The VLDB Journal (2014) 23:627–652
DOI 10.1007/s00778-013-0343-9

REGULAR PAPER

# High availability, elasticity, and strong consistency for massively parallel scans over relational data

**Philipp Unterbrunner · Gustavo Alonso · Donald Kossmann**

Received: 5 February 2013 / Revised: 3 October 2013 / Accepted: 23 October 2013 / Published online: 19 November 2013

**Abstract** An elastic and highly available data store is a key component of many cloud applications. Existing data stores with strong consistency guarantees are designed and optimized for small updates, key-value access, and (if supported) small range queries over a predefined key column. This raises performance and availability problems for applications which inherently require large updates, non-key access, and large range queries. This paper presents a solution to these problems: Crescando/RB; a distributed, scan-based, main memory, relational data store (single table) with robust performance and high availability. The system addresses a real, large-scale industry use case: the Amadeus travel management system. This paper focuses on the distribution layer of Crescando/RB, the problem and theory behind it, the rationale underlying key design decisions, and the novel multicast protocol and replication framework it is composed of. Highlighting the key features of the distribution layer, we present experimental results showing that even under permanent node failures and large-scale data repartitioning, Crescando/RB remains fully available and capable of sustaining a heavy query and update load.

This work was conducted while Philipp Unterbrunner was at ETH Zurich.

P. Unterbrunner (✉)
Snowflake Computing, 107 South B Street, Suite 200,
San Mateo, CA 94401, USA
e-mail: philipp.unterbrunner@snowflakecomputing.com

G. Alonso · D. Kossmann
Systems Group, Department of Computer Science,
ETH Zurich, Universitätstrasse 6, 8092 Zurich, Switzerland
e-mail: alonso@inf.ethz.ch

D. Kossmann
e-mail: kossmann@inf.ethz.ch

**Keywords** Scan-only query processing · Dynamic partial replication · High availability · Atomic multicast

## 1 Introduction

The popularity of the cloud as a deployment platform and the extreme requirements of some enterprise and cloud applications have sparked a proliferation of so-called NoSQL systems. Members of this broad class of systems compromise on some of the functionality of traditional relational databases—typically the relational model and ACID transactions—in favor of scalability, elasticity, and performance for a target workload class. Earlier examples of NoSQL systems, notably Amazon Dynamo [27], abandon strong (external) consistency in favor of eventual (internal) consistency. Over the last few years, however, several NoSQL systems have implemented strong consistency guarantees [8,24,39,60,66,68,77,78], targeting OLTP and Web workloads which traditionally relied on a standard relational database.

What all these systems have in common is that they are designed for small updates, key-value access, or (if supported) small range queries over some predefined key column [72]. Conflicts are assumed to be rare and are detected and resolved either by distributed locking or by optimistic concurrency control techniques. At the end of transactions, atomic commit protocols such as Two-phase Commit [9] or Paxos Commit [42] are used to ensure consistency and durability. Fundamentally, these techniques couple synchronization and execution of operations, which leads to poor performance and availability when operations are long running and/or generate many consistency conflicts. This makes existing systems ill-suited for workloads which require large updates, non-key lookups, and/or large range queries.

Addressing precisely such workloads, we present a data store for the cloud, Crescando/RB, which is able to retain high availability under a heavy load (parallel, full-table scans and concurrent updates), even during large-scale system reconfigurations (multi-gigabyte data repartitioning).[1]

Crescando/RB is fully functional and addresses a real industry use case: the Amadeus travel reservation system. The use case, described in Sect. 2, requires range queries over key and non-key columns, atomic range updates, and also features stringent consistency, data freshness, and availability requirements.

The system consists of three components: Crescando, E-Cast, and Rubberband. Crescando is a scan-based, main memory, relational engine. Data are accessed exclusively via massively parallel, shared scans. The present paper focuses on the distribution layer on top of Crescando, namely the multicast protocol (E-Cast) and the replication framework (Rubberband) that compose it.

Crescando/RB is *not* a key-value store. It is also not a relational database, at least not by itself. Like Crescando, one instance of Crescando/RB implements a single relational table. This is the layer where availability, elasticity, and consistency matter and are difficult to implement. That said, we have in fact built a full SQL query processor on top of Crescando called SharedDB, which is published elsewhere [36,37].

The scan-only execution model of Crescando allows it to atomically perform *relational* queries and updates (arbitrary conjunctions of Boolean predicates), where key-value stores or semi-relational stores (BigTable [20], Spanner [24], Percolator [60]) provide data access only through a few key columns. As shown in a previous paper [74], Crescando is also *much* less sensitive to the combination and selectivity of selection predicates than a conventional relational database. This makes the engine especially suited for real-time business intelligence workloads such as our use case, where key-value stores are too restrictive in their consistency and/or data model, and relational databases lack the required performance, elasticity, availability, and robustness.

The distribution layer of Crescando/RB preserves the full data and consistency model of Crescando. This generality raises a serious problem for distributed execution: transactions may frequently span many or all replicas. The Rubberband replication framework and E-Cast multicast protocol are thus specifically designed for global, long-running, and conflict-heavy operations.

For performance reasons, Crescando/RB does not implement classic transactions (serializable sequences of reads and writes). Instead, the system offers atomic execution of arbitrary multi-record reads and writes (formally, sequential consistency [33]). The key technique to this effect is to handle events in two separate but pipelined stages. First, E-Cast is used to establish uniform agreement on the event order. At some later point, in the second stage, all affected nodes deterministically handle their (partition of the) events in the previously agreed order. This enforces strong consistency and durability. Our implementation aggressively pipelines the two stages across all architectural layers and protocols, overlapping the execution of *thousands* of queries and updates. This unusual design enables high throughput despite scan-based execution, without compromising consistency.

The two-stage execution approach has previously been explored by Thomson et al. in the context of statically partitioned databases [71,72]. In this paper, we show how to extend the two-stage approach to system reconfiguration events (membership changes, data repartitioning). By overlapping the execution of reads and writes with the handling of reconfiguration events, Crescando/RB implements *asynchronous reconfiguration*, thus providing high availability and elasticity.

In summary, Crescando/RB's scan-only, two-stage execution model enables arbitrary, relational queries and updates, where key-value stores provide atomic access only for individual keys or (small) key ranges. As a consequence, transactions in Crescando/RB may span many replicas. Our experimental results show that, in practice, Crescando/RB is able to scale well enough, despite potential global-ordering bottlenecks. We further show that even under node failures and large-scale repartitioning, the system remains available and can sustain a heavy read and write load.

### 1.1 Contributions

This paper makes the following contributions.

- A novel interpretation and formalization of dynamic partial replication as a stateful routing problem.
- A solution for said routing problem: E-Cast. Formally speaking, E-Cast is a dynamic, uniform, causal total order multicast protocol for asynchronous networks with unreliable failure detectors. It handles both message ordering and dynamic membership in a single, wait-free (asynchronous and pipelining) protocol. By plugging different *routing functions* into E-Cast, applications built on E-Cast can implement complex system membership and data replication schemes in a straight-forward manner. This novel design makes E-Cast as easy to use as a clas-

[1] We are not referring to formal availability as in the CAP Theorem [13] and related impossibility results [38]. We are concerned with *practical* availability: a system being responsive even under challenging conditions, i.e., full-table scans or data repartitioning. A system can be formally available but unusable if response times go beyond a certain level.

sic atomic broadcast protocol, while also providing much of the elasticity and performance benefits of a more general group communication toolkit.

- Rubberband, a framework for highly available, elastic data stores built on E-Cast. Rubberband supports atomic queries and updates with and—in contrast to existing systems—*without* key predicates.
- Crescando/RB, a relational data store that combines Rubberband and the Crescando engine into a specialized system for real-time business intelligence workloads. Crescando/RB addresses a real industry use case, the Amadeus travel reservation system.

### 1.2 Outline

The remainder of this paper is organized as follows. We first present the industry use case (Sect. 2) and an architectural overview of Crescando/RB (Sect. 3). We then set our work in scientific context (Sect. 4). Next, we explain the two components of the distribution layer in detail: E-Cast (Sect. 5) and Rubberband (Sect. 6). Finally, we present the results of an experimental evaluation of Crescando/RB (Sect. 7) and conclude (Sect. 8). A formalization and a proof of correctness of E-Cast are provided in the "Appendix."

## 2 Use case

Amadeus is a world-leading service provider for managing travel-related bookings (flights, hotels, rental cars, etc.). Its core service is the global distribution system (GDS), an electronic marketplace that forms the backbone of the travel industry. The world's largest airline carriers and many thousand travel agencies use the GDS to integrate their data. The main database in the GDS contains hundreds of millions of flight bookings. Just for the "hot" bookings, this results in a fact table of several hundred gigabytes in size.

The most important view of this fact table contains one record for every person on a plane. This is the schema used for the performance evaluation in this paper, and we will refer to it as the *Ticket* table. A ticket record is approximately 350 byte in size and consists of 47 columns, many of which are flags with high selectivity (e.g., seat class, wheelchair, dietary preferences).

The Amadeus GDS is a prime example of the growing need in industry for large, in-memory decision support and operational business intelligence systems [61]. The Ticket table is used in a large number of critical data services: reporting passenger lists, analyzing customer profiles of different airlines and airports, but also real-time operational decisions, e.g., flight cancelation in case of severe weather conditions. This requires range queries and, occasionally, range updates over both key and non-key columns. Examples are "give me the list of first-class passengers in a wheel chair departing from SFO next week," or "cancel all tickets departing from LHR to the United States tonight."

The original implementation of the Ticket table in a relational database system had reached a level of complexity where adding views and indices was no longer feasible let alone economical. As an alternative, we have developed Crescando, a main memory, scan-based engine designed for high throughput and robustness to evolving workloads with unpredictable, multi-dimensional predicates. The engine, published in an earlier paper [74], is now in production use at Amadeus, providing data services that were previously infeasible.

Encouraged by the performance and robustness of Crescando, Amadeus seeks to apply the engine to use cases well beyond the capacity and availability limits of a single machine. These new use cases include ticket pricing, flight availability, and log auditing.

What is required is a highly available distribution layer, which partitions and replicates the data in a manner that preserves the full data model and the unique performance properties of Crescando. In some use cases, the primary copy of data must reside in Crescando, so the distribution layer must also arbitrate (i.e., order) conflicting write requests. That is, the distribution layer must provide strong consistency.

To the best of our knowledge, no existing solution meets all the requirements in terms of query and update model, performance, availability, and consistency. We have thus developed our own distribution framework, Rubberband. Crescando/RB combines Crescando and Rubberband into an elastic, highly available, highly robust, relational data store for the cloud.

## 3 System overview

In this section, we provide an overview of the Crescando storage engine and its salient properties, followed by an outline of the distributed system, Crescando/RB.

### 3.1 Crescando

Crescando is a special-purpose relational storage engine. Its unusual design makes it well-suited for real-time business intelligence workloads such as our industry use case, but also creates some hard technical challenges for distributed execution.

Figure 1 shows a sketch of Crescando. Crescando is a *push-based* engine. Queries and updates are pushed in, executed in parallel via dedicated scan threads, and any generated results are pushed out asynchronously in an interleaved stream. All table data are horizontally partitioned and stored in main memory. Threads communicate exclusively through message queues.

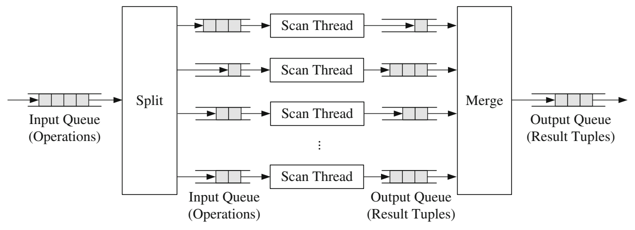


**Fig. 1** Crescando engine architecture (from [74])

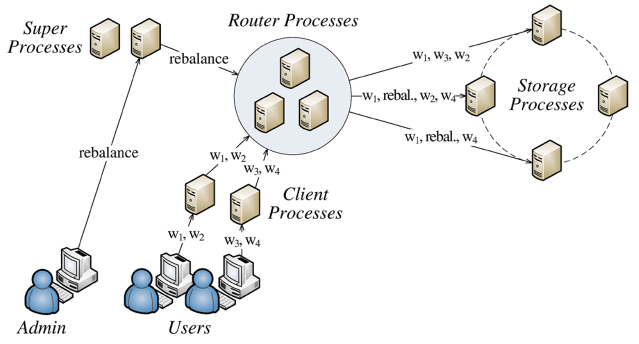


**Fig. 2** Crescando/RB system overview

Scan threads perform heavy batching of pending operations. Short-lived, read-optimized, cache-resident indices are built over the predicates of queries and updates that share a scan cursor. For example, when a scan thread finds a large fraction of pending queries features an equality predicate on a common attribute, say $Q_1$: `airport` = ‘LHR’, $Q_2$: `airport` = ‘LAX’, etc., it builds a hash index over the predicate constants (‘LHR’, ‘LAX’). During execution, the set of queries forms the inner side of an index-join between the data table and the (virtual) query table. Crescando indexes not just queries, but also update and delete operations. Because predicate indices are short-lived (one scan cycle), Crescando can rapidly adapt to changing workloads.

Owing to this unusual design, a single Crescando instance on a commodity machine can handle thousands of ad hoc, *non-key* queries and updates per second [74]. The downside is that the turn-around time for individual operations is in the order of *seconds*, regardless of how few or how many records are touched.

This raises serious problems with respect to distributed concurrency control. Holding distributed locks for several seconds in the face of thousands of large, concurrent reads and writes is a recipe for disaster [41]. As an alternative to distributed locking, optimistic snapshot isolation in various flavors has become a popular choice. But it too becomes expensive for long-running, distributed transactions, especially write transactions. Besides the danger of live lock, read and write sets of all *potentially* conflicting distributed transactions need to be exchanged and intersected before a transaction commits. This creates a compute-heavy serialization bottleneck with many workload-specific design trade-offs, see Hildenbrand [48].

Existing NoSQL data stores sidestep the problem by restricting the query and update model in various ways (see Sect. 4.1), such as requiring update transactions to not cross data partitions, or aborting queries that select too many records. But our main requirement for Crescando/RB is to preserve Crescando's key features: robust performance for *arbitrary* relational predicates. In other words, the goal of our work is to maximize system availability, elasticity, and performance—*without* sacrificing robustness, non-key data access, or strong consistency.

## 3.2 Crescando/RB

A Crescando/RB deployment is a dynamic set of processes, communicating via message-passing over an unreliable network (a private or public cloud). Figure 2 shows the four types of processes: client, super, storage, and router. Client processes issue read and write requests ($w_1$ through $w_4$ in Fig. 2). Super processes issue reconfiguration requests (e.g. *rebalance* in Fig. 2). And each storage process runs an instance of Crescando to hold application data and serve incoming requests. Router processes are discussed momentarily.

