteria. Each category is illustrated with examples of practical systems that employ the approach, and it includes a discussion of the trade-offs: the pros and cons of choosing one category over another. My hope is that this taxonomy will provide a useful structure and vocabulary to researchers and practitioners working with event-based systems, allowing us to more easily communicate and reason about the fundamental design choices of such systems.

Next, in Section 3 I present a personal perspective on two particular families of event-based systems that I have worked on over the last decade: Apache Kafka and its stream processing ecosystem (which I worked on in 2012–2015 during my time as industrial software engineer at LinkedIn), and multi-user real-time collaboration software in the style of Google Docs (which has been the focus of my research since returning to academia in 2015). Reflecting on these systems in the context of the taxonomy of Section 2 helps further illuminate the characteristics of event-based software architecture.

Finally, Section 4 outlines some open problems that, I believe, deserve further attention from researchers.

## 2 A TAXONOMY OF EVENT-BASED SYSTEMS

The following taxonomy categorises the different uses of events based on the flowchart in Figure 1. Such a broad-brush taxonomy cannot necessarily capture all of the nuance of the field, and sometimes the boundaries between categories can be blurry. Nevertheless, I hope that it will be a useful tool for understanding event-based systems.

### 2.1 Notifications and persistence

At a high level, an event can be:

- a notification about the fact that something happened;
- a persistent record of the fact that something happened; or
- both.

With "notification" I mean that shortly after an event occurs, the system invokes application code that may act on the event. Examples of notification events occur in many applications. JavaScript code running in a web browser can register functions to be called when the user does something, such as clicking or typing a key on the keyboard: here the click or the key-press is the event [53]. Most other user interface frameworks have a similar system of dispatching user input events to callback functions or *event handlers*. Many operating systems offer asynchronous (non-blocking) I/O functionality, in which case a thread runs an *event loop* that


*DEBS '21, June 28-July 2, 2021, Virtual Event, Italy*

ACM ISBN 978-1-4503-8555-8/21/06.
https://doi.org/10.1145/3465480.3467835


**Figure 1: A taxonomy of event-based systems.**

processes notifications from the OS when requested I/O operations complete [48]. Discrete event simulations employ this type of event for simulated, rather than physical, occurrences [52]. Reactive and functional reactive programming (FRP) offer higher-level APIs, such as a dataflow programming model, for working with *streams of events* [6, 16]. In all of these examples, an event is something that exists only in-memory in a single process; it is ephemeral, not persistent, in the sense that it is not normally written to disk. Ephemeral events exist only for the lifetime of a process and are not preserved across restarts.

In contrast, persistent events do not necessarily have a notification element. For example, time-series databases are often used to record events that occurred over time, such as periodic readings from a sensor, price updates of a financial asset, and tracking the activity of a person or machine [44]. Another example is the fact table that appears in the star or snowflake schema of data warehouses and business analytics systems: every row in the fact table is typically a timestamped record of an event that happened, such as the purchase of a particular product by a particular customer [32]. In such systems, an event is a value that is recorded in a database for future analysis, but the receipt of the event does not necessarily trigger the immediate execution of any application code. The primary purpose of the events from the user's point of view is to allow the retroactive querying and analysis of the event history, such as examining trends and generating reports showing the change of some metric over time.

Some systems combine these characteristics such that an event is both a persistent record of the fact that something happened, and also a notification (i.e. application code is called to handle the occurrence of an event). I will broadly group such systems under the term *stream processing*, and break them down into subcategories later.

Some databases provide facilities such as *materialized view maintenance* and *continuous queries* (queries whose results are automatically updated as the underlying data changes), which can also be regarded as a form of notification [15, 24]. Many programming models for distributed systems are based on sending and receiving messages, and the receipt of a message can also be seen as a notification event that triggers the execution of application code. For example, the *actor model* is a widely-used approach in which multiple actors or lightweight threads, potentially located on different machines, communicate by sending each other messages [65].

Persistence is a fuzzy concept in practice, because an event may pass through multiple systems, some of which may write it to disk and others may treat it as ephemeral (i.e. it might be lost if a node crashes or if the network is unreliable). For example, some actor frameworks and messaging middleware systems also include persistence mechanisms for actor state and/or messages [7]; however, in many message brokers, messages are stored only until they have been successfully delivered to a consumer, and then deleted. On the other hand, databases typically store data indefinitely until it is explicitly deleted. For the purposes of this taxonomy, the exact persistence characteristics of a system are not crucial, although many of the systems we discuss later lean more towards the database-like, long-term-storage end of the spectrum.

## 2.2 Windowing in stream processing

Zooming in on the systems that provide both persistence and notifications, we can further differentiate them by examining the kind of logic that can be performed in response to an event. The very simplest stream processors use stateless operators, in which every event can be processed independently of any other event, based only on the information in that event (for example, selecting events where a certain property of the event falls within some set of values). However, stateless processing is rarely sufficient, and most interesting applications also require event processing logic that combines information from several events: in database parlance, a *join* or *grouping* of events [12]. Events may be combined by aggregation (e.g. computing the count, sum, or average of some property of a set of events), by emitting compound events that include properties from several input events, or by more complex state machines that we discuss in Section 2.4.

