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:
Kevin De Porre and Elisa Gonzalez Boix. 2019. Squirrel: An Extensible Distributed Key-Value Store. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (META '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358502.3361271

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].

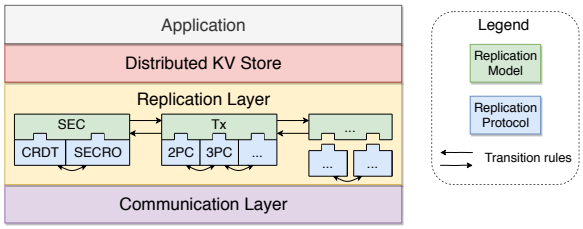

Figure 1. Squirrel's high-level architecture.

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

[1] https://docs.riak.com/riak/kv/2.2.3/developing/data-types.1.html

META '19, October 20, 2019, Athens, Greece

ACM ISBN 978-1-4503-6985-5/19/10…$15.00
https://doi.org/10.1145/3358502.3361271

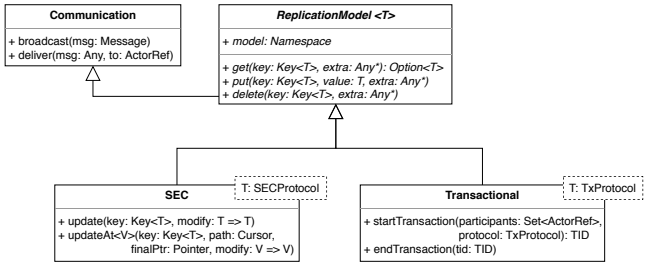

Figure 2. An overview of Squirrel's replication models. Programmers can plug in custom replication models by implementing the abstract `ReplicationModel`.

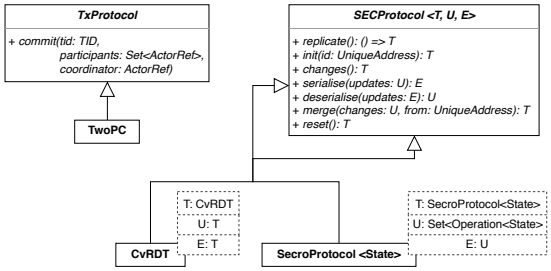

Figure 3. An overview of Squirrel's replication protocols. Programmers can plug in custom replication protocols by implementing the abstract protocol defined by the model.

and retrieve data. The store is built on top of the replication layer which determines where to store the data and how to perform reads and writes. This layer consists of three parts:

Models A replication model dictates the effects of operations on the data being stored. It essentially prescribes what to do with the data (e.g. cache on reads) without specifying how to do it. Therefore, the model distils an interface that must be implemented. A replication model may refine the store API and extend it with additional operations.

Protocols A replication protocol implements the interface of some specific model, i.e. specifies how and where to replicate data, how to manage replicas, etc.

Transitions Since an object's consistency requirements may vary over time [9], transition rules describe how to switch between models and protocols at runtime.

In this paper we focus on replication models and protocols. Transition rules are left as future work.

The replication layer is built on top of the communication layer which reifies two communication primitives: 1) point-to-point communication with causal, exactly-once delivery, and 2) causal order broadcast. The causal delivery property ensures that if a message A could have influenced message B, all processes receive A before B.

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`, 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 [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 [3] 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`

[2] Some code listings in this paper are simplified for presentation purposes.
[3] https://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.

```
1 val system = ActorSystem("my-system")
2 /* set up a cluster of machines */
3 val squirrel = new Squirrel(system)
4 val store = squirrel.storeActor
5 val key = CounterKey("my counter")
6 val counter = Counter()
7 store ! StoreAction(Put(key,counter), SEC_SPACE)
```

