State transfer for active replicas in CORBA

Luis M. Peña[1], Juan Pavón[1]

[1] Dep. Sistemas Informáticos y Programación

Universidad Complutense de Madrid
Ciudad Universitaria s/n
28040 Madrid, Spain

luismp@skynet.be
jpavon@sip.ucm.es

Abstract. Current Fault Tolerant CORBA specification by OMG addresses fault detection, notification, analysis and recovery for object replicas in a distributed environment. This paper addresses the open issue of state transfer in groups of active replicas under an application-controlled consistency style, which must be flexible to cope with different application requirements, such as transfer of large states, non-blocking during state transfer, and coordinator election.

1 Introduction

The Object Management Group (OMG) has just approved the standard for Fault Tolerant CORBA [OMG99]. This deals with replication of objects (instead of servers) in a CORBA environment. The standard tries to support a wide range of requirements, such as passive and active replication modes, control of the creation and the consistency of the replicas by the application or the fault tolerance infrastructure, automatic or application-driven checkpointing, logging and recovery from faults, while keeping minimum impact in the ORB.

This paper addresses an open issue in the specification: the state transfer in groups of active replicas under an application-controlled consistency style, which must be flexible to cope with different application requirements, such as transfer of state in one step or in several (for large states), blocking or non-blocking of server object groups during state transfer, and election of coordinator for the state transfer.

2 Fault Tolerant CORBA

The key component of the Fault Tolerant CORBA architecture for management of the object group is the *ReplicationManager*, which should be also replicated to be fault tolerant. The

ReplicationManager interface inherits three interfaces: *PropertyManager*, *GenericFactory*, and *ObjectGroupManager*.

The *PropertyManager* interface allows the user to define fault tolerance properties of object groups, that basically determine the responsibilities of the infrastructure. The most relevant properties are *ReplicationStyle*, *MembershipStyle*, and *ConsistencyStyle*.

The *ReplicationStyle* sets how the state is maintained on the replicas; it can be *stateless*, *cold passive*, *warm passive*, *active* and *active with voting* (this last one is not supported in the current specification). Passive replications are based on one single member, the primary member, processing every method invocation to the group, which contains as well backup members. In the case of warm replication, backup members are updated periodically with the state of the primary replica; in the case of cold replication, this update is only done when needed for recovery. Under active replication, every replica processes each method invocation, and duplicated requests or replies are suppressed. This style requires a group working under the virtual synchrony model [Birman87].

The *MembershipStyle* determines whether group membership is under the responsibility of the application (*application-controlled*) or the infrastructure (*infrastructure-controlled*). In the latter case, the *ReplicationManager* invokes the required factories to create the members needed to satisfy the group properties, like the initial or the minimum number of replicas. In the first case, the *ReplicationManager* provides the operations to do manually the insertion and extraction of members in groups, to define the primary member and the locations of each group member.

The *ConsistencyStyle* can also be *application-controlled* or *infrastructure-controlled*. When controlled by the application, this must maintain its own consistency at each moment: after each group invocation or when a new replica is added, which must therefore receive the appropriate group's state. Alternatively, when controlled by the Fault Tolerance infrastructure, the application group objects must just implement a basic interface, *Checkpointable*, which provides simple *setState* and *getState* methods.

```
interface Checkpointable {
    State get_state ();
    void set_state (in State s);
};

interface Updateable : Checkpointable {
    State get_update ();
    void set_update (in State s);
};
```

Optionally, objects can implement a specialized interface, *Updateable*, which also defines the operations *get_update*, *set_update*, to store and retrieve the state incrementally. The infrastructure keeps the checkpointing, logging, activation and recovery mechanisms required to maintain a strong consistency and to install the group's state on any new replica. This consistency means, under the passive replication, that after a state transfer to a backup member, this is consistent with the primary member (or the previous primary member, if cold replication).

Under active replication, the strong consistency requires that after each method invocation, every replica has the same state. If the replication is not stateless, the behaviour of each member must be deterministic and each member must start in the same state.

This specification means that when the Fault Tolerance infrastructure must keep strong consistency, replicas are periodically requested for their state. The infrastructure stores the messages processed by every replica and requests the state to the replicas according to the *CheckpointInterval* property (defined in units of 100 nanoseconds). When a new replica is included in the group, the state can be built from the last stored state and the log of messages that have been processed after the state was retrieved.

Cactus [Schlichting93] works under a similar scheme. Each replica must store periodically checkpoints of its state, and keep a trace of the messages processed from the last checkpoint. If a replica becomes active, it can build its state from those checkpoints. This technique is fast when replicas are down during short periods but it requires every replica to spend some processing time periodically, not only in case of a state transfer.

