Passive Replicator: A Design Pattern for Object Replication

Teresa Gonçalves and António Rito Silva
INESC/IST Technical University of Lisbon
R. Alves Redol n°9, 1000 Lisboa, PORTUGAL
Tel: +351-1-3100287, Fax: +351-1-3145843
{tsg,ars}@albertina.inesc.pt

Abstract

This paper describes a pattern for passive object replication in distributed systems. The pattern provides support for the representation of replicated objects, the management of replicated objects and the implementation of several replication policies. It decouples replication from functionality and distribution. It supports different replica consistency criteria.

1 Intent

The intent of the Passive Replicator Pattern is to allow the use of different replication policies for object replication according to the needs of the application being developed in terms of performance/consistency. It allows the support of different levels of performance and different consistency criteria. It decouples replication from functionality and distribution.

2 Motivation

2.1 Background

The reasons for developing this pattern are associated with the increasing use of distributed systems and the associated requirements for application programmers and framework users and developers. Replication is a technique widely used in distributed systems for increasing availability, performance and fault-tolerance [AAH96]. Replication in object-oriented systems is used in application areas such as:

- Replicated data management.
- Management of replicated computations.
- Distributed shared memory.

Objects are characterized by a set of values and its methods, so object replication can be seen as a combination of data replication and execution replication. Object replication models result from combining data replication models with execution replication models. In Object Oriented Systems, data replication models are concerned with the consistency/availability of objects state and execution replication models are concerned with application fault-tolerance. If data is replicated at different sites of a network and if sites are failure-independent then data availability is enhanced. Under normal circumstances sites with replicated data can maintain data consistency, i.e. all replicated data agrees on a single data value at any time (known as strong consistency). If there are communication failures there is a potential for inconsistent data. Data replication models vary according to the consistency and data availability they offer.

Data replication models can be classified into pessimistic (strong consistency), controlled inconsistency (when updates are done to a replica and data consistency of the other replicas does not need to be restored immediately) and optimistic (no consistency is assured, i.e. updates to replicas can be done anywhere, anytime leading to data inconsistency between replicas).

Execution replication models enhance availability, tolerate `Fail-stop` failures and if the execution is replicated and occurs in parallel it also provides guarantees of correct request

processing because it can offers guarantees against `Byzantine-failures`, also called arbitrary fails. Execution replication models can be classified into passive replication (Primary-backup approach) [Budhiraja 93a] and active replication (State-machine approach) [Budhiraja 93b]. In passive replication, methods are executed by one object at one node. The state of the object is mirrored to other object replicas at certain execution points (checkpointing). A usual checkpoint is the the end of the method execution. In active replication methods, are executed by all the replicas.

Some data replication policies can be combined with the passive replication execution: `ROWA (read-one write all)` where a read request can be satisfied by reading any single copy of a data item and a write request requires all copies to be written; `Primary-Copy` where a read request can be satisfied by any copy of a data item and a write request can only be satisfied by the primary copy; `Quorum-Based` where copies of a data item may be given a certain number of votes each,a read/write request can only be satisfied if the sum of the votes of the available replicas is equal or higher than the read/write `quorum` value.

2.2 Example

Consider the example of a Shared Agenda with the following functionality:

- users can create, update and delete appointments and can also create, confirm, shift and cancel meetings, i.e. users can manage appointments and meetings. The difference between meetings and appointments is that a meeting requires several participants, its creator and other participants and an appointment requires a single participant.
- the agenda manager can create new users and delete existing ones.

Consider a Shared Agenda composed by two kinds of sessions:

- **Manager session:** that holds an Agenda Manager and allows the creation of new users and the deletion of existing users.
- **Agenda session:** that allows users to manage appointments and meetings.

Figure 1 illustrates the structure of the Shared Agenda using a Booch class diagram [Booch 94]. An `Agenda Manager` is responsible for the manipulation of several agenda users. Each `Agenda User` can have several `Agenda Appointments` and several `Agenda Meetings`. Both `Manager Session` and `Agenda Session` will use the data of the `Agenda Manager`.

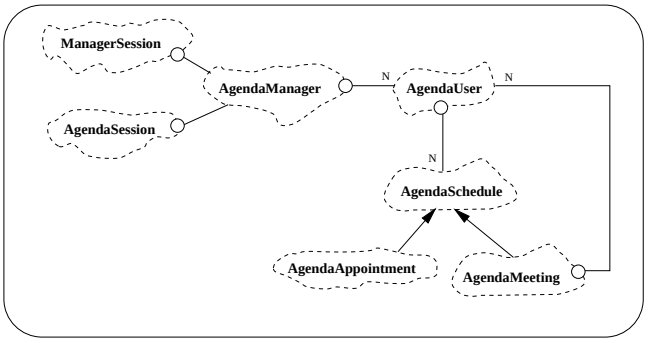

Figure 1: Diagram Structure of Shared Agenda

Figure 2 shows an example of the Shared Agenda where there are two `Agenda Sessions` and a `Manager Session` using a common `Agenda Manager`. `Agenda Sessions` should allow the

consulting of `Agenda Meetings`, in which several users participates, without delays caused by the use of remote data objects. So each `Agenda Session` should have local replicas of the meeting data (in its own address space) allowing fast read access to the meeting data. Modifications on the meeting data associated with an `Agenda Session` should be propagated to the `Agenda Manager` associated with the `Manager Session` and to others `Agenda Sessions` local replicas of meeting data.