Some stream processing systems are designed primarily for applications in which the events that need to be combined occur fairly close together in time. For example, a stock trading system may need to compute the minimum and maximum prices of an asset per hour or per day, or a fraud detection system may need to detect unusual patterns of recent activity on a customer's credit card. Such operations are known as *windowed* joins or aggregations. Several types of window are in use (such as tumbling or sliding windows [2]), but they all have in common that they place a bound on the maximum time interval between any two events that may be combined; any events that are further apart in time than this bound are treated as unrelated [3]. Windowed processing often occurs in business analytics on streams, or in *complex event processing* [50].

On the other hand, some applications need to combine events that may lie arbitrarily far apart in time, and thus windowing is not applicable [15]. For example, imagine a Twitter-like social network in which there are three main types of event: posting a tweet (a message), following a user, and unfollowing a user. When a user logs in, they want to see tweets posted by the users they are following. In a SQL database, this query might be expressed as follows:

```
SELECT tweets.*
FROM tweets
JOIN follows ON follows.followee_id = tweets.sender_id
WHERE follows.follower_id = logged_in_user_id
ORDER BY tweets.send_timestamp DESC
LIMIT 1000
```

In an event-based system, a tweet-posting event corresponds to inserting a row into the `tweets` table, a follow event corresponds to inserting a row into the `follows` table, and an unfollow event corresponds to deleting a row from the `follows` table. However, it may well be that the user followed another user years ago, and thus finding the current list of people the user is following requires going back arbitrarily far in the stream of follow/unfollow events. The join therefore needs to be non-windowed.

The SQL query above is expensive to execute for users who follow many people, since it needs to look up the recent tweets by all followed people and merge them chronologically. For this reason, Twitter constructs a cache that contains the result of the query above for each user (this is known as the *home timeline* [42]). Keeping this cache up-to-date requires a stream processor that combines

the tweet, follow, and unfollow events in order to incrementally maintain a materialized view of the query above.

When a user posts a tweet, the stream processor needs to find all the people who are following the sender and add the new tweet to their home timelines. When user *A* follows user *B*, the stream processor needs to find recent tweets by *B* and merge them into *A*'s home timeline. When user *A* unfollows user *B*, it needs to remove all tweets by *B* from *A*'s home timeline. Executing this stream join efficiently requires indexes that allow looking up the current set of followers for a given user, and the recent tweets for a given user.

## 2.3 Database replication and events

If we focus our attention on systems where the events are persistent notifications without windowing, we may notice a strong similarity to replication in database systems. The goal of replication is to have a copy of the same data on several different machines: that is, any two replicas converge to the same state, and all committed transactions are reflected in that state [14]. Viewed through the lens of event-based systems, we can treat every transaction that modifies the database as an update event, the execution of an update (i.e. reflecting the update in the database state) as the processing of that event, and the replication of data from one machine to another as the propagation of that event through the distributed system. Read-only transactions may or may not correspond to events, depending on the system.

In such a database system, the replication infrastructure ensures that every event corresponding to a committed transaction is eventually processed by every non-faulty replica. We can classify such replication systems into two broad categories: those that arrange the replication events into a log, and those that use some other replication mechanism (such as gossip protocols [47], anti-entropy [19], or clients independently updating several replicas [4]). The key properties of a log are that the events in it are totally ordered (i.e. all replicas observe the log entries in the same order), and it is append-only (i.e. if a replica observes log entry *A* immediately followed by log entry *B*, then there will never be a log entry *C* that appears between *A* and *B* in the total order).

This append-only log is constructed either using a consensus protocol such as Raft [55] or Multi-Paxos [46], or by designating one replica as the *leader* (also known as master or primary) and all other replicas as followers (aka slaves or secondary replicas) [23]. In fact, most consensus protocols are essentially leader-based replication combined with an automatic mechanism for electing a new leader if the current leader fails [29]. The leader decides on the order in which events should be appended to the log, and all other replicas process the events in the order decided by the leader.

In many databases, the exact structure and content of the events in the log has traditionally been an implementation detail of the database system that is not exposed to applications. For example, several database systems use the *write-ahead log* (WAL) for replication, which consists of records indicating how the database's on-disk data structures are changing; others use a separate replication log [28]. More recently, *change data capture* techniques have been developed to extract the data change events (rows inserted, updated, or deleted) from this log and make them available to applications via stream processing systems [1, 17].

## 2.4 State machine replication (SMR)

In WAL-based replication and change data capture, the primary data model used by the application is the database state that can be mutated by the application (e.g. by inserting, updating, or deleting rows in tables), and the data change events are generated automatically as a side-effect of these mutations. It is also possible to swap these roles, making the data change events the primary data model of the application, so that any state changes become a side-effect of processing those events [41]. This is the idea behind *state machine replication* (SMR) [59], and the closely related concept of *event sourcing* [20, 68]. My 2014 talk *Turning the database inside-out* [33] also helped popularise this idea.

In SMR and event sourcing, the application does not directly mutate the state of a database. Instead, the application defines a set of event types that may occur; whenever something happens, an event of the appropriate type is appended to an event log, and is thereafter immutable. (In SMR, an event is also known as a *command*.) All replicas subscribe to this event log, and each replica processes events in the order in which they appear in the log. The event processing function can use arbitrary logic to update the database state, as long as it is deterministic and it depends only on the current database state and the content of the event. Provided that each replica processes the same sequence of events in the same order, and each replica starts in the same initial state (e.g. an empty database), the deterministic event processing logic ensures that all replicas move through the same sequence of states and end up in the same final state [9, 59]. We can think of each replica as a state machine whose state is its copy of the database, and whose state transition function is the event processing function. Put another way, the database state is a materialised view onto the underlying event log [26].

