Kronos: The Design and Implementation of an Event Ordering Service

Robert Escriva
Cornell University
escriva@cs.cornell.edu

Ayush Dubey
Cornell University
dubey@cs.cornell.edu

Bernard Wong
University of Waterloo
bernard@uwaterloo.ca

Emin Gün Sirer
Cornell University
egs@systems.cs.cornell.edu

Abstract

This paper proposes a new approach to determining the order of interdependent operations in a distributed system. The key idea behind our approach is to factor the task of tracking happens-before relationships out of components that comprise the system, and to centralize them in a separate event ordering service. This not only simplifies implementation of individual components by freeing them from having to propagate dependence information, but also enables dependence relationships to be maintained across multiple independent systems. A novel API enables the system to detect and take advantage of concurrency whenever possible by maintaining fine-grained information and binding events to a time order as late as possible. We demonstrate the benefits of this approach through several example applications, including a transactional key-value store, and an online graph store. Experiments show that our event ordering service scales well and has low overhead in practice.

1. Introduction

Time and event ordering are critical to the design of distributed systems. Because this ordering determines the sequence of actions observed by clients, it directly impacts the end-to-end correctness and consistency invariants a system may maintain. Further, constraints placed on the ordering of events can have significant impact on performance by enabling or limiting concurrency.

Because event ordering plays such a significant role, many techniques have been suggested to capture dependencies and ordering in distributed systems. The three most commonly used approaches are Lamport timestamps [23], vector clocks [17, 28], and consensus-based approaches [22, 31]. While these schemes differ in how they capture dependencies (whether they are expressed in a happens-before relationship, a time vector, or an assigned timestamp in a timeline), they share the same structure. Namely, they are instantiated separately within each independent distributed system and track dependencies solely within the purview of that system, often by monitoring communication at the boundaries of internal components. This leads to the following problems:

- False negatives: Because a given system only knows of relationships within its purview, it will miss any dependencies that are formed over external channels [11, 23].
- False positives: Because false negatives have significant consequences, distributed systems often err by conservatively assuming a causal relationship even when a true dependence might not exist. For instance, many vector clock implementations will establish a happens-before relationship between every message sent out and all messages received previously by the same process, even if those messages did not play a causal role.
- Early assignment: Time ordering systems often impose an order too early on concurrent events, thereby reducing the flexibility of the system. For instance, Lamport timestamps and vector clocks order events at the time when timestamps are assigned.
- Composition: Modern networked applications, including almost all high-performance web services, are increasingly built on top of multiple distributed subsystems, and would benefit from a notion of dependence that carries over and composes between independent subsystems.

In this paper, we propose a radically different approach to the management of time dependencies in distributed systems. The main tenets of our approach are threefold. First, we advocate factoring event ordering out of independent subsystems into a shared component that tracks timing dependencies between actions that traverse multiple subsystems. This refactoring creates a "lingua franca" of event ordering which, in turn, enables multiple independent subsys-

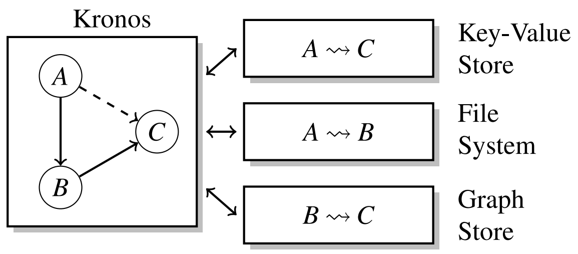
EuroSys 2014, April 13–16, 2014, Amsterdam, The Netherlands.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2704-6/14/04…$15.00.
http://dx.doi.org/10.1145/2592798.2592822

tems to keep track of event ordering relationships without having to agree on and pass event ordering information between other services. Second, we propose keeping track of dependencies at very fine granularity; specifically, we make the case for maintaining the full *event dependency graph*. This yields expressive systems that can distinguish and take advantage of concurrency where available. Finally, we advocate late *time-binding*, that is, picking an absolute order of events that is congruent with constraints as late as possible. Late assignment of time order provides extensive freedom to applications on how to schedule a set of concurrent events whose time order is under-constrained, a situation commonly encountered in practice.

Based on this approach to managing the partial order of events in a factored component, we designed and implemented a fault-tolerant event ordering service called Kronos. Kronos externalizes the task of tracking dependencies from distributed subsystems to capture a global view of dependencies between a set of distributed operations. This architecture enables multiple independent subsystems to share and maintain a unified directed acyclic graph that keeps track of "happens-before" relationships at fine granularity. This graph representation captures ordering relationships at much finer granularity than both Lamport timestamps and vector clocks. Finally and most importantly, Kronos enables applications to query the graph and determine if two events are concurrent, which in turn identifies those instances where the application can make its own decision on how to order these concurrent events.

We have built several applications and examples on top of Kronos. Our first application illustrates how Kronos can be used to improve the user experience in a social network. The second application is an online, strongly-consistent graph store that uses Kronos to order writes and graph traversals. The third application is a transactional key-value store that uses Kronos to serialize transactions in an off-the-shelf key-value store. Finally, we simulate the shop-floor control and fire-alarm examples described by Cheriton and Skeen [11], and demonstrate how Kronos overcomes the problems that they demonstrate.

This paper makes three contributions. First, we introduce a new abstraction for event ordering and propose a new service and minimal API for distributed systems. Second, we describe the implementation of multiple applications on top of Kronos, focusing mainly on the way in which each application exploits the event-ordering provided by Kronos. Finally, we evaluate the properties of a full implementation of Kronos and two of our applications. Experiments show that our graph store achieves throughput that is 59× higher than achieved in an off-the-shelf graph store and our transactional key-value store achieves 94% the throughput of "put-and-pray" (i.e. equivalent number of non-atomic, non-serializable operations) operations using MongoDB, and outperforms a lock-based transactional system by a factor of 3.6×.



A: Alice updates new photos which only her friends may access. The ACL is stored in the key-value store, and the photos themselves are stored on the file system.
B: Alice uploads a photo to the album and tags Bob in the photo. The photo is stored on the file system, and the graph store records that Bob is tagged by the photo.
C: Bob likes Alice's photographs. This action checks the ACL, and records the "like" in the graph store.

Figure 1. A social network built using Kronos, a key-value store, a graph store, and a file system. Each Kronos event corresponds to an action in the application. Kronos ensures that the transitive dependency $A \rightsquigarrow B \rightsquigarrow C$ will be enforced at the key-value store as $A \rightsquigarrow C$, even though the key-value store is unaware of event B.

2. Design

This section describes the design and implementation of Kronos. It defines the core abstractions on which the system is based, describes the API and its implementation, and outlines several deployment optimizations.

2.1 Kronos Abstractions

Kronos is a standalone shared service that tracks dependencies and provides time ordering for distributed applications. The central entity in Kronos is an *event*, an application-determined set of state changes that take place atomically, associated with a unique identifier. Events are akin to basic blocks in programming languages; they may be as fine-grained as the execution of a single instruction or receipt of a single message, or as coarse grained as system-wide state changes spanning multiple hosts. In practice, applications create events that correspond to any number of actions they take internally in response to externally-provided inputs. For example, a transactional key-value store could map each transaction to a Kronos event. Kronos leaves the precise semantics associated with events up to application and concerns itself with establishing a partial order between events.

Internally, Kronos builds and maintains an *event dependency graph*, a directed acyclic graph whose vertices correspond to events and whose edges correspond to *happens-*

before relationships[1]. An edge therefore succinctly represents all the ordering related constraints between events spanning multiple applications.