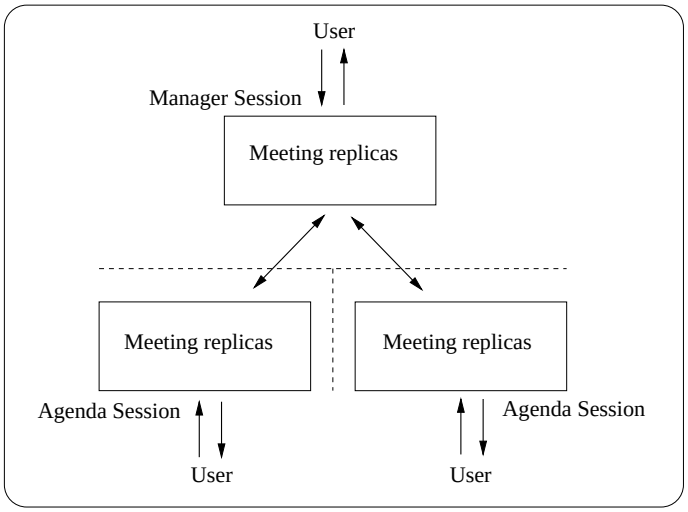

Figure 2: Shared Agenda

2.3 Problem

In many distributed systems, data access performance is critical. This is often solved using replication-based approaches. Because the requirements for data consistency vary depending on applications semantics, it is necessary to be able to support different replication policies. How can you design an approach to replication that supports different policies?

2.4 Forces

The problem must consider the following forces:

- **Access performance versus data consistency:** The data access performance for read/ write operations can vary according to the data consistency expected by the user of the application. Read access performance can be improved by reading local replicas of data. If strong consistency is desired the write access on a replica implies that data updates resulting from the write operation are propagated immediately to the other replicas of the replica set. To achieve strong consistency it is also necessary to assure that operations are serialized.

  If controlled inconsistency models are used the write access on a replica does not imply the immediate update of the other replicas. Data replicas of the same object can be inconsistent allowing the user to specify when replicas consistency will be restored and the resolution of updating conflicts. Optimistic replication models allow write operations to be executed on local replicas achieving write access performance.

  The Shared Agenda is an example of a distributed application where read access performance to meeting data can be improved by reading local data instead of remote data. This example requires strong consistency of meeting data. Updates to meeting data should

be propagated immediately to other replicas because users should always have updated information on their meetings.

Geographical Information Systems browsers deal with large amounts of data transfer. This is an example of a distributed application where keeping data locally (local data caching) means much faster access to data. Local cache copies are only updated when the client requires an updated data version. In this example data inconsistency is prefered as a means of gaining performance.

- **Different replication policies versus single replication policy:** Application requirements may vary demanding the use of different replication policies for different objects. Therefore the use of single policy solutions may not be appropriate.

  Consider a Document Management System where it is possible to have different document types: private (only document creator can view and edit it), shared-read (several users can view the document but only the document creator can edit it), shared (several users can both view and edit the document). This is an example of a distributed application where different replication policies can be used to implement the shared-read and shared document types.

- **Flexibility versus efficiency:** The decoupling of replication issues from object functionality allows adding and replacing replication policies without changing the object's implementation. This increased flexibility has costs in terms of efficiency through the use of indirections.

2.5 Solution

The solution takes into account the forces named above.

The solution is a three-level structure for object replication, namely Policy-Generic, Policy-Specific and Object-Specific. The solution provides an abstraction that allows the use of several replication policies for the support of object replication as an approach to increase performance. That abstraction must allow the definition of different replication policies that may implement different data models of consistency. The Policy-Generic level of the pattern abstracts what is common between replication solutions that implement different replication consistency criteria. It decouples the above abstraction from policies implementation. The Policy-Specific level corresponds to the implementation of replication policies decoupled from application's functionality. The Object-Specific level corresponds to the integration of application's functionality with replication solutions implemented by the above levels.

3 Applicability

Use this pattern when:

- it is necessary to test several different replication policies with minor changes to the code.
- it is necessary to use object-specific replication criteria. This pattern allows data inconsistency between replicas of the same object allowing the user to specify when replicas consistency is restored. For example the user can decided to restore replicas consistency only when executing a particular operation.

4 Structure and Participants

The structure of the pattern is illustrated in Figure 3 as a Booch class diagram using the notation referred in [Booch 94].

The pattern is structured into three different levels:

- **Policy-Generic:** This level abstracts what is common to replication policies and should be supported by a framework concerned with object replication.
- **Policy-Specific:** This level can be customized by programmers and allows the development of several replication policies that may use different consistency criteria.
- **Object-Specific:** This level is object specific, i.e. at this level classes implementation are specified and are integrated with the replication concern by specifying subclasses of `Specific Replica Manager` and `Operation`.

