Squirrel: An Extensible Distributed Key-Value Store

Kevin De Porre  
Vrije Universiteit Brussel  
Brussel, Belgium  
kdeporre@vub.be

Elisa Gonzalez Boix  
Vrije Universiteit Brussel  
Brussel, Belgium  
egonzale@vub.be

Abstract

Distributed key-value (KV) stores are a rising alternative to traditional relational databases since they provide a flexible yet simple data model. Recent KV stores use eventual consistency to ensure fast reads and writes as well as high availability. Support for eventual consistency is however still very limited as typically only a handful of replicated data types are provided. Moreover, modern applications maintain various types of data, some of which require strong consistency whereas other require high availability. Implementing such applications remains cumbersome due to the lack of support for data consistency in today’s KV stores. In this paper we propose Squirrel, an open implementation of an in-memory distributed KV store. The core idea is to reify distribution through consistency models and protocols. We implement two families of consistency models (strong consistency and strong eventual consistency) and several consistency protocols, including two-phase commit and CRDTs.

CCS Concepts: • Information systems → Key-value stores; Data replication tools; Distributed database transactions.

Keywords: distributed key-value stores, replication, data consistency, open implementation

ACM Reference Format:

1 Introduction

Relational databases are widely used to store and query structured data. NoSQL databases emerged as they fit semi-structured and unstructured data. Key-value (KV) stores are a type of NoSQL database that store data as key-value pairs in an associative array. Traditional KV stores favoured consistency over availability, a consequence of the CAP theorem [4]. Since the appearance of Dynamo [8], most stores nowadays focus on availability. However, modern applications no longer exclusively reside on one end of the AP/CP spectrum [5]. Instead, applications are a mix of strongly consistent parts and highly available parts. Some systems may even benefit from a dynamic strategy, switching between consistency models at runtime [9].

Current KV stores do not provide appropriate support for mixed-consistency applications. They typically support only one or two consistency models and a few replicated data types. For instance, Riak, a popular KV store, features high availability but supports only 6 replicated data types 1. This is problematic because apart from embedding objects within maps, replicated data types cannot be composed to build custom ones such as balanced trees.

In this paper, we propose an in-memory distributed KV store that can be customized to replicate objects under different consistency models. Our concrete implementation in Scala, called Squirrel, provides an interface containing the traditional put, get, and delete operations. We showcase how to extend Squirrel with custom consistency models by adding support for highly available objects and distributed transactions. We believe that an open distributed KV store is key to enable mixed-consistency applications since it provides the hooks that are needed to plug in new consistency models and protocols.

2 An Open Distributed KV Store

Figure 1 depicts the architecture of our open implementation of a distributed KV store. Programmers build their applications on top of the store which provides a simple API to store

---

1https://docs.riak.com/riak/kv/2.2.3/developing/data-types.1.html
2.1 Squirrel: An Open Distributed KV Store in Scala

We now describe Squirrel, a concrete implementation of an extensible in-memory distributed KV store that leverages the extensibility of traits and type members in Scala [15]. Squirrel is built around replication models and protocols, which are depicted by the UML class diagrams in Figures 2 and 3. Custom replication models implement the abstract fields and methods (italic font in the class diagram) defined by the ReplicationModel trait, using the communication primitives provided by the Communication trait.

As shown in Figures 2 and 3, Squirrel supports an arbitrary number of replication models and protocols. Replication models are actors that implement the basic store functionality (get, put, and delete) described by the ReplicationModel trait from Listing 1, which corresponds to the abstract ReplicationModel class from Figure 2. All operations take a key followed by an arbitrary number of arguments, allowing concrete models to take additional arguments if needed. Note that keys encode the type of the value. This avoids collisions when objects of different types are added concurrently with the same name. Collisions between objects of the same type are solved by merging them.