Most of the existing group communication systems with support for active replication perform the state transfer from another active replica. *Arjuna* [Parrington95] does automatically a state transfer when a new member joins a group. The state transfer is done in one step, blocking the whole system, and group members must just implement two state operations: *save_state, restore_state*. *Electra* [Maffeis97] uses the same approach, but with a more flexible scheme, allowing transfers in several pieces and still using a simple interface: *get_state, save_state*. In *Phoenix* [Malloth96], only the members involved in the transfer are blocked, but the transfer must be done in just one step, with a similar interface: *get_state, put_state*. *JavaGroups* does not invoke automatically the transfer; this is an operation to be called by the interested member, who can request the state from every group member or just from one, generally the oldest one. The member sending its state must first make a copy of its state, and while this copy is performed, the member is blocked. When the state is requested, the member just sends this temporary copy. It is a transfer in one step, and requires total ordering on the messages; the interface is slightly more complex, to do the state copy: *SaveState, GetState, SetState*.

Maestro, a *C++* interface for *Ensemble* [Hayden98] allows several approaches to the state transfer. A group can specify if it needs a transfer and, if so, it will be invoked automatically. Messages can be classified as safe or unsafe; in the former case, they can be sent during the transfer, otherwise they are blocked, but this means that the relative order between messages is lost. Transfers can be done in several steps, as opposed to the latest versions of Maestro, which include a new state transfer mechanism in one step. State members automatically send their state, being possible to identify the current one. Therefore, any member can update its state using the newest state. The protocol is simple, and has an acceptable performance when the state is small, groups have few members, and changes in the group membership are unusual.

This paper specifies a state transfer interface for those *active* CORBA groups under an application-controlled *ConsistencyStyle*. This interface must allow flexible transfers to be adapted to any generic group. The following section includes the requirements for the state transfer,

with the adaptations needed to the fault tolerance IDL specification, and the section after shows several use cases for a better understanding of this proposal.

3 State transfer requirements

On an active group, two cases require a state transfer. First, when one or more members join the group and need to receive the state from the old members. Second, when the group has been partitioned due to network problems, with the subgroups progressing independently, and the group is re-merged, making a state transfer necessary between the subgroups. This second possibility is not afforded in this paper for two reasons: its complexity and the extra requirements needed to keep the consistency (Fault Tolerance CORBA specification does not address object group partitioning either).

Although every member in the group has at any moment a specific state, only those members sharing a common, initialised state will be considered as *state members*. Those members whose state is not coherent with this global state are considered as *stateless members*.

The simplest state transfer from an active group to one or more joining members consists of the state being sent in one multicast message from one state member. The joining members receive it and the other active members see the message as the trigger to consider that the transfer has finished. The selection of the member that sends the state can be based on the properties of the *view synchrony* model as every group member receives the same ordered list of members, called a *view*. According to this, one possibility is to select the state member with the lowest range in the view. This member is the *coordinator* of the state transfer.

Several circumstances can make this scenario more complex. If the state to transfer is large enough, it should be sent in more than one message, in order to avoid blocking the coordinator while it builds the message with the state (as no updates can be performed on that member in the meantime). The possibility to send the state in several steps implies as well that a transfer can be cancelled due to a failure of the coordinator; in this case, a transfer synchronization is required. The interface should anyway stay simple if the transfer is done in one step.

Other point that can affect the performance of the group is the election of the transfer coordinator. An application can select one member to do every transfer. This member will normally perform worse that the other members, but frees them of doing this task. At the other extreme, a different member can be selected each time a transfer is required, to achieve load balance. Additionally, because a member can be selected to do several transfers, these transfers can be done concurrently or sequentially.

Finally, group blocking must be taken into account while the transfer is performed (i.e., the group does not accept method invocations while the state transfer is performed). If the coordinator can update its state during the state transfer, a re-synchronization will be needed. A simple way to avoid this additional complexity is blocking the group during the transfers, but the performance can degrade dramatically if state transfers are required very often.

3.1 Transfer synchronization

The basic functionality to keep consistency of state is covered by the existing *Checkpointable* interface, enough to get and set the state. However, if the state is transferred in several chunks, some synchronization is needed between the stateless member and the coordinator to identify the chunk being transferred; this synchronization is also needed to restart a previously cancelled transfer. Although the state chunks can include the required synchronization, to define a separate object for that purpose allows for a better and clearer design. This object is called *phase coordination*, and it is defined by the application; from the transfer system's point of view, this *phase* must just flag when the transfer has finished. An example of a *phase* object is one containing the number of chunks to transfer and the next chunk to be transferred, flagging the end of the state transfer when both numbers are equal.

```
abstract valuetype PhaseCoordination {
    boolean isTransferFinished ();
};
```

The definition of this object makes use of the Objects by Value functionality. If an interface would be used, the checking of the end of the transfer would need a remote call. If a structure were used, there would be no possibility to extend the type to include the specific application behaviour, as IDL structures do not allow inheritance.

