17

The Persistent Object Group Service — An approach to fault tolerance of open distributed applications

M. Zweiacker
Swiss Telecom PTT Research & Development
CH-3000 Berne 29, Switzerland
zweiacker@vptt.ch

Fault tolerance is an issue of high importance to distributed systems, a fact that is well identified in the ISO/ITU Reference Model of ODP by the inclusion of failure transparency. The Persistent Object Group Service (POGS) *described in this paper keeps track of the state of a distributed application as far as global checkpoint consistency is concerned. Application objects take checkpoints of their own in a non-coordinated fashion, using the POGS to detect global state inconsistencies. As a consequence of consulting POGS, objects take additional checkpoints that would not have occurred otherwise, but which are necessary to ensure global state consistency. The advantage of the POGS approach lies in the fact that global checkpoint consistency control is separated from the objects that actually do the checkpointing. This is a necessary step on the way to integrating fault tolerance mechanisms in a late stage of the software development process. A prototype of the POGS has been implemented in the Swiss Telecom R&D laboratories, using CORBA as a standard distributed systems technology.*

Keywords: CORBA, distributed systems, checkpointing and recovery, persistence, global checkpoint, state consistency.

1 INTRODUCTION

Fault tolerance is a highly important issue in the world of distributed computing. The gains of distributing a task among a number of collaborating computers is somewhat weakened by the increased probability for a certain portion of that distributed system to fail. Moreover, the breakdown of a single piece might lead to the failure of the entire distributed application. What needs to be done is to find appropriate ways to handle the possible breakdown, preferably by a workaround where the user doesn't notice that a failure has ever occurred. This vision is in accordance with the *failure transparency* proclaimed in the ISO/ITU Reference Model for ODP [1][3].

An approach to work around an application's failure is to make the entire application persistent using *checkpointing and recovery* techniques [3][13]. The Swiss Telecom R&D Dept. has launched a research project to investigate the applicability of the approach to distributed systems using standard architectures and platforms, like the OMG's Common Object Request

J. Rolia et al. (eds.), *Open Distributed Processing and Distributed Platforms*

Broker Architecture CORBA. Moreover, the RM-ODP has foreseen the need for a checkpointing and recovery function in order to support failure transparency [3]. One of the project's goals is to get a clearer picture of the implications of using the checkpointing and recovery approach for distributed systems, and, if applicable, assist further standardization activities.

Applying *distributed checkpointing* introduces the problem of having to coordinate the checkpointing of the individual objects in order to achieve *checkpoint consistency*. Consistency of checkpoints is a prerequisite to using them for the recovery of a distributed system. Inconsistency among the individual checkpoints does not affect the application's progress, but it will certainly impair, or make impossible, the usage of these checkpoints for later recovery of the application. The goal of this Telecom internal project is to specify and implement a service that prevents inconsistencies among the checkpoints of a set of distributed objects, such that the checkpointing and recovery approach can be used to enhance fault tolerance of distributed applications[1]. Furthermore, that service should use standard object technology, like CORBA.

The *Persistent Object Group Service (POGS)* coordinates the checkpointing of objects such that the set of all checkpoints is consistent. However, due to the many possibilities to define an object's state, the POGS does not itself take responsibility for checkpointing objects. Rather, it acts as a guide for the object to decide *when* a checkpoint needs to be taken. Furthermore, the POGS allows interrupt-free operation of the distributed application, or service, that it controls.

The paper is organized as follows: Section 2 introduces persistence as a means to achieve higher availability and robustness of an application. A brief theoretical survey on the consistency of checkpoints in a distributed system is provided in section 3. The architecture of the Persistent Object Group Service (POGS) and an explanation of its interfaces is given in section 4. The project has been guided by a number of requirements that the POGS is expected to fulfill. These are provided in section 5 along with a formal description of the interfaces at the boundary between the POGS and the application objects. In section 6, some notes on the prototype POGS implementation are given. A discussion concludes the paper in section 7.