To avoid the loss of data and/or availability in case of process failures, Crescando/RB replicates each data object. For scalability and performance, the data are partitioned in a way such that each storage process holds some overlapping fraction of the complete data set (i.e., partial replication). The precise mapping of data to storage processes is determined by the *configuration* of the system, which may be altered at runtime through reconfiguration requests. A key feature of Crescando/RB in this respect is that it can be reconfigured at any point, without compromising data consistency, and without stopping read and write processing. Crescando/RB performs repartitioning and fail-over (partial live migration) in a novel manner optimized for high availability under long-running operations.

Client, super, and storage processes are all called *application processes* in the context of E-Cast. Application processes do not communicate directly with one another. (The only exceptions are low-isolation reads, read results, and data shuffling during repartitioning.) Instead, requests are multicasted through the replicated set of *router processes*. These routers give a number of crucial guarantees regarding message ordering and delivery. Most importantly, they guarantee that any message delivered by any two application processes is delivered in the same order by the two processes. This guarantee is called *uniform total order delivery* and allows Crescando/RB to give strong consistency guarantees.

**Table 1** Summary of key problems and their respective solutions in Crescando/RB

| Problem | Solution | Advantage |
|---|---|---|
| Unpredictable workload | Scan-only query processing | No index maintenance, robust performance |
| High access latency | End-to-end pipelining and batching | Throughput independent of latency |
| Large range queries | Read-one-write-all successor replication | No duplicate matching, minimal overhead |
| Non-key queries and updates | Up-front, total order of operations | No deadlocks, no livelocks, robust performance |
| System membership changes | Stateful routing, partial live migration | No blocking, continuous availability |

To be precise, E-Cast delivers operations in uniform total *causal* order. This order is established in a wholly asynchronous, pipelined fashion, and Crescando preserves the order during batched execution of operations inside individual scan threads. Thus, *clients never have to wait for confirmations*, allowing for high throughput, even under high network or execution latency.

Table 1 provides a final summary of the key problems and their solution in Crescando/RB.

## 4 Background and related work

Crescando/RB draws on a wide range of database and distributed systems literature. We focus on the five areas most germane to the paper: NoSQL data stores, push-based query processing, scan-only query processing, state-machine replication, group communication.

### 4.1 NoSQL data stores

The emergence of large-scale, cloud-based services has created a need for elastic, highly available data stores. Over the past years, a number of systems have been developed which radically simplify the programming model (query language, data model, consistency model) to enable a higher degree of parallelism, fault tolerance, and performance. Crescando/RB is an example of this class of systems referred to as "NoSQL" data stores [15].

Some NoSQL data stores abandon strong consistency in favor of weak or *eventual consistency*. Examples of eventually consistent data stores are PNUTs [23], Dynamo [27], SimpleDB [44], and Cassandra [53]. In many use cases, however, including ours, eventual consistency is not acceptable. *Strong consistency* is required.

A common solution is to extend a distributed hash table with agreement and atomic commit protocols such as Paxos [54] and Two-phase Commit [9]. Example systems include MegaStore [8], PNUTs [23], ElasTraS [25], G-Store [26], Scatter [39], and Spinnaker [65]. For scalability, the aforementioned systems shard the data into strictly disjunct key or entity groups. Consequently, consistent transactions cannot cross entity groups. In contrast, Crescando/RB imposes no such restrictions.

Other data stores with support for consistent multi-key or range operations do exist. We know of Spanner [24], Percolator [60], P-Store [66], Scalaris [68], ecStore [77], and CloudTPS [78]. Technically, MegaStore also offers transactions over multiple groups, but these are intended to be an exceptional case [24]. Commercial data store services with consistent multi-key operations include DynamoDB and Windows Azure Table Storage.

In contrast to Crescando/RB, some of these systems support read-modify-write transactions, not just atomic operations. However, *all* of these systems require updates (and often queries too) to carry a key predicate. In addition, commercial services enforce various tight restrictions on the number of objects and replication groups touched by operations, as well as on the maximum runtime of queries. As explained in the introduction, these restrictions cannot easily be removed from existing systems, since their concurrency control and atomic commit protocols rely on conflicts to be, if not known in advance, at least rare and easily detectable.

To the best of our knowledge, Crescando/RB is the only consistent data store—short of a full-blown relational database—which supports reads and writes with arbitrary conjunctions of Boolean selection predicates, key or non-key. And because the Crescando engine accesses data exclusively via massively parallel, shared scans, Crescando/RB is *much* less sensitive to the combination and selectivity of selection predicates than a conventional relational database.

The price to be paid is a lack of read-modify-write transactions (though conditional writes are obviously possible via predicates). This is not a fundamental limitation of our architecture however. It is a concession of the Crescando engine made in the interest of predictable performance. As shown by Thomson et al. [71,72], it is possible to overlap the execution of arbitrarily complex, conditional transactions given a pre-agreed serialization order. As long as the execution is *deterministic*, consistency can be preserved. Thomson et al. report significant throughput benefits in the context of small, static sets of nodes. With Crescando/RB, we extend the idea to a fully dynamic, fault-tolerant (fail-crash) set of nodes and show its additional merits for this case.

## 4.2 Push-based query processing

To answer a given SQL query, relational database systems rewrite the query as a tree of relational operators (join, projection, etc.). This tree defines a data flow, from the leaves (base relations) to the root (the client). In traditional systems, execution starts at the root, recursively *pulling* tuples up through calls to `next()` or some equivalent iterator interface to each operator [40].

Crescando is an example of a more recent line of systems that separate control flow from data flow. Rather than recursively pulling tuples up through the operator tree, these systems connect operators via buffered queues. That is, each operator forms an independent "mini engine." This design has a few advantages. First, it naturally enables both horizontal parallelism (e.g., parallel join) and vertical parallelism (pipelining) [18,74]. Second, individual operators run in tight loops, which leads to high instruction locality [46]. And third, push-based processing makes it easier to share operators across queries (multi-query optimization [69]) [6,14,36,46,74].

Notable examples of push-based systems are DataPath [6], TelegraphCQ [18], and QPipe [46]. In contrast to Crescando, these systems support full SQL, but have not been designed for unpredictable and update heavy workloads. We too have recently implemented multi-table SQL processing on top of Crescando. This system, *SharedDB* [36,37], extends the performance benefits of Crescando to a much wider range of applications. SharedDB has not been ported to Crescando/RB yet, but we expect no major obstacles since Crescando/RB retains the data model, consistency model, and basic interface of a stand-alone Crescando instance.

## 4.3 Scan-only query processing

Crescando is a scan-only engine; that is, data are accessed exclusively through scans, even for updates. To increase throughput, Crescando lets queries and updates share scan cursors. Shared scans have been implemented in the context of disk-based systems such as Red Brick DW [34], DB2 UDB [57], and MonetDB/X100 [79]. Shortly before Crescando, Raman et al. [63,64] had demonstrated that shared scans in main memory can provide predictable, low ("constant") access latency.

In contrast to these systems, Crescando has been designed for large numbers of non-grouping, fairly selective queries over live, concurrently updated data. Consequently, the cost of queries in Crescando is dominated by *predicate evaluation*, not aggregation or disk access. To speed up execution, Crescando builds *indices* over potentially *thousands* of queries and updates sharing a scan cursor. Predicate indexing is a technique that originates in the stream-processing domain [19,32].

## 4.4 State-machine replication

A deterministic, event-driven system can be made fault-tolerant by replicating the (initial) system state over independent processes (replicas), and ensuring that every replica handles the same state-changing events in the same order. This simple and well-known idea is called the *state-machine approach* [67]. In Crescando/RB, it is the set of E-Cast router processes, which forms a replicated state machine. The replicated state is the set of routed messages, their order, and their delivery status.

A replicated state machine implementation must ensure that every replica handles the same events in the same order, regardless of process and network failures. This problem is equivalent to a sequence of *distributed consensus* instances on an event and its predecessor event [43,45]. Paxos [54] and Multi-Paxos [62] are the most popular consensus protocols in practice, and they are found in many of the systems mentioned in Sect. 4.1.

Since consensus requires a majority of replicas to be live [38], a long-running system must either rely on replicas to eventually *recover* after they fail, or must be able to change the set of replicas dynamically. The former is typically implemented via write-ahead-logging to local disks. The latter constitutes a *reconfigurable state machine*. We argue that a reconfigurable state machine is to be preferred over disk-based recovery in a cloud environment, where machines may fail frequently and permanently, and local disks may not be reliable.

Lamport et al. [55,56] describe Paxos variants that can dynamically change the set of replicas. Other algorithms used to the same effect are RAMBO [58], RDS [21], and Zab [51] (the protocol behind ZooKeeper [49]). While conceptually simple, the implementation details of these algorithms are difficult to get right [2,16]. As a result, reconfigurable state machine implementations are rare. Indeed, all the related, strongly consistent data stores mentioned in Sect. 4.1 rely on disks for recovery.

Crescando/RB uses a disk-less, wait-free (asynchronous and pipelining), reconfigurable variant of Multi-Paxos. The system requires no disks,[2] real-time clocks, or perfect failure detectors, so it can run in any cloud environment.

## 4.5 Group communication

As outlined in the introduction, Crescando/RB does not interleave a distributed concurrency control protocol with the execution of queries and updates. Instead, operations are

[2] This does not mean one cannot add disks to Crescando/RB for purposes of disaster tolerance (e.g., general power outage), see Zab [51]. It just means that disks are not a requirement and not in the critical path of our protocols.

pre-ordered and routed by E-Cast. This can be seen as an application of *group communication*.

A group communication toolkit is a combination of a communication service (more-or-less reliable, ordered multicast primitives) and a membership service [22]. The membership service establishes distributed consensus on a sequence of system views, each of which defines a configuration that consists of one or more process groups. Applications can send messages within and across process groups and views, and the communication service gives ordering and delivery guarantees.

Chockler et al. [22] provide an overview of the theory of group communication; Defago et al. give a survey of the many known protocols and their respective guarantees [28]; and Birman [10] provides a historical account. Notable examples of group communication toolkits are Spread [3], Totem [4], JGroups [30], Isis [12], Isis$^2$ [11], Transis [31], QuickSilver [59], and Horus [75].

The E-Cast protocol used by Crescando/RB has been specifically designed for very strong message delivery and ordering guarantees across large numbers of overlapping replication groups with frequent membership changes. In contrast to existing protocols and toolkits, E-Cast does not explicitly maintain groups. Instead, E-Cast models reliable multicast as a stateful routing problem. This novel design makes E-Cast as easy to use as a plain atomic broadcast protocol, while also providing much of the flexibility and performance benefits of a general group communication toolkit.

## 5 E-cast

This section describes E-Cast, the multicast protocol that forms the heart of Crescando/RB.

### 5.1 Motivation

Consider an abstract data store consisting of 3 processes: $A, B, C$ (Fig. 3). The system stores 3 data objects: $x, y, z$. For fault tolerance, 2 copies of each object exist. Each process contains the primary copy of one data object, and a secondary copy of another data object (partial replication). Now, consider the following 3 independently submitted multi-key writes: $W_1 = \{x \leftarrow 1, y \leftarrow 1\}, W_2 = \{y \leftarrow 2, z \leftarrow 2\}, W_3 = \{z \leftarrow 3\}$.

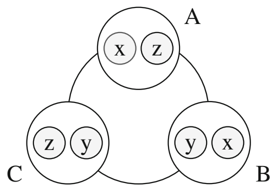


**Fig. 3** Dynamic partial replication

To be *strongly consistent*, the system must behave as if there was a single copy of each data object, and as if the writes were executed strictly in some serial order. If each process could independently choose the order in which to execute writes, the processes may disagree on the final values of $x$, $y$, and $z$. Formally, the system must agree on a partial order relation which determines the order of execution for every pair of conflicting operations [33]. (Here, $W_1$ and $W_2$, as well as $W_2$ and $W_3$ conflict directly, so $W_1$ and $W_3$ conflict transitively.)

If each write could only update a *single* data object or some disjunct group of data objects, one could just let the primary replica of each data object or group *independently* choose a serialization order for writes of the respective data object or group. (This is, in essence, how Spinnaker [65], G-Store [26], and many other consistent key-value stores work.) But with unrestricted multi-key and non-key writes, as required by our use case, the transitivity of the conflict relation means the system generally has to agree on a *total* order of operations. In the example, $W_1$, $W_2$, and $W_3$ conflict transitively, so agreement on a total order is required.

Apart from agreeing on an order relation, processes must also *uniformly execute* the writes that concern them. If the mapping of data objects to processes is static, clients can cache the mapping and send writes directly to the affected processes. But if the mapping can change concurrently (for elasticity or fault tolerance), writes may be lost or received by the wrong processes, leading to loss of uniformity and thus inconsistent data.

We call the resulting problem *dynamic* partial replication. Rubberband, our data store framework described in Sect. 6, solves dynamic partial replication by modeling it as a stateful routing problem. E-Cast in turn implements stateful routing.

### 5.2 System model

We present only a semi-formal system model and problem statement here. A proper formalization and a proof of correctness are provided in the "Appendix."

The system consists of a set of processes, which communicate via messages passed over an asynchronous network. To simplify our protocols, we assume processes communicate through quasi-reliable FIFO channels, as defined by Aguilera et al. [1]. That is, if a correct process $p$ sends a message $m$ to a correct process $p'$, then $p'$ eventually receives $m$, in send order. This guarantee is easily implemented via sequence numbers, as in TCP for example, so it is not an unrealistic simplification.

Processes may crash *silently*, but otherwise behave as specified (no Byzantine failures). We assume processes have access to inaccurate but complete failure detectors. That is, a process may suspect another, correct process of being faulty (i.e., false positive), but every faulty process is eventually suspected of being faulty by every correct process. Such failure detectors are easy to implement with periodic heartbeat messages, so, again, this is not an unrealistic simplification.

It is known that eventual weak accuracy (eventually *not* suspecting a correct process of being faulty) is formally required for liveness of uniform atomic broadcast and thus E-Cast [17]. As is common practice in such protocols, we assume failure detectors are accurate enough to ensure liveness in practice.

## 5.3 Protocol overview

E-Cast is a dynamic, uniform, causal total order multicast protocol. As such, it has no notion of reads, writes, or data objects. It just transmits messages. The protocol is specified in terms of two roles: application and router processes. Application processes submit and deliver messages;[3] routers act as message sequencers and intermediaries between application processes.

Applications (such as Rubberband) define a *routing function* that is plugged into the routers at system bootstrap. The existence of this function distinguishes E-Cast from existing group communication toolkits and is key to its usability. Given a history, i.e., sequence of previously routed messages and a new message to be routed, that function returns a set of processes (the *destination set*) to deliver the message to.

That is, application processes do not explicitly tell E-Cast which processes to deliver a message to. Instead, they put just enough information into the message, so that the routing function can compute the destination set. Because the routing function sees the system configuration at the point in global history where the message will be delivered, it becomes easy to implement complex system membership and data placement schemes.

To agree on a global history of messages, and to ensure uniform delivery in case of router crashes, the routers *atomically broadcast* incoming messages among each other (but not the application processes). A single router, called the *leader*, then forwards the messages in the agreed-upon order to their destination set.