When a state member is called to obtain its state, it must return the *PhaseCoordination* associated to that state, which is transferred to the joining member. As state transfers are now done in several steps, if the state member fails, a new transfer should be started from the beginning. In order to allow the continuation of a cancelled transfer, two new methods are added. *sync_transfer* is called on the stateless member to specify the next *PhaseCoordination* expected; this phase is passed to the coordinator as a parameter to a method *start_transfer* These methods are specified in the interface *StateHandler*, which has no connection with the *Checkpointable* interface. To specify when a group is using the basic functionality or the extended one proposed here, we define a group property *UseBasicStateTransfer*:

```
typedef boolean UseBasicStateTransfer;
interface StateHandler {
 void start_transfer (in Location joining_member,
                      inout PhaseCoordination phase);
 void sync_transfer  (in Location coordinator,
                      inout PhaseCoordination phase);
 State get_state (inout PhaseCoordination phase);
 void set_state (in State s,
                 in PhaseCoordination phase);
};
```

- *start_transfer*: call to the state member when a transfer is going to start, to allow any preprocessing that the application may need to perform, and to synchronize the transfer with the joining member. It returns a *phase* which will be used in the next call to *get_state*.

- *sync_transfer:* call to the stateless member when a transfer is going to start. If there was already a transfer and the previous coordinator failed, this method is called with the last received phase (a nil reference if it is the first transfer), and must return the next expected phase.
- *get_state:* the state will be requested while the *phase* returned does not flag the end of the transfer.
- *set_state:* called on the stateless member.

The previous interface is only valid when the transfer includes one coordinator and only one new member receiving the state. The reason is that the probability of having several members joining the group at the same time, and therefore targeting the same transfer, are usually very low. The coordinator could still be programmed to do several unrelated transfers at same time, but the performance increase obtained could not be appreciable, as the application must support several threads getting the state at the same time, and solve the potential synchronization problems.

3.2 Coordinator election

There are different strategies to select the state member that will be responsible for controlling a state transfer: either the joining member chooses its coordinator member (*pull transfer)*, or the state members decide which of them will make the transfer (*push transfer*). The *pull* protocol performs better [Peña99] when hardware broadcasts are supported, otherwise the *push* protocol is preferred[1].

The election of the coordinator in both cases can be done automatically by the state transfer infrastructure, or driven by the application. In the first case, the simplest way is to let any member be the coordinator. Another possibility is to associate weights to each member, allowing a load balance between the state members or to leave this task to one or several predefined members, those with higher weights.

If the application has the responsibility to select the coordinator, it must know the list of state members. The current Fault Tolerant CORBA specification defines an operation to retrieve the members belonging to one group (*location_of_members*), but this operation just associates statically a string (*Location* is defined as *CosNaming::Name*) to each member. Each member should therefore associate that location to a responsibility degree in the transfer task, and the association would be static, making it difficult to include new members in the group.

Our proposal introduces a new concept: *properties* associated to each member. If the application is going to choose the coordinator, it is presented with a list of the state members, along with their associated properties. If the election is automatic, each member can have a weight

[1] For this reason, we consider the service to be predefined as *push* or *pull*, a group cannot specify its preferred protocol.

associated, and that weight is used by the infrastructure to choose the coordinator. This weight behaves as a predefined property with name FT::COORDINATOR_WEIGHT.

When the coordinator is chosen by the application, its object members must implement the interface *CoordinatorElector*, which defines one method called *get_coordinator*: the locations of the state members are presented, and one location must be returned deterministically[2].

```
interface CoordinatorElector {
  Location get_coordinator (in Locations locations);
};
```

3.3 Member properties

The state of a group can be seen as the state of one member that is the only member in the group. However, properties are associated to each specific member, and should have no visibility outside the group. In other words, properties are not part of that state and no output from the group should be affected by those properties. Only group domain operations should be affected, like the choice of the member that must perform some specific task. Examples of such properties are the location of each member, or their weight or responsibility to perform specific tasks.

Other difference between the global state and the member properties is the required transfer. When a member joins a group, this group must transfer its state. However, the properties must be transmitted in both ways: a new member is considered to have properties but not an initial state.

