Shared State for Actors: Pass-By-Replication Semantics

Unifying Asynchrony and Eventual Consistency

Tim Coppieters
Vrije Universtiteit Brussel
Pleinlaan 2
Brussels, Belgium
tcoppiet@vub.ac.be

Joeri De Koster
Vrije Universtiteit Brussel
Pleinlaan 2
Brussels, Belgium
jdekoste@vub.ac.be

Wolfgang De Meuter
Vrije Universtiteit Brussel
Pleinlaan 2
Brussels, Belgium
wdemeuter@vub.ac.be

ABSTRACT

The actor model prevents the traditional pitfalls of concurrent programming (deadlocks and data-races) by not allowing shared state between different processes. While shared state can be simulated through a set of message exchanges, this quickly becomes complex and renders the program inexpressive. In this paper we present a novel way of integrating shared state into the communicating event-loop model by introducing a new type of object called a *repliq*. When such object crosses actor boundaries it exhibits what we call *pass-by-replication* semantics. The object is deeply copied and the runtime keeps track of the owner of the original *repliq*, the actor that first created it. Actors can access and alter *repliq* objects locally, while the runtime makes sure they are eventually consistent with other replicas of the object. Consistency is guaranteed by eventually executing all the operations of the replicas in the same order, as determined by the original object. Synchronization happens in a background process and external updates become visible in between message processing. This makes sure that the programmer's code is always executed in a consistent snapshot of the object states. We present the semantics, guarantees and restrictions exhibited by *repliq* objects using an implementation in the AmbientTalk actor language. This model maintains the easy and safe local-only execution model for the programmer that is key to the actor model, while providing integrated and expressive means to share state that is eventually consistent.

CCS Concepts

•**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Keywords

ACM proceedings; LaTeX; text tagging

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

1. INTRODUCTION

The actor model is a concurrency programming model intended to provide secure computing capability by nature. One of the core principles of the traditional model is to never explicitly share state between concurrent processes. Every process, called an actor, has its own object heap and can never directly access that of another process. This strict isolation of the different actors ensures that the actor model is free of low-level data races. Actors can communicate with each other by means of exchanging asynchronous messages and because of the asynchronicity of their communication mechanism the actor model is also free of low-level deadlocks. This way the actor model avoids the main pitfalls of the traditional thread model.

In practice, actors are made available either trough a dedicated programming language (for example Erlang [1]) or trough a library implementation (for example, the Akka framework for Scala). On the one hand, actor languages are mostly *pure*, in the sense that they often enforce strict isolation of the different actors. The benefit of this is that low-level data races and deadlocks are ruled out by design. The downside is that they are severely limited in the way they can model access to a shared resource. A common approach to model shared state under these restrictions is by modelling that shared state a delegate actor. Different client actors can then access that shared state by sending asynchronous requests to the delegate actor. However, because clients are forced to use asynchronous communication to access the shared state, this approach leads to an event-driven style of programming and introduces the inversion-of-control problem [9]. On the other hand, actor libraries are mostly *impure*, in the sense that they cannot enforce strict isolation of the different actors. Because of this, developers can use the underlying shared memory as an "escape hatch" to model shared state by allowing actors to obtain direct references to that shared state. It is shown that programmers often step outside of the actor model when possible in order to regain the ability to explicitly share state [13]. However, once the developer chooses to go this route, the benefits of the high-level actor model are lost, and the developer typically has to resort to other ad hoc synchronization mechanisms (e.g. locks) to prevent data races.

Explicit and synchronous access to shared state can be useful both in a shared-memory concurrency setting as well as in a distributed setting. For *concurrent programs* it's sufficient to think about any desktop application that includes editing, saving and displaying of something (e.g. text or images). In order to make the application efficient and re-

sponsive, different processes need to perform the different tasks in parallel.

Any application where people collaborate on the same data makes a good case for shared state in *distributed programs*. This goes from simple chat applications to document and image editing applications. Moreover, more modern applications that allow both collaborating and continuous operation while disconnected, require a more advanced type of shared state. Namely, they require optimistically replicated shared state, also called eventually consistent, which includes a lot more than traditional or pessimistically shared state [11]. First, state and local operations need to be copied. Second, operations need to be applied locally and buffered. Then, the operations need to be synchronized with other copies. Finally, a consistency model needs to be in place to keep the replicas consistent.