Squirrel thus acts as a single KV store but is in fact a collection of stores (one per model). Each model maintains its own store, which serves as a container for objects shared under that model, thereby avoiding name clashes between objects shared under the same key but different models. As a result, programmers have to specify both a namespace (identifying the store) and a key (identifying the object) when reading or writing objects.

```
1 trait ReplicationModel extends Actor
2 with Communication {
3   type T
4   val model: Namespace
5   def get(key: Key[T], extra: Any*): Option[T]
6   def put(key: Key[T], value: T, extra: Any*)
7   def delete(key: Key[T], extra: Any*)
8 }
```

Listing 1. Interface for replication models.

2.2 Setting up a KV Store in Squirrel

Listing 2 shows how to set up a distributed KV store and add a key-value pair under a certain consistency model. First, we use Akka to set up a regular actor system and configure a custom cluster comprising several nodes each running an instance of Squirrel (Line 1). This implies that a Squirrel distributed KV store crosses machine and network boundaries since objects can be replicated across machines spanning several networks. Then on Line 3, we make a new Squirrel instance, passing along our actor system. Finally, on Lines 4 to 7 we fetch a reference to the store actor and add a Counter

2Some code listings in this paper are simplified for presentation purposes.

3https://akka.io/
object by sending the `Put(key, counter)` action wrapped inside a `StoreAction`. Upon receiving a `StoreAction` the store unwraps the action, which is `Put(key, counter)`, and forwards it to the store that is responsible for the provided namespace, in this case `SEC_SPACE`. That is the namespace for objects shared under strong eventual consistency (SEC) [20], a consistency model that is the focus of Section 3.

Listing 2. Setting up and using Squirrel.

In the remainder, we describe the implementation of two custom replication models in Squirrel: SEC in Section 3 and distributed transactions in Section 4. We also introduce a privacy extension for replication in Section 5.

### 3 Strong Eventual Consistency

Strong eventual consistency (SEC) [20] is a variation on eventual consistency that avoids replica synchronisation by relying on some mathematical properties. When replicas shared under SEC receive the same set of operations, possibly in a different order, they are guaranteed to converge. One way to achieve this is to make all operations commute. Commutativity lies at the basis of (operation-based) conflict-free replicated data types (CRDTs) [20]. Other work proposed SECROs [7], an alternative data type ensuring SEC that can deal with non-commutative operations by keeping a totally ordered log of operations.

We observe that data types implementing SEC follow similar implementation strategies. Operations are first applied locally, whereafter the update is asynchronously propagated to the other replicas. How replicas merge incoming updates depends on the data type. Based on this strategy we implement a generic replication model for SEC and some protocols (CRDTs and SECROs) that handle updates in different ways.

#### 3.1 Replication Model for SEC

We now describe a generic replication model for SEC. By design, the model abstracts five crucial aspects, i.e. how to:

- replicate objects,
- buffer updates,
- serialise/deserialise updates,
- merge incoming updates, and
- reset objects.

The first three aspects are properties of the replication protocol while the other aspects are defined by the replicated data type itself. By default, objects are serialised using the Java serialiser, however, programmers can override specific hooks to provide custom serialisers.

Listing 3 shows a skeleton of the replication model for SEC. Internally, the model maintains a store which is a special map CRDT (Line 3) allowing arbitrary nesting of replicated data types (also known as the remove-as-recursive-reset map CRDT [16]). This CRDT requires, however, that all stored objects define a reset operation. On Line 4 the model overrides the abstract type T with a subtype relation, enforcing all objects shared under this model to implement the `SECProtocol` which will be analysed later. The model extends the traditional API (get, put, and delete) from Figure 2 with two additional operations: update and `updateAt`. The former applies a pure function on an object and yields the updated object. Similarly, the latter applies a pure function on a nested object that is identified by a path consisting of three parts: a key identifying an object 0 in the store, a path within 0, and a final pointer to the object. Recall that stores are actors, hence, we extend the actor’s receive partial function to handle `update` and `updateAt` messages sent by users of the store (Lines 17 to 24).

Listing 3. Skeleton of the replication model for SEC.