The fault tolerance specification already defines a *Property* structure, perfectly valid for our purposes. The *PropertyManager* interface provides methods to set properties statically as defaults for any group created by that manager, or dynamically for a specific group.

```
typedef CosNaming::Name Name;
typedef any Value;
struct Property {
        Name nam;
        Value val;
};
typedef sequence <Property> Properties;
```

We define two additional properties, for the use of member properties and the coordinator election: *UseMemberProperties, CoordinatorElectionStyle*.

```
typedef boolean UseMemberProperties;
typedef long CoordinatorElectionStyleValue;
const CoordinatorElectionStyleValue
                        COORD_ELECTION_INF_CTRL = 0;
```

---

[2] On a pull protocol, this election doesn't have to be deterministic, as only one member, the one requesting the state, must select the coordinator.

```
const CoordinatorElectionStyleValue
                        COORD_ELECTION_APP_CTRL = 1;
```

The *PropertyManager* interface granularity allows the specification of properties for groups, not for the members themselves. However, it is possible and effective to specify a new predefined property that associates for each location a set of properties. If the group uses an *infrastructure-controlled* membership, the properties will be transferred to the factories creating the individual group members; otherwise, the application itself must specify the properties when the objects are created.

The *PropertyManager* interface includes a method to modify dynamically the properties of a group, but it would be impractical to modify the properties of a member through this method because the whole group's properties must be specified. Additionally, this operation would be prone to race conditions when several members in the group wanted to modify their own properties.

Therefore, the *PropertyManager* can be used to specify the initial properties of the group members, and the factories (interface *GenericFactory*) receive the properties of a member when this is created.

```
interface GenericFactory {
   Object create_object (in TypeId type_id,
                         in Criteria the_criteria,
                         out FactoryCreationId id);
   void delete_object (in FactoryCreationId id);
};
```

In our proposal, these properties are included in the *Criteria* parameter in an entry named FT::FT_MEMBER_PROPERTIES: a member only receives its own properties, the other's are received during the state transfer. Properties are received and can be dynamically changed through the interface *PropertyHandler*, to be implemented by the object members.

```
interface PropertyHandler {
   Properties get_properties (in Location location);
   void set_properties (in Location location ,
                        in Properties properties);
};
```

To change the properties, a member must invoke *set_properties* on the object group reference, specifying its own location and its new properties.

3.4 Group composition

As stated in the fault tolerance specification, the active replication requires the use of a multicast group communication system providing reliable totally-ordered message delivery and group membership services in a model of virtual synchrony. Under this model, group members usually receive events related to the group membership, like the *view* of the group, with the

ordered list of members. This list can be obtained under the current specification with the method *locations_of_members* in the *ObjectGroupManager* interface.

Nevertheless, a member cannot be considered as belonging to a group until it has received the state. During the state transfer, it is needed to block any action on the coordinator and joining members, as these actions could modify the transferring state, and both members would likely finish in inconsistent states. When the transfer finishes, the blocked messages can be unqueued and processed. Only the coordinator and the joining member are blocked; they can be seen as slower members, not performing any action until the state is transferred.

To be able to know at each moment the real composition of the group, that is, the list of state members, we include an additional method in the *StateHandler* interface:

```
typedef sequence<StateHandler> StateView;
interface StateHandler {
    ...
    void state_view (in StateView s);
};
```

4 Use cases

4.1 *StateHandler* interface

1. The application creates a new member in a group with the property *UseBasicStateTransfer* set to false. An existing state member is chosen as coordinator.
2. The method *sync_transfer* is invoked on the interface *StateHandler* implemented by the joining member. A nil reference is passed as parameter, besides the location of the coordinator.
3. The coordinator's method *start_transfer* is called with the location of the joining member and the phase returned after the call to *sync_transfer*. The coordinator must return a phase object.
4. This phase object is passed back to the coordinator on the next call, to the method *get_state*. It must supply the first chunk and a phase object that will be transferred to the joining member through the method *set_state*.
5. While the phase returned by the coordinator does not flag the end of the transfer, the previous step is repeated.
6. If any of the members involved in the transfer fails, the other member learns about it through the *FaultNotifier*. If the coordinator is the member failing, a transfer is restarted by choosing a new coordinator. The method *sync_transfer* is invoked again on the joining member, but this time it receives the last *PhaseCoordination* object received, instead of a nil reference. It can then choose whether the whole state must be transferred again, or just continue the interrupted transfer, by modifying that object.

7. When the transfer finishes, every member in the group receives a new state view with a call to the method *state_view*.