In this paper we focus on **adding optimistically replicated, or shared, state** to the actor model, by introducing a novel kind of object: a *repliq*. This object has pass-by-replication semantics, which means that when crossing actor boundaries it is replicated. This involves a deep copy of the object and an algorithm that runs in the background to keep it in sync with other replicas.

The repliq that is first created is called the *master* object and a repliq created by replication is called a *replica*. By construction an actor can always treat a *repliq* object as if it were a normal, local object. Repliqs are kept consistent by eventually executing all operations in the same order, as determined by the master, on all replicas. This requires the operations to be reordered and puts certain restrictions on a *repliq* object. Furthermore, reordering is only made visible in between message processing such that the programmer's code always works on a consistent snapshot of the replicas.

Although this work can be applied to most of the actor models, this paper focuses on the Communicating Event Loop Actor model by using an implementation in the AmbientTalk programming language.

In the next section we first briefly explain the communicating event loop model implemented by AmbientTalk. Afterwards, we introduce the the novel *repliq* object by implementing a distributed chat application that works while disconnected. In section 3 we go into the specifics of the semantics, guarantees and restrictions of the model. Chapter 4 elaborates on the consistency model used by repliq objects and motivates the use of eventual consistency. Finally, we discuss future and related work.

2. COMMUNICATING EVENT LOOP

The Communicating Event-Loop (CEL) Actor model was first introduced by the E programming language [10] as a object oriented programming model for secure distributed computing. In this model actors are represented by *vats*. Throughout the rest of this paper the terms *actor*, *event-loop actor* and *vat* are used interchangeably and always refer to an actor as defined by the CEL model. In the CEL model each vat has a single thread of control (the event loop), an event queue and an object heap (See Figure 1).

Each object in the heap of a vat is *owned* by that vat and is only synchronously accessible by that vat. This restriction ensures that actors are strictly isolated from one another. Within a single vat, objects can hold references to other objects and those references are called *near references*. A near reference can be used as in the traditional object oriented style to synchronously invoke methods on the referenced object. Objects can also hold references to objects within another vat and those references are called *eventual references*. An eventual reference cannot be the target of a synchronous method invocation and any attempt to synchronously access an eventual reference will result in a runtime exception. However, an eventual reference can be used for asynchronous communication. Sending an asynchronous message to an eventual reference enqueues that message as an event in the event queue of the actor that owns the target object. Once the event is ready to be processed it is removed from the event queue of the owner by its event loop and is passed to the target object as a synchronous method invocation. This ensures that a vat can only synchronously access objects owned by that vat and that vats remain fully isolated.

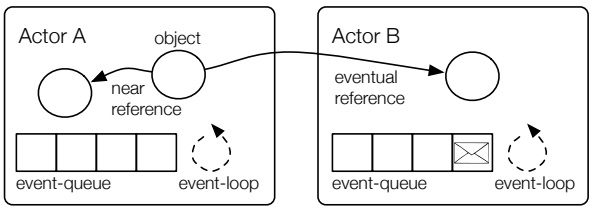


Figure 1: The communicating event-loop model

Any reference that crosses actor boundaries, for example as the argument of an asynchronous message, are passed as an eventual reference. This *pass by eventual reference* semantics distinguishes the CEL model from other actor models where actors typically only have a single entry point or address. In the CEL model a vat can export any number of references to its own objects with potentially different interfaces to other vats. Each of those references shares the same event queue and event loop and serves as an entry point to that vat.

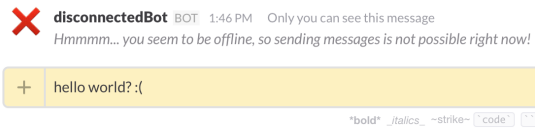
The CEL model was later adopted by AmbientTalk [14]. AmbientTalk is designed as an ambient-oriented programming [6] (AmOP) language. It adds to the CEL model new primitives to deal with disconnecting nodes in a peer-to-peer network where connections are volatile (e.g. a network of cell phones). However, in this paper we do not focus on that aspect of AmbientTalk but rather use it as a vehicle to implement our language abstractions on top of.