2 FAULT TOLERANCE THROUGH PERSISTENCE

This section provides a brief explanation of how checkpointing and recovery is used to enhance fault tolerance of an application. The idea behind the checkpointing and recovery technique is to regularly store an object's state on stable storage, i.e., on a device that is considered safe from durable data loss[2]. The state information of an object is called the object's *local checkpoint* or simply *checkpoint*, whereas the process of bringing it to stable storage is termed the *taking of a checkpoint*. In case the object fails a new object is created, then initialized with the latest state of

---

[1] Sometimes, distributed applications are called such despite that they are in fact single hosted, i.e., they run under the control of a single operating system, and they depend on a single aggregate of hardware. The only relation to distributed computing is the client/server paradigm that they support. Throughout this paper, we do not specifically address that kind of server but, rather, refer to a more general case of distributed computing, i.e., a group of performers — mostly computer programs — cooperate in order to achieve a common goal. They communicate by exchanging messages, and they are likely to be geographically disperse. Moreover, we consider the individual performers as objects in their most common interpretation, introduced in the RM-ODP object model, and the OMG's CORBA, respectively [2][4].

[2] By regularly we mean that checkpoints are taken either on the basis of timely intervals, or as a consequence of the application's progress. In either case, the point in time when a checkpoint needs to be taken is dependent on the application, and on the degree of robustness and fault tolerance that the programmer wants to achieve.

the object found on stable storage. This procedure is called *recovery*. The new object replaces the failed one and resumes its execution as it was put back in time when the original object took the checkpoint (see Figure 1).

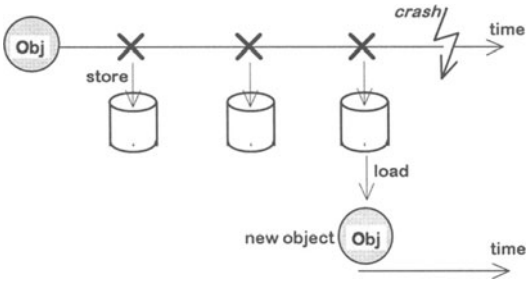

Figure 1 Checkpointing and recovery of a single object.

As a checkpoint represents the object's state, it also preserves the object's history, hence the only work lost is the activity that took place after the last checkpoint had been taken and the moment when the object failed. All previous activities are reflected in the object's state, and only those that happened after the latest checkpoint will need to be repeated in order to make the new object a replacement for the failed one. As an example, consider a word processor with the auto-save option turned on (the program will automatically save the edited document in predefined intervals). Should anything bad happen to the computer, like a system crash, there is at least a large portion of the document stored, if not all of it.

Checkpointing and recovery can be used to enhance robustness and availability of an object. In conjunction with other distribution transparencies, checkpointing and recovery can allow crashed systems to recover without the users to notice any interrupt while interacting with their application (except perhaps some delay during recovery).

3 DISTRIBUTED CHECKPOINTS

In this section, the necessary conditions to assure global state consistency in a distributed system are described, and the major differences between distributed checkpointing and a transactional system are briefly discussed.

3.1 Consistency Criteria

In a distributed application, all objects need to take checkpoints in order to form a *global checkpoint* which is a collection of local checkpoints, one for each member of a group of objects. The global checkpoint represents the state of the entire object group if and only if it is *consistent*. Consistency among the local checkpoints means that, after recovery, the re-loaded state of the entire group is one that could have occurred in the past. This introduces the problem of having mutually inconsistent states stored in the checkpoints. From the theory we learn that a global checkpoint is consistent if all pairs of checkpoints are mutually consistent [8][9][18]. Inconsistencies come as a direct consequence of the message flow between the objects. They arise whenever certain temporal relations between local checkpoints and message transfers occur. One can always construct situations where two objects of a distributed application form the above mentioned temporal relation, making their checkpoints mutually inconsistent.

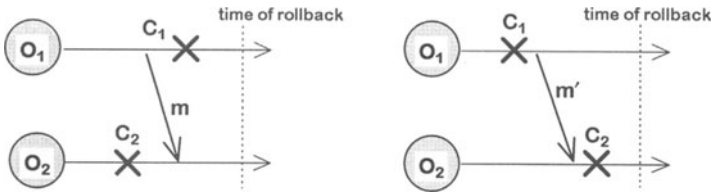

Figure 2 Missing message and orphan message.