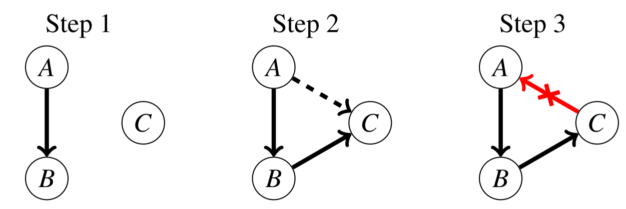
The central task of Kronos, then, is to enable applications to quickly order events along a timeline using the event dependency graph. Kronos provides interfaces by which applications may create new events, establish relationships between events, and query for pre-existing relationships. Each of these methods translates to an operation on the graph. When the application creates a new event, Kronos creates a new vertex in the event dependency graph. Similarly, when the application establishes a happens-before relationship, Kronos constructs a directed edge between the two vertices. To check for a pre-existing relationship between two events, Kronos looks for a directed path between them. The direction of the path directly encodes the happens-before relationship. The absence of a path between two events indicates that they are concurrent.

To permit applications using Kronos to make decisions that rely upon the timeline, Kronos upholds two invariants called the *coherency* and *monotonicity* invariants. The coherency invariant ensures that the events can be arranged into a possible timeline by ensuring that the graph is free of cycles. The existence of a path between two events in the graph implies that Kronos has made a series of commitments that force one event to necessarily succeed the other, in which case Kronos communicates this ordering to applications so that they can act accordingly. The coherency invariant prevents logical contradictions within the timeline represented by the event dependency graph.

Kronos's monotonicity invariant ensures that happens-before relationships, once established, are incontrovertible. Applications may safely commit to a particular time order once established by Kronos, as subsequent operations can only further constrain, but never violate, established dependencies. This enables Kronos clients to be able to issue side-effects and produce user-visible output based upon Kronos responses. In practice, the monotonicity invariant is easily upheld by omitting from the Kronos interface any means by which paths may be removed from the event dependency graph.

Using the Abstraction To see how the event dependency graph may be used by applications, consider a social network that allows users to upload, tag, and like photographs of each other. This application stores users' photos in a file system, records tags and "likes" in a graph store, and maintains ACLs in a key-value store. When Alice uploads her photos to the application, it stores her photos in the file system and updates the key-value store to store the ACLs. Similarly, when Alice uploads a photo in which she's tagged Bob, the application stores the photo on the file system and records in the graph store that Bob is tagged in the photo. Finally, Bob can like the photo, which records Bob's actions in the graph store only after checking that Bob is permitted to do so by the ACLs stored in the key-value store. Since the system is consists of three separate components, in the absence of order, it is possible for the ACLs setup by Alice in the first step to be improperly retrieved in the third step, potentially exposing her photos to an unintended audience.

[1] We use the terms *dependency* and *happens-before relationship* synonymously throughout this paper. The term *causal relationship* is related but more specific and not synonymous; a happens-before relationship can emerge without a causal relationship.



A: Alice updates new photos to an album.
B: Alice uploads a photo to the album and tags Bob.
C: Bob likes Alice's photographs.

Figure 2. As dependencies are added between events, edges are added to the event dependency graph. The application adds dependencies between A, B, and C insteps 1 and 2. Kronos prohibits the application from adding the dependency $C \rightsquigarrow A$ in step 3 because the application already established $A \rightsquigarrow B \rightsquigarrow C$.

This example social network application can use Kronos to ensure that this disastrous situation is reliably avoided. Each user-facing change to the social network is represented in Kronos as an event. Thus, when Alice initially uploads her photos or tags Bob, or when Bob likes Alice's photo, the application creates an event in Kronos to represent the user's interaction with the service. Individual components of the social network application will each process a different subset of these events, and each can impose an order on the subset they process. Kronos can then maintain an application-wide consistent timeline that spans all events, as shown in Figure 1. Figure 2 illustrates how the application may incrementally build the timeline within Kronos. After Alice's actions are recorded by Kronos, Kronos ensures that Bob's request will be correctly ordered after Alice's actions.

2.2 Kronos API

Applications interact with Kronos through a simple API (Table 1) designed around the event and dependency abstractions. This API enables applications to manipulate, refine, and query the event timeline represented by the event dependency graph. Kronos's API also permits atomic batching for efficiency, and conditional operations for additional application-level control.

Broadly speaking, the Kronos API is split into event-oriented calls and traversal-oriented calls. The former allow applications to create and manage events and control

`create_event()`	Create a new event and return a unique identifier e.
`acquire_ref(`e`)`	Increment the reference count on e.
`release_ref(`e`)`	Decrement the reference count on e.
`query_order([(`e_1`, `e_2`), ...])`	Check the relationship between event pairs e_i e_j in specified list, returning $e_i \rightsquigarrow e_j$, $e_j \rightsquigarrow e_i$, or *concurrent* for each.
`assign_order([(`e_1`, ` *order*`, `e_2`, ` *must/prefer*`), ...])`	Create the set of relationships $e_i \rightsquigarrow e_j$ in specified list, if possible.

Table 1. The Kronos API. Applications primarily use `query_order` and `assign_order` to establish dependencies.

the garbage collection mechanism, while the latter to help discover precedence relationships between events of interest to the application.

Event Creation Applications can add events to the Kronos timeline with the `create_event` call, which creates a new vertex and returns a globally unique identifier. This identifier may be passed to subsequent calls to query the graph or establish happens-before relationships with the event.

Dependency Creation The fundamental purpose of Kronos is to enable applications to establish a time order for events. It does this by permitting applications to incrementally refine the timeline with new pairwise dependencies between events. Kronos ensures that any refinement specified by the application is logically coherent, and maintains the abstraction's invariants; it does not permit the application to perform any refinement that violates them.

Dependencies may be created at any time during the lifespan of the event dependency graph. For instance, in our social network application, each time Alice and Bob interact with the service, Kronos assigns the interaction a unique event identifier, and orders this event identifier with respect to other events that the application has previously created. These additional ordering constraints enable Kronos to clarify the order of events in the timeline without withdrawing from any previously upheld guarantees. Consequently, events may be ordered by the application long after the interaction that precipitated the event's creation.

Applications may use the `assign_order` call to establish a dependency between a pair of events. On each call to `assign_order`, Kronos maintains the coherency invariant by implicitly performing a graph traversal on the event pair. Any operations that request an order that contradicts the result of the traversal are aborted by Kronos and the client is informed of the true order of operations.

To enable a wide array of application behaviors, the Kronos API enables applications to express how to deal with requests that contradict a previously established order. Kronos applications may specify two kinds ordering behavior: `must` and `prefer`. A `must` ordering conveys a hard constraint from the application that two events must be ordered in a specific way. Applications can use `must` constraints to store pre-existing relationships within Kronos, such as relationships that arise from the natural execution of the system. For instance, when an application deletes an object, the delete is necessarily ordered after the preceding create. If a `must` request cannot be satisfied, Kronos aborts the entire `assign_order` request without any side effects and returns an error to the application. In contrast, a `prefer` ordering preference indicates that the application would prefer that the events be ordered as specified in the request, but is willing to accept a reversal if previously established constraints make the request impossible. For example, applications typically prefer to respond to events in their arrival order, as long as doing so does not violate timing constraints. The application can use the `prefer` option to instruct Kronos to maintain the arrival order where possible and reorder them when necessary. This permissive ordering is invaluable to applications that can reorder events, as it improves performance while maintaining correctness.

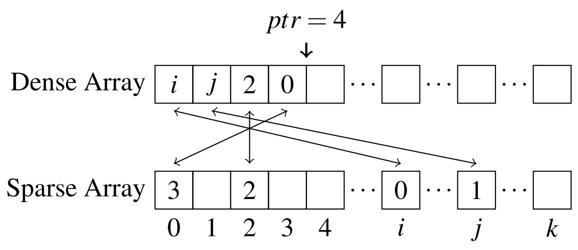


Figure 3. A diagram of the set data structure used to track visited vertices. A vertex i is in the set if and only if `sparse[i] < ptr && dense[sparse[i]] == i`. Adding an element to the set is done with `sparse[i] = ptr; dense[ptr++] = i;`. Clearing the set is done in constant time by setting `ptr = 0`.