Different actor models employ different message passing semantics to ensure isolation. Traditionally, any object that is transmitted across actor boundaries is either copied (e.g. SALSA [15]), proxied (e.g. E [10]) or immutable (e.g. Erlang [1]). Other approaches employ techniques based on ownership types [**?**] to ensure that any reference to a mutable object is globally unique, regardless of whether it is passed between different actors [**?**]. This paper proposes an alternative and novel message passing semantics where certain objects are *passed by replication*.

3. OFFLINE AVAILABLE CHAT

Although offline capability can be useful (e.g. working on an air plane) or even critical (e.g. in disaster-relief scenarios where network connectivity is sparse), supporting it is far from trivial. As a consequence a lot of the modern

distributed/web applications today, often a lack such functionality. Even for example in a simple chat application like Slack (maintained by a billion dollar company), we can see that the programmers prefer to not allow disconnected operations:



The reason is that supporting this is actually a very daunting task. Namely, it requires a deep change from the traditional, centralized application state to a decentralized application state with accompanying complexities. First, state and computations need to be replicated to the interested nodes (e.g. the client's computer). Second, operations need to be tentatively executed locally and buffered. Then, if a connection is available, operations need to be synchronized between client and server. Finally, a consistency model needs to be present to prevent divergence by the concurrent operations. When using an ad-hoc solution for such application, this introduces a tremendous amount of accidental complexity.

The model we introduce hides all these complexities from the programmer by pushing them into the programming model. In order to introduce and demonstrate the repliq model we implement a simple offline available chat with multiple rooms/channels. The employed language is AmbientTalk [14], to which our model is added using meta programming. It is an object oriented language built upon the principles of prototype-oriented programming [**?**]. It uses a combination of curly-brace styled programming languages and keyword syntax like Smalltalk. Concurrency and distribution is at the core of the runtime by implementing the CEL model as described above. Closures are described in terms of blocks : `{ | param1, param2 ... | body }`. An object is initialized using the `object: { <body> }` keyword and its fields are initialized according to the the variables (using `def name := exp`) defined in the body of the initializer block. By cloning an object (using `clone: parent with: { <body> }`) a new object can be created that inherits from the parent object. Finally, AmbientTalk includes a meta-object protocol that uses mirrors [2] in order to override and intercept behaviour of an object and an actor [**?**]. This will be extensively used in the next section in order to embed the proposed model.

We now start by implementing the server of the chat application in Listing 3, which consists of:

- Creating a repliq object called `Channel` that contains all the the data and operations that will be replicated from the server to the clients. It contains a `messages` field that holds the messages of the channel and an `add(msg, user)` method that allows a user to post a message. Sending a message is done by adding a new repliq object, dedicated to that message, to the list of `messages`.
- Creating an object that keeps track of all the different channels by mapping clones of the Channel repliq to their name. It starts off with one channel, the main "general" chat.
- Exporting the API of the server in order to allow clients to retrieve a channel with `getChannel(name)` or create a new one with `createChannel(name)`.

```
def Channel := repliq: {
  def messages := []
  def add(msg, user) {
    this.messages = insertBy(this.messages,
      repliq: { |text, user|
        text: text,
        date: now(),
        user: user
        }, { |msg| msg.date })
    }
}

def channels := Map.new();
channels["general"] = clone: Channel;
}

export("chat server", object: {
  def getChannel (name) { channels[name] }
  def createChannel(name) {
    channels[name] = clone: Channel
  }
})
```

Listing 1: Server implementation of the offline available chat.

Second, we use this server to implement the client in Listing 1, which consists of:

- Retrieving a reference to the server API by looking it up by its name.
- Defining a `displayChannel` method that displays and continuously updates the channel with given name as follows:
  - An asynchronous message call to the `getChannel` method (using the message send operator `<-`) retrieves the desired channel from the server.
  - When the promise of the message resolves, the repliq that represents the general channel is replicated and made available to the client. The given closure is executed and from that point on the `channel` object can be used like a normal, local object.
  - Using a declarative, reactive rendering system, the messages from the `channel` repliq are displayed on the screen. In short, the rendering system has a `render` call that starts a process to display something on the screen. This executes an accompanying closure that can use the `display` method to declare what should be shown on the screen. The `display` function can take any expression that can be rendered into text or pre-defined HTML objects in this case. The `render` method will monitor the given repliqs and re-execute to closure whenever one of them changes and adjust the screen accordingly [1]

[1]This works much like Facebook's React framework (and in fact uses React underneath). Yet, the advantage here is that the rendering framework knows exactly when the objects have changed and thus when to re-render. React has to rely on the programmer keeping some state and explicitly notifying the framework when it might have changed.