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 `O` in the store, a path within `O`, and a final pointer to the object [4]. 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).

```
1  type Cursor = Seq[Pointer[Storable]]
2  class SEC extends ReplicationModel {
3    var store: RRMap[String] = RRMap.empty[String]
4    override type T <: SECProtocol
5    override val model: Namespace = SEC_SPACE
6    override def get(key: Key[T], extra: Any*) =
7      /* ... */
8    override def put(key: Key[T], value: T,
9      extra: Any*) = /* ... */
10   override def delete(key: Key[T],extra: Any*) =
11     /* ... */
12   def update(key: Key[T], modify: T=>T) = /* .. */
13   def updateAt[V <: SECProtocol]
14     (key: Key[T], path: Cursor,
15      finalPtr:Pointer[V], modify: V=>V) = /* .. */
16
17   def extension: Receive = {
18     case Update(key, modify, extra @ _*) =>
19       update(key, modify, extra:_*)
20     case UpdateAt(key, path, fPtr, modify) =>
21       updateAt(key, path, fPtr, modify)
22   }
23   override def receive: Receive =
24     super.receive orElse extension
25 }
```

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.

```
1 override def get(key: Key[T], extra: Any*) =
2   store.get(key)
3 override def put(key: Key[T],
4                  value: T, extra: Any*) = {
5   store = store + (key, value)
```

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

```
6     replicate(key, value)
7   }
8   def update(key: Key[T], modify: T => T) = {
9     val newStore = store.update(key, modify)
10    val value = newStore.get(key).get
11    propagate(key, value)
12  }
```

Listing 4. Operations of the SEC model.

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.

```
1   def replicate(key: Key[T], value: T) = {
2     val fn = value.replicate()
3     val msg = NewReplica(key, fn, selfAddress)
4     broadcast(msg)
5   }
6   def propagate(key: Key[T], value: T) = {
7     val updates = value.changes
8     val str = value.serialise(updates)
9     val msg = StateChange(key, str, selfAddress)
10    broadcast(msg)
11  }
```

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 [5] (Line 7), serialises them (Line 8) and wraps the serialised updates inside a `StateChange` message which is broadcast over the cluster (Lines 9 and 10).

When these messages arrive at a Squirrel instance, the replica must be added to the store or the update applied on a replica. Listing 6 shows how we extend the model's `receive` partial function to handle incoming `NewReplica` and `StateChange` messages.

```
1   override def receive: Receive =
2     super.receive orElse extension orElse {
3       case NewReplica(key, repBuilder, from) => {
4         val selfAddress =
5         Cluster(context.system).selfUniqueAddress
6         val rep = repBuilder().init(selfAddress)
7         val map = RRMap.empty[String] + (key, rep)
8         val store = store.merge(map, from)
9       }
10      case StateChange(key, changes, from) => {
11        val oldVal = store.get(key).get
12        val updates = oldVal.deserialise(changes)
13        val newVal = oldVal.merge(updates, from)
14        store = store.update(key, (_: T)=>newVal)
15      }
16    }
```

Listing 6. Handling incoming replicas and updates.

Upon receiving a `NewReplica` message, the model constructs a new replica using the wrapped function (Line 6), adds the replica to a fresh map CRDT (Line 7), and merges that map into the current store (which takes care of conflicts in case the entry already exists). Upon receiving a `StateChange` message, the model deserialises the changes and 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 implementing the protocol shown in Listing 7.

```
1   trait SECProtocol {
2     type T <: SECProtocol
3     type U; type E;
4     // Properties of the protocol
5     def replicate(): () => T
6     def init(id: UniqueAddress): T
7     def changes(): U
8     def serialise(updates: U): E
9     def deserialise(updates: E): U
10    // Properties of the data type
11    def merge(changes: U, from: UniqueAddress): T
12    def reset(): T
13  }
```

Listing 7. Interface for SEC protocols.

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 lets 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

[5] An update can be some state, an operation, etc.

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.

```
1  trait CvRDT extends SECProtocol {
2    type T <: CvRDT; type U = T; type E = T
3    override def replicate() = () => this
4    override def changes(): T = this
5    override def serialise(updts: T): T = updts
6    override def deserialise(updts: T): T = 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 [6]. 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.