We have implemented our own atomic broadcast protocol using an asynchronous (heavily pipelined), reconfigurable variant of Multi-Paxos [62], a well-known protocol for solving a sequence of consensus problems. E-Cast is more efficient and scalable than atomic broadcast by itself, because it confines the unavoidable consensus problem to the (small) set of routers. Application processes do not participate in consensus.

[3] We write "submit" and "deliver" rather than "send" and "receive" to distinguish between the interface of E-Cast and that of the lower-level network protocol.

**Algorithm 1**: Application Process

```
1  state variables
2      $t_{max} \leftarrow -1$ ;                    /* max ts of any delivered msg */
3      $W \leftarrow \langle\rangle$ ;          /* send window (queue of unstable msgs) */
4  /* submit/deliver/acknowledge are external events            */
   /* send/receive means quasi-reliable FIFO unicast            */
5  upon submit msg $m$ do
6      append $m$ to $W$
7  upon receive $\langle$"stable", id($m$)$\rangle$ do
8      remove $m$ from $W$
9      acknowledge $m$ to user
10 upon receive $\langle$"deliver", $m$, $t\rangle$ from router $r$ do
11     if $t > t_{max}$ then
12         $t_{max} \leftarrow t$
13         deliver $m$ to user
14     send $\langle$"confirm", id($m$)$\rangle$ to $r$
15 periodically
16     choose some router $r$ believed to be correct
17     foreach msg $m \in W$, in order do
18         send $\langle$"route", $m\rangle$ to $r$
```

Also, membership of router processes is separate from that of application processes. Router membership is solved as part of atomic broadcast. Application processes in contrast announce process failures and other membership changes by submitting messages to E-Cast. The ability of applications to run custom failure detection protocols and announce failures through regular messages is an important feature. It obviates the need for a complex membership protocol between E-Cast and the application.

## 5.4 Application process algorithm

Pseudo-code for application processes is given in Algorithm 1. We write [[X]] to refer to line number X.

When a user submits a message $m$ [[5]] to an application process $a$, that message is first buffered in $a$'s send window $W$ [[6]]. Periodically, the contents of that window are sent—in submission order—to some router (any router is safe) [[16–18]]. Eventually, the routers will forward $m$ to all its destination processes, which will eventually deliver and confirm the message [[14]], or fail.

By definition, a message becomes *stable* once it has been delivered by all correct (non-failed) destination processes. Routers collect confirmations to detect when the message $m$ is stable. Once $m$ is stable, the routers inform the submitter of $m$, process $a$, by sending it a protocol message $\langle$"stable," $m\rangle$. Upon receiving the latter [[7]], $a$ acknowledges successful delivery to the user and removes $m$ from the send window [[8–9]].

Every message $m$ has a unique timestamp $t$, representing its position in the global history of messages, as decided by the routers. Application processes deliver messages strictly in timestamp order [[11–13]]. Any incoming message is

**Algorithm 2**: Router Process

```
1  state variables
2      $Q \leftarrow \langle\rangle$ ;                                  /* learned m-seq */
3      $W \leftarrow \langle\rangle$ ;          /* send window (queue of learned, unstable
       msgs) */
4      $C \leftarrow \langle\rangle$ ;     /* map of msgs to destination procs that have not
       yet confirmed and are not suspects */
5  /* send/receive means quasi-reliable FIFO unicast                  */
   /* propose/learn means FIFO atomic broadcast                       */
6  upon receive ⟨"route", $m$⟩ do
7      if $m$ not stable then
8          propose ⟨"route", $m$⟩ to all routers
9      else
10         send ⟨"stable", id($m$)⟩ to submitter($m$)
11 upon receive ⟨"confirm", id($m$)⟩ from app. proc. $a$ do
12     if $C[m]$ exists then  $C[m] \leftarrow C[m] \setminus \{a\}$
13 upon learn ⟨"route", $m$⟩ from some router do
14     if $m$ not in $Q$ then  append $m$ to $Q$ and $W$
15 upon learn ⟨"stable", id($m$)⟩ from some router do
16     remove $m$ from $W$ and remember $m$ as stable
17 periodically if self is leader then
18     foreach $m \in W$, in order, where $C[m]$ not exists do
19         $C[m] \leftarrow$ destsetpfx($Q$,$m$) \ suspects($Q$)
20         $t \leftarrow$ the position of $m$ in $Q$
21         foreach $a \in C[m]$ do
22             send ⟨"deliver", $m$, $t$⟩ to $a$
23 periodically  foreach $m$ where $C[m]$ exists do
24     $C[m] \leftarrow C[m]$ \ suspects($Q$)
25     if $C[m] = \{\}$ then
26         propose ⟨"stable", id($m$)⟩ to all routers
27         destroy $C[m]$
```

either delivered or discarded immediately. Nonetheless, E-Cast guarantees gap-free (loss-less) message delivery.

The reason is that when a message with timestamp $t$ is received, the receiver is guaranteed to have previously received any message with timestamp $t' < t$. This holds because of the following rule obeyed by all processes: *when sending a message to some process, first send any preceding, unstable message*. And since the network channels are quasi-reliable FIFO channels, application processes do not have to wait for messages to become stable before sending subsequent messages. This feature is critical for high throughput.

Some important optimizations are missing from the pseudo-code for simplicity. Notably, an application process $a$ need not resend any message $m$ to a router $r$, if $a$ had sent $m$ to $r$ before. So to avoid wasting bandwidth, an application process should keep track of which messages have been sent and only choose a new router if it believes that its current router has failed.

### 5.5 Router process algorithm

Pseudo-code for router processes is given in Algorithm 2. We write "m-seq" as an abbreviation for "sequence" or "queue of messages." It is the main task of routers to ensure that messages are delivered in some global, total order. To this end, each router maintains an active replica of an m-seq $Q$, which represents the global history of messages and their order.

When a router *receives* a message $m$ which is not known to be stable, it *proposes* the message. That is, it atomically broadcasts $m$ to all routers, including itself ⟦8⟧. When the router later *learns* $m$ ⟦13⟧, it appends $m$ to its m-seq $Q$ and send window $W$. By the definition of atomic broadcast, all routers learn the same messages in the same order, regardless of failures. Hence, there is agreement on $Q$ and a global timestamp for each message in $Q$ ⟦20⟧.

Routers not only order messages, but also ensure uniform delivery. That is, every correct (non-failed) process in the destination set of a message $m$ shall deliver $m$. To this end, the lead router (see below) periodically sends all messages in its send window to their respective destination sets ⟦22⟧, as determined by the application-defined destination function: `destsetpfx` ⟦19⟧.

Once all application processes in the destination set of a message $m$ have confirmed $m$ ⟦12⟧ or have become suspects (failed) ⟦24⟧, $m$ is proposed stable via atomic broadcast ⟦26⟧. Eventually, every router learns that the message $m$ is stable, and $m$ is removed from everyone's send window ⟦15⟧. Like application processes, routers can handle an arbitrary number of pending (unstable) messages. As a result, E-Cast can achieve very high throughput, independent of the network latency.

One aspect that may be surprising is that routers treat messages as stable even if *every* destination is a suspect ⟦25⟧. The idea is that if this is relevant to the submitter of a message, the application can easily inform it by defining the routing function in a way that delivers it (or some management process) the previous message(s) that caused all destinations to be suspects.

To keep the pseudo-code concise, we do not show how routers remember and later identify messages as stable ⟦16⟧. Our implementation uses per-sender sequence numbers (not to be confused with the global timestamps). Routers maintain a high watermark of sequence numbers for the stable messages of each unsuspected application process. To check whether a message is stable, a router simply compares the message's sequence number to the sender's watermark.

For efficiency, only one router (the *leader*) forwards each message to its destination set. Given an atomic broadcast protocol between routers, leader election is easy to implement. However, leader election is purely an optimization. Any router may safely consider itself a leader at any time.

The pseudo-code omits some other optimizations. For instance, a router need not repropose a message $m$ which is already in $Q$ when $m$ is received. Further, to avoid application processes having to repeat "route"-messages in order to receive a "stable"-message, a router process can safely send a message ⟨"stable," id($m$)⟩ to the application process that submitted $m$, as soon as the router learns ⟨"stable," id($m$)⟩.

A complete proof of correctness of E-Cast is provided in the "Appendix."

### 5.6 Implementation

E-Cast has been implemented in roughly 10,000 lines of Erlang [5]. The Erlang process and distribution model matches our formal model, so Erlang was a natural choice. The codebase of the implementation is substantially larger than the pseudo-code might suggest, because the implementation has to deal with many real-world issues such as failure detection, congestion control, and distributed garbage collection. We can only provide a high-level overview here. A much more detailed account is available elsewhere [73].

Figure 4 visualizes the implementation stack. Proceeding bottom-up in the stack, TCP is abstracted away by the Erlang process and channel model. Erlang nodes communicate through asynchronous FIFO channels. We added sequence numbers where necessary to make these channels fully reliable. Using reliable FIFO channels, we have implemented an asynchronous (heavily pipelined), reconfigurable variant of Multi-Paxos [62], a protocol for solving a sequence of consensus problems.

This Multi-Paxos variant is used for uniform FIFO atomic broadcast (ufabcast). Ufabcast in turn is used to replicate router processes; more precisely, ufabcast is used to replicate a common sequence of routed messages, i.e., m-seq, between router processes, see Algorithm 2. Conceptually, the destination function takes this m-seq as input at every step. Implementing this directly would require unbounded memory and would be very inefficient, however. So, instead, in our implementation, applications (such as Rubberband) plug a user-defined `Config` module into the routers. The `Config` module defines `apply, destset`, and `suspects` functions over config objects and messages. Routers maintain a sequence of config objects, created by successively *applying* learned messages to the latest config.

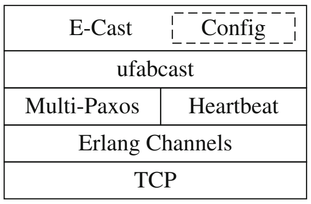


**Fig. 4** E-cast implementation stack

Config objects can be seen as checkpoints of the routing configuration, as determined by the m-seq they represent. Routers only ever store distinct `Config` objects, so there is no space overhead if the configuration does not change. Config objects are garbage-collected when all messages that rely on them for routing have become stable. For a concrete `Config` module, see Sect. 6.3.

Dynamic router membership is implemented in a manner roughly similar to Zab [51]. Routers can join and leave the system at any time. They form a reconfigurable state machine, see Sect. 4.3. To detect router failures and also for garbage collection of stable Paxos instances, ufabcast uses a simple heartbeat protocol. A crashed router can be replaced in a matter of seconds, as demonstrated in the experimental evaluation. This is a very useful property when building highly available, long-running systems in a cloud environment.

So, while router membership is implemented on the level of ufabcast, application membership is determined by the sequence of messages routed through E-Cast. The two are wholly independent. But in both cases, there is no static configuration and no disk-based recovery (though this could be added for disaster tolerance).

## 6 Rubberband

In this section, we present Rubberband, a novel framework for elastic data stores with strong consistency. Rubberband relies on E-Cast for reliable, ordered message transfer. In doing so, it can offer atomic queries and updates with arbitrary combinations of key and non-key predicates. In contrast to related systems (cf. Sect. 4.1), Rubberband even allows updates *without* key predicates. This enables Crescando/RB to scale to a large set of storage processes without restricting the query and data model of Crescando. In brief, Rubberband

- delivers writes uniformly, in causal total order (yields sequential consistency [33]);
- is completely wait-free (clients can safely submit a high-rate stream of reads and writes without waiting for confirmations);
- elastically scales to many storage processes;
- allows data stores to be continuously available, even during reconfiguration (data repartitioning);
- tolerates permanent, silent failure of any process;
- does not require stable storage (disks), perfect failure detectors, or real-time clocks.

In this paper, we present the results of Rubberband over Crescando (forming Crescando/RB), but we have also built a prototype of Rubberband over Memcached, a popular key-value engine [35].

### 6.1 Process types

Rubberband defines and implements the following three types of processes, each of which is also an E-Cast application process.

*Client Processes* submit read and write messages. Depending on the type of message and the chosen isolation level (cf. Sect. 6.7), an individual read or write may be transmitted through E-Cast, or sent directly to one or more storage processes.
*Storage Processes* each store a partition of application data, as explained below. Delivered read and write messages are forwarded to the storage engine (Crescando) in an order that preserves consistency. Conversely, result tuples emitted by the storage engine are streamed back directly to the respective client process, which in turn delivers them to the user.
*Super Processes* submit *configuration messages* on behalf of users. Configuration messages change the mapping of keys to storage processes, as explained in Sect. 6.4. Super processes maintain a replica of the latest system configuration (set of processes and mapping of keyspace partitions to storage processes). Super processes are also responsible for failure detection of client, storage, and super processes.

Recall Fig. 2 on page 4 for an exemplary instantiation of Rubberband that shows all types of processes.

### 6.2 Data placement

Rubberband dynamically partitions the data along a user-defined key domain, using a consistent hashing scheme popularized by Chord [70]. Unlike in Chord, however, messages are not dynamically routed along the ring. Instead, (most) messages are transmitted through the E-Cast router processes.

Storage processes are arranged in a logical ring. Each storage process is assigned a unique key called its identifier (ID). The ID determines a process' position on the ring. If a process $s$ is assigned the ID $k$ and its (in key order) successor $s'$ is assigned the ID $k'$, then $s$ holds the primary replica of all data objects with key greater-or-equal $k$ and less-than $k'$. We call this range of keys the *primary partition* of $s$.

In order to tolerate permanent process failures, Rubberband uses *successor replication* [52]. This scheme is widely used in other NoSQL data stores, including Dynamo [27] and Cassandra [53].

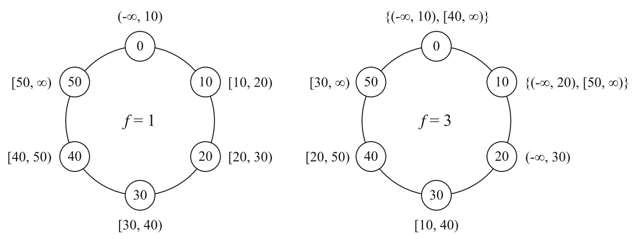


**Fig. 5** Successor replication

A replication factor of $f$ means the $f - -1$ successors of a process $s$ on the ring hold a replica of every object in $s$'s primary partition. The resulting union of key ranges is called the *partition* of a process. Because partitions of successive processes overlap, it requires the correlated failure of $f$ successive processes for data loss to occur. An example for replication factors $f = 1$ and $f = 3$ is shown in Fig. 5. Users can choose an arbitrary replication factor at system initialization. The replication factor becomes fixed and applies to the entire system, though this could easily be generalized.

Rubberband performs read-one-write-all (ROWA) replication. Thus, for a replication factor of $f$, a point read is served by 1 storage process, and a point write is served by $f$ storage processes. In general, every range read is processed by a minimal set of storage processes whose union of partitions fully covers the queried range. Every range write is processed by every storage process whose partition overlaps with the written range.