– This first displays a loading screen.

  Then, when the `channel` is replicated a list of all the messages and an input field and button to send a new message is displayed. A message contains the time, sender and text and uses the `msg.confirmed` field to display the message in black when it is confirmed by the server or grey when it is still tentative.

– Sending a new message is done by simply calling the `add` method. It uses a built-in method `current_user` that evaluates to a special repliq object that represents the currently logged in user.

```
def server := import("chat server")

def DisplayChannel(name) {

  render({ display("loading...") });

  server<-getChannel(name).then({| channel |
    // stateless display of the messages
    render({
        def ul := UL: channel.messages.map({  msg |
          LI: "(#{msg.time})
               #{msg.sender} :
               #{msg.text}"
          color: msg.confirmed ?
                 Colors.black
                 Colors.gray });
      def inpt := INPUT: "type your message";
      def btn := BUTTON: "send"
                 click: {
                 list.add(inpt.value, current_user)};
      display(ul, inpt, btn);
    }, channel)
  });
}
DisplayChannel("general")
```

Listing 2: Client implementation of the offline available chat.

While this is all the code required to write a simple chat application that works while disconnected, a lot is going on in the runtime. The following section describes how the replication algorithm works.

4. REPLIQ SEMANTICS

A repliq object represents a part of the program (fields + methods) that can be replicated to other actors. These objects have a number of key characteristics:

Isolated The normal object heap is not accessible from within a repliq object. Repliqs can only reference built-in *immutable data types* (such as int, string, list, etc.) and *other repliq objects*. On the other hand, normal objects can have references to repliq objects.

Pass-By-Replication When the repliq object crosses actor boundaries (when it is used as an argument to a message call), it is said to be passed by replication.

The **isolation** of the repliq objects is enforced by the following two properties:

1. A repliq object has *no lexical scope* and fields can only be manipulated from within the repliq its own methods. This means that values can only flow into the object when they are used as arguments to a method call.
2. Normal objects cannot be used as arguments to message calls. Only immutable data types and pointers to other repliq objects of which the master repliq lives on the same actor as the target's master.

These restrictions are required in order to maintain the guarantee of always being able to execute repliq operations locally and maintaining eventual consistency without requiring consensus between actors. This will be further elaborated in the next section.

The **pass-by-replication** semantics are defined as follows:

1. The repliq object is (recursively) deeply copied to the receiving actor, including the entire transitive reach of referenced repliq objects. Such a copied object is called a *replica*. The original repliq created using the `repliq: { body }` construct is called the *master* object[2].
2. The receiving actor keeps track of the master object for each replica it creates.

In order to visualize this, we give an example of how the object heaps of actors implementing the chat application could look like in Figure 2. The rectangles represent actor object heaps: blue circles are normal objects, purple objects are master repliqs and green objects are replica repliqs. A solid arrow represents a local pointer, a dashed arrow a far reference and a dotted arrow a hidden link from the replica repliq to its master. The purple objects on the server represent the different chat rooms and their messages. Each client has a replica of a different thread, i.e. they are both visiting a different chat room. The normal (blue) objects represent pure actor-local state. For the clients this could for example be the text they are currently typing. In the server's case this could be client meta or session data. The repliq (purple/green) objects represent the data and operations that are shared between the actors for collaboration.

The key property that the repliq model guarantees is that local pointers are always locally and synchronously accessible, even if it is a repliq object. This means repliq objects can be acted on as if they were just local, like in the `channel.add(current_user, msg)` example. On the other hand, far references always need to be accessed asynchronously using the <- operator and promises. In order to maintain this property it is thus of utmost importance that normal objects can never be accessed by repliqs. Otherwise, when a repliq is replicated to another actor, this object pointer could be translated into a far reference (at best) and operations would no longer be guaranteed to be local.