We now turn attention to the actual implementation of the operations. Listing 4 only shows the implementation of get, put, and update. The other operations are similar since they adhere to the aforementioned implementation strategy.


---

4A pointer can be an index in a list, a key in a map, etc.

The get operation fetches the value that is associated to the provided key, from the underlying store. The put operation adds the key-value pair locally whereafter it replicates the entry across the cluster. Similarly, the update operation updates the key-value pair locally, fetches the updated value and propagates it across the Squirrel cluster.

Listing 5. Replicating objects and propagating updates.

Listing 5 shows how entries are replicated and updates are propagated. To replicate an entry, the model calls the object’s replicate method which is part of the object’s replication protocol (Section 3.2) and returns a function for constructing replicas (Line 2). As such, the object does not need to be serialisable. Instead, the function encloses only the information that is needed to reconstruct the object. Lines 3 and 4 wrap the function inside a NewReplica message that is broadcast to the other nodes of the Squirrel cluster. Each time an operation is applied on an object, the object’s protocol stores the update. Periodically, the model fetches the updates, serialises updates, merges them into the object (Lines 12 and 13). How to merge the changes is defined by the protocol. The updated object then replaces the old object in the store (Line 14).

3.2 Protocols for SEC

Recall that the replication model for SEC makes abstraction from five crucial aspects (how to replicate objects, buffer updates, (de)serialise updates, merge updates and reset objects). Objects shared under SEC must provide a concrete implementation for all aspects by applying the protocol shown in Listing 7.


The SECProtocol defines three abstract types (Lines 2 and 3): T is the object’s type and thus must be a subtype of this protocol, U is the type of updates, and E is the type of serialised updates. The protocol also defines a number of abstract methods. The replicate method returns a function that constructs replicas of this object. The init method serves to initialise those new replicas with a unique ID, otherwise they still contain the ID of the original object. The changes method returns the latest updates. The serialise and deserialise methods let protocols customise how updates are sent over the network. The merge method takes some updates, incorporates them in the object, and returns the updated object. The reset method simulates deletions by resetting the object to its initial state. In the remainder
of this section, we use this protocol to implement two data types guaranteeing SEC.

3.2.1 CRDT Protocol

We now extend Squirrel with state-based CRDTs [20] by implementing the protocol from Listing 7. State-based CRDTs are objects whose state form a join semilattice. Updates are propagated by shipping the CRDT’s entire state to the other replicas. Incoming states are merged into the local replica by taking the least upper bound (LUB) of the local and received state. The merge operation is defined by the data type itself.

```scala
1 trait CvRDT extends SECProtocol {
2   override def replicate(): Unit = this
3   override def changes(): CvRDT = this
4   override def serialise(updts: CvRDT): CvRDT = updts
5   override def deserialise(updts: CvRDT): CvRDT = updts
7 }
```

Listing 8. State-based CRDTs.

Listing 8 shows the implementation of the protocol for state-based CRDTs. replicate yields a closure that returns the CRDT object itself. The changes method simply returns the object and no custom serialisation is used. CRDT objects must thus be serialisable and both abstract type variables U and E are equal to T, which is the object’s own type.

Note that implementing operation-based CRDTs is simple as it only requires modifying the changes method to return the operations instead of the state. Appendix A shows how we use the CvRDT protocol to implement custom CRDTs.

3.2.2 SECRO Protocol

We now extend Squirrel with SECROs [7], an alternative data type for SEC. SECROs are objects that keep a totally ordered log of operations that respects causality and preserves application-level invariants specified via preconditions and postconditions. Operations are added to the local log and asynchronously propagated to the other replicas that, in turn, add them to their log, yielding identical logs at all replicas.

```scala
1 trait SecrProtocol[State] extends SECProtocol {
2   this: Secro[State] =>
3   type T <: SecrProtocol[State]
4   override type U = Set[Operation[State]]
5   override type E = U
6   var buffer = Set[Operation[State]]
7   override def changes(): U = {
8     val tmp = buffer
9     buffer = Set[Operation[State]]
10    tmp
11    }
12   override def serialise(updts: U): E = updts
13   override def deserialise(updts: E): U = updts
7 }
```


We previously focused on strong eventual consistency, a consistency model that guarantees high availability and low latency but introduces temporal inconsistencies. We now focus on models that guarantee strong consistency by means of distributed transactions.