The only way to prevent the objects from taking inconsistent checkpoints is to introduce a control mechanism whose responsibility is either the avoidance or the alarming of a possible inconsistency. To illustrate how inconsistency can occur, refer to Figure 2: the horizontal lines represent the history of an object concerning checkpointing and messaging, with time running from left to right. The crosses mark the point in time when a checkpoint has been taken. The arrows running from one horizontal axis to another denote a message.

A message is termed *missing* if the sending of m is recorded in the sender's checkpoint C_1, while it is not recognised in the recipient's checkpoint C_2 (Figure 2, left). This kind of message is not critical as far as global state consistency is concerned. A missing message can be dealt with by introducing a logging mechanism with the objects. The objects O_1 and O_2 might be rolled back to the checkpoints labeled C_1 and C_2, respectively. The consistency is preserved by forcing O_2 to read from its log all messages that it had received after the checkpoint, and marked as sent by O_1, including m in our case.

An *orphan* message is not recorded as sent in the sender's checkpoint, but its acceptance is well recognised in the recipient's checkpoint (Figure 2, right). Upon the recovery from the two checkpoints, O_1 would re-send message m' which is recorded as already being received by O_2! This situation makes the two checkpoints useless in their combination, therefore they are termed inconsistent. If checkpointing is to be an efficient technique to achieve fault tolerance in a distributed system, it must not allow orphan messages to be stored with the checkpoints.

Note that orphan (and lost) messages have to do with the problem of consistent checkpointing alone. A message cannot be classified as "missing" or "orphan" by its content or by some other property but the relations shown in Figure 2.

If we allowed orphan messages to occur in the checkpoints, the recovery procedure would have to find a set of local checkpoints with no orphan messages in either pair. This would lead to a backwards propagating search, with some probability of never finding a suitable set of checkpoints. This phenomenon is called the *domino effect* [8][18][19]. As the purpose of distributed checkpointing and recovery is to save as much as possible of the application's history, a *coordinator* must be introduced that avoids the domino effect by preventing messages from becoming long-term orphans. The solution lies in the introduction of extra checkpoints that avoid the production of orphans. These would have to be injected by the coordinator that detects or anticipates a (potential) orphan relationship. The coordinator must have a global view on the distributed system in order to take appropriate measures, like checkpoint injection. There are basically the two approaches to realize the coordinator:

- Let it take full control over the application's execution, and coordinate checkpointing as a global event. No orphan message will ever occur, as the checkpoints are taken all at the same time, with no system related messages passing between the objects meanwhile[3].

[3] Speaking of the "same time" must be considered with care, as there is no such thing like a common clock in an asynchronous distributed system. Nevertheless, we can identify timely intervals where, for a specific purpose, we can assume there to be a temporal correspondence, mainly based on the happened-before relation introduced in [17].

- Let the coordinator trace the message flow between the objects. When an inconsistent situation is about to be produced, the coordinator would instruct the affected objects to take extra checkpoints in order to keep the global checkpoint consistent.

The first approach is straightforward and simple. Instructions to take checkpoints can occur at any time, and the produced set of checkpoints is guaranteed to be consistent. However, there is a serious drawback with *coordinated checkpointing*: the entire distributed application will be stopped for checkpointing, a situation which is deemed unacceptable in many cases.

The second solution allows interrupt-free operation of the application, but is far more complex to realize: each object requires a logging mechanism, and the message flow has to be traced. Yet, it is the preferred solution to realize the coordinator, called *Persistent Object Group Service (POGS)*. The POGS allows the objects to take checkpoints at their own schedule, and forces a few additional ones in order to avoid inconsistent checkpoints.

There exist algorithms that solve the inconsistency problem for distributed checkpointing. They ensure that none of the checkpoints, those taken on the object's own schedule as well as those explicitly introduced by the POGS, will be inconsistent in the long run. In [9], the authors present an entire theoretical framework to describe consistency for distributed checkpointing. Based on this framework, Baldoni et al. provide an algorithm to prevent the forming of so called *rewinding paths*, a message flow pattern in a distributed system that is equivalent to having orphan messages. These rewinding paths are made impossible by the introduction of additional checkpoints by the algorithm. We have used this algorithm to implement a prototype of the POGS in the Swiss Telecom R&D laboratories.