Through this guarantee, the programmer can always syntactically see which code will be executed locally, synchronously and which code will execute remotely, asynchronously. This is important, because they have significantly different behaviour [8]

Now, the idea is that replicas are changed locally, making them diverge from the master repliq. Yet, whenever contact

[2]The word *master* will be used for both the actor that owns the original object and the original object itself, depending on the context.

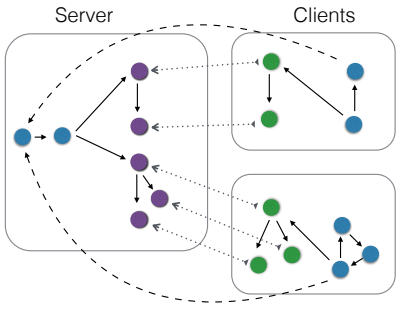


Figure 2: An example of three actors and their object heap for the chat application. Blue = normal object, purple = master repliq, green = replica repliq, solid arrow = local points, dashed arrow = far reference, dotted arrow = replica-master link.

is made between the concerning actors, they will exchange the changes that were performed in order to converge again. This means that changes performed on a replica will eventually be visible on all replicas of the same master repliq. This is what is called optimistic data replication or eventual consistency [11]. How changes are propagated and replicas converge is discussed next.

5. EVENTUAL CONSISTENCY

As mentioned earlier, the fields of a repliq can only be changed by a method of the repliq itself. On top of that repliq objects have no lexical scope and can perform no side-effecting operations other than altering its own fields or calling another repliq method. Because of this restriction, the entire state of a repliq can be determined by the accumulation of all repliq method invocations. In order to use and reason about the replication model it is key to understand this concept. When reasoning about a normal object it is common to only think of the object its current state. Operations on that object are just temporary computations, executed once to change the object its state. In contrast, when reasoning about a repliq it is important to know that its state is made up of a sequential ordering (and execution) of repliq methods. It might namely happen that the ordering of methods changes or methods get added externally, thus changing the state of the repliq.

In order to achieve eventual consistency between all the replicas of a repliq it suffices to establish the same order of method invocations on that repliq. It is the job of the actor that owns the master object to determine the order of the method invocations. The replication protocol relies on the GSP algorithm (Global Sequence Protocol) [5] in order to achieve global ordering of the repliq methods, as determined by the master repliq. While the entire algorithm is out of scope of this paper, we give a simplified, yet sufficient version. It consists of the following key parts:

Log Every actor has a log of method invocations for each actor of which it it has replicas. This log is called the *tentative log*[3].

Versioning Every field of an object has two versions: the committed state and the tentative state. The committed state is the value of that field as confirmed by the master object. The tentative state is the state of that field after applying all the tentative operations on confirmed state.

Reading The default behaviour of reading a field (using `object.field`) is returning the tentative state. Using a dedicated `object..field` operator, the committed value can be retrieved.

Recording The runtime intercepts every *root method invocation*, adds it to the tentative log and executes it. Root method invocations are defined by calls initiated from within a normal object, and not from within a repliq method. Repliq methods invoked from within another repliq method are evaluated without recording.

Replay A synchronization algorithm will use the previous building blocks to establish eventual consistency between the master and replica repliq objects. At the heart this consists of:

1. The replica repliq notifies the master repliq of invocations performed on the replica.
2. The master object invokes the invocation locally and broadcasts this to all the replicas.
3. In the actors of the replicas the received operation will be removed from the tentative operations (if it was there), reset the tentative states to the committed state, replay the received invocation, commit the tentative state and replay the remaining tentative operations.

Using the extensive meta object model we were able to embed first class repliq objects, employing this algorithm, into AmbientTalk. The crux of this implementation relies on implementing mirrors, allowing us to change the behaviour of reading/assigning fields and invoking methods. We now use the core of this implementation in order to elaborate on the semantics of the model. Note that some of the specifics of the implementation (for example creating and installing the mirrors and creating/copying a repliq) are left out for conciseness, but they should not harm the correctness of the algorithm and the understanding of the algorithm.