To extract the target key or key range(s) from a given read or write message, Rubberband calls a user-defined callback function. By providing a function that does hashing, applications can perform hash partitioning instead of range partitioning when desired.

The combination of successor replication and ROWA is unusual; symmetric replication [52] and quorum reads being more common in related systems. Successor replication was chosen over symmetric replication, because every process here covers a single, consecutive range of keys (with wrap-around at key 0). Thus, even a large range query in Crescando/RB can usually be mapped to a single range query in the Crescando engine of every target process. And as explained in Sect. 6.5, quorum reads perform poorly for range queries due to the necessary duplicate matching.

### 6.3 Message routing with E-cast

Rubberband defines a variety of message types. The two main types of messages are *data messages* and *configuration messages*. Every data message is either a *read message* or a *write message*.

A configuration message is either a *membership message* or a *keyspace message*. A membership message changes the set of client, storage, or super processes that are consid-

**Algorithm 3**: Rubberband Config Module Sketch

```
function apply(m, c) begin
    if m is a configuration message and sender of m is an
    unsuspected process then
        update (side-effect free) the config c and return the
        new config
    else
        return the original config c
end
function destset(m, c) begin
    if sender of m is an unsuspected process then
        if m is a data message then
            p ← the set of key ranges (partition) read or
            written by m
            if m is a read message then
                (deterministically) find and return a set of
                unparted, unsuspected storage processes
                in c whose union of partitions fully covers p
            else m is a write message
                find and return the set of all unparted,
                unsuspected storage processes in c whose
                partitions overlap with p
        else m is a configuration message
            return the union of the set of all super
            processes in c, and the set of all unsuspected
            client and storage processes in c that are
            affected by m
    else
        return {}
end
function suspects(m) begin
    if sender of m is an unsuspected process then
        return the set of all processes suspected by m
    else
        return {}
end
```

ered part of the system, while keyspace messages change the mapping of keyspace IDs to storage processes. The different membership and keyspace messages and their effects are explained in Sect. 6.4.

With the exception of low-isolation reads (cf. Sect. 6.7), all messages are transmitted through E-Cast. As explained previously, each router process maintains a local m-seq object. Whenever a router process learns a new message, it appends it to its local m-seq object. An m-seq object internally maintains a sequence of config objects, created by successively *applying* the appended messages to the latest config. When all messages pertaining to a specific config have become stable, that config is no longer needed and is garbage-collected.

In the case of Rubberband, a config object consists of a set of client process references, super process references, and a mapping of keyspace identifiers to storage process references. E-Cast requires `Config` modules to define the following callback functions: `destset`, `apply`, and `suspects`. The Rubberband `Config` module and the implementation of these callback functions are sketched in Algorithm 3.

The `destset` function computes the destination set of data messages in read-one-write-all (ROWA) manner. The destination set of a configuration message is simply the set of all affected processes, which always includes the set of super processes, since these must maintain a replica of the current system configuration.

The `apply` function works as follows. Whenever a data message is applied to a config object, it returns the original config (reads and writes do not change the system configuration). But when a configuration message is applied, it returns a new config which reflects the effect of the message and may change the routing of subsequent messages. For example, when a *suspect message* $m$ is applied such that suspects$(m) = \{s\}$ where $s$ is some storage process in the keyspace, then Rubberband will stop routing subsequent read messages to $s$ and instead use the neighbors of $s$ on the keyspace, which hold replicas of the data partition assigned to $s$.

The `suspects` function informs E-Cast of failed application processes. Messages sent by suspected processes have no effect, which ensures that suspected processes cannot corrupt the system state. The concrete implementation of the config module does not remember which processes are suspected (there may be many in a long-lived system), but instead remembers which unsuspected processes are currently part of the system.

Note that the `destset` function must be deterministic to ensure agreement on the destination set of a message across multiple router processes. In the case of read messages, there are typically many possible, correct destination sets due to multiple replicas being available for each data object. A naïve solution is to always choose the replica with the lowest identifier, but this would lead to poor load balance. As a better solution, Rubberband client processes tag every read message with a random *replica offset*. This offset determines which storage process to choose in every replication group.

### 6.4 Asynchronous reconfiguration

There are two types of configuration messages: membership and keyspace messages. Membership messages change the set of application processes that are considered part of the system; for example, a storage join message adds a (standby) storage process to the system. Keyspace messages change the assignment of identifiers to processes, and thereby the placement of the data. Rubberband defines four types of keyspace messages:

*Expand* Assign an ID to a standby storage process, which upon delivery of the message becomes active.
*Contract* Contract the keyspace by removing an active storage process from it.

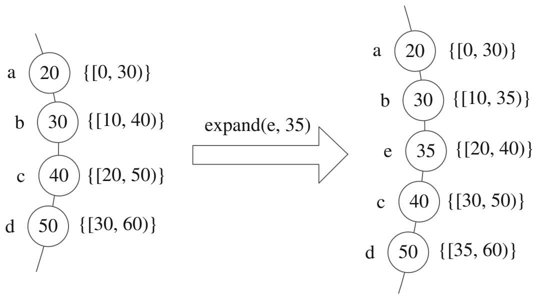


**Fig. 6** System reconfiguration: keyspace expansion

*Replace* Replace an active storage process with a standby storage process, assigning the same ID to it.

*Rebalance* Assign a different ID to an active storage process, thereby rebalancing the keyspace.

Expand and contract messages alone would be sufficient to perform any keyspace transformation. Replace and rebalance messages have been added to lower the number of reconfiguration steps and thus the amount of data shuffling for these frequent transformations.

Keyspace messages contain a list of instructions, each of which tells a specific storage process to copy, receive, or delete some partition of data. These instruction lists are created by the super process which submits the configuration message. This is because for a given keyspace transformation, there can be multiple correct instruction lists, and choosing the "best" of these is a non-trivial optimization problem, discussed in Sect. 6.5.

Consider Fig. 6 for an example with replication factor $f = 3$. On the left, a part of a larger (irrelevant) keyspace is shown, containing 4 storage processes $a$, $b$, $c$, and $d$, which have been assigned the IDs 20, 30, 40, and 50, respectively. A user has requested to *expand* the keyspace (elastic scale-out), assigning ID 35 to storage process $e$. On the right, the keyspace after the transformation is shown. Process $e$ now holds the partition $\{[20, 40)\}$, and the partitions of $b$, $c$, and $d$ have shrunk. Consequently, $e$ needs to receive copies of all data objects in the partition $\{[20, 40)\}$, and $b$, $c$, and $d$ need to delete part of their data. One possible list of instructions to that effect is this (the notation should be self-explanatory):

$\langle \text{copy}, b, e, \{[20, 40)\}\rangle$
$\langle \text{delete}, b, \{[35, 40)\}\rangle$
$\langle \text{delete}, c, \{[20, 30)\}\rangle$
$\langle \text{delete}, d, \{[30, 35)\}\rangle$

When a storage process receives a keyspace message, it projects out the instructions which concern itself and passes them on to the storage engine (Crescando) for execution. In the example, storage process $b$ passes $\langle\langle \text{send}, e, \{[20, 40)\}\rangle, \langle \text{delete}, \{[35, 40)\}\rangle\rangle$ to its storage engine. Process $e$ passes $\langle \text{receive}, b, \{[20, 40)\}\rangle$. The engines of $b$ and $e$ then shuffle the data through direct connections (not E-Cast). We call the act of shuffling data in this manner *partial live migration* [7].

Data consistency relies on every storage engine executing its instructions atomically, in the same total order, also with respect to read and write messages that are delivered before and after partial live migration. We emphasize that it is *not* necessary to stop the system during reconfiguration. Crescando/RB *never waits*. Messages (reads, writes, reconfigurations) can be submitted at any point and will be promptly delivered and executed. An arbitrarily long sequence of configuration messages can be submitted before even the first reconfiguration has been completed.[4]

The fact that E-Cast and Rubberband do not wait during reconfiguration (messages keep being delivered) enables *asynchronous reconfiguration*; that is, processes not involved in partial live migration are not impacted by it. As long as there is at least one available process for each key in the keyspace, the entire keyspace can be read and written. This allows Crescando/RB to scale elastically and to provide continuous availability, despite its very strong consistency model.

## 6.5 Reconfiguration as an optimization problem

The downside of this approach is that there is no safety net in the case that partial live migration fails. If the sender of a data partition crashes before the transmission is complete, the receiver is left in an incomplete state. All other replicas may have already been updated, so the receiver must declare itself failed as well.

In principle, this problem can be mitigated by giving storage processes the ability to time travel; that is, the ability to recover to a previous state. But time travel is an expensive feature that is not supported by all storage engines, including Crescando. So instead, we treat reconfiguration as an optimization problem where the optimization metric is the probability of data loss.

While there is a wide range of literature on data and load balancing in NoSQL data stores, we are not aware of any prior work which treats shuffling a large amount of data atomically as an optimization problem by itself. We believe this is due to two main facts. First, most systems hash partition data over virtual processes (assigning many different, virtual IDs to a

[4] There is one small caveat. Since the instruction list is created by the super process submitting a keyspace message, correctness relies on the fact that the configuration does not change concurrently, from the point where the super process submits a keyspace message to the point where the message is delivered. Rubberband solves the problem by timestamping every keyspace message with the configuration version at the point where the message is created. In the (rare) case where the configuration has changed concurrently, a conflict is detected and the keyspace message has no effect.

single physical machine [27,70]). Consequently, adding or removing a physical machine generates less of a hot spot than in range repartitioning. Second, most NoSQL data stores are not designed for large range queries or multi-key updates. A solution to data shuffling based on multi-version read quorums and anti-entropy is more appropriate in that context [27,29,53].

Generally speaking, the problem of atomically shuffling large partitions of data among a small set of processes manifests in the context of *elastic range partitioning with strong consistency*. Besides Crescando/RB, only Scalaris [68] and ecStore [77] even support this combination to the best of our knowledge.

We are not aware of any data on Scalaris' or ecStore's availability and performance during repartitioning. However, both systems use multi-version quorum reads and writes. Thus, they remain available only as long as there is a quorum (majority) of processes available in every replication group. (Available here means "able to serve requests," not just "responds to ping.") Multi-version quorum reads also become *very* expensive for large reads, due to the necessary duplicate matching. In contrast, Crescando/RB remains available with only a *single* process in each replication group; there is no need for duplicate matching, and the Crescando engine is specifically designed for large reads and writes.

Returning to Rubberband, every keyspace message contains a list of instructions for the affected storage processes. But depending on the replication factor, many different instruction lists are possible. Let us revisit the reconfiguration example discussed previously (Fig. 6). Assume now that at the point the transformation is requested, process $d$ has been declared *failed* (through an earlier message). It is still in the keyspace, but is unable to participate in partial live migration. Figure 7 shows two of the remaining solutions.

In Solution 1, process $c$ is chosen to send partition $\{[20, 30)\}$ to the new process $e$. Failure of $c$ during transmission would then leave $c, d$, and $e$ in a failed state. As

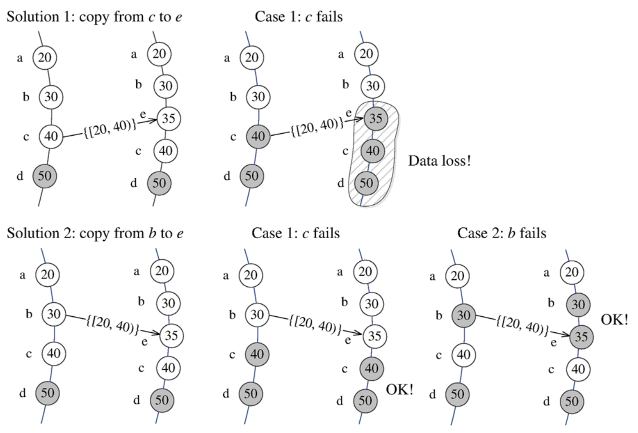


**Fig. 7** Transitive failures during partial live migration

explained previously, for a replication factor of $f$, the failure of $f$ successive storage processes causes data loss. In the example, $b$ may already have deleted $\{[35, 40)\}$ to reflect the new data placement, so the data objects in $\{[35, 40)\}$ are lost. In contrast, if process $b$ is chosen to copy $\{[20, 30)\}$ to $e$ (Solution 2), then the additional failure of any single process cannot cause data loss. For any failure case, at least one process is live in every replication group. Solution 2 is thus preferable.

The example is relatively simple, but things get complicated for higher replication factors and particularly for contract messages, where multiple storage processes need to receive data. There can be chains of dependencies between processes, so the failure of a single process can cause a whole chain of processes to fail transitively.

Rubberband models each possible solution as a directed graph, with an edge connecting every pair of ⟨sender, receiver⟩ processes involved in partial live migration. When a super process is asked to issue a keyspace message, it first enumerates all minimal[5] solutions that achieve the desired keyspace transformation. For practical replication factors ($f \leq 5$), the number of minimal solutions is small enough for full enumeration.

The super process then simulates all possible single process failures by flooding the different solution graphs. Based on this simulation, the solutions are ranked. Put simply, the optimizer picks the solution which for any additional single process failure leaves the smallest number of (transitively) failed processes in any group of $f$ successive processes. That is, a solution that has 3 groups with 2 failed processes each is preferred over a solution with 1 group with 3 failed processes.

The ranking favors solutions that involve the smallest number of processes of any individual replication group. In particular, it avoids solutions that involve *all* storage processes of a replication group. Besides improved fault tolerance, this has another positive effect. Whenever possible, there is at least one storage process per replication group that is not busy with partial live migration. That process is able to execute reads and writes, leaving the system fully available.

## 6.6 Consistency versus robustness versus scalability

As explained in Sect. 5.1, strong consistency requires agreement on a partial execution order covering every pair of conflicting operations [33].

The related NoSQL data stores discussed in Sect. 4 create the partial order on demand, during execution of queries and updates, by distributed locking or by optimistic concurrency

[5] A non-minimal solution is a super-graph of another solution. Such solutions are not considered, because they are more susceptible to transitive process failures and thus more at risk of data loss.

control schemes. This approach allows throughput to scale linearly with the number of processes in the ideal case. However, throughput is also very *sensitive* to conflicts. The reason is that the approach couples transaction execution with conflict resolution, which results in multi-phase protocols (atomic commit), high concurrency control overhead, and consequently low per-process parallelism and throughput.

Crescando is a scan-only engine. Both queries and updates are performed by parallel, heavily shared scans. While a Crescando instance can serve thousands of operations per second, individual operations have a turn-around time of one or more seconds. This design excels on workloads with an inherently large number of conflicts and need for full-table scans, such as the flight reservation workload of our use case (Sect. 2).

However, the "on-demand" approach to concurrency control followed by related systems is poised to fail in the context of full-table scans with many conflicts. So, instead, in Crescando/RB, we rely on E-Cast to deliver read and write messages uniformly, in causal total order. The individual storage processes then only have to execute operations in (the equivalent of) delivery order to make the resulting system sequentially consistent. The E-Cast protocol makes no attempt to detect conflicts and construct a minimal partial order. Instead, consistency is derived from the fact that all operations—conflicting or not—are brought into a *total* order. The downside of this approach is an upper limit on write throughput; the E-Cast protocol forms a so-called *consensus bottleneck*.