For performance reasons, Kronos does not attempt to discover the minimal set of `prefer` reversals to render a suggested `assign_order` request coherent with respect to the existing event dependency graph. Instead, Kronos applies all `must` edges before `prefer` edges, thereby ensuring that a `prefer` edge is never established ahead of a `must` and thus will never cause an order assignment to abort when it could have been satisfied. Once all `must` edges are satisfied, the `prefer` edges are applied in the order specified by the application. An application can have some degree of control over which `prefer` edges are prioritized through the order in which they appear in the `assign_order` request. Not providing a guarantee of optimality avoids an NP-complete problem while providing a degree of control to the programmer.

Kronos provides a powerful primitive reminiscent of test-and-set atomic instructions that enables applications to specify a mix of `must` and `prefer` operations that execute as one atomic batch. Clients may specify constraints to check with the `must` flag set. Should all of the constraints be met, the batch will be applied atomically, but if any constraint is not met, the batch will be aborted without effect. A mixed batch of `must` and `prefer` operations resembles conditional test-and-set, where the `must` operations act as a conditional, and the entire batch will succeed or fail atomically. These atomicity guarantees enable safe yet concurrent use of the Kronos service without requiring an external lock service [9, 20].

Graph Traversal The `query_order` call enables applications to discover happens-before relationships captured by Kronos. This call takes a pair of events, e_1 and e_2, and returns whether $e_1 \rightsquigarrow e_2$[2], $e_2 \rightsquigarrow e_1$, or they are concurrent. To do this, Kronos performs a standard breadth-first search (BFS) to discover paths between e_1 and e_2.

The Kronos implementation pays careful attention to the cost of creating new events and happens-before relationships. BFS is potentially a costly operation, whose latency can be $O(|V|)$ where $|V|$ is the number of events managed by the system. Since a naive BFS would either require $\Omega(|V|)$ operations to initialize a visited bit field in every vertex or else dynamically allocate memory, and since $|V|$ can be large, Kronos instead uses a technique that makes use of uninitialized memory [7] to make the running time of BFS proportional to the number of vertices traversed. To avoid dynamic allocation, and linear initialization costs, Kronos preallocates all memory required for graph traversal at the time of vertex creation by creating two arrays, `dense` and `sparse`, of size $|V|$. The sparse array corresponds to vertices, and maintains indices into the dense array, which, in turn, indexes back into the sparse array. Initially, `ptr` is set to 0. When BFS visits a node i for the first time, Kronos sets `sparse[i]` to `ptr`, sets `dense[ptr]` to `i` and increments `ptr`. Checking to see if a node i has been visited can then be accomplished by checking if `sparse[i] < ptr` and `dense[sparse[i]] == i`. This optimization enables the core traversal algorithm in Kronos to require no memory allocation and only a single cache line worth of initialization.

2.3 Garbage Collection

The event dependency graph abstraction described so far will grow without bound as long as the distributed system is active. Kronos employs garbage collection to enable clients to safely shrink the event dependency graph. A critical invariant that Kronos maintains is that all events that could be submitted as arguments to any of the Kronos API calls remain within the graph, since they can be used as starting points in traversal operations. Kronos enables clients to dictate exactly which events can be used as arguments by exposing a reference counting API to clients.

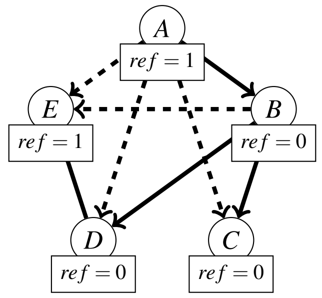


Figure 4. Kronos uses reference counting to determine when it is safe to collect events. Because events are collected after their dependencies are collected, B, C, and D remain in the graph despite their 0 reference count.

Kronos associates a reference count with each event and enables clients to acquire and release references through the `acquire_ref` and `release_ref` calls. Each time a client acquires a handle to an event, the reference count is incremented. Clients may at any time release the handle through a call to `release_ref`, which decrements the reference count. Once an event's reference count reaches zero, the event may be garbage collected. Overall, this reference counting mechanism ensures that all events that can be named by clients have non-zero reference counts and are pinned in memory.

To preserve transitive happens-before relationships, Kronos does not garbage collect events until their dependencies are garbage collected. For example, Figure 4 shows an event dependency graph in which multiple events remain in memory despite having zero references. Event A pins events B, C, and D into memory, delaying their garbage collection until after `release_ref` is called on A.

Garbage collection is strict: each `release_ref` call performs a topological sort on the graph, removing vertices with zero references and their outgoing edges. Thus, a single `release_ref` call garbage collects a subset of all vertices with zero references. In our example above, this means that once A's reference count goes to 0, A, B, C, and D will be collected immediately. The acyclic property of the graph ensures that the operation will complete in bounded time, and that all vertices may be eventually collected.

The Kronos API exposes no means of removing edges to applications because doing so would violate the monotonicity invariant. Edges are removed only after their source vertex is garbage collected. This ensures that edges persist until they may no longer affect any traversal.

[2] $e_1 \rightsquigarrow e_2$ may be read as e_1 happens before e_2

2.4 Fault Tolerance

Kronos achieves fault tolerance by replicating the event dependency graph with state machine replication. Applications may treat Kronos as a single, logically centralized service, and, due to state machine replication, the graph will be transparently maintained on several physical servers simultaneously. Because the Kronos API is entirely deterministic, each API call directly corresponds to a state transition in the replicated state machine.

Kronos replicates the event dependency graph using chain replication, which guarantees linearizability [40]. The exact number of Kronos replicas in the chain is a deployment specific decision and should reflect the maximum number of simultaneous faults the system is likely to experience. A system looking to tolerate f faults deploys $f+1$ replicas. In response to a replica failure, Kronos requests reconfiguration of the chain via a coordination service [9, 20]. Both the normal case and failure case performance behavior follow from the standard chain replication protocol.

The functionality provided by chain replication is not fundamental to Kronos's design and could easily be provided by other strongly consistent replication protocols. We use chain replication because the linear nature of the chain allows transactions to be pipelined at line rate without the fan-out/fan-in exhibited by Paxos-based techniques.

2.5 Scaling and Caching

The replicas necessary for fault tolerance provide a natural way to scale the system. Kronos can perform traversals on potentially stale replicas for improved parallelism. Only traversals which indicate that events are concurrent must execute on an up-to-date copy of the graph. The monotonicity invariant upheld by Kronos guarantees that any ordered answer returned by a stale replica is indistinguishable from the answer that would be returned had the query executed on the latest version of the graph.

Similarly, the monotonicity invariant permits widespread caching of traversal results without sacrificing correctness. Kronos and applications are free to cache the results of traversals where doing so can improve performance. For example, Kronos can maintain an internal cache of traversal results for high-degree vertices in order to improve traversal efficiency. Applications can freely pass around traversal results related to events within the messages used to commit the events.

3. Applications

In this section, we examine illustrative distributed applications to describe exactly how these systems use Kronos in practice. To simplify exposition, we present these applications in their most simple form, omitting implementation details about caching and batching in favor of straightforward explanations of how they interact with Kronos.

```
def post_message(user, message):
    e = kronos.create_event()
    for friend in friends_of(user):
        enqueue_in_timeline_for_user(timeline=friend,
                                     source=user,
                                     message=message,
                                     event=e)

def reply_to_message(user, message, in_reply_to):
    e = kronos.create_event()
    kronos.assign_order([(in_reply_to, ’->’, e, ’must’)])
    for friend in friends_of(user):
        enqueue_in_timeline_for_user(timeline=friend,
                                     source=user,
                                     message=message,
                                     event=e)

def render_timeline(user):
    # messages is a list of (id, message) pairs
    messages = get_messages_enqueued_for(timeline=user)
    # message_pairs is every pair of message ids selected
    # from the messages
    message_pairs = all_pairs([m.id for m in messages])
    orderings = kronos.query_order(message_pairs)
    # This will perform a topological sort of the messages
    # to ensure that sorted_messages abides by the partial
    # orders specified within orderings.  The remaining
    # messages will be unaffected by the sort, enabling
    # them to be displayed in their arrival order
    sorted_messages = topological_sort(messages, orderings)
    return sorted_messages
```

Figure 5. Pseudocode for maintaining social network timelines with Kronos. Users may post messages, which appear on timelines in the order in which the system processes them. When users use the social network's reply mechanism, the network uses Kronos to order the messages. Users' timelines are rendered with respect to the order recorded within Kronos, ensuring that conversations flow naturally.