3.2 Relationship with Transactions

Transactions are a powerful technique to maintain the consistency of an application's progress with its specification. In order to allow intermediate state changes during a transaction to be undone, processes (objects) take checkpoints. However, the consistency criteria for a transactional system do not necessarily coincide with those for the global system state. In particular, the scope of a transaction does in general not span all objects, but only a few. It might be useful to report to the POGS the checkpoints that are taken due to a transaction, and to integrate those into the global state record. But, with a transactional system alone, we would not be able to assure global state consistency for a distributed system as described previously.

4 THE PERSISTENT OBJECT GROUP SERVICE (POGS)

After a short introduction into the basics of global distributed state theory in Section 3, we are ready to describe the architecture of a POGS whose purpose is the detection and alarming of inconsistent states in a distributed system. In particular, we find that the POGS must be given the opportunity to *trace the entire message flow* between the objects. Note that the desired checkpoint coordination could as well be distributed. In fact, the algorithm presented in [9] is aimed at being distributed in the objects. The fundamental idea of a POGS is the separation of the consistency responsibility from the objects that take checkpoints, hence it represents a stand-alone service that can be included into any distributed application.

As shown in section 3, the relevant information for a coordinator is the temporal relationship between existing checkpoints and the messages that pass between the objects. More precisely, the POGS must know the identity of the sending and the receiving object of a message, and indicate to the receiving object that it must take a checkpoint *prior* to processing the message which — if no checkpoint were taken — would introduce inconsistency. This introduces the following

synchronization problem: how does the POGS ensure that the receiving object will get the checkpoint decision on time?

The POGS must indicate the checkpoint decision such that the receiving object can take a checkpoint before processing the (otherwise orphan) message. It becomes clear, that the POGS architecture is dependent on the way message tracing is achieved. Figure 3 depicts the options:

- *Message relay.* All messages are routed through the POGS, i.e., the sender submits a message to the POGS with an indication of the destination object. The POGS would append the checkpoint decision to the message prior to forwarding it to the destination object.
- *Pre-notification.* Each time an object sends a message, it indicates to the POGS the destination object identity and asks for the appropriate checkpoint decision. The sender would then submit a combined message, carrying the actual system message data, and the checkpoint decision for the destination object. The latter would only process the message after having obeyed the checkpoint decision.
- *Post-notification.* The receiving object keeps the incoming message in a cache, then consults the POGS whether it needs to take a checkpoint before further processing the message.

All of the presented options are suitable from a theoretical point of view, as they ensure timely delivery of the checkpoint decision to the destination object. However, there are practical reasons why post-notification is the preferred option. Relaying implies that all system messages go twice across the wire, from the sender to the POGS, and from the POGS to the destination object. Pre-notification is an unnecessary complication of what can be achieved through the post-notification option.

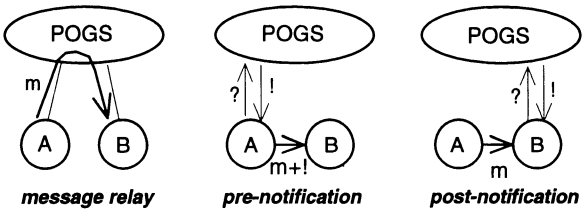

Figure 3 Message tracing options.

Apart from reporting the receipt of a message, the objects need to notify the POGS each time they take an unforced checkpoint, i.e., a checkpoint that was taken as a result of the application's progress or any other decision that does not regard the POGS. This information is necessary for the POGS to keep track of the checkpoints stored by the objects, hence to have a consistent view on the checkpoint landscape in the system.

As the POGS alone ensures checkpoint consistency, the objects do not have to take other checkpoints than those forced by it. Programmers can, if they wish, include their own, orthogonal checkpointing schedule for the application objects. However, they may as well do without it. It is important to mention that the POGS' coordination task may easily lead to a situation where a certain object is never requested to take a checkpoint simply because it would not introduce inconsistency. If only based on the POGS, such an object might never be checkpointed. Thus, relying on the POGS only is a design decision that must be taken carefully if fault tolerance is of importance to the application.