This design for the worst case wastes resources if the overwhelming majority of operations would not need to be ordered. But it is consistent with the design of the Crescando storage engine; in that, users are given strong, clear, and predictable performance guarantees. In fact, our experimental results show that Crescando/RB can reach 20,000 consistent reads and writes per second. This is 10× the load on the Wikipedia back-end [68], and about 20× the load of our heaviest use case. In short, for many applications, E-Cast is not a bottleneck.

That being said, there is a way to scale our approach with the number of processes for workloads that predominantly consist of single-key operations. The idea is to categorize reads and writes into single-key and multi-key operations. For sequential consistency, it is then sufficient to reach agreement on (i) a global execution order of all multi-key operations, (ii) one execution order for every set of single-key operations with the same key, and (iii) an interleaving of every individual single-key order with the global multi-key order.

Single-key operations need not go through E-Cast to achieve this, but can be sent directly to the affected replication groups, which order them locally. We propose a variant of chain replication for this [76]. The resulting system can be expected to scale linearly with the single-key load. Because performance of E-Cast is sufficient as-is, we have not yet implemented this extension, but a protocol sketch along with proofs of the central theorems is given in Unterbrunner [73].

## 6.7 Read isolation levels

Rubberband allows clients to trade read consistency for performance and scalability. This is useful; for example, for analytic queries which tolerate approximate answers. Such queries are common in our use case and other operational business intelligence workloads. We have currently implemented three levels of read isolation, which users can choose from for every individual read.

*Sequential Read* Every sequential read sees a globally consistent snapshot, which is more recent than any snapshot seen by any preceding sequential read by the same client, and reflects the effects of any preceding write by the client, but not the effects of any successive write by the client. This guarantees *sequential consistency* [33] with respect to sequential reads, hence the name.

*Snapshot Read* Snapshot reads are brought into total order with respect to writes. Thus, snapshot reads see a globally consistent snapshot of the data. No guarantees concerning the *specific* snapshot version with respect to operations submitted before and after are given though (i.e., no causal order).

*Basic Read* Basic reads scale linearly with the number of storage processes. However, *multi-key* basic reads may see partial effects of concurrent writes.

Sequential reads are sent through E-Cast like any other message. In contrast, snapshot reads use a protocol shortcut. Every snapshot read is sent to (what the client believes is) the current lead router, which timestamps and injects it into the sequence of writes it forwards, but *without* atomically broadcasting the read to other routers. Delivery is not guaranteed (no uniformity) in case if the router crashes or is not the lead router, but the timestamp is used to guarantee that a consistent snapshot is returned if successful.

Basic reads are sent directly to the destination processes, so there is no protocol bottleneck. Clients attach the configuration (version) they expect every destination to be in. When a process receives a basic read, it checks whether the configuration matches (i.e., there has been no concurrent reconfiguration). In this case, it executes the read; otherwise, it rejects it. Thus, any successful basic read is executed by a "correct" set of processes according to a single configuration. A multi-key basic read that is executed by multiple storage processes may see partial effects of concurrent writes, however, since individual storage processes may have executed slightly different prefixes of the global write sequence at the point where they execute the basic read.

## 7 Experimental evaluation

In this section, we provide comprehensive experimental evidence of the scalability, elasticity, and fault tolerance of Crescando/RB.

We decided not to include a comparison with a baseline system for the following reasons. Crescando/RB's support for arbitrary conjunctions of Boolean query and update predicates (specifically, *lack* of key predicates) sets it functionally apart from key-value stores and semi-relational data stores. These systems simply cannot execute the queries and updates we are interested in. Conversely, Crescando/RB, by design, cannot match the latency and extreme scalability of these systems when simple key-value access is sufficient. Relational database management systems on the other hand *can* execute the queries and updates that Crescando/RB was designed for. But, as we have shown before [74], performance is poor and highly unpredictable.

### 7.1 Platform

Except for some experiments on Amazon EC2, all experiments were conducted on a private cluster of 30 dual Intel Xeon L5520 2.26 GHz (i.e., $2 \times 4$-core) machines with 24 GB RAM, connected through 1 Gbit Ethernet. The machines were running Linux, kernel version 2.6.32. Crescando/RB was running on Erlang/OTP[6] R14B02. Crescando/RB and the OTP libraries were compiled to native code, using the High-Performance Erlang Compiler (HiPE) [50]. The Crescando storage engine, written in C++, was compiled into a stand-alone executable using GCC 4.6.1. Crescando/RB storage processes communicated with their Crescando engine instances via Unix pipes (called *ports* in Erlang parlance).

### 7.2 Workload and configuration

Storage processes were loaded with Amadeus Ticket data from our main use case (cf. Sect. 2). A Ticket table contains flight booking information, one record per passenger on a plane. Each record is about 350 Byte.

Unless otherwise indicated, experiments were run with 3 routers, 1 client, 1 super, and 16 storage processes. The data were hash partitioned by *rloc*, a 6-character alphanumeric string which uniquely identifies a travel booking (but there are generally multiple Ticket records associated with one *rloc*). For most of the experiments, each read or write message contained a relational query (or update, respectively) with a single equality predicate on *rloc*. Storage process identifiers were uniformly distributed over the *rloc* domain. Thus, for a replication factor $f$, each read was delivered to 1 storage process, and each write was delivered to $f$ storage processes. We also present results for range operations and range partitioning.

Note that whenever we write *transaction* in this section, we mean the system activity from the point where a client submits a read or write message, to the point where the client receives the confirmation that the operation has been executed by all affected (and non-failed) storage processes. This definition is not to be confused with ACID transactions in a relational database.

### 7.3 Write scalability

First, we investigated how write throughput changes with an increasing number of storage process. Each Crescando engine instance was running a single scan thread and was bulk-loaded with 2 GB of Ticket data. We measured write throughput for 1 to 32 storage processes and replication factors $f = 1$, 3, 5, and $n$, where $n$ is the number of storage processes. In the case of $f = n$ (full replication), every storage process had to execute every single write, so E-Cast had to atomically broadcast every write message. Figure 8 shows the results. Each data point was averaged over 5 runs of at least 3 min each. Error bars show the standard deviation.

For cases where $n \leq f$, the data and workload was fully replicated across all storage processes, so write throughput was limited to that of a single storage process: 5,500 TPS. For $n > f$, however, Rubberband was able to *partition* the workload over the storage processes (multicasting every write operation to exactly $f$ processes). Since every storage process contained a fixed amount of data, but had to handle a reciprocally decreasing amount of transactions, throughput increased linearly with the number of storage processes, up to the point where the E-Cast routers became a bottleneck. For $f = 3$, throughput plateaued at about 20,000 TPS and 12 storage processes.

The full-replication case ($f = n$) is equivalent to the case where write operations do not have predicates on the parti-

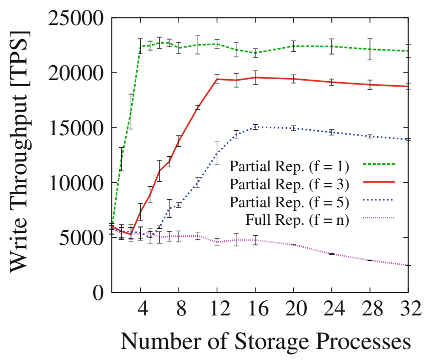


**Fig. 8** Write throughput: 3 router, 1 client, vary storage, vary rep. fact. $f$

[6] OTP is the reference collection of Erlang virtual machine, standard libraries, and tools.

tioning attribute, so that every storage process must execute every write. It shows the throughput of Crescando/RB for *global write transactions* under any replication factor. It also shows the expected behavior of any naïve, broadcast-based solution to replication (and thus the benefits of selective routing by E-Cast).

Global write throughput was hovering around 5,000 TPS between 1 and 16 storage processes. For 32 processes, it was still around 2,500 TPS. Note that in realistic workloads, the majority of writes have a key predicate and need not be broadcasted.

In summary, Crescando/RB easily meets our (write) throughput requirements. For reference, the live Crescando instance deployed by Amadeus currently sustains around 1,000 Write TPS. Should it ever become necessary, it is possible to scale throughput further, by treating single-key writes differently, see Sect. 6.6.

## 7.4 Read scalability

We repeated the previous scalability experiment for a replication factor $f = 3$, measuring read throughput (instead of write throughput) for each of the three isolation levels offered by Crescando/RB: sequential, snapshot, and basic read. In addition to hash partitioning the data by *rloc* (the default used throughout the experiments section), we also investigated random partitioning. Random partitioning represents the "worst case" for any workload, where no operation carries a predicate that matches the data partitioning.

### *7.4.1 Hash partitioning*

Crescando/RB performs read-one-write-all replication (ROWA). Consequently, for hash partitioning (Fig. 9, left chart), throughput increased linearly with the number of processes, up to the point where a protocol limit was hit. A single storage process could handle 4,500 to 5,500 read transactions per second (TPS). The system was able to scale up to 25,000 sequential read TPS or 110,000 snapshot read TPS. Basic read throughput scaled linearly without hitting any bottlenecks, since client processes send basic reads directly to the destination storage processes.

### *7.4.2 Random partitioning*

Under random partitioning, every read is necessarily *global* and has to be executed by $\lceil n/f \rceil$ storage processes, where $n$ is the number of processes. In an ideal system (with no networking cost), throughput would plateau at $f * t$ for $n \geq f$, where $t$ is the throughput per storage process. Figure 9 shows that Crescando/RB remains close to this upper limit (the dashed line) for basic and snapshot reads. For sequential reads, throughput begins to drop off noticeably for $n \geq 21$.

## 7.5 Range reads and range writes

A key feature of Crescando/RB is its support for range operations. Under range partitioning, Rubberband ensures that range operations are multicasted to a minimal set of storage processes. We verified this as follows.

We range partitioned the data by *dateOut* (departure date). To avoid load imbalance and keep the results interpretable, we replaced the original *dateOut* column with one that is uniformly distributed. In addition to an equality predicate on *rloc* (which is useless for purposes of restricting the destination set of messages), we gave every read or write operation a range predicate on *dateOut*. We then loaded 32 storage processes with 2 GB Ticket data each and measured throughput for a varying range predicate coverage $c$; that is, fraction of the *dateOut* domain covered by each predicate.

Figure 10 shows the results for range writes and range reads at all three isolation levels. For $c = 0$, each read is executed by exactly 1 storage process, and each write is executed by exactly $f = 3$ storage processes. Consequently, performance is similar to that of hash partitioning with matching equality predicates (Fig. 9, left chart). Conversely, for $c = 1$, each read is executed by $\lceil n/f \rceil = \lceil 32/3 \rceil = 11$ storage processes, and each write is executed by all $n = 32$ storage processes. Performance is similar to that of random partitioning (Fig. 9, right chart). For any $0 < c < 1$, Rubberband ensures that E-Cast routes each read or write message to the

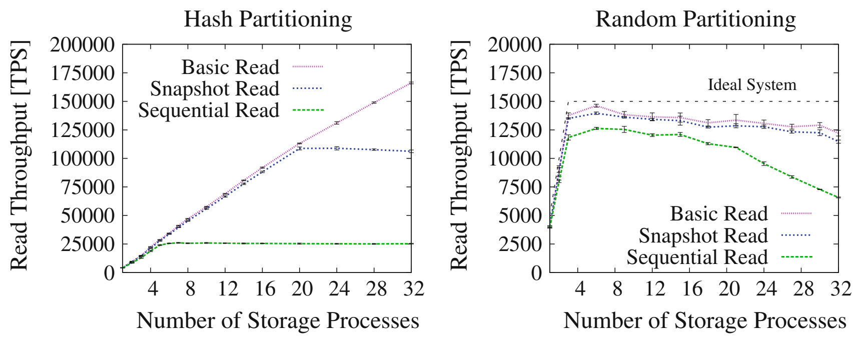


**Fig. 9** Read throughput: $f = 3$, 3 router, 16 client, vary storage, vary isolation, vary partitioning

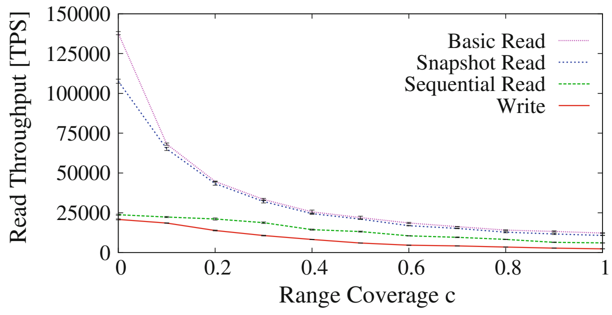


**Fig. 10** Read throughput: $f = 3$, 3 router, 1 client, 32 storage, vary range coverage $c$

minimal set of storage processes to cover the range predicate, and throughput indeed degrades gracefully.

Note that for real workloads, range predicates typically cover a small fraction of the domain. Thus, Crescando/RB can provide good performance in the common case, without falling off a performance cliff when operations occasionally have large range predicates or do not match the chosen partitioning at all.

## 7.6 Elasticity

Another key feature of Crescando/RB is its ability to scale elastically; that is, the ability to change the mapping of identifiers to storage processes (and thus the data placement) without loss of consistency or availability. To test this, we loaded a single Crescando/RB storage process with 16 GB of data. We then added another storage process every 5 min, always expanding the keyspace, up to a total of 16 processes, each running on a separate machine. The replication factor was set to $f = 3$. (Thus, for 16 storage processes, each process was holding roughly 3 GB data.) We then removed one storage process every 5 min, always contracting the keyspace, until the system was scaled back to just 1 storage process holding all the data.

### *7.6.1 Throughput*

The experiment was repeated four times while measuring throughput at a 1-s sampling interval; one iteration using a pure write workload, and one more iteration for each read isolation level. We used 16 client processes spread over 4 machines to make sure the system was saturated. Figure 11 shows the resulting traces.

The step shape of the traces is an artifact of (the lack of) load balancing. To keep the experiment simple, we had chosen not to rebalance the keyspace after each process addition or removal. Only in case of 1, 2, 4, 8, and 16 processes was the load perfectly balanced.

Throughput scaled elastically as storage processes were added or removed. For each type of workload or isolation level, the corresponding trace shows the same throughput limits reported previously. Note that in contrast to the read and write scalability experiments, each Crescando instance here was given all 8 cores of the respective machine to run on. Therefore, the throughput for 1 storage process was similar to the previous experiments, despite the larger data volume (16 GB on 8 cores as opposed to 2 GB on 1 core).

In all four traces, one can see a brief dip in throughput every 5 min, at the point where the keyspace was expanded or contracted. The longest dips appear where the system was scaled from 1 to 2 storage processes. At this point, the first storage process had to copy all 16 GB of data to the second process. Copying 16 GB required roughly 160 s (i.e., the 1 Gbit network was saturated). As more processes were added, less data had to be shuffled each time, and the dips in throughput became less pronounced.