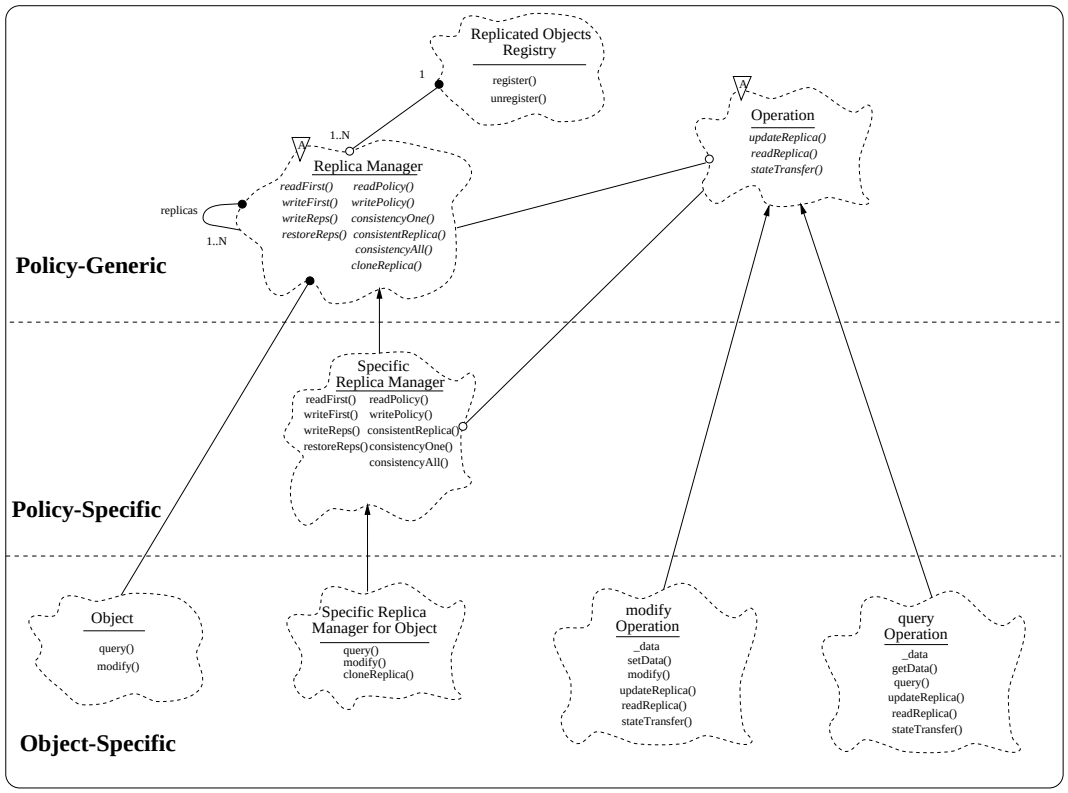

Figure 3: Diagram Structure of the Passive Object Replication Pattern

The participants are described below:

- **Replicated Objects Registry:** It maintains a registration of the sets of replicated objects. Replicas belonging to the same replica set are identified by a common designator the `Replica Object Identifier`. This class is used by the `Replica Manager` objects to interact with the set of replicas.
- **Replica Manager:** This class encapsulates replicated objects and abstracts different replication policies, that may use different consistency criteria, to control the consistency/-availability of replicated objects. There is one replica manager object associated with each replicated object. The replicated objects are isolated from replication issues which are handled by the corresponding replica manager object. The replica manager class has also a list of references to all the other replica managers of the replica set. This list is used basically to provide support to the implementation of replication policies.

  The `readFirst` operation is used to calculate one replica object from the replica set on which read methods will be executed. The `writeFirst` operation is similar to the `readFirst` operation but calculates the replica object on which update methods will be

executed. The `writeReps` operation is used to calculate the replicas on which to checkpoint the object state of the object updated by the `update` method. The `restoreReps` is used to calculate the replicas that may be inconsistent and should be updated when it is necessary to restore replicas consistency. The `readPolicy` and `writePolicy` are template methods used to implement respectively the read operation and write operation of a replication policy. The `consistentReplica` is used to find out a consistent replica from which to copy the consistent state when restoring replicas consistency. The `consistencyAll` operation is used to restore the consistency of all replicas. The `consistencyOne` operation is used to restore the consistency of one replica. The `cloneReplica` is used to support the cloning of the object associated with one `ReplicaManager` object.

- **Object:** This class represents a specific (concrete) class which can be replicated. The replicas of an object are instances of this class. Replicas of the same object are identified by a common designator the `Replica Object Identifier` and belong to the same replica set. The data associated with an object is always manipulated by its methods. The methods can be classified into query (read) methods and update (write) methods.

- **Specific Replica Manager:** This class implements a specific replication policy providing implementations for the `readFirst`, `writeFirst`, `writeReps`, `restoreReps`, `readPolicy`, `writePolicy`, `consistencyOne`, `consistencyAll` and `consistentReplica` operations of the specific policy. The specific policy data is defined here. There can be one or more of these classes by policy.