3.1 Social Network

Social networks are often built around the notion of providing users with a timeline of activity drawn from their social circles. A user's timeline captures both public posts and personal interactions between users, displaying social activity along the timeline. While much of the activity in a social network is generated independently, there are certain classes of interaction where the user expects ordering to be preserved. For instance, communication between users should be preserved within the timeline—the timeline should never show a reply earlier in the timeline than the message to which it is replying.

Kronos provides a straightforward way to ensure that users' timelines reflect these communication patterns without enforcing a total order on all timeline activity. The social network may assign to each timeline post a Kronos event identifier, and then record communication patterns in Kronos with `assign_order`. When displaying user's timelines, the application can issue a corresponding `query_order` call to detect the partial order between events. Figure 5 shows pseudocode for this social network application.

3.2 Graph Store

Graph structured data is ubiquitous and analysis of these large graphs has prompted the development of specialized storage systems that directly store and maintain these graphs [8, 18, 27, 33, 37]. We have used Kronos to build a horizontally scalable data store for graph-structured data called KronoGraph. KronoGraph is built around a sharded architecture where the graph data is partitioned across servers. The KronoGraph API enables applications to incrementally build and maintain graph-structured data and perform isolated queries on the graph.

KronoGraph permits updates and queries to the graph that span multiple hosts; consequently, KronoGraph needs to apply operations in the same order across multiple hosts. In the absence of ordering, graph queries that are concurrent with updates could be applied in different orders at different hosts simply because the underlying messages used to transmit the operations arrive in a different order on each host. For example, imagine a graph consisting of edge $A-B$, where the application removes $A-B$ and adds $B-C$ as one update. An incorrect implementation could indicate that C is reachable from A, when, in fact, there was no instance in time when that was true.

The intuition behind KronoGraph is that shard servers process updates and queries in their natural arrival order, except in cases where Kronos indicates that the natural arrival order would not form a coherent timeline. To do this, KronoGraph assigns to each update or query a unique Kronos event identifier as it enters the system. Upon receipt of a new update or query operation, a shard server determines which vertices and edges are relevant to the operation, and gathers the event identifiers for all previous operations that affected these vertices and edges. The shard server then constructs a batch `assign_order` call to Kronos that `prefers` that each of these previously-processed events be ordered prior to the current operation.

Given the information available, the preferred order specified within an `assign_order` call is the most efficient ordering for the events. Should this order be satisfiable, the shard server may perform the operation immediately, without reordering it with respect to previously applied operations. Sometimes, the preferred order cannot be satisfied. For instance, if a pair of events arrive on two different shard servers in a different order, the first shard server's `assign_order` call will fix the order between these events. The second shard server's `assign_order` call must necessarily indicate a reversal to match the order returned in the first call. KronoGraph shard servers can tolerate a reversed order that does not match their preferred ordering by reordering operations on the graph.

For updates, shard servers maintain version information for each vertex and edge in the graph to order the updates. Vertices and edges contain a list of modifications and their associated event identifiers, sorted by the relative order of events. When Kronos upholds the ordering specified in the `assign_order` call, the shard server simply appends the update to the list. Should Kronos indicate a reversal, the shard server inserts the update into its sorted position within the list. The coherency invariant prevents cycles in the order, ensuring that it is always possible to insert into the list and maintain its sorted order.

For queries, shard servers decide on their execution time using the information returned from the Kronos assign order call. If the assign order call succeeds with no reversals, the KronoGraph shard server should execute the query on the graph that contains all previous updates. When Kronos indicates a reversal within the timeline, the shard server can construct an older version of the graph that omits all updates that happen after the query. Updates that are ordered strictly later than the query can easily be masked because of the timeline information maintained alongside the graph.

Kronos ensures that the shard servers execute queries in matching order even as the queries traverse multiple shard servers. Every `assign_order` call orders a query with respect to some subset of updates. Kronos ensures that all shard servers order a given query the same way with respect to a given update; subsequent iterations of a query refine its place within the timeline by ordering it with respect to additional updates. Localized queries that traverse a small portion of the graph are ordered only with respect to updates on the same portion, and will likely remain concurrent with respect to updates occurring elsewhere in the graph.

While a straightforward implementation of KronoGraph would query Kronos once per vertex or edge during a query, these costs may be avoided with judicious use of batching and caching. Upon receipt of a query operation, the KronoGraph shard server optimistically selects the events for vertices and edges in the graph could be traversed by the query operation, and requests that Kronos order the query consistently with respect to these optimistically chosen events. This permits KronoGraph to reduce the total number of calls to Kronos, and enables queries to traverse larger portions of the graph between calls.

Internally, KronoGraph relies upon caching to avoid unnecessary calls and to limit the size of each batched call. Each KronoGraph server independently maintains an LRU cache of the pairwise order between events. Because of the monotonicity invariant, KronoGraph servers may actively pre-fill this cache with transitive relationships. For example, if KronoGraph queries Kronos and sees that $u \rightsquigarrow v$, and the cache already contains $v \rightsquigarrow w$, the KronoGraph server can infer that $u \rightsquigarrow w$ without another call to Kronos.

3.3 Transactional Key-Value Store

Key-value stores have recently emerged as widely-used components in distributed services, mainly due to the high performance and scalability they offer. Existing key-value stores, however, achieve high performance by limiting their API; specifically, they restrict their clients to operate on a single object at a time. We have used Kronos to build a transactional key-value store that provides ACID transactions, where each transaction may update multiple objects atomically and with full serializability.

Transactional key-value operations are inherently difficult because transactions may span multiple hosts. Without coordination, concurrently executing transactions would be processed in a different order on different hosts, violating serializability. One approach to adding this coordination would be to assign a total order across all transactions, where the total order ensures that transactions execute in the same order across all hosts. While such an approach would safely ensure serializability, it would do so at the expense of concurrency. Transactions which operate on disjoint sets of keys are able to execute concurrently, but the system would expend resources enforcing a total order across these keys.

The key insight in our prototype key-value store is to create a new Kronos event for each transaction, and to order transactions that read or write the same keys using Kronos. This enforces a partial order across all transactions using the event dependency graph, and ensures that transactions are serializable, without actually serializing them. Servers incrementally build the dependency graph by establishing an order between transactions within their purview. Upon receipt of a transaction, a server examines the keys within its partition, and issues an `assign_order` call specifying that the transaction must be ordered after the last transaction which read or wrote each key. Should the `assign_order` call fail, the transaction will abort without effect.

Globally, the event dependency graph captures and enforces all dependencies between transactions. The system does not enforce any order between transactions not already ordered by the event dependency graph, as these transactions' individual operations may be applied in any order without violating serializability. Put another way, any topological sort of the event dependency graph will yield a schedule of transactions that is equivalent to the actual execution that produced the event dependency graph. This permits maximum flexibility between transactions, without requiring that they be applied in a total order.

3.4 System Integration

Cheriton and Skeen, in their paper on the limitations of causally and totally ordered communication support (CATOCS) [11], provide several example applications which critically rely upon time and event ordering.

One example from CATOCS is a manufacturing environment where machines are directed to "start" and "stop" processing orders by multiple control units. These control units communicate via a common database that does not preserve causality across requests. Consequently, the "start" and "stop" messages issued by control units may arrive in an unconstrained order, allowing the machines to "start" processing when they should "stop", or vice-versa. Kronos provides a solution to this problem, where each "start" or "stop" message maps to a Kronos event. Control units explicitly preserve order in Kronos with `assign_order`, and clients can verify the correct order of messages with `query_order`.