The architecture of the POGS is depicted in Figure 4. The POGS and the objects are modeled according to the CORBA architectural guidelines [4]. Each of the objects has its own mechanism to log messages and store checkpoints. The interaction between the objects and the POGS is based on the post-notification option. The objects use the `POGScheck` interface to notify the arrival of a new message and to report checkpoints. As the overall purpose of the POGS is fault tolerance, it must be able to issue instructions to the objects to rollback. Each object has

therefore a `POGSrecover` interface to accept the rollback instruction. The `POGSadmin` interface is used to administer the object group, such as the registration of an object to an object group (called a context). Having distinct names for object groups allows the POGS to control many groups independently.

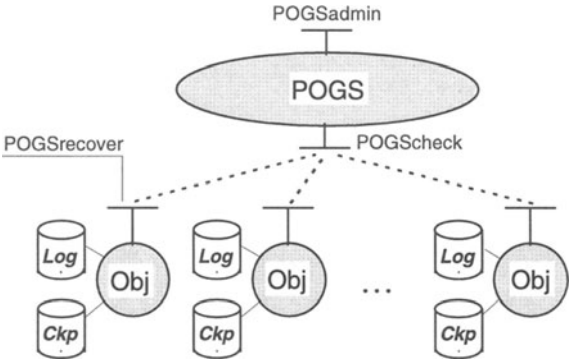

Figure 4 Architecture of the Persistent Object Group Service (POGS).

Regarding the interaction between the objects and the POGS, there are a number of responsibilities that the objects must take:

- indicate membership to a group of consistently checkpointed objects (registration);
- indicate every checkpoint taken apart from those decided by the POGS;
- consult and obey the checkpoint decision each time a message arrives.

Programmers are free to define what information is relevant to determine the state of an object without the functionality of the POGS being affected. The use of standard storage mechanisms, like the CORBA Persistent Object Service (POS), is just one possibility. More advanced checkpointing procedures might free the programmer from the details of state storage, and capture the entire object, including the relevant state data. As the POGS does not prescribe the choice of a particular checkpointing procedure, programmers can create one that adapts to the application's needs. One of the goals of the POGS was to abstract from the checkpointing of the individual objects and only serve as the coordinator of checkpoints. Another design goal was to specify and implement the POGS such that it allows cooperating objects to rely on a third-party decision about checkpointing and not care about the algorithm that implements the decision. The separation between the checkpoint decision and the objects that actually take checkpoints is one of the primary achievements that the POGS fulfills.

Having a centralized service as the key component to achieve fault tolerance raises the obvious question of how safe the approach is. In the presented architecture, the POGS appears to be a single point of failure as far as fault tolerance is concerned. Should the POGS crash the application objects would be without guidance of when to take checkpoints. However, they would still accomplish the intended task for which the application was designed, though without being fault tolerant for some period of time. It is important to note that the inclusion of fault tolerance through the POGS does by no means affect the normal progress of an application. It is up to the engineering to include additional measures that enhance robustness and availability of the POGS itself (through local checkpointing of the POGS, or object replication, for instance).

Another option could be to enhance interaction semantics between the POGS and the application objects, such that the latter take "safe" checkpoints (checkpoints that might be unnecessary but which are taken to be completely sure that no inconsistency can occur in the system) as soon as the POGS is found to be unavailable, then report the checkpointing activities that occurred during this period of non-guidance to the POGS when it has recovered.

5 REQUIREMENTS AND SPECIFICATION

The project has always aimed at keeping the POGS an easy-to-use service. The following list of requirements has served as a guideline to specify the POGS:

- *Checkpoint coordination.* The core functionality of the POGS is the coordination of checkpoint and recovery procedures for a set of objects. It allows the objects to apply their own schedule for taking checkpoints. However, when an object receives a checkpoint instruction from the POGS it must obey this instruction in order to keep the global checkpoint consistent.
- *Recovery coordination.* The POGS allows any object to initiate the recovery action. The objects will be giving guidance in finding the checkpoint that they need to load for recovery.
- *No fault detection.* The POGS does not perform fault detection. The functionality of the POGS is limited to checkpointing and recovery coordination.
- *No taking of checkpoints.* It is the responsibility of each application object to define a suitable checkpointing procedure. The POGS does not take checkpoints of the objects.
- *Networked Service.* The POGS must be specified using the OMG Interface Definition Language IDL.
- *User Transparency.* The existence of the POGS should be transparent to the users of a distributed application. An application user is by no means involved in the interactions between the POGS and the objects. The only observation a user can do is notice high availability and interrupt-free operation of the application.
- *Explicit usage.* The programmers of a distributed application use the POGS explicitly, i.e., they take the responsibility to include code for the objects to interact with the POGS. It is important to note that the project has never aimed at combining the functionality of the POGS with existing, old software, that possibly lacks access to its source code. The POGS is definitely not targeted at legacy applications of that kind.