- **Operation:** This class is used by a `Specific Replica Manager` class to encapsulate the execution of different methods (`modify` and `query`) of replicated objects as objects. This class has methods like `updateReplica`, `readReplica` and `stateTransfer`. The method `updateReplica` is responsible for executing a `modify` method on the replicated object and saving the result on its data fields. The method `readReplica` is responsible for executing a `query` method. The method `stateTransfer` is responsible for state transfer, i.e. copying the result saved in the data fields to other replicated objects.

- **modify Operation:** This class is a derived class of the `Operation` class. The fields of this class are the fields updated by the `modify` operation. This class has to implement the methods for setting those fields `setData`. The method `readReplica` is redefined as a empty method.

- **query Operation:** This class is also a derived class of the `Operation` class. The fields of this class are the fields used by the `query` operation and this fields are returned by the execution of the `query`. This class redefines the abstract methods `updateReplica` and `stateTransfer` as empty methods.

- **Specific Replica Manager for Object class:** This class implements the interface of the `Object` class. An application that wants to use the replicated object uses the corresponding object of the `Specific Replica Manager for Object` Class. This class controls the access to the replicated object. This class redefines the abstract method `cloneReplica` for cloning an object.

5 Collaborations

The collaboration between participants can be seen by analyzing four main aspects: adding replicated objects, executing non-updating and updating methods on replicated objects and restoring the consistency of inconsistent replicated objects.

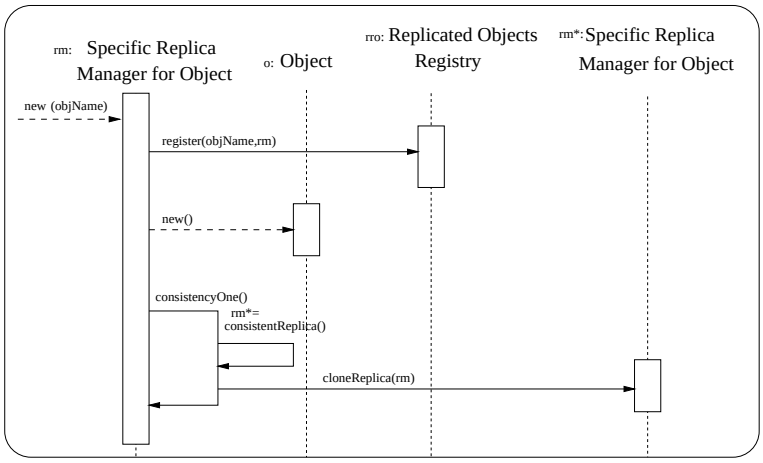

Figure 4: Collaboration among participants for adding a replicated object

5.1 Adding Replicated Objects

Figure 4 illustrates the collaboration between participants in the *Passive Replication* pattern for creating a replicated object. This figure shows that the process of creating a replicated object corresponds basically to the creation of a replica manager, the registration of the object in an instance of the class `Replicated Objects Registry`, the creation of an object that will be associated with the above replica manager and the initialization of the replica calling the `consistencyOne` method.

When the `Specific Replica Manager for Object` object calls the `register` operation besides registering the `Specific Replica Manager for Object` class instance it also adds to other replica managers (field replicas) of the "same" object a reference to the registered replica manager.

The method `consistencyOne` is responsible for cloning an object initializing the fields of the replicated object, when it joins to the replica set, reading up-to-date field values by finding out the more recent replica. The method implementation is dependent on the policy used. In the case of *primary-copy* replication policy it corresponds to initializing the replica with the values obtained by reading the primary replica.

5.2 Executing Updating Methods

Figure 5 illustrates the collaboration between participants in the Passive Replication pattern for executing an update method on a set of replicated objects. Figure 5 shows that only one replica of the object executes the method invocation, updating the remaining replicas after the end of the method invocation execution. These collaborations are divided into two phases:

1. **Executing the method invocation by one replicated object:** The execution of an updating method using the *Passive Object Replication* begins with a call of the updating member function on an instance of the `Specific Replica Manager for Object` class. The instance of the `Specific Replica Manager for Object` is selected by invoking the operation `writeFirst`. This function selects the replica on which the update method operation is executed. Its result depends on the replication policy being used.

   The update method function should be implemented by creating a new instance of the `Operation modify` class with parameters values equal to the update parameters followed by calling the `writePolicy` method with the created instance of `Operation modify` as parameter. The implementation of the `writePolicy` method corresponds to reading the update data information and executing the updating method on the replicated object

associated with the `Specific Replica Manager for Object` instance and saving the result on the data structure associated with the `Operation modify` class instance.

2. **State transferring for the remaining objects of the replica set:** This phase corresponds to updating a set of the remaining replicas of the replicated object updated in phase `1`, in order to preserve the consistency criteria. It calls the `writeReps` method which returns the remaining replicas to be updated. This method returns different results depending on the replication policy being used. Each one of the replicas is updated with the values recorded on the `modify Operation` structure on phase `1`.