### *7.6.2 Transaction duration*

We repeated the experiment a few more times, but measuring transaction duration ("latency") for a fixed background workload of 2,000 write TPS. Figures 12 and 13 show latency scatter-plots for writes and snapshot reads, respectively (sampling interval 10 s). In each figure, the left chart shows the range between 0 and 180 s of latency, while the right chart zooms in on the range between 0 and 4 s. The data underlying each pair of charts are identical. We performed the experiment also for sequential reads and basic reads, but the results did not differ much from those of snapshot reads. We thus omit those charts.

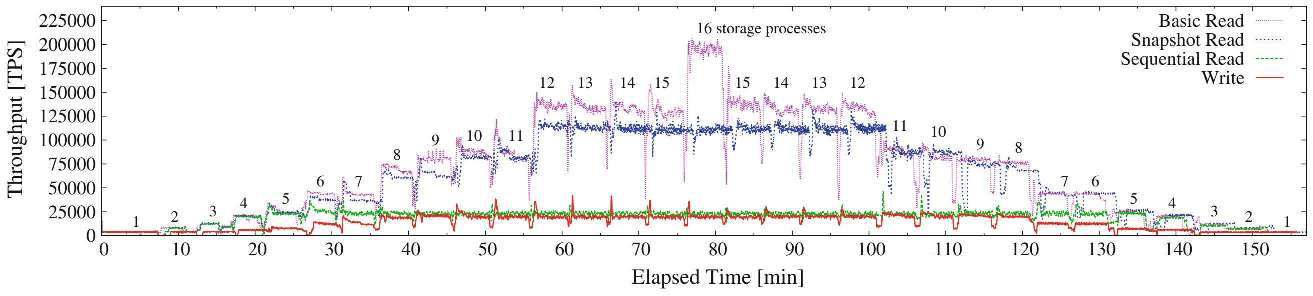


**Fig. 11** Throughput: $f = 3$, 3 router, 16 client, elastic storage, vary workload type

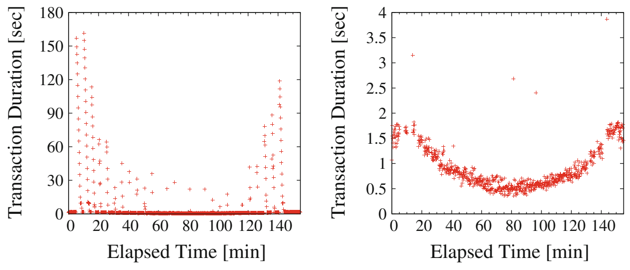


**Fig. 12** Write duration: $f = 3$, 2,000 background write TPS, 3 router, 1 background client, 1 latency client, elastic storage

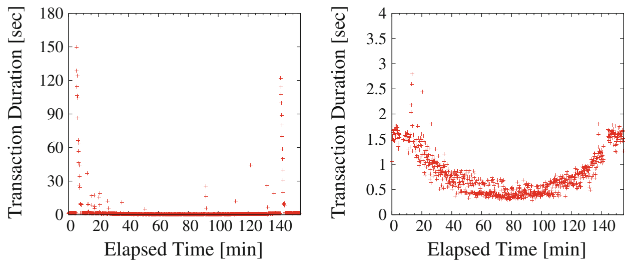


**Fig. 13** Snapshot read duration: $f = 3$, 2,000 background write TPS, 3 router, 1 background client, 1 latency client, elastic storage

The vast majority of transactions completed within 1.75 s for 1 storage process, down to 0.75 s for 16 storage processes. E-Cast needs only a few milliseconds to reliably deliver messages, see Sect. 7.9. Latency was thus dominated by the time required by Crescando to scan the data. Note that a throughput of 2,000 write TPS and a transaction duration of 0.75–1.75 s means that up to 3,500 write operations were being processed in parallel at any given moment.

During reconfiguration, some transactions were active for significantly longer periods of time (up to 160 s), because processes were busy with partial live migration. In the case of snapshot reads (or any other read), transactions were very rarely delayed. The reason is that for $f = 3$, there was almost always one replica that was not busy, and Rubberband automatically forwards reads to non-busy replicas whenever possible.

In all our experiments, we considered a transaction complete only after *every* affected process had executed it. Consequently, the measured write duration became very long during reconfiguration. But as explained in Sect. 6.4, it is quite irrelevant when the last storage process executes a given write. As soon as E-Cast acknowledges that a write message is stable (which requires just a few milliseconds), the write is guaranteed to be durable.

Moreover, the effects of a write can be seen much sooner than when the *last* storage process executes it. It is sufficient that *a single* (non-busy) storage process executes the write, since Rubberband forwards subsequent reads to non-busy processes. It follows that Crescando/RB is able to remain *fully available* during reconfiguration, provided a sufficient degree of replication.

### *7.6.3 Elasticity on EC2*

We repeated the elasticity experiments on 20 Amazon EC2 instances of type `cc1.4xlarge`. Each of these instances had 23 GB of memory, 2 Intel Xeon X5570 2.93 GHz, quad-core CPUs with hyper-threading, and a 10 Gbit Ethernet interface. 1 EC2 instance was running the client process, and 3 EC2 instances were each running 1 router process. The remaining 16 EC2 instances were each running 1 storage process.

The throughput trace did not differ much from that produced by our lab cluster, so we omit it here. Despite the higher clock rate of the EC2 machines (2.93 GHz as opposed to 2.26 GHz), throughput per storage process was identical to our lab cluster. This can be attributed to the overhead of virtualization on EC2. Peak system throughput on EC2 was lower, however, at 15,000 write TPS as opposed to 20,000 write TPS. Again, much of that may be due to virtualization, but part of it can also be attributed to the increased network latency and variance, which resulted in a larger number of in-transit messages, which in turn put slightly more load on the (compute-bound) router processes.

The latency trace (also omitted) was also very similar to that produced by our lab cluster. The additional network latency and variance on EC2 were unnoticeable, since transaction duration was dominated by Crescando's data scans. However, copying all 16 GB of data to a newly launched storage process required just over 100 s on EC2 as opposed to 160 s on our lab cluster. On EC2, Crescando engines were shuffling data at a rate of around 160 MB/s. The reason is that the EC2 instances had 10 Gbit network interfaces as opposed to our 1 Gbit lab network.

In summary, Crescando/RB shuffles data at a rate that saturates a 1 Gbit network. The system remains available during this period, assuming a sufficient replication factor. Throughput and latency promptly stabilize after each reconfiguration. The system also works well in a public cloud. Increased network latency and variance do not have a serious effect on performance.

## 7.7 Tolerance to storage process failures

Next, we investigated how Crescando/RB behaves in case of storage process failures. To this end, we started 6 storage processes and loaded them with 12 GB of data in total. The replication factor was again set to $f = 3$, so each storage process was holding about 6 GB of data.

We first measured how snapshot read throughput changed under different failure scenarios. As Fig. 14 shows, the system originally sustained just over 40,000 TPS. After

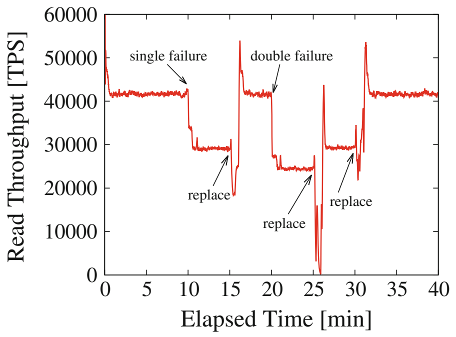


**Fig. 14** Snapshot read throughput: $f = 3$, 3 router, 4 client, 6 storage

10 min, one storage process was killed. Some partition of the keyspace was now covered by just 2 rather than 3 replicas. Consequently, throughput dropped to just under 30,000 TPS.

The failed storage process was replaced 5 min later. While the replacement storage process was receiving data, throughput dropped some more. However, the system remained available, and after just 1 min (required to copy 6 GB of data), throughput went back to the original 40,000 TPS.

Twenty minutes into the experiment, 2 storage processes were killed at the same time. Throughput dropped to about 25,000 TPS. After 25 mins, one of the failed processes was replaced. At that point, all storage processes of one of the replication groups had either failed or were busy with partial live migration. As a result, throughput briefly dropped to zero. After 1 min, one of the failed storage processes was successfully replaced, and throughput increased back to 30,000 TPS. Finally, after 30 min, the second failed process was replaced, and throughput returned to the original 40,000 TPS.

Note that throughput was briefly higher after each reconfiguration. This is because a number of pending operations had queued up at the previously busy storage processes. The corresponding Crescando engines could then operate at maximum efficiency while catching up with the rest of the system.

The experiment was repeated for a fixed background workload of 8,000 Write TPS, while snapshot read latency was being measured (5 s sampling interval). The results are shown in Fig. 15. As before, the right chart merely zooms in on the range between 0 and 4 s of latency. Both charts show the same data.

The vast majority of reads was completed within 1 s. Process failures had no visible effect on latency. The reason is that the write load per process does not change as long as the keyspace is not modified. The only latency spike at minutes 25 to 26 shows that part of the keyspace was briefly not available when the system was recovering from the double-process failure.

In summary, throughput degrades gracefully when storage processes fail. Crescando/RB remains available, even during process replacement, assuming a sufficient replication factor. After process replacement, the system quickly stabilizes, both in terms of throughput, and in terms of latency.

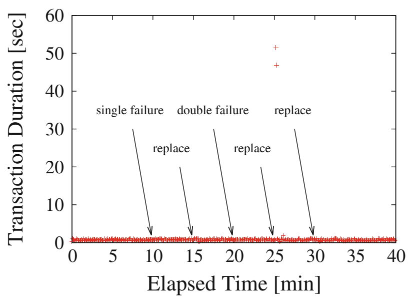
## 7.8 Tolerance to router process failures

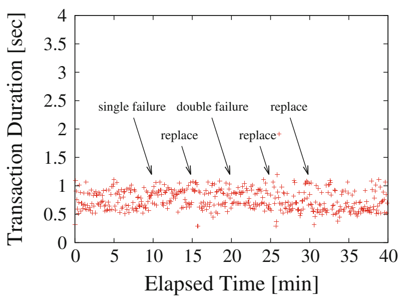
An important remaining question is how well E-Cast can tolerate router failures. Figure 16 shows a write throughput trace ($f = 3$) for various failure scenarios.

The trace starts with 3 router processes. After 30 s, the lead router was crashed. By design, all client processes talk to the lead router, so throughput immediately dropped to zero. The heartbeat protocol used by routers was configured to declare routers faulty after 3 s. Since there was uniform agreement on the sequence of messages, and every router at any point was ready to send those messages, there was no state to be transferred after the router crash. Accordingly, E-Cast elected a new lead router and resumed normal operation after exactly 3 s. Note that throughput after the failure was slightly higher than before. The reason is that router replication becomes cheaper for lower numbers of routers.

Sixty seconds into the experiment, the crashed router was replaced and throughput went back to the original number, and 90 s into the experiment, a follower router (i.e., not the leader) was crashed. Shortly thereafter, E-Cast stopped transmitting messages. This is because E-Cast ensures that no router falls behind too far in processing messages. After 3 s however, the routers reached agreement on the new membership, the failed router was removed from the system, and the remaining routers resumed normal operation, and 120 s into

**Fig. 15** Snapshot read duration: $f = 3$, 8,000 background write TPS, 3 router, 4 background client, 1 latency client, 6 storage





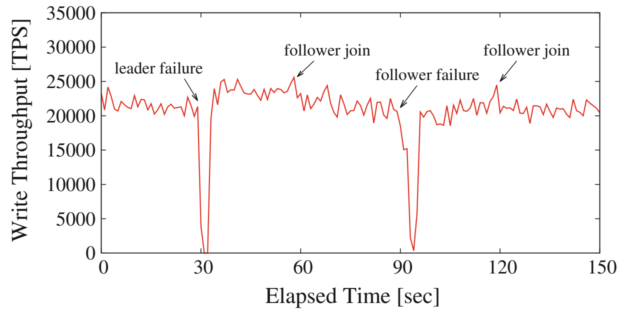


**Fig. 16** Write throughput: $f = 3$, 3 router, 1 client, 16 storage

the experiment, a third router joined again, and the system quickly stabilized at the original throughput.

In summary, E-Cast rapidly recovers from router failures, even at peak load. Together with the fact that E-Cast has higher throughput in "degraded" mode than under normal operation, this means that Crescando/RB will not become overloaded as a result of router failures.

### 7.9 Other results

The following is a list of other interesting results, which we present only in summarized form for space reasons.

Crescando/RB scales well with the number of clients, owing to E-Cast's built-in congestion control. Throughput remained unchanged for up to 1,024 clients, at which point it began to drop logarithmically. E-Cast throughput also remained unchanged for payload sizes up to 1,024 Byte. For larger payloads, the 1 Gbit network interface of the lead router became a throughput bottleneck. Since write messages in Crescando/RB are compact, relational updates, and read results are transmitted through direct connections, outside of E-Cast, bandwidth is not an issue in Crescando/RB, however.

In terms of protocol latency, we found that on average, E-Cast uniformly delivered and acknowledged messages within 2 ms at around 10,000 TPS. At peak throughput, latency reached over 30 ms, but it stayed below 10 ms for up to 95 % of peak load. Unlike most protocols with similarly strong ordering and delivery guarantees, E-Cast does not require write-ahead-logging on stable storage (disks). Consequently, the transmission delay of E-Cast is very low.

## 8 Conclusions

We have presented Crescando/RB, a scan-based data store, which combines strong consistency guarantees with high availability, elasticity, and very high throughput. Crescando/RB is the first data store, short of a full-blown relational database, with support for atomic queries and updates with arbitrary conjunctions of Boolean selection predicates (key or non-key).

The key to this combination of features lies in Crescando/RB's distribution and execution models. In the first stage of input processing, an efficient multicast protocol called E-Cast establishes uniform agreement on the input order. At some later point, in the second stage, the affected processes deterministically execute their (partition of the) input in the previously agreed order, using massively parallel, shared scans. Crescando/RB aggressively pipelines the two stages of processing, resulting in high throughput and continuous availability during process failures and large-scale system reconfigurations.

The presented system is fully functional. It addresses a large-scale industry use case in the context of the Amadeus travel reservation system, and we hope the techniques we have developed will enable and inspire many more exciting systems and services for the cloud.

**Acknowledgments** It requires the help of many people to build a real system. We specifically thank our student collaborators Khalid Ashmawy, Janick Bernet, Georgios Giannikis, Simon Loesing, and Ralph Steiner for their contributions. We would also like to thank our industry collaborators at Amadeus, Dietmar Fauser and Jeremy Meyer, who participated in the initial design of Crescando and have given us continuous feedback on the design of Crescando/RB. Their team at Amadeus has productized Crescando, and without the experience of Crescando in production, this work would not have been possible. This work has been funded in part by the Amadeus IT Group, as part of the Enterprise Computing Center of ETH Zurich (www.ecc.ethz.ch).