An advantage of this approach is that well-designed events often capture the intent and meaning of operations better than events that are a mere side-effect of a state mutation [68]. For example, "student $x$ cancelled their enrollment in course $y$ for reason $z$" is a clear descriptive event, whereas "one row was deleted from the `enrollments` table, the `available_places` field of a course was incremented, and one row was added to the `cancellation_reasons` table" is much less clear and embeds a lot of irrelevant detail about the current database schema. An event log thus makes it easier for the people who work with a system to understand how it got into a particular state, which can help with audit and debugging [9].

Another advantage of an event log is that it can be replayed in order to rebuild the resulting database state. If the application developers wish to change the logic for processing an event, for example to change the resulting database schema or to fix a bug, they can set up a new replica, replay the existing event log using the new processing function, switch clients to reading from the new replica instead of the old one, and then decommission the old replica [34]. This sort of retroactive change in business logic is usually not possible in databases that rely on state mutation as their primary data model. Moreover, it is easy to maintain several different views onto the same underlying event log if needed. As long as the rate of updates is not too high, it is often feasible to retain the event log indefinitely and replay it occasionally as needed. In systems with a high event rate such replay may not be feasible.

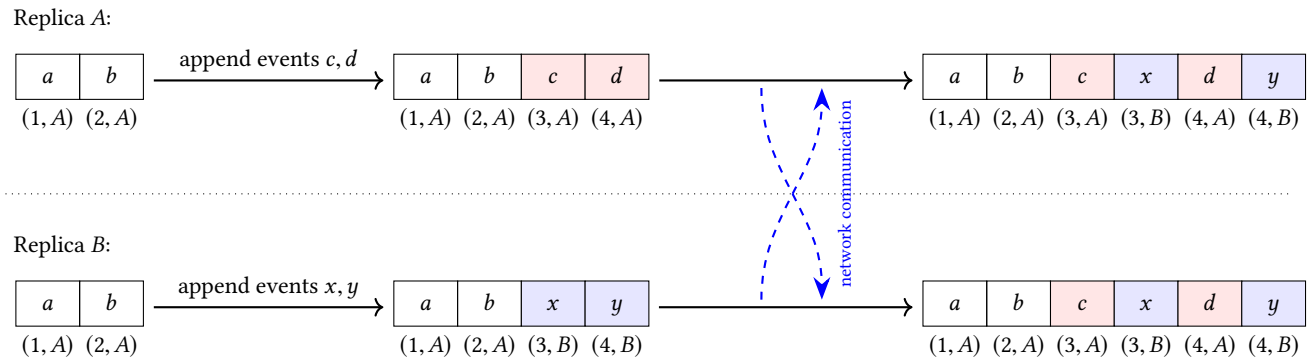


**Figure 2: Obtaining a total order on events using Lamport timestamps, which are** (*counter*, *replicaID*) **pairs. Two events with the same** *counter* **are ordered by** *replicaID*; **here we assume that** $A < B$.

Blockchains and distributed ledgers also use SMR, in which case the chain of blocks (and the transactions therein) constitutes the event log, the ledger (e.g. the balance of every account) is the resulting state, and smart contracts or the network's built-in transaction processing logic are the state transition function [66].

The downside of the event sourcing/SMR approach is that it is less familiar to most application developers than mutable-state databases, and the level of indirection between the event log and the resulting database state adds complexity in some types of applications that are more easily expressed in terms of state mutations. In applications with a high rate of events, storing and replaying the log may be expensive.

If permanent deletion of records is required (e.g. to delete personal data in compliance with the GDPR right to be forgotten [62]), an immutable event log requires extra care. Proposed solutions include periodically rewriting the log to remove any records that need to be deleted, or encrypting personal data with a per-user key that can be deleted if that user requests deletion of their data: a record that cannot be decrypted is widely regarded as being equivalent to a deleted record [63]. If the state of other systems is derived from an event log, personal data can also be removed from those downstream systems by first deleting personal data from the log and then replaying the events.

## 2.5 Partially ordered events

Both WAL-based replication and SMR depend crucially on the assumptions that all replicas process events in exactly the same order. Within a single datacenter, this is a reasonable assumption, since leader-based replication and consensus protocols work well in this setting. However, if replicas are distributed across multiple geographic locations, or if the network between replicas is unreliable, constructing a log becomes expensive, since appending an event to the log requires at waiting least one network round-trip to the leader and/or a quorum of replicas. In the extreme case, if we want a replica to be able to generate and process events even while it is completely disconnected from all other replicas (a system that is "available" and "partition-tolerant" in the sense of the CAP theorem [21]), the assumption of a totally ordered log becomes impossible to satisfy [13, 18].

In a system that allows disconnected operation, the strongest order we can guarantee is a causal order [5]. This is a *partial* order, in contrast to the total order of a log. In a causally ordered system, some events *happen before* other events [45], and those events are processed in the same order by all replicas. However, other events may be *concurrent*, which means that neither happened before the other; in this case, different replicas may process those events in a different order [10]. A partially ordered form of replication is also known as *optimistic replication* [58].