```
1  trait SecroProtocol[State] extends SECProtocol {
2    this: Secro[State] =>
3    type T <: SecroProtocol[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
14 }
```

Listing 9. State-based CRDTs.

[6] We rely on Scala's spores [13] to ensure safe serialisation of closures.

Listing 9 shows the implementation of the protocol for SECROs. On Line 4, the abstract type `U` is assigned the concrete type `Set[Operation[State]]` because SECROs propagate operations on state. Every time an operation executes locally it is added to a buffer (Line 6). The `changes` method on Lines 7 to 11 returns the operations accumulated by the buffer and clears it. The protocol does not apply a custom serialisation scheme, therefore, the abstract type `E` equals `U`. The protocol does not implement the `replicate` method, leaving the implementation up to the SECROs themselves. For example, the list SECRO replicates its internal list.

4 Distributed Transactions

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.

4.1 Transactional Model

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.

```
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,
```

```
12      extra: Any*) = /* ... */
13    override def delete(
14      key: Key[T],
15      extra: Any*) = /* ... */
16    override def receive: Receive =
17      super.receive orElse {
18        case StartTx(participants, protocol) =>
19          startTx(participants, protocol)
20        case EndTx(tid) => endTx(tid)
21      }
22  }
```

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).

```
1   trait Coordinator extends Actor {
2     type T
3     case class Tx(participants: Set[ActorRef],
4                   protocol: TxProtocol)
5     var store: Store[T]
6     var transactions: Map[TID, TxDetails] = Map()
7     def startTx(participants: Set[ActorRef],
8                 protocol: TxProtocol) = {
9       val tid = id()
10      val tx = Tx(participants, protocol)
11      transactions += tid -> tx
12      sender ! NewTransaction(tid)
13    }
14    def endTx(tid: TID) = {
15      transactions.get(tid) match {
16        case None => sender ! UnknownTx(tid)
17        case Some(Tx(participants, protocol)) =>
18          protocol.commit(tid, participants, self)
19      }
20    }
21    def onDecision(tid: TID, decision: Vote) = {
22      val tx = transactions.get(tid).get
23      tx.participants.foreach(p => deliver(msg,p))
24      transactions -= tid
25    }
26  }
```

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).

```
1   trait TxProtocol {
2     def commit(tid:TID, participants:Set[ActorRef],
3                coordinator: ActorRef)
4   }
```

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 [7]. The `doAbort` method discards the transaction locally.

```
1   sealed trait Vote
2   case object Commit extends Vote
3   case object Abort extends Vote
4
5   trait Participant extends DecisionLogic {
6     type T
7     var store: Store[T]
8     var tentatives: Map[TID, Store[T]]
9     def vote(tid: TID): Vote = /* ... */
10    def doCommit(tid: TID) =
11      store = tentatives.get(tid).get
12    def doAbort(tid: TID) = tentatives -= tid
13  }
```

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.

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

[7] The actual implementation is more complex because the tentative store may contain stale entries due to overlapping transactions that got committed.

```
7       Future.sequence(answerFuts)
8         .map(answrs => answrs.forall(_ == Commit))
9         .onComplete(_ match {
10          case Success(decision) => {
11            val consensus =
12              if (decision) Commit else Abort
13            coordinator ! Outcome(tid, consensus)
14          }
15          case Failure(_) =>
16            coordinator ! Outcome(tid, Abort)
17        })
18    }
19  }
```

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.

```
1  akka {
2    actor { provider = "cluster" }
3    cluster { roles = ["scope:EU"] }
4  }
```

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.

```
1  trait ScopedProp extends ReplicationModel {
2    val cl = Cluster(context.system)
3    val selfScopes = parseScopes(cl.selfMember)
4    var scopes: Map[Member, Set[String]] = Map()
5    var objScopes: Map[Key[T], String] = Map()
6    override def preStart(): Unit = {
7      super.preStart()
8      cl.subscribe(self,
9                   classOf[ClusterDomainEvent])
10   }
11   override def put(key: Key[T], value: T,
12                    extra: Any*) = {
13     val scope = extra.head.asInstanceOf[String]
14     if (!selfScopes.contains(scope)) {
15       sender ! CreateFailure(key, model)
16     }
17     else {
18       objScopes += key -> scope
19       super.put(key, value, extra.tail)
20     }
21   }
22   override def broadcast(msg: MessageWithKey) ={
23     val scope = objScopes.get(msg.key).get
24     scopes
25       .filter(_._2.contains(scope))
26       .keys.map(getActorRef(_))
27       .foreach(to => deliver(msg, to))
28   }
29   def updateScope(m: Member) = /* ... */
30   def receiveClusterEvents: Receive = {
31     case m: MemberEvent => updateScope(m.member)
32     case state: CurrentClusterState =>
33       state.members.foreach(updateScope(_))
34     case _: ClusterDomainEvent => // ignore
35   }
36   override def receive: Receive =
37     receiveClusterEvents orElse super.receive
38 }
```

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 [8].