Another CATOCS example is a fire alarm system wherein the order in which "fire" and "fire out" messages are processed is critical. The key problem is that a delayed "fire out" message could lead an extinguisher to believe that multiple fires were extinguished, leaving fires to burn indefinitely. Again, Kronos provides a natural solution wherein each "fire" and "fire out" message is recorded as a Kronos event. The system records in Kronos a happens-before relationship between each pair of "fire" and "fire out" events. The resulting event dependency graph will consist of isolated pairs of vertices connected by single edges. It enables all entities to determine which fires are still burning no matter the order in which messages are delivered.

It is common for manufacturing environments to enhance fire alarm systems with a kill-switch that safely shuts down machinery during emergencies. One approach to this would be to modify all components involved, and tightly couple them together. Another approach, facilitated by Kronos, is to introduce a fail-safe component that couples the control units with the fire alarm via the event dependency graph. The fail safe responds to "fire" messages by issuing "stop" requests, and using Kronos to order the "stop" message after the "fire" message. On receipt of the corresponding "fire out" message, the fail safe orders the "stop" before the "fire out", and orders the subsequent "start" message after the "fire out". Thus the fail-safe automatically stops and restarts processing machines in response to fires without changes to either the fire alarm or manufacturing environment.

4. Evaluation

We have fully implemented Kronos to provide the functionality detailed in Section 2, and have built multiple applications on top of it. In the first half of our evaluation, we examine the performance of our sample applications to demonstrate that it is feasible to build real-world applications using Kronos. In the second half of our evaluation, we use microbenchmarks to investigate important aspects of Kronos's design, paying careful attention to performance, scalability, and resource usage. We finish our evaluation with a brief demonstration of Kronos's fault-tolerance.

Our experimental setup consists of fourteen well-provisioned servers. Each server is equipped with two Intel Xeon 2.5 GHz E5420 quad-core processors and 16 GB of RAM. All servers are running 64-bit Debian 7 and are connected via gigabit Ethernet.

4.1 Applications

In this section we attempt to answer the question, "Is it practical to build applications on Kronos?" The performance of the resulting applications is important in evaluating whether Kronos is a suitable choice for each application, but should not be the only deciding factor. For small- to medium-sized

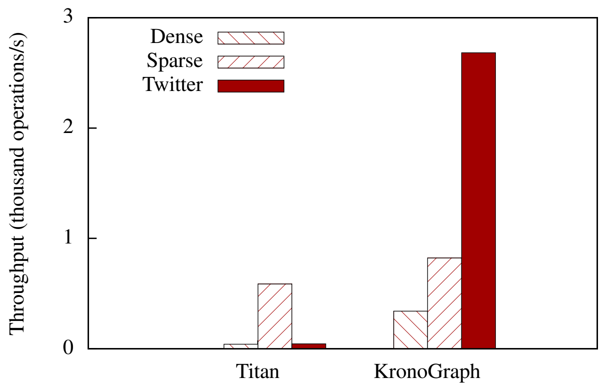


Figure 6. Titan and KronoGraph performing friend recommendation calculations on a mutating graph in a 95% read/5% write workload. KronoGraph outperforms Titan by a factor of 59× for the Twitter social network. Kronos enables KronoGraph to perform queries that are fully isolated from the ongoing write operations, while Titan uses locking to make the same guarantee.

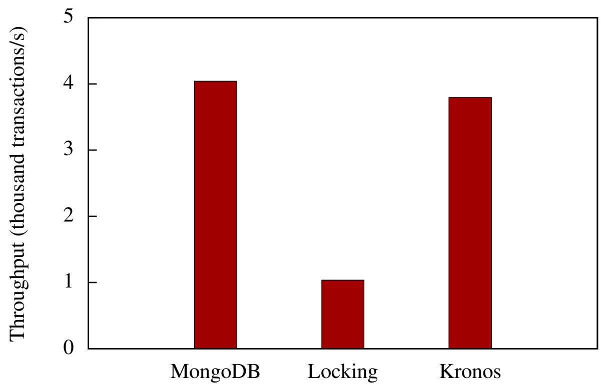


Figure 7. Transactional chains are fully three times faster than locking-based implementations and achieve 94% of the throughput of a "put-and-pray" approach built on MongoDB. This graph shows a sample banking application performing transfers between accounts.

applications, the composition property provided by Kronos may be worth any overhead that affects performance.

For all of the application-specific benchmarks, we deployed a single instance of Kronos on its own server, to ensure that the cost of interacting with Kronos includes all relevant communication cost. The remaining servers in the cluster deploy the application itself. We evaluate fault tolerance overheads separately.

4.1.1 Graph Store

We first evaluate KronoGraph, our graph store built on top of Kronos. For an accurate comparison, we compare KronoGraph to Titan [37], another online graph store that permits users to query and incrementally alter the graph. Titan employs lock-based techniques to provide isolation guarantees comparable to KronoGraph. We omit comparisons to other notable graph systems [18, 27, 33] because they do not support online operation and are thus incomparable to Titan and KronoGraph.

Intuitively, queries and updates in KronoGraph should be strictly less expensive than in Titan because Titan's lock-based techniques inhibit concurrency, while KronoGraph exploits late time binding in Kronos to allow non-blocking behavior. Titan's locks decide the order of graph operations; the first process to grab a lock is implicitly ordered earlier than later lock-holders. KronoGraph explicitly manages the order of graph operations, and consequently can perform multiple operations simultaneously, and resolve their order in one call to Kronos.

To characterize the difference in behavior between Titan and KronoGraph, we implemented a friend recommendation application in a social network on top of both systems. Our application represents the social network as a graph where individuals are represented by vertices, and edges symbolize friendship. The application makes friend recommendations on the basis of maximizing mutual friendship. For a given input, the algorithm will return the user with the most number of friends in common. This mimics the behavior of many social networks, where the structure of the graph is used to make further recommendations to users [39].

We ran both of our friend recommendation algorithms on a subset of the Twitter social network [29]. This graph consists of 81,306 individuals with 1,768,149 friendship links. For both implementations, we ran 32 parallel clients with a workload generator that produced a mixed workload that performed a friend recommendation 95% of the time, and introduced new individuals or friendships to the graph the remaining 5% of the time. We can see in Figure 6 that the KronoGraph friend recommendation algorithm outperforms the Titan recommendation algorithm by a factor of 59×.

The performance gap between KronoGraph and Titan is largely related to the density of the graph. We generated two random graphs of varying density to use as inputs to our friend recommendation algorithm to confirm this hypothesis. The denser of the two graphs had an average degree of 100, while the sparser graph had an average degree of 10. We can see in Figure 6 that KronoGraph outperforms Titan by a factor of 8.3× and 1.4× respectively.

The variation in KronoGraph's performance across the three different graphs gives us deeper insight into the performance characteristics of the system beyond raw differences in throughput. Because the number of calls made to Kronos is related to the number of operations submitted by Kronos clients, we would expect that a bottleneck around Kronos would limit the throughput and restrict it from varying with the density of the graph. Batching and caching in Krono-

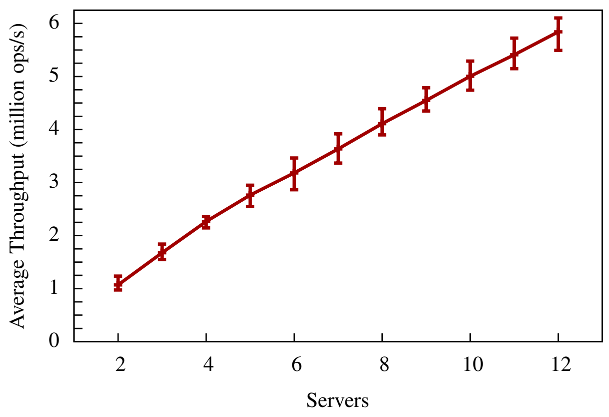


Figure 8. Kronos is a scalable system. This graphs shows the aggregate throughput achieved by a fixed number of clients calling `query_order` on a graph where each edge participates in, on average, 5 happens-before relationships. Aggregate throughput is measured across a 30 s window and the tight error bars show the 5th and 95th percentiles for throughput observed throughout the window.