Since replicas are not guaranteed to process events in the same order, deterministic event processing is no longer sufficient to ensure that replicas end up in the same state. However, in some applications it is possible to ensure that whenever two events are concurrent, processing them is commutative (for example, adding numbers); in this case, the nondeterministic order of processing is no problem, and the replicas can nevertheless converge. Section 2.6 expands on this idea.

In a partially ordered system it is still possible to enforce a total order on events after the fact, as illustrated in Figure 2. We do this by attaching a logical timestamp to each event; Lamport timestamps [45] are a common choice. These timestamps consist of a counter, which is incremented for every event, together with the globally unique ID of the replica that generated the event. In Figure 2, two replicas initially have the same two events $a$ and $b$ with timestamps $(1, A)$ and $(2, A)$ respectively. During a period when the two replicas are disconnected from each other, replica $A$ generates two events $c$ and $d$ with timestamps $(3, A)$ and $(4, A)$ respectively, while concurrently replica $B$ generates events $x$ and $y$ with timestamps $(3, B)$ and $(4, B)$. After connectivity is restored, the replicas learn about each others' events, and merge them into a total order. This order is defined by first sorting events by the counter portion of their timestamps, and then breaking ties by sorting by replica ID (here we assume $A < B$).

Timestamp ordering produces a totally ordered sequence of events, but it is not a log, because new events are not always appended to the end. In Figure 2, when replica $B$ receives event $c$ from

$A$, it must insert $c$ ahead of its existing events $x$ and $y$ in its event sequence. Similarly, replica $A$ must insert $x$ between its existing events $c$ and $d$. In SMR, the total order of events is fixed as soon as an event is processed, but with timestamp ordering, the order may need to be revised as more events are received from other replicas.

It is nevertheless possible to use timestamp-ordered events in an SMR-like approach: that is, to use a deterministic function to apply events one by one to the current replica state in timestamp order. Assuming every replica eventually receives every event, all replicas will eventually have the same timestamp-ordered sequence of events, and thus all replicas will go through the same sequence of states and end up in the same state. However, when a replica processes events out of timestamp order (inserting an event somewhere in the middle of the timestamp-ordered sequence), it must be able to roll back the replica state to the state at the time corresponding to the insertion position, apply the new event, and then replay the events whose timestamps are greater than that of the new event [64]. This approach is known as *time warp* [31].

The cost of this rollback-and-replay process depends on the degree to which events are received out-of-order. If the replicas receive most events in ascending timestamp order, and they only occasionally need to reorder the most recent couple of events, the cost can be modest [43]. On the other hand, if replicas might generate large numbers of events while disconnected from each other, the cost of merging $n$ events into a linear sequence may become as great as $O(n^2)$.

Moreover, if processing an event may have external side-effects besides updating a replica state – for example, if it may trigger an email to be sent – then the time warp approach requires some way of undoing or compensating for those side-effects in the case where a previously processed event is affected by a late-arriving event with an earlier timestamp. It is not possible to un-send an email once it has been sent, but it is possible to send a follow-up email with a correction, if necessary. If the possibility of such corrections is unacceptable, optimistic replication cannot be used, and SMR or another strongly consistent approach must be used instead. In many business systems, corrections or apologies arise from the regular course of business anyway [27], so maybe occasional corrections due to out-of-order events are also acceptable in practice.

## 2.6 Conflict-free Replicated Data Types (CRDTs)

In the time warp approach, like in SMR and event sourcing, the events are the primary data model for the application, and the replica state is derived from the events. Similarly to Section 2.4, we can also choose to swap these roles, so that the mutable replica state is the application's primary data model, and the events are generated automatically as a side-effect of mutating this state. If we take the mutable-state approach in the context of a partially ordered system, we obtain a technique called *Conflict-free Replicated Data Types* or CRDTs [60].

CRDTs have been defined for a number of common abstract data types: sets, maps, lists, trees, graphs, and so on [61]. Applications may mutate these structures through the operations provided by the data type's interface: for example, a set or a list can be mutated by inserting or deleting elements; a map can be mutated by assigning a value to a key or by deleting a key-value pair. These mutations can take place even while a replica is disconnected from the rest of the system.

In operation-based CRDTs, the CRDT algorithm tracks any mutations to a data object and generates events (usually called *operations*) describing the changes. These events are partially ordered; causal ordering, like in Section 2.5, is a common choice [22]. When a replica receives an event generated by another replica, it invokes the CRDT algorithm to update its state. This algorithm is carefully designed such that applying concurrent events is commutative: that is, the final state is the same, regardless of the order in which a replica applies a set of concurrent events. By relying on commutativity, CRDTs ensure that replicas converge to a consistent state, without requiring that all replicas process events in the same order.

Many CRDT algorithms have behaviour that can equivalently be expressed in the time warp model [37]. However, CRDTs have the advantage that they usually do not require the rollback-and-replay process of time warp, so they can offer higher performance [57]. The state mutation model of CRDTs is a good fit for applications where the users are able to more or less directly manipulate the state in question: for example, in a text editor, users can insert or delete text anywhere in the document; and in graphics editing software, users can create, delete, move, or modify graphical objects anywhere within the picture. In such applications, the level of indirection provided by event sourcing is not needed, since the events would only express low-level state updates anyway (e.g. "insert character $A$ at position $x$", or "change the coordinates of object $A$ to $(x, y)$").