Applications require distributed transactions to coordinate updates spanning several participants with ACID guarantees. Such transactions involve a coordinator and a fixed set of participants. For each transaction the system undertakes the following steps:

1. Start transaction
2. Apply tentative updates at the participants
3. End transaction
4. Decide whether to commit or abort the transaction

The first three steps are general, the last step depends on the atomic commit protocol (e.g. two-phase commit [3]).

In the remainder of this section we extend Squirrel with transactions, enabling arbitrary atomic commit protocols to be plugged in. To this end, we implement a new replication model (i.e steps 1 to 3 from the list above) and abstract the decision making process into a separate replication protocol.

We then implement two-phase commit and show how to plug it into our transactional model.

4 Distributed Transactions

Conceptually, the transactional model extends the store with operations to start and end transactions. The put and delete operations expect an additional argument, namely the transaction id. This id is passed through the method’s extra parameter which accepts a variable number of arguments.

```scala
1 trait Transactional extends ReplicationModel {
2   with Coordinator with Participant {
3     type T
4     var store: Store[T] = Map()
5     var tentatives: Map[TID, Store[T]] = Map()
6     override val model: Namespace = Tx_SPACE
7     override def get(
8       key: Key[T],
9       extra: Any*): Option[T] = /*...*/
10    override def put(
11      key: Key[T], value: T,
```
Listing 10. Skeleton of the transactional model.

Listing 10 shows the structure of the transactional model. Every node can be both a participant and a coordinator; therefore, Line 2 of the transactional model mixes in functionality from both the Coordinator and Participant traits.

In this model, the put and delete operations operate on a tentative copy of the store. The model stores a tentative copy for each transaction (Line 5). In addition to the traditional operations, this model provides operations to start and end transactions (Lines 16 to 21). The actual implementation of these operations is provided by the Coordinator trait that is mixed-in (shown in Listing 11).

```
trait Coordinator extends Actor {
  type T
  case class Tx(participants: Set[ActorRef],
                 protocol: TxProtocol)
  var store: Store[T]
  var transactions: Map[TID, TxDetails] = Map()
  def startTx(participants: Set[ActorRef],
              protocol: TxProtocol) = {
    val tid = id()
    val tx = Tx(participants, protocol)
    transactions += tid -> tx
    sender ! NewTransaction(tid)
  }
  def endTx(tid: TID) = {
    transactions.get(tid) match {
      case None => sender ! UnknownTx(tid)
      case Some(Tx(participants, protocol)) =>
        protocol.commit(tid, participants, self)
    }
  }
  def onDecision(tid: TID, decision: Vote) = {
    val tx = transactions.get(tid).get
    tx.participants.foreach(p => deliver(msg, p))
    transactions -= tid
  }
}
```

Listing 11. The transaction coordinator.

Listing 11 shows the functionality for coordinators. The startTx method takes a set of participants and an atomic commit protocol as input. The method then generates a unique ID for the transaction, stores the details of this transaction in the transactions map, and returns a NewTransaction message containing the ID of the newly created transaction. The endTx method fetches the transaction’s protocol and calls its commit method (see TxProtocol in Figure 3), thereby initiating the atomic commit protocol. When the protocol reaches a final decision the coordinator’s onDecision method is invoked. This method informs all participants of the final decision (Line 23). Since the communication layer guarantees eventual delivery we can already discard the transaction locally (Line 24).

```
trait TxProtocol {
  def commit(tid: TID, participants: Set[ActorRef],
              coordinator: ActorRef) = {
  }
}
```

Listing 12. Interface for transactional protocols.