## 9 Appendix 1: Formalization

### 9.1 Stateful routing (SR)

#### *9.1.1 System model*

The system consists of a set of processes $P$, which are sequences of discrete events. Submitting or delivering a message is an event. Let $M$ be the universe of messages. Define $M^\Sigma \subseteq M$ as the set of submitted messages, and $M^\Delta \subseteq M$ as the set of delivered messages. Every process $p \in P$ submits a set of messages $M_p^\Sigma \subseteq M^\Sigma$ and delivers a set of messages $M_p^\Delta \subseteq M^\Delta$. By definition, a process $p$ is *faulty* iff $p$ crashes, or a process $p'$ successfully submits a message declaring $p$ faulty.

Every message $m \in M^\Sigma$ is submitted exactly once.[7] Let sub$(m)$ denote the submission event for $m \in M^\Sigma$, and Dlv$(m, p)$ denote the set[8] of delivery events for $p \in P$ and

[7] One can use unique message IDs to this effect.

[8] Dlv$(m, p)$ is a set, since an (incorrect) protocol may deliver a message $m$ multiple times to some process $p$.

$m \in M_p^{\Delta}$. The causal order relation $e \rightarrow e'$ over events $e, e'$ is defined as the minimal, transitive relation where both $e$ and $e'$ are events of some process $p$ and $e$ precedes $e'$ in the execution of $p$, or $e = \text{sub}(m)$ and $e' \in \text{Dlv}(m, p)$ for some $p \in P$ and some $m \in M$.

We refer to a sequence of messages as an *m-seq*. Let $\mathbb{Q}$ be the universe of possible m-seqs over $M$. An m-seq $Q \in \mathbb{Q}$ is a pair $Q = \langle M_Q, \rightarrowtail_Q \rangle$, where $M_Q \subseteq M$ is a set of messages, and $\rightarrowtail_Q$ is a strict total order relation over $M_Q$. A function prefix$(Q, m)$ is defined, where $Q$ is an m-seq, and $m$ is a message. If $m \in M_Q$, the function yields the prefix of $Q$ up to but excluding $m$. If $m \notin M_Q$, then prefix$(Q, m) = Q$.

A user-defined function destset$(Q, m)$ is assumed, which maps every pair of m-seq $Q$ and message $m$ to a finite set of processes destset$(Q, m) \subseteq P$ called the *destination set* of $m$ according to $Q$. We define the shorthand destsetpfx$(Q, m) \overset{\text{def}}{=} \text{destset}(\text{prefix}(Q, m), m)$

The user-defined destset function sets our model apart from related work in the area of group communication. This function allows one to express arbitrary data models and replication schemes without the use of explicit groups.

### 9.1.2 Predicates over message sequences

In order to give a succinct formal problem statement, we define the following predicates over m-seqs.

**valid(Q)** *Every message in $M_Q$ is submitted, and no message is delivered that is not in $M_Q$.*

**minimal(Q)** *Every process that delivers a message m is in the destination set of m according to Q.*

**complete(Q)** *For every $m \in M_Q$, every correct process in the destination set of m according to Q delivers m.*

**gap-free(Q)** *For every message $m \in M_Q$ submitted by a process p, every message $m'$ submitted previously by p is also in $M_Q$.*

**submit-ordered(Q)** *$\rightarrowtail_Q$ is an extension of the order relation $\rightarrow$ over the submission events of $M_Q \cap M^{\Sigma}$.*

**delivery-ordered(Q)** *$\rightarrowtail_Q$ is an extension of the order relation $\rightarrow$ over the delivery events of $M_Q \cap M^{\Delta}$.*

**terminated(Q)** *Every message m submitted by a correct process p is in $M_Q$.*

### 9.1.3 Problem statement

**Correct routing** *There is a valid, minimal, complete, gap-free, submit-ordered, delivery-ordered, terminated m-seq Q.*

A protocol is a *correct* solution for **Stateful Routing** iff it transfers messages between processes in a manner that Correct Routing holds. This requires that any message $m$ is delivered, in the right order, to the processes which the application considers destinations of $m$. The following (generalizations of) familiar properties follow directly from Correct Routing.

**Validity** *Every delivered message has been submitted.*

**Uniform causal order delivery** *If the submission of $m \in M^{\Sigma}$ causally precedes the submission of $m' \in M^{\Sigma}$ then no process $p \in P$ delivers $m'$ before m.*

**Uniform total order delivery** *For any pair of messages $m, m' \in M^{\Delta}$, if some process $p \in P$ delivers both m and $m'$, and delivers m before $m'$, then any process $p' \in P$ that delivers m and $m'$ delivers m before $m'$.*

**Uniform agreement** *Any message m delivered by any process is delivered by every correct process in the destination set of m, in accordance with some m-seq Q.*

**Termination** *Any message m submitted by a correct process is delivered to every correct process in the destination set of m, in accordance with some m-seq Q.*

Uniform agreement states that if any process $p$ delivers a message $m$, every process (that is supposed to) will eventually deliver $m$, *even if* process $p$ and/or the submitter of $m$ fail immediately after $p$ delivers $m$. The combination of uniform total order delivery, uniform causal order delivery, and uniform agreement yields sequential consistency [33], the distributed system equivalent of linearizability [47].

If destset$(Q, m) \overset{\text{def}}{=} P$ for any $Q$ and $m$, then the problem is identical to uniform atomic broadcast with causal order delivery [28]. Stateful Routing is thus a proper generalization of uniform atomic broadcast.

## 9.2 Dynamic partial replication as stateful routing

We show how an algorithm that solves Stateful Routing (SR) can be used as a solution for dynamic partial replication (DPR). Let $W$ be the universe of writes. Let every client process $c \in C$ submit writes $W_c$ in the form of messages. Let every client process submit each write exactly once. Let every storage process $s \in S$ execute delivered writes $W_s$ exactly once, in delivery order. Then, the global execution order corresponds to the global delivery order.

Let us further refer to the set of data objects stored by a storage process $s$ as a table $T_s$. We write $T_s^0$ to refer to the initial table of $s$ (before $s$ executes any writes). We write $T_s^Q$ to refer to the table of $s$ after the sequence of writes (m-seq of messages) $Q$ has been applied to $T_s^0$.

A *correct* solution for **Dynamic Partial Replication** executes writes in a manner that the following guarantees hold.

**Integrity** *No storage process s ever holds a table that cannot be reached by applying some sequence consisting only of submitted writes to* $T_s^0$.

**Strong consistency** *An order relation over* $W$*, call it* $<$*, exists, which is an extension of the write submission order of every client process, and the execution order of every storage process.*

The $<$ relation is exactly the serialization order discussed in Sect. 5.1. For practical reasons, we further require that the serialization order is compatible with the write order of every client.

**Atomicity** *If a storage process* $s \in S$ *executes a write* $w \in W$*, at some point the table of every correct storage process* $s' \in S$ *is in a state* $T_s^Q$ *that is reached by applying a sequence of writes* $Q$ *which includes* $w$.

A process is defined *correct* iff it is not faulty. In this work, we only consider crash failures (no Byzantine failures), but the problem statement is independent.

**Durability** *For any write* $w \in W_c$ *of a correct client process* $c \in C$*, at some point the table of every correct storage process* $s \in S$ *is in a state* $T_s^Q$ *that is reached by applying a sequence of writes* $Q$ *which includes* $w$.

If every write $w$ is delivered to every correct storage process $s$, then Atomicity and Durability follow immediately from Correct Routing. However, this *broadcast*-based solution does not scale with the number of storage processes. Instead, we want every write to be delivered only to those processes whose table is actually affected by the write. Let us write $Q \circ w$ to express the extension of write sequence $Q$ by $w$. The following is a formal definition of a *destset* function that avoids broadcast.

**Definition 1** (*perfect destset*) destset$(Q, w)$ $\stackrel{\text{def}}{=} \left\{ s \in S \mid T_s^Q \neq T_s^{Q \circ w} \right\}$

We call the above destset function *perfect* in that it routes writes *exactly* to the set of storage processes whose tables are affected. While appealing in theory, a perfect destset function is impractical. Fortunately, perfect destset functions are not required. A destset function may return more storage processes than affected, since writes to these additional storage process have no effect (other than overhead). We call a destset function *correct* iff it always returns some superset of storage processes of a perfect destset function.

In an implementation of a correct destset function, there is thus a trade-off between the amount of routing state and the number of writes that are delivered unnecessarily. As a concrete example, Rubberband routers map a *range* of data object keys to each storage process (range partitioning). The destset function of Rubberband routes each write to exactly those storage processes whose key range overlaps with the key range(s) of the write, see Algorithm 3. This avoids broadcast of messages in most cases, but obviously, a specific write may not have an effect if no object for the given key exists.

**Theorem 1** *Under the described reduction, a solution for Stateful Routing (SR) is a solution for Dynamic Partial Replication (DPR).*

*Proof* Correct Routing guarantees the existence of an m-seq $Q = \langle W, < \rangle$. Validity implies that $\forall s \in S : W_s \subseteq \bigcup_{c \in C}(W_c)$. Integrity follows immediately. Strong Consistency follows from Uniform Total Order Delivery and Uniform Causal Order Delivery. By definition of destset, Atomicity follows from Uniform Agreement, and Durability follows from Termination. □

SR in fact makes stronger guarantees than required for DPR. For instance, DPR only requires total FIFO order delivery, while SR guarantees total causal order delivery. Also, SR guarantees that m-seqs are gap-free and complete, which together is a stronger uniformity guarantee than required by DPR. DPR "only" requires that the effects of a given update are applied to all or none of the processes' tables. But SR additionally guarantees that for any write that is applied, every write *previously submitted* by the same client is also applied.

These guarantees may not seem very interesting from a formal point of view, but they are very useful in practice. For instance, gap-freeness allows applications to pipeline a large number of writes in order to maximize throughput over high latency network links, while still, in the event a client process crashes, having the guarantee that some prefix (and not an arbitrary subset) of the pending writes will be applied.

## 10 Appendix 2: Proof of correctness of E-cast

Our goal is to prove the following theorem:

**Theorem 2** *E-Cast is a correct solution for Stateful Routing.*

Router processes propose and learn messages via uniform atomic broadcast with FIFO delivery (*ufabcast*). It is easy to see that by definition of uniform atomic broadcast [28], all router processes maintain agreement on an ever-growing m-seq $Q$. Our implementation of ufabcast (cf. Sect. 5.6) upholds that guarantee even when routers join or leave the system.

To prove that E-Cast solves Stateful Routing, we need to prove that Correct Routing holds. That is, we prove that the m-seq $Q$ that the dynamic set of routers agree on is valid, minimal, complete, gap-free, submit-ordered, delivery-ordered, and terminated. In what follows, we write [[AX]] to refer to line X of the application process algorithm, and [[RY]] to refer to line Y of the router process algorithm, respectively.

**Lemma 1** *$Q$ is valid.*

*Proof* To appear in $Q$, a message $m$ must be learned by some router $r$ [[R13]]. To be learned by $r$, $m$ must previously be proposed by some router $r'$ [[R8]]. For $m$ to be proposed by $r'$, $r'$ must previously receive a message $\langle$"route," $m\rangle$ [[R6]]. Assuming no Byzantine failures, an application process $a$ must previously send the message $\langle$"route," $m\rangle$ to $r'$ [[A18]], which in turn implies that $m$ must previously be submitted [[A5]]. □

**Lemma 2** *$Q$ is minimal.*

*Proof* No application process ever delivers a message $m$ for which it did not receive a $\langle$"deliver," $m, t\rangle$ message [[A10]]. No router process ever sends a message $\langle$"deliver," $m, t\rangle$ to any application process that is not in $\text{destsetpfx}(Q, m)$ [[R19]], [[R22]]. Minimality of $Q$ follows immediately. □

**Lemma 3** *$Q$ is submit-ordered.*

*Proof* It is to show that for any pair of messages $m, m' \in M_Q$ where $\text{sub}(m) \rightarrow \text{sub}(m')$, it holds that $m \rightarrowtail_Q m'$. We first prove that this holds if a single application process submits both $m$ and $m'$ (Case 1, FIFO Order). We then prove the general case (Case 2, Causal Order).

Case 1: Assume $m, m' \in M_a^\Sigma$ for some application process $a$. For the message $m'$ to be learned by any router process [[R13]], it must first have been proposed by some router process [[R8]]. Let $r$ be the first router process to propose $m'$ (according to the order enforced by atomic broadcast). To propose $m'$, $r$ must first have received $\langle$"route," $m'\rangle$ from $a$ [[R6]]. At that point, either $m$ was already learned or not.

If $m$ was already learned, then $m$ must be in $Q$ at that point, so clearly $m \rightarrowtail_Q m'$. If $m$ was not already learned, then $m$ can also not have been acknowledged [[R10]]. Since application processes send messages in submit order [[A6]], [[A18]]; and since processes communicate through quasi-reliable FIFO channels; $r$ must have received $\langle$"route," $m\rangle$ before it received $\langle$"route," $m'\rangle$ [[R6]]. Therefore, $r$ must have proposed $m$ before $m'$ [[R8]]. Since the atomic broadcast protocol guarantees FIFO order delivery, every router must learn $m$ before $m'$ [[R13]]. It follows that $m \rightarrowtail_Q m'$.

Case 2: Assume $m$ and $m'$ were submitted by two different application processes. Without restriction of generality, say $m$ was submitted by application process $a$, and $m'$ was submitted by application process $a'$. For $\text{sub}(m) \rightarrow \text{sub}(m')$ to hold, there must be a message $m''$ submitted by $a$, such that $\text{sub}(m) \rightarrow \text{sub}(m'')$ or $m = m''$, and $m''$ is delivered to at least one application process. Otherwise there could be no chain of causality between $\text{sub}(m)$ and $\text{sub}(m')$. But to be delivered to any application process [[A13]], $m''$ must first be sent by some router [[R22]], which implies that $m''$ must first be learned by some router [[R13]]. Therefore, $m''$ must precede $m'$ in $Q$ [[R14]]; that is, $m'' \rightarrowtail_Q m'$. If $m = m''$, then obviously $m \rightarrowtail_Q m'$. If $m \neq m''$, then Case 1 applies and $m \rightarrowtail_Q m''$. By transitivity of $\rightarrowtail_Q$, it holds that $m \rightarrowtail_Q m'$.

In both Case 1 and Case 2, $m \rightarrowtail_Q m'$. □

**Lemma 4** *$Q$ is delivery-ordered.*

*Proof* It is to show that for any pair of messages $m, m' \in M_Q$, where the delivery of $m$, call the event $e$, precedes the delivery of $m'$, call the event $e'$; i.e., $e \rightarrow e'$, it holds that $m \rightarrowtail_Q m'$. We first prove that this holds for any single application process that delivers both $m$ and $m'$ (Case 1). We then prove the general case (Case 2).