A disadvantage of CRDTs is that they support only the predefined operations offered by the data type's interface: for example, while list CRDTs allow elements to be inserted or deleted, most do not have good support for reordering list items [35]. In contrast, the time warp model allows any deterministic, pure function to be used for processing events, making it more flexible.

# 3 PRACTICAL EXAMPLES

In this section I offer a personal perspective by briefly discussing practical applications of the models from Section 2. I draw examples from two areas I have directly worked on: stream processing with Apache Kafka and related tools, and collaboration software that allows several people to work together on a shared document.

## 3.1 The Kafka Ecosystem

Apache Kafka is a publish/subscribe message broker based on event logs [41, 67]. It is widely used in enterprises where some teams want to publish streams of events relating to their business operations, and other teams want to subscribe to and process those event streams [40]. The Kafka ecosystem includes stream processing frameworks (Kafka Streams, Flink [12], Samza [38, 54]), a SQL-based stream query engine (ksqlDB), and tools for change data capture that obtain event streams from external systems such as databases (Kafka Connect, Debezium [1]). Kafka-based stream processors are used for both windowed and non-windowed processing.

Every event in Kafka belongs to a *topic*, and subscribers choose which topics to listen to. For scalability reasons, Kafka provides not just one log: every topic consists of a configurable number of separate logs (called *partitions* or *shards*). All events within a

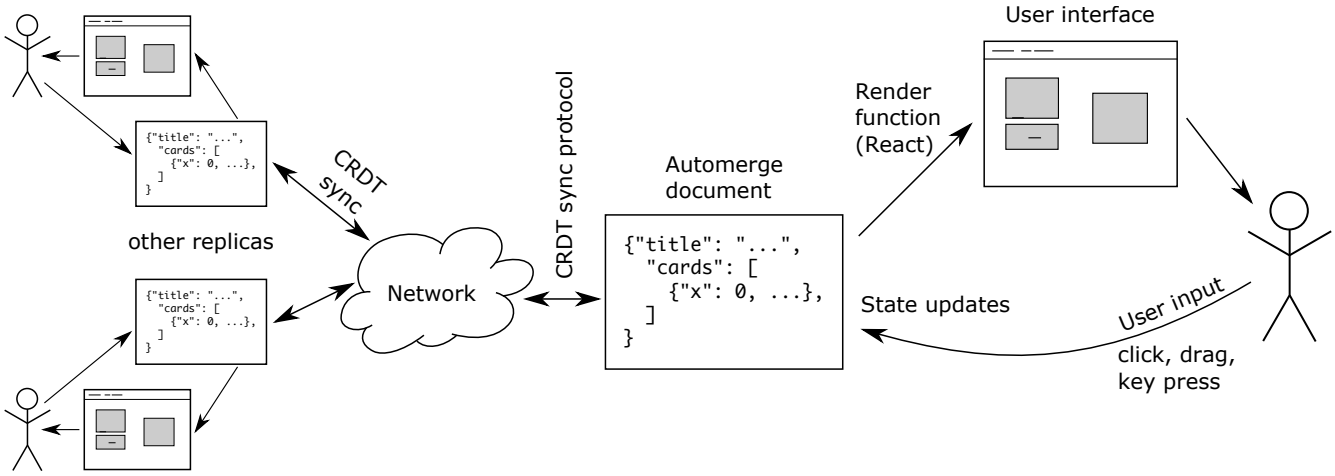


**Figure 3: A functional reactive programming model for local user input and collaboration with remote users. Figure from [25].**

given partition are totally ordered. This partitioning design has the advantage that different partitions can be handled by different nodes without requiring coordination; it has the disadvantage that there is no defined ordering of events across different partitions.

Therefore, when Kafka is used in an event sourcing/SMR style, we have to either use a single partition (limiting the scalability of the system), or break the state of the system into independent partitions to match the partitioning of the event logs. For example, if the state can be broken up by entity (so that each event relates to one entity, and the state of an entity is determined only by the events relating to that entity), then we can ensure that all events relating to the same entity are placed into the same partition. Each event log partition can then be processed independently to obtain the state for the entities within that partition.

The situation is more complicated if the system cannot be neatly broken down into separate partitions: for example, if one event may relate to multiple entities (e.g. it represents the transfer of money from one entity to another), then it is no longer sufficient to process the events in each partition independently. *Online event processing* (OLEP) [36] is an approach for handling such multi-partition interactions by breaking them down into multiple stream processing stages.

In some applications, an event needs to be checked against the current state of the system to determine whether it is allowed. For example, an event representing the booking of a seat in a theatre may be allowed only if that seat is not already taken. In a mutable-state database model, such a validation would be performed by a transaction that first reads the current state of the database (whether the seat is available), and then atomically writes its update only if the check succeeded. Kafka does not natively support such validations, but it is possible to implement them using a two-stage stream processing pipeline: an initial event represents only the *intention* to perform a certain action; then a stream processor joins that event with the current state to determine whether the action is permitted, and if so, emits a new event to a stream of validated events [36].

## 3.2 Local-first Collaboration Software

Over the past several years, my collaborators and I have been working on new foundations for collaboration software, that is, applications that allow several users to collaboratively modify a shared file. The file could be a text document, a drawing, a spreadsheet, a to-do list, or many other types of data. Software that runs on mobile devices needs to work offline: for example, you should be able to add an item to your to-do list even if your phone currently has no cellular data coverage. It is therefore clear that this type of application operates in the partially ordered, non-windowed, persistent part of the taxonomy.