First, whenever a field is created for a repliq, we intercept this call and create a special kind of field. This is done by overriding the `makeField(name, value)` method in the mirror of a repliq (see Listing 5).

```
// included into the mirror of a repliq
def makeField(nam, val) {
  object: {
    def name := nam;
    def committed := val;
    def tentative := val;
    def readField() {
      tentative
```

[3]In the context of a replica, "the tentative log" will refer to the tentative log of the actor to which the master of the replica belongs.

```
    };
    def writeField(val) {
      tent := newVal
    };
    def resetToCommit() {
      tent := comm
    };
    def commit() {
      comm := tent
    }
  }
}
```

Listing 3: The code that intercepts the creation of a field in a repliq object (by implementing this method in the mirror of a repliq).

This includes keeping track of the committed and tentative value and providing some methods to manipulate these values. Whenever a field is read, the `readField()` method is called on the mirror, which will return the tentative value in this case.

Using these versioned fields we can now implement recording, exchanging and replaying of method invocations in order to obtain eventual consistency. This is all implemented in the mirror of an actor that uses repliq objects (see Listing 3).

At the hearth of this algorithm lies the `invoke(rec, inv)` method, which overrides the behaviour of a method invocation, providing the receiver object and a reification of the invocation itself. First we check whether the receiver of the invocation is a master repliq or not. This is done by getting the mirror of the object using the `reflect: object` construct. On the mirror we can use the `isRepliq` and `isMaster` field, which will be set appropriately upon creation of the repliq object.

When the object is a **replica**, the `invokeReplica` method will check whether a repliq method is already being evaluated by checking the `recording` field. If not, this means a repliq method is being invoked from within a normal object. This results in the following:

1. The actor is marked as being in "record" mode by setting the `recording` field.
2. The invocation is recorded by adding it to the tentative log of the master actor of the receiver, using `logReplica`.
3. The actual invocation of the method is performed on the receiver object, using the application operator `rec <+ inv`.
4. The actor is no longer marked as being in "record" mode.
5. The result of the invocation is returned.

If another repliq method is already being evaluated, this means this method is a result of that evaluation. As a result, this invocation does not need to be recorded and it is normally executed. Because a repliq object can only get access to repliq objects that have the same master actor, it suffices to only record the root call. If this restriction would not apply, all method invocations would need to be recorded and sent to the other actors. Yet, then these two actors would need to coordinate with each other in order to agree on the order of execution, requiring a consensus algorithm. That in turn would drastically complicate the consistency algorithm. This is the reason the repliq referencing rule is in place.

When the object is a **master** repliq, the `recording` field is again used to check whether this is an invocation from a normal object or from within a repliq. If it is invoked from within a normal object, the context is first set to recording, so other invocations know they are already part of a repliq invocation. Then the algorithm proceeds as follows:

1. `markReplicas`: for each replica (of this receiver) its actor, it is checked whether it already has an entry in the `marked` HashMap. If it has not, the current invocation is put in there.
2. The actual method is executed.
3. `unmarkAndBroadcast`: for each actor that was `marked`, it is checked whether it is marked for the current invocation. If it is, that actor is notified of the confirmation of this invocation and it is unmarked.
4. The recording state is set to the state before this invocation.

```
// Mirror of the actors that use repliq
{
  def recording := null;
  def tentative := [];
  def marked := HashMap.new();
  def tentative := HashMap.new();

  def invoke(rec, inv) {
    def om := reflect: rec
    if: om.isRepliq.and: om.isMaster then: {
      invokeReplica(rec, inv)
    } else: {
      invokeMaster(rec, inv)
    }
  }

  def invokeReplica(rec, inv) {
    if: (!recording) then: {
      recording = inv;
      self.logReplica(rec, inv);
      def result := rec <+ inv;
      recording = nil;
      result
    } else: {
      rec <+ inv;
    }
  }

  def logReplica(rec, inv) {
    def master := (reflect: rec).masterActor;
    tentative[master] += [rec, inv]
  }

  def invokeMaster(rec, inv) {
    def wasRecording := recording;
    if: (!recording) then: {
      recording := inv;
    }
    markReplicas(rec, inv);
    def result := rec <+ inv;
    unmarkAndBroadcast(rec, inv);
    recording := wasRecording;
    result
  }

  def markReplicas(rec, inv) {
```

```
    foreach: { | rep |
      def actor := (reflect: rep).actor
      if: (marked[actor] != nil) then: {
        marked[actor] = [rep, inv];
      }
    } in: (reflect: rec).replicas
  }

  def unmarkAndBroadcast(rec, inv) {
    foreach: { | actor |
      [mrec, minv] = marked[actor];
      if: (minv == inv) then: {
        actor <- confirmed(rec, inv)  ;
        marked[actor] = nil;
      }
    } in: marked.keys
  }

  def confirmed(rec, inv) {
    def om := (reflect: rec);
    self.recording := [rec, inv];
    om.resetToCommit();
    rec <+ inv;
    om.commit();
    tentative[om.masterActor].remove(inv);
    foreach: { |[rec, inv]|
      rec <+ inv
    } in: tentative[om.masterActor]
    self.recording := nil;
  }
}
```