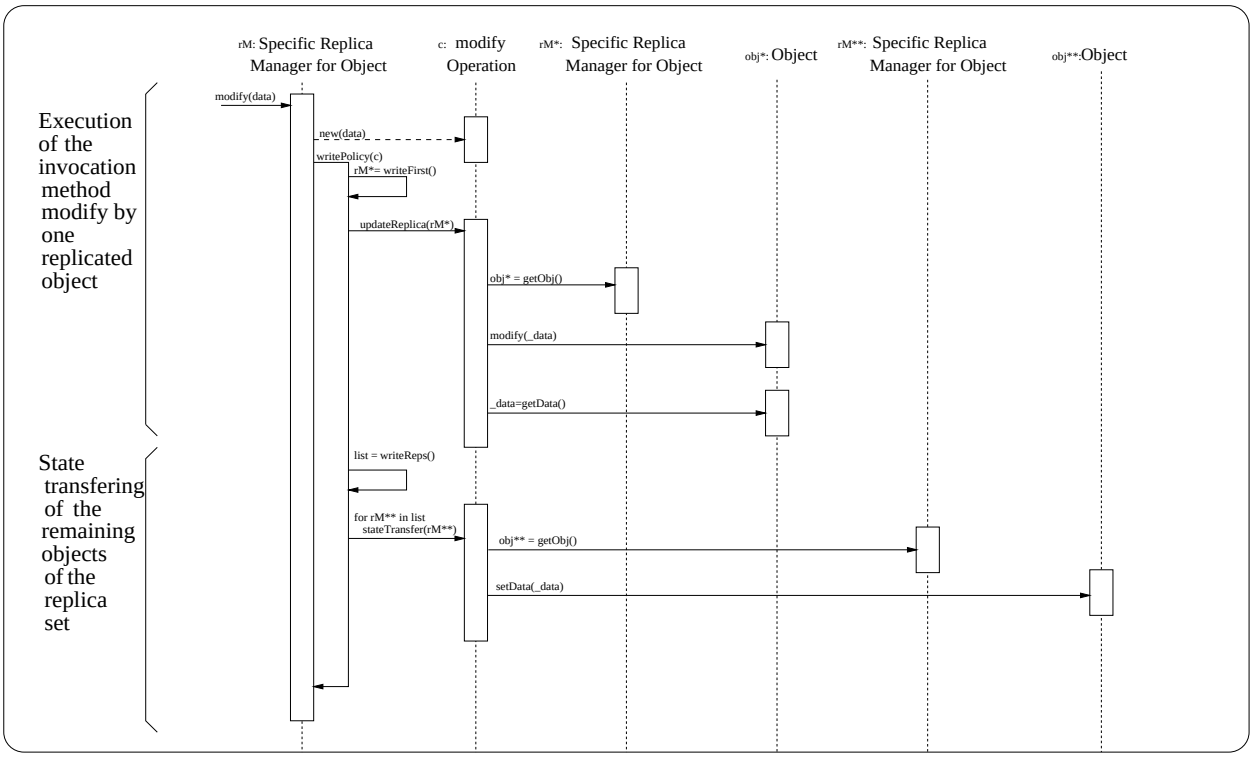

Figure 5: Collaboration among participants for an updating method

5.3 Executing Querying Methods

Figure 6 illustrates the collaboration between participants in the Passive Replication pattern for executing a non-update method. The execution of the operation corresponds to selecting a replica on which the operation will be executed. The selection of the replica depends on the replication policy used. In a primary-copy replication policy it can be the replica itself while in a quorum replication policy it is a replica belonging to the replica quorum for the read operation which has the greatest update number.

5.4 Restoring Replicas Consistency

Figure 7 illustrates the collaboration between participants in the *Passive Replicator* pattern for restoring replicas consistency. The set of replicas updated by phase `2` of an update operation may not be equal to all the remaining replicas of one replicated object updated by phase `1`. As such, some inconsistency between replicas of the same object may exist temporarily. The method `consistencyAll` is responsible for cloning an object restoring the fields of replicated objects,

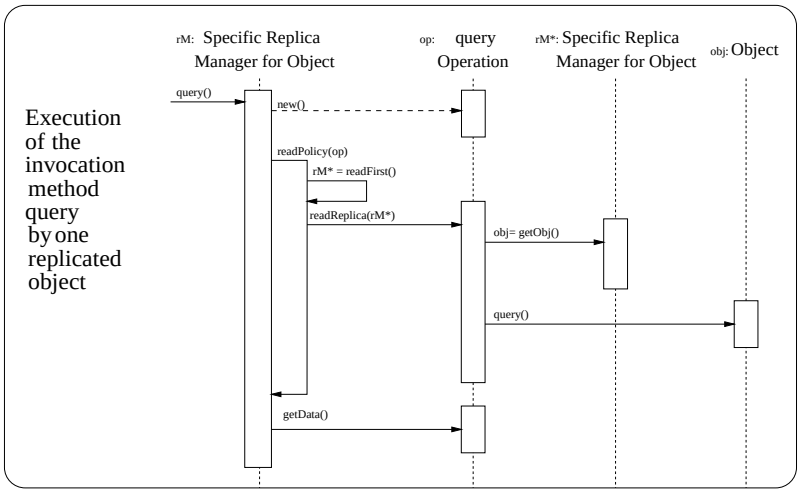

Figure 6: Collaboration among participants for a query method

when it is executed. This method calls the `consistentReplica` to find out from which replica it should copy the state. It calls the `restoreReps` to find out on which replicas consistency will be restored.