More specifically, we are designing this software according to the *local-first* approach [39], in which every end-user device from which a user accesses their data is treated as a full-fledged replica using its local on-device storage. When a user modifies their data, they immediately update the replica on their local device, even if it is offline, and any updates are synced to the replicas on other devices the next time a network connection is available.

We use CRDTs to implement this approach, and thus the events describing data updates are generated by the CRDT algorithm as a side-effect of a mutation of a replica's state. As explained in Section 2.6, this model is a good fit for collaboration software where user input takes the form of state mutations. To this end we have developed a CRDT library called Automerge,[1] which provides a JSON data model. This data model is sufficient for implementing a variety of applications [25, 39].

CRDT-generated events also serve as notification that a document has changed, which enables real-time collaboration among several users editing the same document. We handle such events using a *functional reactive programming* model illustrated in Figure 3. The current state of the user's application is represented as an Automerge CRDT data structure. A rendering function takes this data structure and translates it into the corresponding user interface elements that should be displayed on screen [25]. In web

[1] https://github.com/automerge/automerge

applications, Facebook's React[2] is a popular library for performing this kind of rendering.

The rendered user interface has attached event handlers that are called when the user interacts with the respective user interface elements (these events are ephemeral). When such an event handler is called, it does not update the user interface directly, but rather it mutates the Automerge CRDT state to reflect the user input, and then the rendering function is called again to update the user interface. This functional reactive programming model makes the software easy to reason about, because data flows only one way: the user interface state is always derived from the Automerge state, not the other way around.

When a user thus mutates the Automerge state, the CRDT generates a change event, persists it locally, and uses a messaging middleware or sync protocol to send it to all other devices that have a replica of that document. When a replica receives such an event from a remote user, it uses the CRDT logic to update the local Automerge state, and then calls the rendering function in exactly the same way as it does for a change made by the local user. Using the same code paths and data flows for real-time collaboration as for local user input significantly simplifies the programming model. This approach is discussed in more detail in our paper on PushPin [25], one of our local-first software prototypes.

The change events generated by the CRDT represent the editing history of the document; we can think of these events as being similar to commits in a Git commit history. Persisting these events provides useful capabilities: we can reconstruct the state of the document at any past moment in time, and compute the differences between versions of the document to visualise the change history. We can also support collaboration workflows in which one user suggests changes to a document and another user can accept or reject the changes: in Git terms, one user can create a *branch* (a set of commits that are not yet part of the main document version), and another user can choose whether to *merge* it. Any number of branches can exist at the same time, and users can tentatively combine them to see what the document state would be if the branches were merged. Enabling such advanced workflows is a direct consequence of representing the application state as a partially ordered set of change events.

# 4 CONCLUSIONS AND FUTURE WORK

Events are used to great effect in a wide variety of systems, and this paper has provided a systematic overview of the field of event-based software. The taxonomy of Section 2 categorises event-based systems based on some simple criteria: whether the events are persistent, whether they act as notifications (triggering the execution of application code), whether there is a time bound on events that can be joined, whether the events are totally or partially ordered, and whether the application's primary model for expressing changes is by mutating state or by generating events. As we saw in this paper, there is no one true way: all of the categories in the taxonomy have important use cases. I have given examples of those use cases, and highlighted some of the key trade-offs that determine the pros and cons of different categories depending on the situation.

[2]https://reactjs.org/

I will close by outlining some open questions and challenges for future research.

## 4.1 Multi-version/Branching Workflows

Most replicated systems are based on the assumption that every replica should immediately process every event it receives, so that its state is as up-to-date as possible. However, sometimes this is not actually what we want: as suggested in Section 3.2, we may want to tentatively make some changes to a copy of a dataset, allow several users to inspect and discuss these changes, and then decide later whether to merge them into the primary copy of the dataset. In source code repositories we do this all the time with branches, merges, and pull requests; why can we not do the same with other forms of data?

Database transactions support a weak form of multi-version concurrency control by allowing an uncommitted transaction's writes to be either committed or rolled back by aborting the transaction [8]. However, most databases do not allow one user to share the state of an uncommitted transaction with another user, and most databases do not allow the user to find out what data has changed in an uncommitted transaction (the equivalent of `git diff`). Moreover, in most database systems, an uncommitted transaction may hold locks and thus prevent other transactions from making progress.

Partially ordered event-based systems are well placed to support such branching-and-merging workflows, since they already make data changes explicit in the form of events, and their support for concurrent updates allows several versions of a dataset to coexist side-by-side. CRDTs provide us with a mechanism for merging such diverged versions of a dataset. However, there are many open questions around how such systems should handle data versioning, including comparing and visualising differences between versions of a dataset, and which internal representations systems can use to efficiently operate on such multi-versioned data.

## 4.2 Data Model Changes

A challenge in many event-based systems is how to handle changes in the schema or data format [56]. The problem is especially pronounced in systems where we cannot guarantee that all replicas are running the same version of the software. For example, in applications that run on end-users' devices, it is up to the user to decide when to install a software update; thus, a user who is hesitant to install updates may be running a much older version of an application than a user who always installs the latest version. Nevertheless, those users should be able to interoperate as much as possible, which means that any data format changes must be both forward and backward compatible. The challenge becomes even greater if users are able to customise the software they are running, e.g. through end-user programming [30].