Listing 4: The code that intercepts the creation of a field in a repliq object (by implementing this method in the mirror of a repliq).

When an actor receives a `confirmed` notification, it proceeds as follows:

1. All the fields their tentative values are reset to the committed value using `resetToCommit()` on the receiver's mirror, which on its turn will call this on all its fields.
2. The actual invocation is executed.
3. All the receiver's fields their committed values are set to the current tentative value using `commit()`.
4. The invocation is removed from the tentative invocations, if it was there.
5. The tentative operations of the receiver's master actor are re-executed.
6. The recording mode is exited again.

The algorithm makes sure that whenever an actor executes an invocation on a master object, it will broadcast the top-most invocations to the effected replicas. In other words, whenever a master method is invoked that method will be broadcast to all the actors that have a reference to a replica of this repliq. If they don't have a reference to this repliq, the invocations that resulted from this invocation will have the same recursive behaviour until a top-most invocation is found of which the actor does have a reference. This behaviour is achieved by the interplay of `invokeMaster` and `marked`.
To exemplify this with the chat application, say that a client got each message of a channel separately, without the actual channel object. If a method is now invoked on a channel object that has pointers to these objects, this method will not be propagated to that client because it has no reference to that repliq. Yet, if this method performs other method invocations on the messages of the channel (for example increasing all the timestamps), the client does need to get these individual invocations. Otherwise, the message replicas will not converge with the master message repliqs. On the other hand, if another client does have a reference to the channel object, it should not receive all the individual invocations on the messages. Instead it should only receive the invocation of the channel object, i.e. the top-most invocation of which it has a reference.

Using a combination of the GSP protocol and some proofs over the delivery of messages from masters to replicas and vice-versa we can proof eventual consistency between master and replica repliqs is attained. The proof itself is outside of the scope of this paper.

6. DISCUSSION

Some important remarks need to be made about the presented algorithm/model.

First, the given implementation of the algorithm is not **fault tolerant or robust**, i.e. when actors crash and recover or when messages are lost eventual consistency can no longer be guaranteed. Also, the given algorithm assumes reliable broadcast to maintain eventual consistency, which is a strong requirement on the underlying communication model. In order to provide fault-tolerance and not require reliable broadcast, the implementation needs to employ the GSP protocol. This includes using more different types of logs, propagation in numbered rounds and persisting some state. Also, where GSP describes a master-replica relationship between two entities, this implementation requires each actor to perform both the master and the replica part (depending on which object it is acting for). Including all of this in the paper would be too repetitive w.r.t. the GSP protocol and would render it less algorithm. The algorithm described in this paper is a semantically much easier to understand and really demonstrates the core. Furthermore, given this algorithm and the original GSP protocol, the reader should be able to easily reconstruct the fault-tolerant version.

Second, given how the algorithm is structured in this paper, invocations that are performed in the same method are not atomically propagated. This means that whenever a master method performs multiple other invocations and these sub-invocations are propagated, an actor can observe state where one of these invocations is applied but others aren't yet. Instead, it is better to provide the programmer with a **snapshot isolation model** [7], where these updates are all applied transactionally. This is again achieved by the batching of operations in rounds as explained in GSP.

Third, up till now the model assumed that the master repliq always has a list of all actors that have a replica of this repliq. This is easily constructed by notifying the master actor whenever such replica is constructed. Yet, one has to be careful, because this replica can now be missing out on updates from the master while the notification is in-flight/process. Again, by numbering the rounds, these anomalies can be easily detected and restored. What is more interesting though is the deconstruction of this list.