Listing 12 shows the interface for transactional protocols. Such protocols implement a commit method that reaches the participants in an attempt to commit the given transaction. The final decision (commit or abort) must be reported to the coordinator which in turn informs the participants.

Finally, Listing 13 shows the implementation of participants. The participant’s vote method decides to commit or abort the transaction. The doCommit method replaces the store by the transaction’s tentative store. The doAbort method discards the transaction locally.

```
trait Participant extends DecisionLogic {
  type T
  var store: Store[T]
  var tentatives: Map[TID, Store[T]]
  def vote(tid: TID): Vote = /* ... */
  def doCommit(tid: TID) =
    store = tentatives.get(tid).get
  def doAbort(tid: TID) = tentatives -= tid
}
```

Listing 13. Participant of a transaction.

4.2 Two-Phase Commit

We now showcase how to plug in custom atomic commit protocols. As an example, we implement the well known two-phase commit protocol (2PC) [3]. Our implementation is resilient to arbitrary network failures but cannot cope with device failures. Robustness against device failures requires extending the protocol with persistence.

```
class TwoPC(implicit t: Timeout)
  extends TxProtocol {
  override def commit(tid: TID,
                      ps: Set[ActorRef],
                      coordinator: ActorRef) = {
    val answerFuts = ps.map(_ ? Decide(tid))
  }
```

The actual implementation is more complex because the tentative store may contain stale entries due to overlapping transactions that got committed.
To showcase the extensibility of our distributed KV store, we describe an extension that controls the propagation of waits either for all votes to arrive or for the voting phase to time out. In the actual implementation the protocol aborts the transaction immediately when receiving an abort vote.

5 Scoped Replicas

To showcase the extensibility of our distributed KV store, we describe an extension that controls the propagation of replicated data types. This extension is highly relevant in the context of the new GDPR legislation. Simply put, personal data of EU citizens may only be transferred to non-EU countries if they have an adequate level of protection. Such restrictions greatly complicate geo-replication.

In this section, we propose a static scoping mechanism for replicated data as a simple means to control data propagation. In our model each node owns one or more scopes and objects belong to exactly one scope. Objects are replicated only to nodes owning the object’s scope. Likewise, subsequent updates are only propagated within the boundaries of the scope. Note that we deliberately choose static scopes. Allowing scopes to change at runtime quickly leads to privacy issues, especially in a weakly-consistent setup.

Listing 15. Configuring a node with the EU scope.

In Akka, members of a cluster have a “role” property describing their capabilities. We encode the scopes of a node in its role property, as shown in Listing 15. Every time a member joins the cluster, Akka notifies subscribers about this member and its roles. We use this information to keep track of each member’s scopes. When replicating objects or propagating updates, we only contact members of the scope.

Listing 16. Implementation of the scoping mechanism.

Listing 14 shows our implementation of 2PC. The TwoPC class extends the TxProtocol trait and provides a concrete implementation of commit. On Line 6, commit asks each participant to vote on the transaction. If everyone agrees to commit (Line 8) the protocol commits the transaction (Lines 10 to 14). However, if some participant votes abort or the votes do not arrive timely, then the protocol aborts the transaction (Lines 15 to 17). This implementation always waits either for all votes to arrive or for the voting phase to time out. In the actual implementation the protocol aborts the transaction immediately when receiving an abort vote.

Listing 14. Two-phase commit.

Listing 14 shows our implementation of 2PC. The TwoPC class extends the TxProtocol trait and provides a concrete implementation of commit. On Line 6, commit asks each participant to vote on the transaction. If everyone agrees to commit (Line 8) the protocol commits the transaction (Lines 10 to 14). However, if some participant votes abort or the votes do not arrive timely, then the protocol aborts the transaction (Lines 15 to 17). This implementation always waits either for all votes to arrive or for the voting phase to time out. In the actual implementation the protocol aborts the transaction immediately when receiving an abort vote.