In order to describe the interactions at the boundary between the POGS and the objects, the `POGSadmin` interface, the `POGScheck` interface, and the `POGScoord` interface need to be specified. At the time of editing this paper, the specifications were rather unstable and not suitable for publishing. However, we decided to give an indication of what an IDL for these interfaces might look like. In order to enhance comprehension, textual explanations are also provided with the specification. For the sake of simplicity, error handling statements have been left out:

```
interface POGSadmin {
      short ContextCreate (in string context_name);
      short ContextDelete (in string context_name);
      short Register (in string context_name,
            in string obj_id, in string rec_id);
      short Deregister (in string context_name, in string obj_id);
};
```

The `POGSadmin` interface supports the management of object groups, called *contexts*. A context comprises the identifications (or names) of all objects that belong to a group that is subject to checkpoint consistency control. The `ContextCreate` operation allows to setup a new context that is then referable by the string variable `context_name` in all subsequent communications with the POGS. `Register` and `Deregister` allow an object to get bound and unbound to a context, respectively. Being bound to a context is the precondition for an object to consult the POGS for checkpoint decisions.

The rec_id interface identifier denotes the callback interface to be used by the POGS in case the application needs to perform recovery from checkpoints. It represents the POGSrecover interface of the registered object.

```
interface POGScheck {
      short Check (in string obj_id, in string sender_id);
      void CheckpointNow (in string obj_id, in short ckpt_seq_nr);
};
```

The POGScheck interface comprises the operations Check and CheckpointNow. Check is used to consult the POGS upon the arrival of a new message from another object of the context. The reply value is used to interpret the checkpoint decision, which possibly forces to take a checkpoint before extracting the message from the input cache. CheckpointNow is used to report to the POGS that the object has taken an unforced checkpoint.

```
interface POGSrecover {
      short Recover (in short ckpt_seq_nr);
};
```

POGSrecover is the callback interface to be used in case the application needs to rollback. The POGS issues the checkpoint sequence number ckpt_seq_nr with the Recover operation as a parameter to indicate the appropriate checkpoint to be loaded for recovery. It is important to note that this interface has to recover *automatically* if the supporting server crashed in order to ensure that distributed recovery can take place. After the recovery has completed, it is the object's responsibility to read from the message log all missing messages that result from the set of recovered checkpoints in the system.

6 IMPLEMENTATION NOTES

CORBA based applications normally use a client-server interaction scheme, i.e., with every request message there is an associated reply message. However, the fault tolerance approach presented in this paper applies to *messaging systems* that do not imply response messages as far as the underlying communication protocol is concerned. If we allowed RPC-like interaction, every missing request message would imply an orphan response message. Therefore, the scope of applications is narrowed to those that use messaging as the primary communication paradigm. In a CORBA context, this is equivalent to having only oneway operations (operations with a void return value). Otherwise, the POGS would have to be modified to not only disallow orphan messages, but missing messages as well.

A prototype of the POGS has been implemented in the laboratories of the Swiss Telecom R&D, using Iona's orbix™, a CORBA compliant development environment available for a large range of hardware and software platforms. The implementation of the prototype has been guided by the philosophy that the application objects should as much as possible be isolated from the checkpointing activity. Therefore, all POGS-related activity is kept away from the objects by a proxy that takes care of the communication with the POGS, as well as the message logging. It is the proxy who controls the POGSrecover interface. As a consequence, it must be able to recover the object, i.e., it needs the appropriate authorization for the creation and deletion of the object. The proxy approach is sketched in Figure 5.

Regarding the implementation, message logging and message caching are no longer distinct concepts. Each incoming message is logged. Caching a message, which is necessary in the course of consulting the POGS, is equivalent to logging, but not yet delivering it to the destination object.