In systems that are based on immutable events, one promising approach is to use bidirectional functions (lenses) to convert between different versions of a data model, which allows different replicas to use different state representations while still being able to interoperate [49]. Open questions include how far this type of conversion can extend, how to support a range of different data models, how to make this programming model more accessible to application developers, and how to make it efficient.

### 4.3 Data Change Notifications Everywhere

There is growing commercial interest in systems that perform materialized view maintenance on top of event streams: startups in this area include Materialize[3] (based on differential dataflow [51]), RelationalAI,[4] and Event Store.[5] A common thread in these efforts is wanting to provide a higher-level programming model with strong semantics, whereas the Kafka ecosystem has generally prioritised low-level work on scalability and fault tolerance [11].

Despite this progress, the core issues I raised in *Turning the database inside-out* in 2014 [33] are still unsolved. Most application logic is still executed in a request-response model: when the application receives a request, it queries a database and returns the result as of that point in time, but the client cannot subscribe to be notified whenever the query result changes (other than by making repeated requests, i.e. polling, which is slow and inefficient). Incrementally maintained materialized views offer the potential to replace the request-response paradigm with a publish-subscribe paradigm, where a client can obtain not only the current state of some resource, but also subscribe to a stream of events that provide a low-latency notification whenever that state changes.

Our use of FRP for collaboration software (Section 3.2) is one instantiation of this idea in the context of application software running on the end user's device. However, we are yet to see a comparable change in the architecture of today's cloud-based service-oriented/microservices systems. Changing this status quo requires innovations both in the programming model (how do application developers express how the state of a system must change as a result of underlying events?) and in the execution (how does the system efficiently implement these operations?). Incrementally maintained materializations of SQL queries (such as the example in Section 2.2) are a good start, but more is needed to fully realise this vision, such as incorporating arbitrary business logic into the view materialization process.

## ACKNOWLEDGMENTS

Thank you to Jean Bacon, Jamie Brandon, Mariano Guerra, Ian Lewis, and Eiko Yoneki for feedback on a draft of this article. I am grateful for support from a Leverhulme Trust Early Career Fellowship, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, RelationalAI, SoftwareMill, Talent Formation Network, and Adam Wiggins.

## REFERENCES

[1] [n.d.]. Debezium. https://debezium.io/
[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076
[3] Tyler Akidau, Slava Chernyak, and Reuven Lax. 2018. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. O'Reilly Media.
[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. *J. ACM* 42, 1 (Jan. 1995), 124–142. https://doi.org/10.1145/200836.200869
[5] Hagit Attiya, Faith Ellen, and Adam Morrison. 2015. Limitations of Highly-Available Eventually-Consistent Data Stores. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 385–394. https://doi.org/10.1145/2767386.2767419
[6] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *Comput. Surveys* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666
[7] Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Technical Report MSR-TR-2014-41. Microsoft Research. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/
[8] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Transactions on Database Systems* 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998
[9] Dominic Betts, Julián Domínguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. 2012. *Exploring CQRS and Event Sourcing*. Microsoft. http://aka.ms/cqrs
[10] Kenneth P Birman, André Schiper, and Pat Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems* 9, 3 (Aug. 1991), 272–314. https://doi.org/10.1145/128738.128742
[11] Jamie Brandon. 2021. Internal consistency in streaming systems. https://scattered-thoughts.net/writing/internal-consistency-in-streaming-systems/
[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (Dec. 2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf
[13] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647
[14] Bernadette Charron-Bost, Fernando Pedone, and André Schiper (Eds.). 2010. *Replication: Theory and Practice*. Vol. 5959. Springer LNCS. https://doi.org/10.1007/978-3-642-11294-2
[15] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Foundations and Trends in Databases* 4, 4 (Dec. 2012), 295–405. https://doi.org/10.1561/1900000020
[16] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 411–422. https://doi.org/10.1145/2491956.2462161
[17] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, Boris Shkolnik, Sajid Topiwala, Alexander Pachev, Naveen Somasundaram, and Subbu Subramaniam. 2012. All Aboard the Databus! Linkedin's Scalable Consistent Change Data Capture Platform. In *3rd ACM Symposium on Cloud Computing (SoCC)*. https://doi.org/10.1145/2391229.2391247
[18] Susan B Davidson, Hector Garcia-Molina, and Dale Skeen. 1985. Consistency in Partitioned Networks. *Comput. Surveys* 17, 3 (1985), 341–370. https://doi.org/10.1145/5505.5508
[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *21st ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 205–220. https://doi.org/10.1145/1294261.1294281
[20] Martin Fowler. 2005. Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html
[21] Seth Gilbert and Nancy A Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601
[22] Victor B F Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). https://doi.org/10.1145/3133933
[23] Jim N Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. In *ACM SIGMOD International Conference on Management of Data*. ACM, 173–182. https://doi.org/10.1145/233269.233330
[24] Ashish Gupta and Inderpal Singh Mumick (Eds.). 1999. *Materialized Views: Techniques, Implementations, and Applications*. MIT Press.
[25] Peter van Hardenberg and Martin Kleppmann. 2020. PushPin: Towards production-quality peer-to-peer collaboration. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*. ACM. https://doi.org/10.1145/3380787.3393683
[26] Pat Helland. 2015. Immutability Changes Everything. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)*. http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf
[27] Pat Helland and Dave Campbell. 2009. Building on Quicksand. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*. https://database.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf