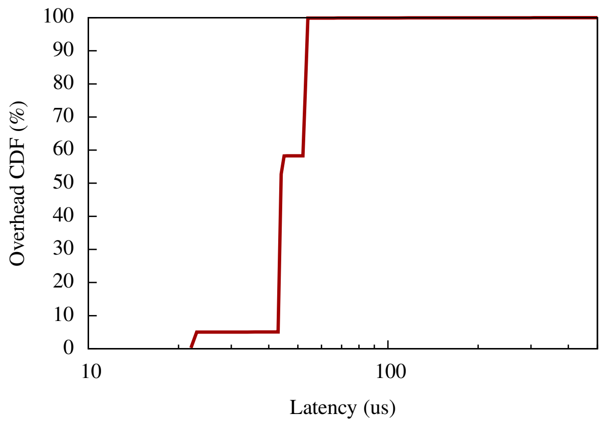


Figure 9. Kronos quickly creates events. Kronos can create a new event in less than 57 µs 99% of the time.

Graph are effective, and prevent Kronos from becoming a bottleneck. In our Twitter experiment, approximately 13.4% of operations required a Kronos traversal.

4.1.2 Transactions

We want to evaluate a transactional key-value store that provides ACID semantics built using Kronos. To evaluate this application, we developed a prototypical banking application, similar to the one used in nearly every database textbook to illustrate transactions [13]. Our application processes users' debits and credits, and transfers money between bank accounts.

For comparison, we implemented the banking application on top of two other data stores for a total of three comparable bank applications. Our first bank application is built on the popular MongoDB NoSQL data store, where account transfer consists of two independent write operations to MongoDB. Because MongoDB does not offer transactional semantics—it is only eventually consistent—this application is likely to encounter undesirable behavior, such as incomplete money transfers and lost deposits. Our second bank application uses locking techniques, such as those used in Percolator [32], to synchronize access to individual accounts and provide fully-serializable semantics. Finally, we implemented transactional semantics using Kronos as described in Section 3.3. We used HyperDex [15] as the underlying key-value store in the second and third implementations.

Figure 7 shows the throughput each implementation was able to achieve when accessed by 64 concurrent clients. We see that the Kronos-based variant outperforms the lock-based variant by a factor of 3.6×. The Kronos-based transactional key-value store achieves 94% the throughput of the non-transactional, eventually-consistent MongoDB deployment. This comparison provides an advantage to MongoDB, as MongoDB provides relatively weak guarantees, while the Kronos-based transactional key-value store provides ACID transactions.

4.2 Micro-Benchmarks

In order to further explore the design decisions made in Kronos, we present several microbenchmarks each of which explores a different aspect of Kronos's design. Kronos provides tools for explicit event creation and ordering. We first examine the performance and scalability of order-related API calls as they are by far the most costly aspect of Kronos. We then investigate the time and resource costs associated with event creation, dependency creation, and event garbage collection. Except where noted, these results do not include the overhead of state machine replication, as it is largely separable from Kronos's implementation.

Scalability Our first experiment measures how additional servers enable Kronos to handle additional `query_order` requests. In this experiment, we pre-loaded Kronos with a random graph over 10,000 vertices with 50,000 edges, and varied the number of replicas used for satisfying `query_order` requests. Each client performs random `query_order` requests on the graph, checking for preexisting relationships. We deployed 64 clients that concurrently query the replicas of the graph using the `query_order` API. Figure 8 shows that Kronos scales well; each additional server enables the system to respond to proportionally more `query_server` requests.

Dependency Creation When assigning order between two events, the dominating cost is graph traversal. Once Kronos traverses the graph, the cost of actually recording the dependency is nearly trivial. We measured the time taken to create dependencies that require no traversal, and found that, across 1 million events, dependency creation completes

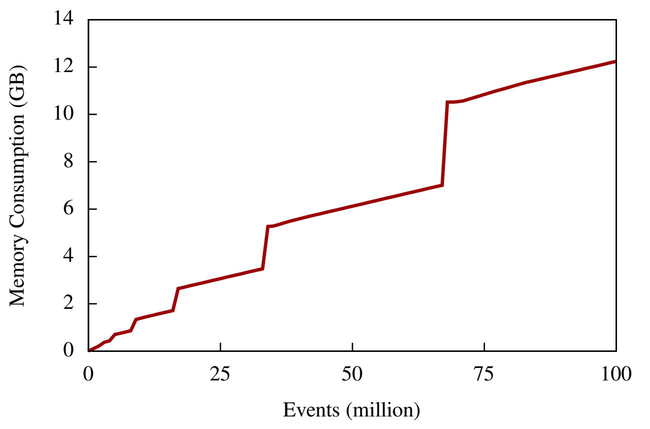


Figure 10. Kronos's memory consumption scales linearly as events are added. In this graph, a single client adds a total of 100 million events sequentially, maintaining a reference to each one. The memory usage is the maximum resident set size of the process. Discontinuities in the graph are directly related to array-doubling in the implementation.

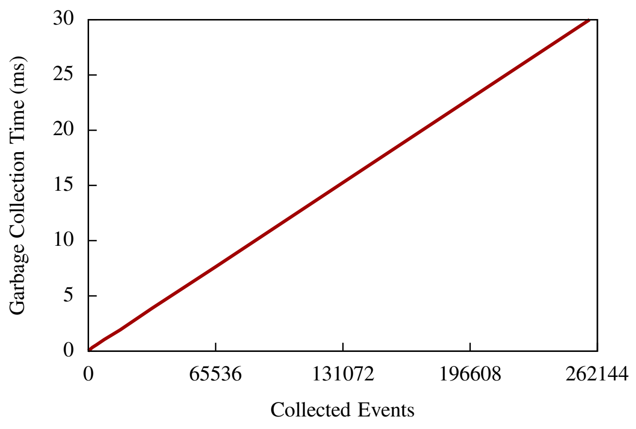


Figure 11. Garbage collection is efficient even for the absolute worst case event dependency graph. In this experiment, fixed length paths are created in the dependency graph such that releasing a reference to the first event in the path garbage collects the entire path.

in 49 µs 14.7% of the time, and 50 µs the remaining 85.3% of the time. These numbers also reflect the additional cost of creating events above and beyond the cost of a `query_order` operation.

Event Creation The next experiment examines the overhead of event creation and shows that Kronos creates events in constant time. Figure 9 shows a CDF of event creation latency for 100 million events. Kronos completes a majority of `event_create` operations in 44 µs and 99% of operations in less than 57 µs. These measurements include all allocation necessary to create the new event. For this experiment, the client uses the Kronos Python bindings to create and acquire references to the events. The event creation latency was measured by timing 10,000 sequential calls to `event_create`, with no parallelism in the calls. To avoid confounding effects relating to network latency and to isolate the performance of the server itself, the client and server are co-located on the same machine.

Memory Consumption Because Kronos allocates all memory used by a vertex at event creation time, it is important to quantify this cost. Figure 10 shows that 100 million events occupy 12 GB of RAM and fit within main memory of a single server. The reported memory consumption includes all memory necessary to track unique event identifiers, perform traversal using the BFS algorithm and maintain one reference per event. Applications will only allocate more memory when adding edges, where each edge occupies 8 B of space. The implementation dynamically allocates memory to grow and shrink while remaining proportional to the number of events and dependencies in the system.