4.2 Coordinator selected by the application

1. A member is added to a group that has defined the property CoordinatorElectionStyleValue as COORD_ELECTION_APP_CTRL.
2. If the fault tolerance service uses *pull* transfer, the method *get_coordinator* is invoked on the joining member. Under *push* transfer, the method is invoked on every state member. In both cases, the member must choose the coordinator between one of the members passed to the method as parameter.

4.3 Coordinator selected by the infrastructure

1. A member is added to a group that has defined the property CoordinatorElectionStyleValue as COORD_ELECTION_INF_CTRL.
2. If the group does not use member properties, any state member can be chosen as coordinator.
3. If the group is using member properties, state members are accessed to obtain the properties of the current members (method *get_properties* under the *PropertyHandler* interface). If property FT::COORDINATOR_WEIGHT is defined, the infrastructure selects as coordinator the member with lower relation *scheduled_transfers* / *weight*. If the property is not defined, any state member can be chosen as coordinator.

4.4 Use of properties under an infrastructure-controlled membership style

1. The application invokes the *create_object* method of the *GenericFactory* implemented by the *ReplicationManager*, supplying the type_id and properties through the *Criteria* parameter.
2. The *ReplicationManager* obtains the fault tolerance properties for the object group from the *PropertyManager* of the type_id specified. These properties can be overridden by others specified in the *Criteria* parameter under the entry called FT::FT_PROPERTIES.
3. The *ReplicationManager* decides the locations at which to create the members of the object group.
4. For each location, if the group properties define *UseMemberProperties* as true, an entry called FT::FT_MEMBER_PROPERTIES is searched, and inside this entry, the properties for the specific location are obtained.
5. The member properties, when defined, are passed to the object being created with the *create_object* method of the appropriated factory, under the *Criteria* parameter.

6. Once the object is created, it receives the properties of the other group members through successive calls to the method *set_properties* on the *PropertyHandler* interface, once for each existing state member.

4.5 Use of properties under an application-controlled membership style

1. Members created directly by the application using the local factory, receive their properties through an entry called FT::FT_MEMBER_PROPERTIES in the *Criteria* parameter.
2. Members created by the *create*_member method in the *ObjectGroupManager* interface implemented by the *ReplicationManager* receive their properties in the same way, as that method defines as well a *Criteria* parameter (the local factory is called indirectly).
3. Once the object is created, it receives the properties of the other group members through successive calls to the method *set_properties* on the *PropertyHandler* interface, once for each existing state member.

5 Conclusions

This proposal introduces a flexible state transfer scheme complementing the basic one defined in the current fault tolerance CORBA specification. The domain of this proposal is composed by the active groups using an application-controlled consistency style, where the logging and recovery mechanisms defined in the specification are not suitable enough.

The basic idea supporting the scheme flexibility is the splitting of the state to be transferred in several chunks. The application can also decide how to handle interrupted transfers, or which state member is more appropriate to coordinate a specific state transfer. It is also possible to define member properties, as a feature separated from the concept of group's state.

These features are specified with a high adaptation to the fault tolerance specification, and there is no need for additional changes to the CORBA specification. Most of these features are configurable through the use of object group properties, and only three new interfaces are defined: *StateHandler*, to handle state transfers, *PropertyHandler*, to allow the dynamic change of properties, and *CoordinatorElector*, to allow the selection of transfer coordinators by the application.

References

[OMG99] Fault Tolerant CORBA, Joint Revised Submission, OMG TC Document orbos/00-01-19, December 20, 1999.

[Birman87] K. Birman, and T. Joseph, “Exploting Virtual Synchrony in Distributed Systems”, *Proceedings of the Eleventh Symposium on Operating Systems Principles*, Austin, November 1987.

[Schlichting93] R. Schlichting**,** S. Mishra, and L. Peterson "A Communication Substrate for Fault-tolerant Distributed Programs", *Distributed System Engineering, vol. 1, pp. 87-103,* December 1993

[Parrington95] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The Design and Implementation of Arjuna", *USENIX Computing Systems Journal, Vol 8, No 3,* 1995

[Maffeis97] S. Maffeis and D. C. Schmidt. "Constructing Reliable Distributed Communication Systems with CORBA", *IEEE Communications Magazine 14(2),* February 1997.

[Malloth96] "Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks" *PhD Thesis No. 1557, Swiss Federal Institute of Technology of Lausanne (Switzerland)* September 1996.

[Hayden98] M. Hayden, "The Ensemble System Cornell University**"** *Technical Report, TR98-1662,* January 1998

[Peña99] L.M. Peña, J. Pavon, "Sensei: Transferencia de Estado en Grupos de Objetos Distribuidos", *Computacion y Sistemas Vol.2, No 4 pp.191-201,* April-June 1999.