[3] https://materialize.com/
[4] https://www.relational.ai/ – disclosure: RelationalAI financially supports my work
[5] https://www.eventstore.com/

[28] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Foundations and Trends in Databases* 1, 2 (Nov. 2007), 141–259. https://doi.org/10.1561/1900000002

[29] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: have we reached consensus on distributed consensus?. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*. ACM. https://doi.org/10.1145/3380787.3393681

[30] Ink & Switch. 2019. End-user programming. https://www.inkandswitch.com/end-user-programming.html

[31] David R Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404 – 425. https://doi.org/10.1145/3916.3988

[32] Ralph Kimball and Margy Ross. 2013. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd ed.). John Wiley & Sons.

[33] Martin Kleppmann. 2015. Turning the database inside-out with Apache Samza. https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html

[34] Martin Kleppmann. 2017. *Designing Data-Intensive Applications*. O'Reilly Media.

[35] Martin Kleppmann. 2020. Moving Elements in List CRDTs. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*. ACM. https://doi.org/10.1145/3380787.3393677

[36] Martin Kleppmann, Alastair R Beresford, and Boerge Svingen. 2019. Online Event Processing. *Commun. ACM* 62, 5 (May 2019), 43–49. https://doi.org/10.1145/3312527

[37] Martin Kleppmann, Victor B F Gomes, Dominic P Mulligan, and Alastair R Beresford. 2018. OpSets: Sequential Specifications for Replicated Datatypes (Extended Version). https://arxiv.org/abs/1805.04263

[38] Martin Kleppmann and Jay Kreps. 2015. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Engineering Bulletin* 38, 4 (Dec. 2015), 4–14. http://sites.computer.org/debull/A15dec/p4.pdf

[39] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You own your data, in spite of the cloud. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, 154–178. https://doi.org/10.1145/3359591.3359737

[40] Jay Kreps. 2013. The Log: What every software engineer should know about real-time data's unifying abstraction. http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

[41] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: a Distributed Messaging System for Log Processing. In *6th International Workshop on Networking Meets Databases (NetDB)*.

[42] Raffi Krikorian. 2012. Timelines at Scale. In *QCon San Francisco*. https://www.infoq.com/presentations/Twitter-Timeline-Scalability/

[43] Roland Kuhn. 2021. Local-First Cooperation. https://www.infoq.com/articles/local-first-cooperation/

[44] Ajay Kulkarni and Ryan Booz. 2020. What the heck is time-series data (and why do I need a time-series database)? https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/

[45] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[46] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (Dec. 2001), 51–58.

[47] João Leitão, José Pereira, and Luís Rodrigues. 2007. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast *(37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks)*. IEEE, 419–429. https://doi.org/10.1109/dsn.2007.56

[48] Linux Programmer's Manual. [n.d.]. select(2) – Linux manual page. https://www.man7.org/linux/man-pages/man2/select.2.html

[49] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. 2021. Cambria: Schema Evolution in Distributed Systems with Edit Lenses. In *8th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*. ACM, Article 8. https://doi.org/10.1145/3447865.3457963

[50] David C Luckham. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley.

[51] Frank McSherry, Derek G Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[52] Jayadev Misra. 1986. Distributed Discrete-Event Simulation. *Comput. Surveys* 18, 1 (March 1986), 39–65. https://doi.org/10.1145/6462.6485

[53] Mozilla Developer Network. [n.d.]. Event reference. https://developer.mozilla.org/en-US/docs/Web/Events

[54] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[55] Diego Ongaro and John K Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (ATC)*. USENIX.

[56] Michiel Overeem, Marten Spoor, and Slinger Jansen. 2017. The dark side of event sourcing: Managing data conversion. In *24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 193–204. https://doi.org/10.1109/SANER.2017.7884621

[57] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. 2019. Putting Order in Strong Eventual Consistency. In *IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2019)*. Springer, 36–56. https://doi.org/10.1007/978-3-030-22496-7_3

[58] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *Comput. Surveys* 37, 1 (March 2005), 42–81. https://doi.org/10.1145/1057977.1057980

[59] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *Comput. Surveys* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[60] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[61] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

[62] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and Benchmarking the Impact of GDPR on Database Systems. *Proceedings of the VLDB Endowment* 13, 7 (March 2020), 1064–1077. https://doi.org/10.14778/3384345.3384354

[63] Ben Stopford. 2017. Handling GDPR with Apache Kafka: How does a log forget? https://www.confluent.io/blog/handling-gdpr-log-forget/

[64] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 172–182. https://doi.org/10.1145/224056.224070

[65] Ivan Valkov, Natalia Chechina, and Phil Trinder. 2018. Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka. In *33rd Annual ACM Symposium on Applied Computing (SAC)*. ACM, 218–225. https://doi.org/10.1145/3167132.3167144

[66] Marko Vukolić. 2015. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *IFIP WG 11.4 International Workshop on Open Problems in Network Security (iNetSec)*. Springer, 112–125. https://doi.org/10.1007/978-3-319-39028-4_9

[67] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1654–1655. https://doi.org/10.14778/2824032.2824063

[68] Alexey Zimarev. 2020. What is Event Sourcing? https://www.eventstore.com/blog/what-is-event-sourcing