That is, when does the master repliq knows that an actor is no longer interested in updates of this repliq. In our current implementation we use a keep-alive mechanism for this. For every replica that is alive in the object heap, it will poll the master actor every X seconds. Thus, if the actor crashes or it garbage collects this replica, stops polling the master actor. On the other side, if the master actor has not heard from a particular replica for Y seconds, it will discard this replica as a dependency. If an actor recovers its object state after a crash, it can start polling the master replica again. Using the numbered rounds, a recovery procedure can be execute to let the replicas catch up with the current state. As an effect, the proposed replication model embeds a **robust, automatic pub/sub system** between the master and replica repliqs, by means of object presence in the heap. To demonstrate the power of this feature, take for example a web applications. When a particular page is being displayed on the screen, the actor will be subscribed to changes of that screen, because the render function is using those repliqs that constitute the page. Then, when another page is displayed, the previous repliqs are no longer used and garbage collected. This also means the master actor will automatically know that the client does no longer require updates for this data, but now needs updates for the next page. In contrast, using current technologies one would need to manually maintain these subscriptions.

Finally, we haven't discussed this in the paper, but the fact that **invocations are replayed** by the server and other replicas can be exploited for **security** and other applications. When a method is replayed, it can be replayed in context of the user that originally invoked it. This means that whenever the `current_user` method is used within a repliq method, it evaluates to a special user object that represents the user that invoked the method. In order to demonstrate this, we can add functionality to the chat application that allows users to edit their messages. Of course, you do not want other people to be able to edit your message, only the owner of the message can. By adding a `update` method that will only update the text if the `current_user` (i.e. the user that invokes the `update` method) is the owner of the message (see Listing 6).

```
def createMessage(text, user) {
  repliq: { |text, user|
        text: text,
        date: now(),
        user: user,
        update(newText) {
          if (current_user == self.user) {
            self.text = newText;
          }
        }
  }
}
```

Listing 5: Example of how the replaying of invocations can be used for adding security checks.

7. RELATED WORK

TODO **Eventual Consistency models**
Existing eventual consistency models such as cloud types [3], crdt's [12] etc. have no complete integration with an asynchronous programming model and thus provide no guarantees/semantics what happens when the two are combined (e.g. when replica's are sent over the wire etc.)
TODO **Shared State in Actor Models**
Here focus on distribution, but also useful for concurrency. Similar to concurrent revisions [4]

8. CONCLUSION

In this paper we presented a way to add shared state to the actor model. This is done by optimistically replicating a special type of object, called a `repliq`. By doing so, we introduced optimistic replication as a first class value to an actor language. Because the shared state and operations are replicated and can always be accessed and executed locally, the isolated properties of the actor model are maintained. We demonstrated how this reduces the overhead of programming offline available applications by implementing a chat example. Furthermore, we provided the core of the implementation using the meta protocol in AmbientTalk. This implementation demonstrates how to handle logging, ordering and replaying of operations in order to maintain eventual consistency of the replicated objects. It also solves a common problem with optimistic replication, which is the composition of replicated objects. Although the composition is restricted to repliq objects of which the master object is the same actor. By embedding replication into the object heap, an automatic pub/sub system can be maintained such that replicas receive updates as long as a reference to that replica exists. Finally, we also showed that the replay of invocation model can be used to add security by adding access control checks in the repliq methods.

9. REFERENCES

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *ACM SIGPLAN Notices*, volume 39, pages 331–344. ACM, 2004.

[3] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *European Symposium on Programming*, pages 67–86. Springer, 2012.

[5] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[8] R. Guerraoui and M. E. Fayad. Oo distributed programming is not distributed oo programming. *Communications of the ACM*, 42(4):101–104, 1999.

[9] J. D. Koster, S. Marr, T. V. Cutsem, and T. DŒşHondt. Domains: Sharing state in the communicating event-loop actor model. *Computer Languages, Systems & Structures*, 45:132 – 160, 2016.

[10] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in e as plan coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag.

[11] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, Mar. 2005.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] S. Tasharofi, P. Dinges, and R. E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *European Conference on Object-Oriented Programming*, pages 302–326. Springer, 2013.

[14] T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, SCCC '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[15] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, Dec. 2001.