5 Scoped Replicas

To showcase the extensibility of our distributed KV store, we describe an extension that controls the propagation of replicated data types. This extension is highly relevant in the context of the new GDPR legislation. Simply put, personal data of EU citizens may only be transferred to non-EU countries if they have an adequate level of protection. Such restrictions greatly complicate geo-replication.

In this section, we propose a static scoping mechanism for replicated data as a simple means to control data propagation. In our model each node owns one or more scopes and objects belong to exactly one scope. Objects are replicated only to nodes owning the object’s scope. Likewise, subsequent updates are only propagated within the boundaries of the scope. Note that we deliberately choose static scopes. Allowing scopes to change at runtime quickly leads to privacy issues, especially in a weakly-consistent setup.

Listing 15. Configuring a node with the EU scope.

In Akka, members of a cluster have a “role” property describing their capabilities. We encode the scopes of a node in its role property, as shown in Listing 15. Every time a member joins the cluster, Akka notifies subscribers about this member and its roles. We use this information to keep track of each member’s scopes. When replicating objects or propagating updates, we only contact members of the scope.

Listing 16. Implementation of the scoping mechanism.

Listing 16 shows the implementation of the scoping mechanism. We override the actor’s preStart hook on Lines 6 to 10 to subscribe to cluster events. The actor’s receive partial function is extended to handle incoming cluster events (Lines 30 to 37). Each time a new member joins the cluster we fetch his scopes and add them to the scopes map. We also override the put operation such that it 1) expects an additional scope argument (Line 13), 2) authorises the user to put objects in the provided scope, 3) stores the object’s scope, and 4) delegates the call to the actual model using super (Line 19). Finally, the trait overrides broadcast (Lines 22 to 28) such that it fetches the scope of a message and sends the message only to the members of that scope.
The ScopedProp trait presented in this section is designed to be mixed in after a concrete replication model.

```scala
class ScopedSEC extends SEC with ScopedProp {
  override val model: Namespace = SCOPED_SPACE
}
```

Listing 17. Extending SEC with scopes.

Listing 17 uses the ScopedProp trait to define a ScopedSEC store that extends the store for SEC (from Section 3) with scopes. Within this newly created store, existing CRDTs and SECROs can be reused without modifications.

6 Related Work

As argued in the introduction, NoSQL databases typically only support a few consistency models, e.g. Riak focuses on SEC and provides only 6 CRDTs which cannot be composed arbitrarily. Cassandra[^9], does allow programmers to fine tune the consistency and availability of the database by choosing from a fixed set of consistency levels. In contrast to Squirrel, programmers cannot implement custom consistency levels.

We focus the discussion of related work on replicated databases which allow users to choose between consistency levels for an operation. Examples include DynamoDB [8], Yahoo PNUTS [6], and CRAQ [21], all of which support strong and eventually consistent reads. However, users are limited to the consistency levels provided by the database and have no means to extend the database as with Squirrel.

Pileus [22] is a replicated KV store that allows programmers to declare service level agreements (SLAs) about consistency guarantees and latency requirements. Based on these SLAs, Pileus dynamically adapts reads to provide an optimal service. Pileus thus switches between consistency models implicitly, whereas Squirrel uses explicit transition rules.

Gemini [11] supports RedBlue consistency, a consistency model where commutative operations are labeled blue, and others are labeled red. Blue operations are executed under eventual consistency and red operations are executed under strong consistency. Due to the commutativity requirement for blue operations, Gemini’s support for eventual consistency is limited to CRDTs. Sieve [10] improves upon Gemini by taking into account application semantics when labelling operations. It also extends the scope of blue operations to non-commutative operations by turning them into commutative ones. To this end, programmers pick the CRDT semantics that suit the operation from a fixed set of semantics. Sieve does not support custom semantics.

MixT [12] proposes mixed-consistency transactions to manipulate data with different consistency levels within a single database transaction. Using information flow analysis MixT can break down mixed-consistency transactions into subtransactions for each consistency level and still guarantee atomicity. Whereas MixT focuses on manipulating data using different consistency levels within one transaction, Squirrel focuses on supporting arbitrary replication models to coexist within an application.