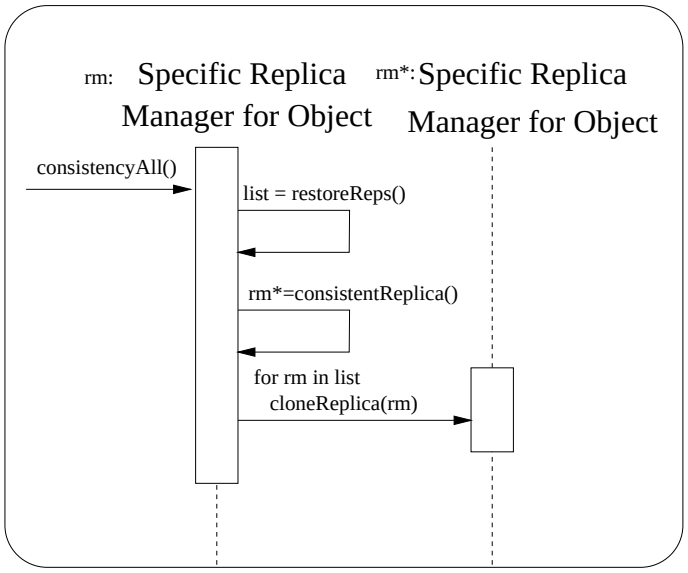

Figure 7: Collaboration among participants for restoring replicas consistency

6 Consequences

The pattern provides the following benefits:

- **Low amount of application specific code related with the replication of objects :** To replicate objects using an implemented specific replication policy the programmer only needs to develop the `Specific Replica Manager for Object` class and the `Operation` class for each method operation of the class to which the replicated object belongs. The development of this classes requires a low amount of code.

- **Independence of the replication policy used:** The replication of objects is independent of the replication policy used. It is possible to try different replication policies without changing the application specific code related with the replication of objects, namely the code necessary for the implementation of the classes `Specific Replica Manager for Object` and `Operation`. For testing the application with different replication policies it is only necessary to change the `Specific Replica Manager` classes.

- **Clear separation between the concrete objects and the Specific Replica Managers for Objects:** The concrete objects do not know that they are replicated. The pattern encapsulates the concrete objects from replication issues such as the management of replicated objects and the replication policy used.

- **Decoupling from synchronization:** The implementation of some replication policies requires atomicity of the `writePolicy` operation when implementing some replication policies, i.e either the `writePolicy` operation succeeds in all replicated objects or the operation aborts. The synchronization is necessary also to assure that the result of executing two concurrent operations is equal to executing the two operations sequentially. The replication concern is presented in this paper isolated from operation atomicity and concurrency control issues since these can be seen as separate concerns which can be incorporated with the replication concern.

The pattern has the following drawbacks:

- **Great number of classes:** For each update and query method of each class it is necessary to create a derived class of the `Operation` class. This can lead to a great number of classes.

- **Fault-Tolerance:** The *Pattern for Passive Object Replication* is not fault-tolerant. This pattern does not tolerate *fail-stop* fails. The replication concern is presented here isolated from fault-tolerance. This pattern can be used to improve read performance.

7 Implementation

This section describes several implementation variations of the pattern.

7.1 Policies

- **Policy specific:** Passive replication can use several different replication policies and models. Replication models describes the user´s expectations concerning consistency/availability of replicated objects:

  - *pessimistic:* the result of a reading operation on replicated objects always returns the result of the last update. To implement a pessimistic policy it is necessary that when changes are done to a replicated object using, `passive replication`, that those changes are done atomically to all remaining replicated objects of the replica set. To implement a particular policy one should define a derived class from `Replica Manager`, i.e define a `Specific Replica Manager` class and redefine the virtual methods of the super class.

    To implement the `ROWA` (Read-One-Write-All) replication policy the `readPolicy` should correspond to reading the replica accessed by the `query` method and the `writePolicy` should correspond to writing the replica accessed by the `update method` and transferring the updated state to the remaining replicas. The `readPolicy` for this policy corresponds to calling the `readReplica` method with the parameter returned by the `readFirst` method. The `readFirst` operation should return the Replica Manager accessed by `query`. The `writePolicy` corresponds to calling the

`updateReplica` method with the parameter returned by the `writeFirst` method and to call the `stateTransfer` method with the parameter returned by the `writeReps` method. The `writeFirst` operation should return the Replica Manager accessed by the `update`. The `writeReps` method should return all the remaining replicas of the replica set which were not updated by the update call. The `restoreReps` and `consistencyAll` should be defined as empty since this policy does not allow inconsistency between replicas.