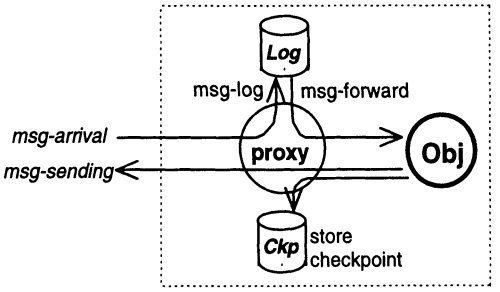

Figure 5 The proxy shields the object from the POGS and from other objects.

In order to enable consistent recovery, the POGS must at all times be aware of the current checkpoint sequence numbers of the objects. The `CheckpointNow` operation therefore carries the checkpoint sequence number `ckpt_seq_nr` with its notification, so the POGS can double-check this value with its own knowledge. Conversely, the reply value of the `Check` operation carries the new sequence number of the checkpoint that the POGS just decided. A negative return value would then indicate that no checkpoint is required.

Finally, the consistency algorithm used to implement the POGS is derived from the distributed algorithm described in [9], which is based on a sound theoretical framework that allows to describe state consistency in a distributed system. The algorithm ensures that only a minimal number of additional checkpoints will be taken (i.e., decided by the POGS) in order to keep the entire system consistent.

7 DISCUSSION

From the experiences so far, we conclude that there are the two perspectives to view the checkpointing and recovery techniques for fault tolerance: There are (**i**) the theoretical constraints that make checkpointing and recovery a useful thing in the context of distributed systems, and (**ii**) the practical implications of using the checkpointing and recovery approach.

(**i**) From a theoretical perspective, we found that the frameworks to describe consistent global states in a distributed system are setup. The real work is now in the adaptation of the theory to useful interaction schemes between the entities that want to perform checkpointing and those that take care of the global state consistency. One approach is to have a distributed algorithm in close conjunction with the objects, such that each object is made aware of possible inconsistencies by its own interactions with other objects. This would imply that application programmers tailor the objects to apply checkpointing and recovery along with the necessary consistency checks in an early stage of the software development process, a solution that might impact portability and openness.

Having distributed state consistency checked by some third-party unit is the preferred approach regarding current trends in standard architectures and platforms. Unfortunately, the de-coupling of the checkpointing from the consistency check introduces the need for additional calls to that certain coordinator for the sake of global state consistency. Every message passing between any two objects implies the need for a non-system message to the POGS, unless the POGS is used as a message relay (the latter solution would not decrease the number of messages required to complete the consistency check). Such overhead leads to a decrease of the application's performance depending on the number of messages in the system (not necessarily the number of

objects). The performance penalty certainly depends on the nature of the application that uses the POGS. A "well formed" application would let the objects heavily compute on a problem while only exchanging few messages.

To conclude the theoretical part of the discussion, we find that distributed checkpointing is not a self-contained problem. Rather, it poses a more general problem, namely how to find the most effective means to observe the state of a distributed application.

(ii) The practical implications of applying checkpointing and recovery techniques to distributed systems are mostly related to the requirement that the POGS be a self-contained open service. It is not sufficient to regard the POGS as just an oracle from which distributed state information can be obtained: what we need to do is try to shield the programmers of distributed applications from the internals that come from the appliance of failure transparency mechanism, including those that rely on checkpointing and recovery.

One option to achieve this ambitious goal is to provide *infrastructure objects* that are used to shield the application objects from lower level coordination activities. The proxy approach has shown to be a promising solution, but there should be no need for programmers to implement a proxy on their own. Rather, the proxy — or infrastructure object — should be an integral part of the service (the POGS in our case). This leads us to the question: is it desirable to have a service that is capable of delivering the required infrastructure object to the users of that service? We believe that Java™ has already answered that question with YES. The possibility to download software components from the service provider to the user is a valuable approach to the proxy problem. A downloadable proxy for the POGS might then offer the `POGSrecover` interface, and connect to the application object. It would coordinate with the POGS without the object taking notice. We believe that this is a feasible approach and we are willing to investigate further in this direction.