```
1 class ScopedSEC extends SEC with ScopedProp {
2   override val model: Namespace = SCOPED_SPACE
3 }
```

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

[8] Such that the trait's super refers to the actual replication model.
[9] http://cassandra.apache.org

show the implementation of a state-based increment-only counter (G-Counter) CRDT [19].

```
1  type ID = UniqueAddress
2  @SerialVersionUID(100L)
3  case class GCounter(
4    selfID: ID, incs: Map[ID, Int] =
5      Map().withDefaultValue(0)) extends CvRDT {
6    override type T = GCounter
7    override def init(id: ID) = copy(selfID = id)
8
9    def value = incs(0)(_ + _)
10   def increment(): GCounter =
11     copy(incs = incs.updated(selfID,
12                              incs(selfID)+1))
13   override def merge(that: T, from: ID): T = {
14     val mergedIncrements =
15       mergeMaps(this.incs, that.incs)
16                (_.getOrElse(0) + _.getOrElse(0))
17     copy(incs = mergedIncrements)
18   }
19   private def mergeMaps[K, V]
20     (m1: Map[K, V], m2: Map[K, V])
21     (merge: (Option[V], Option[V]) => V) = {
22     val keys = m1.keySet ++ m2.keySet
23     keys.foldLeft(Map.empty[K, V])((map, key) =>
24       map + (key -> merge(m1.get(key),
25                           m2.get(key))))
26   }
27 }
```

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

[1] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *10th European Conference on Computer Systems (EuroSys '15)*. Article 6, 16 pages.

[2] Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, Joao Leitao, Nuno Preguiça, et al. 2016. Geo-Replication: Fast If Possible, Consistent If Necessary. *IEEE Data Engineering Bulletin* 39, 1 (2016), 12.

[3] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Eric Brewer. 2000. Towards Robust Distributed Systems. In *19th Annual ACM Symp. on Principles of Distributed Computing (PODC '00)*. 7.

[5] Eric Brewer. 2012. CAP Twelve years later: How the Rules have Changed. *Computer* 45 (02 2012), 23–29.

[6] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1277–1288.

[7] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. 2019. Putting Order in Strong Eventual Consistency. In *Distributed Applications and Interoperable Systems*, José Pereira and Laura Ricci (Eds.). Springer International Publishing, Cham, 36–56.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP '07)*. 205–220.

[9] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 253–264.

[10] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292.

[11] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278.

[12] Matthew Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 226–241.

[13] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *ECOOP 2014*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–333.

[14] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2018. A CAPable Distributed Programming Model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2018)*. ACM, New York, NY, USA, 88–98.

[15] Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 41–57.

[16] Nuno Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. *arXiv preprint arXiv:1806.10254* (2018).

[17] Aleksandar Prokopec. 2016. Pluggable Scheduling for the Reactor Programming Model. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*. ACM, New York, NY, USA, 41–50.

[18] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. 2014. Parallel Actor Monitors: Disentangling Task-level Parallelism from Data Partitioning in the Actor Model. *Sci. Comput. Program.* 80 (Feb. 2014), 52–64.

[19] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages.

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer-Verslag, Grenoble, France, 386–400.

[21] Jeff Terrace and Michael J. Freedman. 2009. Object Storage on CRAQ: High-throughput Chain Replication for Read-mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, Berkeley, CA, USA, 11–11.

[22] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 309–324.