- *controlled inconsistency:* when an update operation on a replicated object does not immediately imply the update of the others replicas from the replicated object set. A limit is established concerning data inconsistency above which consistency is repaired calling the `consistencyAll` or `consistencyOne` method of the `Specific Replica Manager` Class. Consider the implementation of a combined policy such as Primary-Copy with controlled inconsistency where updates are always done on the primary-copy and the consistency between primary-copy and secondary-copies is restored only when executing a update operation `X`. To implement this replication policy the `readFirst` and `writeFirst` operations should return respectively the accessed Replica Manager by the query or update method and the Replica Manager associated with the primary-copy. The `writeReps` operation should return an empty list. This implies that the execution of update methods is not propagated to the remaining replicas. The implementation of the `X` operation can be implemented as the execution of the operation preceded by calling the `consistencyAll` operation. The `restoreReps` operation returns all the Replica Managers associated with the secondary-copies. The `consistencyAll` operation is implemented by copying the state of the primary-copy object to all the replicas returned by the `restoreReps`.
- *optimistic:* when an update operation can occur anywhere anytime. The availability in this case is higher and the consistency is lower. To implement a particular policy one should define a derived class from `Replica Manager`, i.e. define a `Specific Replica Manager` class and redefine the virtual methods of the super class. In this case replica updates do not need to be consistent. In case of conflicting-updates the conflict resolution it is left to the implementation of `consistencyAll`.

- **Object specific:** The definition of Objects is isolated from the replication issues and these can be defined independently of the rest. The integration of Objects with the replication concern is done through the definition of a `Specific Replica Manager for Object` class with the same interface of the Object. The implementation of query and update methods corresponds to creating new `Operations`. It is necessary to implement the `updateReplica`, `readReplica` and `stateTransfer` methods as illustrated in figure 5 and 6.

7.2 Distribution integration

Distribution can be added to the pattern to support distributed replicated objects. The `Replica Manager` class can be instrumented in order to support distribution of replicated objects. This instrumentation consists of combining the `Replica Manager` class with distributed proxies [Silva 97] such that the communication between distributed Replica Managers is done through distributed proxies.

There are two possible ways for adding the distribution concern to the replication concern:

- by using delegation. The `Replica Manager` class will reference a distributed proxy. The distributed proxy will be responsible by all the low level details involved with communications encapsulating the communication mechanisms.
- by using inheritance. It will be needed to create a new class `Distributed Replica Manager` derived from both `Replica Manager` and `Proxy` classes. Replication and distribution can be achieved by using the `Distributed Replica Manager` class.

7.3 Synchronization integration

Synchronization can be added to the pattern to support the execution of concurrent operations on replicated objects. This can be done by using a framework that supports concurrency mechanisms such as locks or time stamps or by using the *Customizable Object Synchronization* pattern [Silva 96].

8 Sample Code

8.1 Shared Agenda Application

In the Shared Agenda Application, replication will be applied to the agenda meetings.

The class `AgendaMeetingInt` represents the interface abstraction of class `AgendaMeeting`.

```
class AgendaMeetingInt
{
  public:
   virtual void addAgendaUser (AgendaUser* user) = 0;
   virtual AgendaUser* getAgendaOwner(void) = 0;
};
```

Class `ReplicaManagerPAgendaMeeting` corresponds to the implementation of a `Specific Replica Manager for Object` class, supporting the replication of class `AgendaMeeting` using the Primary-Copy replication policy. This class implements the interface of the `AgendaMeeting` class and accesses by the Shared Agenda Application are done through instances of this class.

```
class ReplicaManagerPAgendaMeeting:public PrimaryReplicaManager<AgendaMeeting>,public AgendaMeetingInt
{
  public:
    ReplicaManagerPAgendaMeeting (AgendaUser*,AgendaDate&,AgendaTime&, ReplicaObjectIdentifier);
    void addAgendaUser (AgendaUser*);
    AgendaUser* getAgendaOwner (void);
};
```

The update (write) method `addAgendaUser` is implemented by the replica manager as below:

```
void ReplicaManagerPAgendaMeeting::addAgendaUser(AgendaUser* agU)
{
  addAgendaUserOperation c(agU);

  writePolicy(&c);
}
```

The query (read) method `getAgendaOwner` is implemented by:

```
AgendaUser* ReplicaManagerPAgendaMeeting::getAgendaOwner(void)
{
  getAgendaOwnerOperation c;
  AgendaUser* owner;

  readPolicy(&c);
  owner = c.getAgendaOwner();
  return owner;
}
```