Garbage Collection Kronos's strict garbage collection scheme introduces minimal overhead. Because Kronos uses

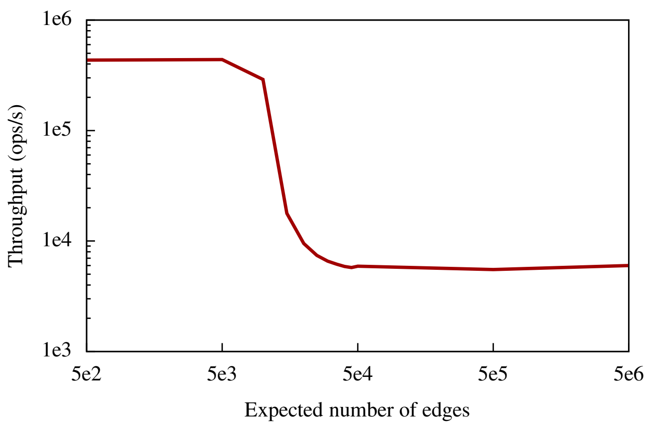


Figure 12. Kronos is fast for sparse graphs. This graph shows the aggregate throughput of `query_order` operations on Erdös-Rényi graphs with 10,000 vertices and varying numbers of edges.

strict garbage collection, the cost of releasing the final reference to a single event is proportional to the total number of events collected. Figure 11 shows worst case garbage collection behavior of Kronos. For this experiment we control the number of events to be garbage collected by a single `release_ref` call. As expected, the time taken to perform strict garbage collection grows linearly in the number of events to be collected.

Impact of Graph Structure The cost of graph traversal is dependent upon the structure of the graph itself. Intuitively, sparse graphs are quicker to traverse as the likelihood of touching many vertices becomes lower as the graph becomes sparser. On the other hand, processing dense graphs will necessarily involve a longer traversal as more vertices

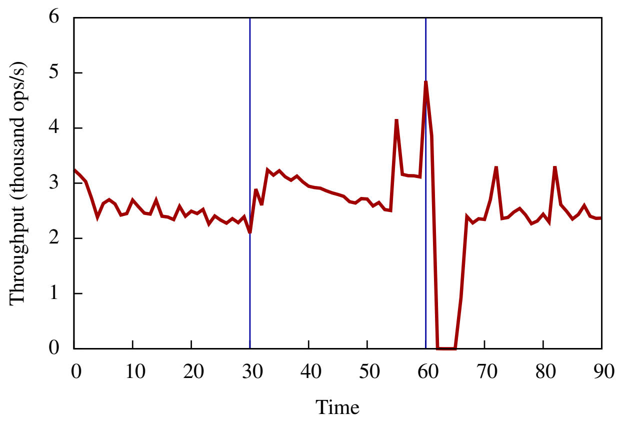


Figure 13. Kronos automatically recovers from failures. This graph shows the effects of server failure in a 3-server Kronos deployment. At the 30 s mark, the middle server in the chain is killed. Another server is brought into the cluster to take its place at the 60 s mark.

belong to large connected components. To test the behavior of Kronos on a variety of sparse and dense graphs, we generated random event dependency graphs conforming to the Erdös-Rényi Model [14]. Under this model, any two points in the graph are connected with probability p. Accordingly, these graphs have between 500 (p = 0.00001) and 5,000,000 ($p = 0.1$) vertices, with larger values of p corresponding to denser graphs. Figure 12 demonstrates the impact of graph density on the throughput of `query_order` operations on a single instance of Kronos. For relatively sparse graphs where each vertex belongs to, on average, less than 3 happens-before relationships, Kronos can perform hundreds of thousands of queries per second. As the density of the graph increases, Kronos's throughput approaches a stable point where additional edges do not alter throughput. The majority of applications will likely resemble sparse dependency graphs as most applications do not need to impose a total order across all events, but instead order small groups of events together.

4.3 Fault Tolerance

Kronos uses chain replication to provide fault tolerance. As a test of its fault tolerance capabilities, we examined the performance of a 2-fault tolerant Kronos cluster. The underlying chain replication algorithm automatically removes failed servers from the cluster and can integrate new servers transparently. Figure 13 shows the throughput of the 2-fault tolerant cluster as a server fails and is re-added to the system. At the 30 s mark, the middle server in the chain is killed. The system recovers quickly and stays available for further operation. At the 60 s mark, a new server is introduced at the tail of the chain and begins the healing process to restore the service to being 2-fault tolerant. Overall, Kronos remains available and provides high throughput when servers are removed or re-added. This graph includes all the overhead of our unoptimized state machine replication implementation.

5. Related Work

Previous sections introduced Kronos, an efficient event-ordering service. To our knowledge, Kronos is the first system to abstract event-ordering into a generic, reusable service for building applications. While prior work often addresses event-ordering at the storage or communication levels, Kronos provides a more general abstraction that enables these applications and more.

Broadly speaking, prior work may be divided into the following categories.

Causality Capturing Techniques: Determining the ordering of events is a classic distributed systems problem with many well-known solutions. The problem was originally articulated as the motivation for Lamport timestamps [23] which captures happens-before relationships and provides a total ordering across events. However, Lamport timestamps can create spurious relationships that do not affect the correctness of the application. This problem cannot be addressed by increasing the granularity at which Lamport timestamps are maintained, because any partial order of Lamport timestamps is still too coarse-grained and cannot capture all relationships between events.

Vector clocks [17, 28] permit finer-grained partial orders than Lamport timestamps by establishing a partial order across events. Vector clocks use a vector of logical clocks to capture happens-before relationships. They enable more parallelism in the partial order than Lamport timestamps, but consume more space to achieve this. In the worst case, vector clocks require as many entries as parallel processes in the system [10] and exhibit significant overhead in deployments where there is a high-rate of server or process churn. The trade-off inherent to vector clocks is that the incidence of false relationships is inversely proportional to the granularity at which the vector clock is maintained.

There has been much work on improving vector clocks. Clock Trees [2] provide support for nested fork-join parallelism, Plausible Clocks [38] offer constant size timestamps while retaining accuracy close to vector clocks. Hierarchical Vector Clocks [21] provide more compact timestamps that adapt to the structure of the underlying network. While these techniques improve the trade-off between granularity and performance, they still restrict the kinds of dependencies an application may specify, and are not fully general.

Kronos takes an entirely different approach as compared to timestamp-based systems in how it captures causality. It maintains an explicit event dependency graph to track causality relationships and offers fine grain control to the application. By externalizing event/dependency handling and management and providing a unified API, Kronos simplifies event-ordering management for applications and enables dependencies to span application boundaries.

Consensus Protocols: Consensus protocols enable applications to construct a total order across all events. Examples of consensus protocols include Paxos [22], a crash-fault tolerant consensus protocol; Viewstamped replication [31] which operates in a primary-backup fashion; Tango [4], which replicates in-memory data structures using a shared, totally-ordered log; and multi-phase commit protocols [24, 34], a class of protocols that ensure all participants in a distributed transaction agree on whether to commit or abort by special-casing consensus [19].

Kronos permits applications to maintain a partial order across events in the system, increasing the flexibility with which events may be ordered. Of course, applications may always institute a total order across all events using Kronos.

Chain Replication: Kronos uses chain replication [40] to replicate its data. Kronos's reads from stale replicas resemble the apportioned queries in CRAQ [35]. Unlike CRAQ, the implementation used in Kronos does not require querying the chain tail to validate reads; the monotonicity invariant ensures that if a query returns a result, the result is as valid as if it were generated by the tail itself.

Embedded Causal Consistency: Many systems internally manage event ordering and track inter-process communication to provide causal consistency. Representative storage system examples include Bayou [36], a replica management system that exchanges logs between servers, allows for connection disruptions without preventing progress, and manages conflict resolution of causally conflicting operations through a set of user specified merge procedures. Depot [26] and SPORC [16] are cloud storage systems which employ variants of Fork-Join-Causal or Fork* consistency to enable practical cloud applications which can operate on untrusted cloud servers. Causal multicast [5, 6] protocols respect causal order when delivering messages to applications. Causality is also useful for supporting speculative execution [30], and bug and fault detection [1]. Externalizing event ordering to Kronos enables causal consistency guarantees that span multiple applications.