Case 1: The timestamp $t$ of any message $\langle$"deliver," $m, t\rangle$, sent by any router process, is the position of $m$ in $Q$ [[R20]]. Application processes deliver messages strictly in timestamp order [[A11]]. It follows that for any application process $a$, for any pair of messages $m, m' \in M_a^\Delta$ where $a$ delivers $m$ before $m'$, it holds that $m \rightarrowtail_Q m'$.

Case 2: For $e \rightarrow e'$ to hold, there must be a message $m''$ submitted by $a$, such that $e \rightarrow \text{sub}(m'') \rightarrow e'$. By analogous argument to the proof of submit-orderedness, it must hold that $m \rightarrowtail_Q m''$, and $m'' \rightarrowtail_Q m'$. By transitivity of $\rightarrowtail_Q$, it holds that $m \rightarrowtail_Q m'$. □

**Lemma 5** *$Q$ is gap-free.*

*Proof* Choose any pair of messages $m, m'$ submitted by an application process $a$, where $\text{sub}(m) \rightarrow \text{sub}(m')$, and $m'$ is in $M_Q$. It is to show that $m$ is also in $M_Q$. We assume that $m$ is not in $M_Q$ and prove by contradiction.

Since $m$ is not in $M_Q$, no router could have sent a "deliver"-message for it [[R22]], and $m$ could not have been acknowledged [[A9]]. Application processes send "route"-messages in submission order [[A6]], [[A18]]. Thus, for any router $r$ to which $a$ sends a message $\langle$"route," $m'\rangle$, $a$ must first send a message $\langle$"route," $m\rangle$.

Processes communicate through quasi-reliable FIFO channels. Thus, any router that receives a message $\langle$"route," $m'\rangle$ must first receive a message $\langle$"route," $m\rangle$ [[R10]]. Since $m$ is not in $M_Q$ and thus obviously not stable, any router process that receives and proposes $m'$ must also, previously, receive and propose $m$ [[R8]]. Since the atomic broadcast algorithm preserves FIFO order, for $m'$ to be learned, $m$ must be learned first [[R13]] and appended to $Q$ [[R14]]. Since $m'$ is in $M_Q$, $m$ must be in $M_Q$ as well; a contradiction. □

**Lemma 6** *$Q$ is complete.*

*Proof* We prove by contradiction. Assume $Q$ is *not* complete; i.e., there is some message $m \in M_Q$ that is not delivered by some correct application process $a \in \text{destsetpfx}(Q, m)$.

Assuming at least one correct router process that eventually considers itself a leader, every message $m \in M_Q$ is eventually sent to and received by every correct application process, including $a$ [[R22]]. To be precise, $a$ must receive a message ⟨"deliver," $m, t$⟩ [[A10]]. Consider the first time $a$ receives this message. For $a$ not to deliver $m$, $a$ must have previously received another message ⟨"deliver," $m', t'$⟩ where $t' > t$ [[A11]].

Router processes send "deliver"-messages in order of $Q$, that is timestamp order [[R20]], [[R22]]. Since $a$ had not received and confirmed $m$ yet [[A14]], and $a$ is a correct process, the router process that sent ⟨"deliver," $m', t'$⟩ could not have removed $a$ from the pending destinations of $m$ [[R11]], [[R12]]. Thus, $r$ must previously have sent ⟨"deliver," $m, t$⟩ to $a$ as well. Processes communicate through quasi-reliable FIFO channels, so $a$ cannot have received ⟨"deliver," $m', t'$⟩ before ⟨"deliver," $m, t$⟩. A contradiction. □

Note that as a generalization of Lemma 6, E-Cast even guarantees that *faulty* application processes see some prefix of $Q$, i.e. gap-free delivery to faulty processes. This guarantee is useful in practice, because it means that processes that are incorrectly suspected by others will not see an inconsistent system state.

This guarantee holds, because for every pair of messages $m, m' \in M_p^{\Delta}$ for some process $p$ (faulty or correct), where $m$ precedes $m'$ in $Q$, any router $r$ will always send ⟨"deliver," $m, t$⟩ to $p$ before sending ⟨"deliver," $m', t'$⟩. This is because the set of suspected processes can only grow, not shrink. If $p$ becomes a suspect in the destination set of $m$, it certainly becomes a suspect in the destination set of $m'$. Thus, if a router stops sending ⟨"deliver," $m, t$⟩ to $p$ even though $m$ is unconfirmed, it also stops sending ⟨"deliver," $m', t'$⟩ to $p$.

**Lemma 7** *$Q$ is terminated.*

*Proof* Any correct application process $p$ will repeatedly send any submitted message $m \in M_p^{\Sigma}$ to some router until $m$ is known to be stable and is acknowledged to the user [[A18]], [[A8]]. Assuming $p$ eventually sends $m$ to a correct router $r$, $m$ is eventually received by $r$ [[R6]]. Assuming liveness of the atomic broadcast algorithm, $m$ is eventually proposed by $r$ [[R8]] and learned by every correct router process [[R13]]. Thus, $m$ is in $Q$. □

*Proof* Theorem 2 follows by definition from Lemmas 1–7.

## References

1. Aguilera, M.K., Chen, W., Toueg, S.: Heartbeat: a timeout-free failure detector for quiescent reliable communication. In: Proceedings of WDAG '97, pp. 126–140 (1997)
2. Aguilera, M.K., et al.: Reconfiguring replicated atomic storage: a tutorial. Bull. EATCS **102**, 84–108 (2010)
3. Amir, Y., Danilov, C., Stanton, J.: A low latency, loss tolerant architecture and protocol for wide area group communication. In: Proceedings of ICDSN '00, pp. 327–336 (2000)
4. Amir, Y., et al.: The totem single-ring ordering and membership protocol. ACM Trans. Comput. Syst. **13**, 311–342 (1995)
5. Armstrong, J.: Erlang. CACM **53**, 68–75 (2010)
6. Arumugam, S., et al.: The datapath system: a data-centric analytic processing engine for large data warehouses. In: Proceedings of SIGMOD '10, pp. 519–530 (2010)
7. Ashmawy, K.: Partial live migration in scan-based database systems. Master's thesis, ETH Zurich (2011)
8. Baker, J., et al.: Megastore: providing scalable, highly available storage for interactive services. In: Proceedings of CIDR '11, pp. 223–234 (2011)
9. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, MA (1987)
10. Birman, K.: A history of the virtual synchrony replication model. In: Charron-Bost, B., Pedone, F., Schiper, A. (eds.) Replication: Theory and Practice, pp. 91–120. Springer, Berlin (2010)
11. Birman, K.P., et al.: Overcoming cap with consistent soft-state replication. IEEE Comput. Mag. **12**, 50–58 (2012)
12. Birman, K.P., Renesse, R.V.: Reliable distributed computing with the ISIS toolkit. IEEE Computer Society Press, Silver Spring, MD (1994)
13. Brewer, E.A.: Towards robust distributed systems. In: Proceedings of PODC '00, p. 7 (2000)
14. Candea, G., Polyzotis, N., Vingralek, R.: A scalable, predictable join operator for highly concurrent data warehouses. PVLDB **2**(1), 277–288 (2009)
15. Cattell, R.: Scalable sql and nosql data stores. SIGMOD Rec. **39**(4), 12–27 (2010)
16. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proceedings of PODC '07, pp. 398–407 (2007)
17. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. JACM **43**(2), 225–267 (1996)
18. Chandrasekaran, S. et al.: Telegraphcq: an architectural status report. IEEE Data Eng. Bull. **26**(1), 11–18 (2003)
19. Chandrasekaran, S., Franklin, M.J.: Streaming queries over streaming data. In Proceedings of VLDB '02, pp. 203–214 (2002)
20. Chang, F., et al.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**, 4:1–4:26 (2008)
21. Chockler, G.V., et al.: Reconfigurable distributed storage for dynamic networks. J. Parallel Distrib. Comput. **69**(1), 100–116 (2009)
22. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33**(4), 1–43 (2001)
23. Cooper, B.F., et al.: Pnuts: Yahoo!'s hosted data serving platform. PVLDB **1**(1), 1277–1288 (2008)
24. Corbett, J.C., et al.: Spanner: Google's globally-distributed database. In: Proceedings of OSDI '12, pp. 251–264 (2012)
25. Das, S., Agrawal, D., Abbadi, A.E.: Elastras: an elastic transactional data store in the cloud. In: Proceedings of HotCloud'09 (2009)
26. Das, S., Agrawal, D., Abbadi, A.E.: G-store: a scalable data store for transactional multi key access in the cloud. In: Proc. SoCC '10, pp. 163–174 (2010)
27. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. In: Proc. SOSP '07, pp. 205–220 (2007)
28. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: taxonomy and survey. ACM Comput. Surv. **36**(4), 372–421 (2004)
29. Demers, A., et al.: Epidemic algorithms for replicated database maintenance. In: Proc. PODC '87, pp. 1–12 (1987)
30. Dept, B.B., Ban, B.: Adding group communication to Java in a non-intrusive way using the ensemble toolkit. Technical report, Cornell University (1997)

31. Dolev, D., Malki, D.: The transis approach to high availability cluster communication. CACM **39**, 64–70 (1996)
32. Fabret, F., et al.: Filtering algorithms and implementation for very fast publish/subscribe systems. In: Proceedings of SIGMOD '01, pp. 115–126 (2001)
33. Fekete, A., Ramamritham, K.: Consistency models for replicated data. In: Charron-Bost, B., Pedone, F., Schiper, A. (eds.) Replication: Theory and Practice. Springer, Berlin, pp. 1–17 (2010)
34. Fernandez, P.M.: Red brick warehouse: a read-mostly rdbms for open smp platforms. In: Proceedings of SIGMOD '94, p. 492 (1994)
35. Fitzpatrick, B.: Distributed caching with memcached. Linux J. **124**, 5 (2004)
36. Giannikis, G., Alonso, G., Kossmann, D.: Shareddb: killing one thousand queries with one stone. PVLDB **5**(6), 526–537 (2012)
37. Giannikis, G., Makreshanski, D., Alonso, G., Kossmann, D.: Workload optimization using shareddb. In: Proceedings of SIGMOD '13, pp. 1045–1048 (2013)
38. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002)
39. Glendenning, et al.: Scalable consistency in scatter. In: Proceedings of SOSP '11, pp. 15–28 (2011)
40. Graefe, G.: Volcano—an extensible and parallel query evaluation system. IEEE Trans. Knowl. Data Eng. **6**(1), 120–135 (1994)
41. Gray J., et al.: The dangers of replication and a solution. In: Proceedings of SIGMOD '96, pp. 173–182 (1996)
42. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006)
43. Guerraoui, R., Schiper, A.: The generic consensus service. IEEE Trans. Softw. Eng. **27**(1), 29–41 (2001)
44. Habeeb, M.: A Developer's Guide to Amazon SimpleDB. Addison-Wesley Professional, Reading, MA (2010)
45. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University (1994)
46. Harizopoulos, S., Shkapenyuk, V., Ailamaki, A.: Qpipe: a simultaneously pipelined relational query engine. In: Proceedings of SIGMOD '05, pp. 383–394 (2005)
47. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
48. Hildenbrand, S.: Scaling out column stores: data, queries, and transactions. Ph.D. thesis, Diss. 20314, ETH Zurich (2012)
49. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of USENIXATC '10 (2010)
50. Johansson, E., et al.: The development of the hipe system: design and experience report. Int. J. Softw. Tools Tech. Transf. **4**(4), 421–436 (2003)
51. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: Proceedings of DISC '09, pp. 245–256 (2009)
52. Ktari, S., Zoubert, M., Hecker, A., Labiod, H.: Performance evaluation of replication strategies in dhts under churn. In: Proceedings of MUM '07, pp. 90–97 (2007)
53. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**, 35–40 (2010)
54. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
55. Lamport, L., Malkhi, D., Zhou, L.: Stoppable paxos. Technical report, Microsoft Research (2008)
56. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News **41**(1), 63–73 (2010)
57. Lang, C.A., et al.: Increasing buffer-locality for multiple relational table scans through grouping and throttling. In: Proceedings of ICDE '07, pp. 1136–1145 (2007)
58. Lynch, N., Shvartsman, A.A.: Rambo: a reconfigurable atomic memory service for dynamic networks. In: Proceedings of DISC '02, pp. 173–190 (2002)
59. Ostrowski, K., Birman, K., Dolev, D.: Quicksilver scalable multicast (qsm). In: Proceedings of NCA '08, pp. 9–18 (2008)
60. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of OSDI '10, pp. 1–15 (2010)
61. Plattner, H.: A common database approach for oltp and olap using an in-memory column database. In: Proceedings of SIGMOD '09, pp. 1–2 (2009)
62. Prisco, R.D., Lampson, B.: Revisiting the paxos algorithm. In: Proceedings of WDAG '97, pp. 111–125 (1997)
63. Qiao, L.: Main-memory scan sharing for multi-core cpus. PVLDB **1**(1), 610–621 (2008)
64. Raman, V., et al.: Constant-time query processing. In: Proceedings of ICDE '08, pp. 60–69 (2008)
65. Rao, J., Shekita, E.J., Tata, S.: Using paxos to build a scalable, consistent, and highly available datastore. PVLDB **4**, 243–254 (2011)
66. Schiper, N., Sutra, P., Pedone, F.: P-store: genuine partial replication in wide area networks. In: Proceedings of SRDS '10, pp. 214–224 (2010)
67. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. **22**(4), 299–319 (1990)
68. Schütt, T., Schintke, F., Reinefeld, A.: Scalaris: reliable transactional p2p key/value store. In: Proceedings of ERLANG '08, pp. 41–48 (2008)
69. Sellis, T.K.: Multiple-query optimization. ACM Trans. Database Syst. **13**(1), 23–52 (1988)
70. Stoica, I., et al.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of SIGCOMM '01, pp. 149–160 (2001)
71. Thomson, A., Abadi, D.J.: The case for determinism in database systems. PVLDB **3**(1–2), 70–80 (2010)
72. Thomson, A., et al.: Calvin: fast distributed transactions for partitioned database systems. In: Proceedings of SIGMOD '12, pp. 1–12 (2012)
73. Unterbrunner, P.: Elastic, reliable, and robust storage and query processing with Crescando/RB. Ph.D. thesis, Diss. 20272, ETH Zurich (2012)
74. Unterbrunner, P., Giannikis, G., Alonso, G., Fauser, D., Kossmann, D.: Predictable performance for unpredictable workloads. PVLDB **2**(1), 706–717 (2009)
75. van Renesse, R., et al.: Horus: a flexible group communication system. CACM **39**, 76–83 (1996)
76. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of OSDI '04, pp. 91–104 (2004)
77. Vo, H.T., Chen, C., Ooi, B.C.: Towards elastic transactional cloud storage with range query support. PVLDB **3**, 506–514 (2010)
78. Wei, Z., Pierre, G., Chi, C.-H.: CloudTPS: scalable transactions for web applications in the cloud. IEEE Trans. Serv. Comput. **5**(4), 525–539 (2012)
79. Zukowski, M., et al.: Cooperative scans: dynamic bandwidth sharing in a dbms. In: Proceedings of VLDB '07, pp. 723–734 (2007)