To encapsulate the execution of the methods `addAgendaUser` and `getAgendaOwner` the classes `addAgendaUserOperation` and `getAgendaOwnerOperation` had to be implemented one for each method.

```
class addAgendaUserOperation: public Operation <AgendaMeeting>
{
  private:
   AgendaUser* _aUser;
  public:
    addAgendaUserOperation (AgendaUser*);
    void updateReplica (ReplicaManager<AgendaMeeting>*);
    void stateTransfer (ReplicaManager<AgendaMeeting>*);
    void readReplica (ReplicaManager<AgendaMeeting>*);
};
```

```
class getAgendaOwnerOperation: public Operation <AgendaMeeting>
{
  private:
   AgendaUser* _aOwner;
  public:
    getAgendaOwnerOperation (void);
    AgendaUser* getAgendaOwner (void);
    void updateReplica (ReplicaManager<AgendaMeeting>*);
    void stateTransfer (ReplicaManager<AgendaMeeting>*);
    void readReplica (ReplicaManager<AgendaMeeting>*);
};
```

The method `readReplica` of class `addAgendaUserOperation` is implemented as an empty method. The methods `updateReplica` and `stateTransfer` are implemented as below:

```
void addAgendaUserOperation::updateReplica(ReplicaManager<AgendaMeeting>* rM)
{
   AgendaMeeting* obj;

   obj = rM->getObj();
   obj->addAgendaUser(_aUser);
}

void addAgendaUserOperation::stateTransfer(ReplicaManager<AgendaMeeting>* rM)
{
   AgendaMeeting* obj;

   obj = rM->getObj();
   obj->addAgendaUser(_aUser);
}
```

The methods `updateReplica` and `stateTransfer` of class `getAgendaOwnerOperation` are implemented as empty methods. The method of the class `readReplica` is implemented as below:

```
void getAgendaOwnerOperation::readReplica(ReplicaManager<AgendaMeeting>* rM)
{
  AgendaMeeting* obj;

  obj = rM->getObj();
  _aOwner = obj->getAgendaOwner();
}
```

9 Known Uses

The usage of replication as a means of increasing data access performance is widely used in Distributed Database Systems and in Distributed Systems. Most of the solutions provided in these systems restrict the use of replication policies to a set of policies that can be supported by the policy-specific level of this pattern.

Arjuna [MCL] is an example of an object-oriented programming system that supports object replication to tolerate fails. The Arjuna supports the *pessimistic* data replication model and the `passive` and `active` execution models.

This pattern was developed in the DASCo [Silva 95] scope. The replication concern is considered with DASCo concerns of synchronization, distribution and naming.

10 Related Patterns

- Simple Shared Object Pattern [Ott 96] is a pattern for object replication across different address spaces using the primary-copy policy for replicating the objects with immediate update (strong consistency). This pattern allows the use of the primary-copy replication policy decoupled from application's functionality and distribution. This pattern can be seen as a particular implementation of the Passive Replicator pattern integrated with distribution where the policy used is the primary-copy policy and the copies belong to

different address spaces. The Passive Replicator pattern allows, besides the primary-copy policy the use of other replication policies that vary according to data consistency/access performance that they offer.

- Observer Pattern [Gamma 95] is a pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated independently. The primary-copy replication policy can be related with the Observer pattern where the primary replica is the `subject` and the other replicas are the `observers`. This pattern can be seen as a particular implementation of the Passive Replicator pattern where the policy used is the primary-copy and distribution is not an issue.

  This pattern does not decouple subject functionality from observer notification. The Passive Replicator pattern allows besides the implementation of the primary copy policy the use of other replication policies and the consideration of the distribution concern.

- Command Pattern [Gamma 95] is used to implement the `Operation` class as an abstract class which declares an interface for executing the `Operation` methods. It is used to decouple the invocation from the knowledge to perform the execution.

- Singleton Pattern [Gamma 95] is used in the implementation of the `Replicated Objects Registry` class.

11 Acknowledgments

The authors would like to thank Neil Harrison for his valuable comments during the shepherding process.

References

[AAH96] Bharat B. Bhargava Abdelsalam A. Helal, Adbelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.

[Booch 94] Grady Booch. *Object-Oriented Analyis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[Budhiraja 93a] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S.J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 8. Addison-Wesley, 1993.

[Budhiraja 93b] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Replication management using the state machine approach. In S.J. Mullender, editor, *Distributed Systems, 2nd Edition*, ACM-Press, chapter 7. Addison-Wesley, 1993.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[MCL] Santosh K. Shrivastava Mark C. Little. *Replicated K-Resilent Objects in Arjuna*. IEEE.

[Ott 96] Robert Ott. Simple shared object. *Object Currents*, 1(3), March 1996.

[Silva 95] António Rito Silva, Pedro Sousa, and José Alves Marques. Development of Distributed Applications with Separation of Concerns. In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference*, pages 168–177, Brisbane, Australia, December 1995.

[Silva 96] António Rito Silva, João Pereira, and José Alves Marques. Customizable Object Synchronization Pattern. In *The 1st European Conference on Pattern Languages of Programming, EuroPLoP '96 (Washington University technical report #WUCS-97-07)*, Kloster Irsee, Germany, July 1996.

[Silva 97] António Rito Silva, Francisco Assis Rosa, and Teresa Gonçalves. Distributed Proxy: A Design Pattern for Distributed Object Communication, September 1997. Submitted to the Fourth Conference on Pattern Languages of Programs, PLoP '97.