Application-Level Dependencies: Many systems rely on application-specific mechanisms to resolve and order events. Dynamo [12] is an eventually consistent key-value store that improves availability by using vector clocks to resolve concurrent writes. COPS [25] provides low-latency geo-replication using causal consistency and uses an application-specific conflict resolution mechanism to merge conflicting writes. Others have advocated for explicit causality by suggesting that applications explicitly select the subset of happens-before relationships that the data store should preserve to uphold application-level invariants [3].

These approaches are complementary to Kronos because Kronos provides a general method for event ordering in the form of a service. Applications may use Kronos within the application-defined handlers of causally-consistent data stores. Further, applications may explicitly, and directly, declare happens-before relationships in Kronos. Unlike other forms of application-level dependency management, Kronos permits the development of reusable components that naturally compose to achieve application-specific guarantees.

6. Conclusion

This paper proposed a new abstraction for tracking and managing dependencies between events in a distributed system. This abstraction opens the door for a new class of services in distributed systems, namely, event ordering services, which enable applications to explicitly manage and refine the possible timeline of events within the system. These new services provide a lingua franca for timeline management, enabling multiple independently developed components to form one integrated system that uses a common interface for time and event ordering. This approach facilitates the implementation of high-performance distributed systems that can provide strong guarantees by identifying potential cases of concurrency wherever possible. End-to-end performance benchmarks on Kronos-aware applications, such as a strongly consistent graph store and a fully-serializable transactional key-value store, demonstrate these performance gains. The graph store achieves up to 59$\times$ higher throughput than commercially available systems, and the key-value store can achieve 94% the throughput of non-transactional implementations. Example applications show that using the Kronos approach to build distributed systems can simultaneously offer high performance, enable reusable components, and uphold strong guarantees in the end application.

Acknowledgments

We'd like to thank our shepherd, Rodrigo Rodrigues, and our anonymous reviewers for their helpful feedback. This material is based upon work supported by National Science Foundation under Grant No. CNS-1111698, by Intel under Grant No. 10176559, the VMWare Graduate Fellowship program, and the National Science and Engineering Research Council of Canada.

References

[1] M. Attariyan and J. Flinn. Using Causality To Diagnose Configuration Bugs. In Proc. of *USENIX,* Boston, MA, June 2008.

[2] K. Audenaert. Clock Trees: Logical Clocks For Programs With Nested Parallelism. In *IEEE Transactions on Software Engineering,* 23(10), 1997.

[3] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The Potential Dangers Of Causal Consistency And An Explicit Solution. In Proc. of *SoCC,* San Jose, CA, Oct. 2012.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed Data Structures Over A Shared Log. In Proc. of *SOSP,* Farmington, PA, Nov. 2013.

[5] K. P. Birman, A. Schiper, and P. Stephenson. Fast Causal Multicast. Cornell University, Technical Report TR90-1105, 1990.

[6] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal And Atomic Group Multicast. In *ACM ToCS,* 9(3), 1991.

[7] P. Briggs and L. Torczon. An Efficient Representation For Sparse Sets. In *ACM LoPLaS,* 2(1-4), 1993.

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebooks Distributed Data Store For The Social Graph. In Proc. of *USENIX,* San Jose, CA, June 2013.

[9] M. Burrows. The Chubby Lock Service For Loosely-Coupled Distributed Systems. In Proc. of *OSDI,* Seattle, WA, Nov. 2006.

[10] B. Charron-Bost. Concerning The Size Of Logical Clocks In Distributed Systems. In *Information Processing Letters,* 39(1), 1991.

[11] D. R. Cheriton and D. Skeen. Understanding The Limitations Of Causally And Totally Ordered Communication. In Proc. of *SOSP,* Asheville, NC, Oct. 1993.

[12] 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 Proc. of *SOSP,* Stevenson, WA, Oct. 2007.

[13] R. A. Elmasri and S. Navathe. Fundamentals Of Database Systems. Addison-Wesley, US, 2010.

[14] P. Erdös and A. Rényi. On The Evolution Of Random Graphs. In *Mathematical Institute of the Hungarian Academy of Sciences,* 5(17–61), 1960.

[15] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proc. of *SIGCOMM,* Helsinki, Finland, Aug. 2012.

[16] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In Proc. of *OSDI,* Vancouver, Canada, Oct. 2010.

[17] C. J. Fidge. Logical Time In Distributed Computing Systems. In *IEEE Computer,* 24(8), 1991.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In Proc. of *OSDI,* Los Angeles, CA, Oct. 2012.

[19] J. Gray and L. Lamport. Consensus On Transaction Commit. Microsoft Research, Technical Report MSR-TR-2003-96, 2004.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-Free Coordination For Internet-Scale Systems. In Proc. of *USENIX,* Boston, MA, June 2010.

[21] D. A. Khotimsky. Hierarchical Vector Clock: Scalable Plausible Clock For Detecting Causality In Large Distributed Systems. In Proc. of *ICATM,* Colmar, France, 1999.

[22] L. Lamport. The Part-Time Parliament. In *ACM ToCS,* 16(2), 1998.

[23] L. Lamport. Time, Clocks, And The Ordering Of Events In A Distributed System. In *CACM,* 21(7), 1978.

[24] B. Lampson and H. E. Sturgis. Crash Recovery In A Distributed Storage System. Xerox Parc, Palo Alto, CA, Technical Report, 1976.

[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 Proc. of *SOSP,* Cascais, Portugal, Oct. 2011.

[26] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage With Minimal Trust. In Proc. of *OSDI,* Vancouver, Canada, Oct. 2010.

[27] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System For Large-Scale Graph Processing. In Proc. of *SIGMOD,* Indianapolis, IN, June 2010.

[28] F. Mattern. Virtual Time And Global States Of Distributed Systems. In Proc. of *PDA Workshop,* Chateau de Bonas, France, Oct. 1989.

[29] J. J. McAuley and J. Leskovec. Learning To Discover Social Circles In Ego Networks. In Proc. of *NIPS,* Lake Tahoe, CA, Dec. 2012.

[30] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink The Sync. In Proc. of *OSDI,* Seattle, WA, Nov. 2006.

[31] B. M. Oki and B. Liskov. Viewstamped Replication: A General Primary Copy. In Proc. of *PODC,* Toronto, Canada, Aug. 1988.

[32] D. Peng and F. Dabek. Large-Scale Incremental Processing Using Distributed Transactions And Notifications. In Proc. of *OSDI,* Vancouver, Canada, Oct. 2010.

[33] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In Proc. of *SOSP,* Farmington, PA, Nov. 2013.

[34] D. Skeen and M. Stonebraker. A Formal Model Of Crash Recovery In A Distributed System. In *IEEE Transactions on Software Engineering,* 9(3), 1983.

[35] J. Terrace and M. J. Freedman. Object Storage On CRAQ: High-Throughput Chain Replication For Read-Mostly Workloads. In Proc. of *USENIX,* San Diego, CA, June 2009.

[36] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts In Bayou, A Weakly Connected Replicated Storage System. In Proc. of *SOSP,* Copper Mountain, CO, Dec. 1995.

[37] Titan Distributed Graph Database. `http://thinkaurelius.github.io/titan/`.

[38] F. J. Torres-Rojas and M. Ahamad. Plausible Clocks: Constant Size Logical Clocks For Distributed Systems. In *Distributed Computing,* 12(4), 1999.

[39] J. Ugander and L. Backstrom. Balanced Label Propagation For Partitioning Massive Graphs. In Proc. of *WSDM,* Rome, Italy, Feb. 2013.

[40] R. van Renesse and F. B. Schneider. Chain Replication For Supporting High Throughput And Availability. In Proc. of *OSDI,* San Francisco, CA, Dec. 2004.