Other work on mixed consistency models uses invariants to classify operations as safe or unsafe [1, 2]. Safe operations execute fast, whereas unsafe operations are synchronised in order not to violate application level invariants. These consistency models are currently not supported by Squirrel but can be added using Squirrel’s metaprogramming constructs.

**Actor-based approaches.** We now compare Squirrel to related work on actor languages. Previous work explored an open implementation of the actor scheduler [17, 18]. Those approaches focus on the extensibility of scheduling policies in a concurrent setting. However, distributed concerns such as the serialisation of objects across actors are not reified, making it difficult to implement a distributed KV store.

Similar to our work, CAPtain [14] supports both consistency and availability by means of consistent and available objects. CAPtain’s meta-object protocol (MOP) allows experts to change the model’s internals, for instance to customise replication. However, the granularity of CAPtain’s MOP is on the level of actors and does not explicitly target replication. This forces programmers to resort to lower-level abstractions in order to implement high-level replication code. In contrast to CAPtain, Squirrel’s replication protocols are designed specifically for replication. We show that these protocols ease the development of replicated data types by adding support for CRDTs and SECROs.

7 Conclusion and Future Work

In this work, we study how to improve the flexibility of in-memory distributed KV stores with regard to replication and data consistency. We propose an open implementation of a distributed KV store featuring a customisable replication model. Multiple models can coexist, each of which can have several implementations or “protocols”. Programmers can also implement their own storable data types.

We prototyped the proposed architecture in Scala, resulting in the Squirrel distributed KV store. We validate Squirrel’s metaprogramming facilities by implementing strong eventual consistency and distributed transactions. We also introduce a scoping mechanism for replicated data types that requires no modifications to the data types.

Although the envisioned architecture allows programmers to switch between models and protocols at runtime, the design and implementation of transition rules which allow objects to move from one store to another (e.g. from the SEC store to the transactional store) remains future work.

A An Increment-Only Counter CRDT

Throughout the paper we discussed the replication protocol for CRDTs. We show how programmers can use this protocol to implement custom storable CRDTs. As an example we

[^9]: Such that the trait’s super refers to the actual replication model.
[^10]: http://cassandra.apache.org
show the implementation of a state-based increment-only counter (G-Counter) CRDT [19].

```scala
1 type ID = UniqueAddress
2 @Serializable(UID(100L)
3 case class GCounter(
4   id: ID, incs: Map[ID, Int] = Map().withDefaultValue(0))
5 extends CvRDT {
6   override type T = GCounter
7   override def init(id: ID) = copy(id == id)
8   def value = incs.getOrElse(0) + _
9   def increment() = incs = incs.updated(id, incs + (id + 1))
10 override def merge(that: T, from: ID) = T {
11   mergedIncrements = mergeMaps(this.incs, that.incs)
12   (_ - getOrElse(0) + _).getOrElse(0)
13   copy(incs = mergedIncrements)
14 }
15 private def mergeMaps[K, V](m1: Map[K, V], m2: Map[K, V])
16   (merge: (Option[V], Option[V]) => V) = {
17     val keys = m1.keySet ++ m2.keySet
18     keys.foldLeft(Map.empty[K, V])((map, key) =>
19       map += (key -> merge(m1.get(key), m2.get(key))))
20   }
21 }
22
Listing 18. A G-Counter CRDT.

Listing 18 shows the implementation of the G-Counter. In order for the GCounter class to be a state-based CRDT it extends the CvRDT protocol from Listing 8. The counter maintains an incs map that counts the number of increments per replica. On Line 6 the counter assigns the abstract type T to the concrete type GCounter. The merge method on Lines 13 to 18 takes the maximum of each entry in its own and the received increment maps. Instances of the GCounter class are now storable under the SEC model.

Acknowledgments
Kevin De Porre is funded by an SB Fellowship of the Research Foundation - Flanders. Project number: 1S98519N.

References
```