The relation of the POGS with existing *CORBAservices* still needs to be studied [6]. Certainly, the POS (Persistent Object Service) is a suitable candidate to assist checkpointing. The POS would also support the storage of a checkpoint on a different location than the object that it represents. The separation of the checkpoint data from the object certainly increases the checkpoint's availability in case of a failure of the object. Another CORBAservice to consider for integration with the POGS is Externalization, which offers the means to externalize and internalize the state of an object.

The message logging approach for consistent distributed checkpointing somewhat contradicts with the rules given in the checkpointing and recovery function of the RM-ODP. In the ODP standards, coordinated checkpointing has been implicitly regarded as the only way to achieve consistent global states [3]. We believe that the experiences with the POGS will lead to a deeper understanding of the checkpointing and recovery function. In fact, we are convinced that, through this project, the RM-ODP standards text on the checkpointing and recovery function can be improved.

REFERENCES

[1] ISO/IEC Draft International Standard 10746-1 | ITU-T Recommendation X.901, Reference Model of Open Distributed Processing, Part 1: Overview, 1995

[2] ISO/IEC International Standard 10746-2 | ITU-T Recommendation X.902, Reference Model of Open Distributed Processing, Part 2: Foundations, 1995

[3] ISO/IEC International Standard 10746-3 | ITU-T Recommendation X.903, Reference Model of Open Distributed Processing, Part 3: Architecture, 1995

[4] *The Common Object Request Broker: Architecture and Specification*, The Object Management Group, Revision 2.0, July 1995

[5] *Common Facilities Architecture*, The Object Management Group, Revision 4.0, Nov. 1995

[6] *CORBAservices: Common Object Services Specification*, The Object Management Group, Revised Updated Edition, March, 1996

[7] ALLISON C. ET AL., Wanted: An Application Aware Checkpointing Service. WARP Report W2-94, Division of Computer Science at the University of St Andrews

[8] BALDONI R., HELARY J.M., MOSTEFAOUI A., RAYNAL M., On Modeling Consistent Checkpoints and the Domino Effect in Distributed Systems. IRISA Publication Interne No. 933, Institut National de Recherche en Informatique, May 1995

[9] BALDONI R., HELARY J.M., MOSTEFAOUI A., RAYNAL M., Consistent Checkpointing in Message Passing Distributed Systems. IRISA Publication Interne No. 925, Institut National de Recherche en Informatique, May 1995

[10] BEGUELIN A., SELIGMAN E., High-Level Fault Tolerance in Distributed Programs. Report CMU-CS-94-223, School of Comp. Sc., Carnegie Mellon University, Pittsburgh, PA 15213

[11] BIRMAN K.P., GLADE B.B., Reliability Through Consistency. IEEE Software, May 1995, pp. 29-41

[12] CHANDI K.M., LAMPORT L., Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3 (1):63-75, July 1985

[13] DECONINCK G., VOUNCKX J., LAUWEREINS R., PEPERSTRAETE J.A., Survey of Backward Error Recovery Techniques for Multicomputers Based on Checkpointing and Rollback. *Proceedings of IASTED Int. Conference on Modelling and Simulation*, Pittsburgh, PA, May 10-12, 1993, pp. 262-265

[14] FIDGE J., Timestamps in message passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, Feb. 1988, pp. 55-66

[15] FROMENTIN E., RAYNAL M., Local states in distributed computations: a few relations and formulas. IRISA Publication Interne No. 796, Institut National de Recherche en Informatique, Jan 1994

[16] JOHNSON D.B., ZWAENEPOEL W., Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing. In *Proceedings of seventh ACM Symposium on Principles of Distributed Computing*, ACM, Aug 1988

[17] LAMPORT L., Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21 (7): 558-565, July 1978

[18] KOO R., TOUEG S., Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Eng., Vol. SE-13, No. 1, January 1987*

[19] NETZER R.H.B., XU J., Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, 2, 1995, pp. 165-169

[20] SILVA L.M., SILVA J.G., Global Checkpointing for Distributed Programs. *Proceedings of the IEEE 11th Symposium on Reliable Distributed Systems*, Houston, Texas, USA, Oct 1992, IEEE Computer Society Press, pp 155-162.

[21] STROM R.E., YEMINI S., Optimistic Recovery: An asynchronous approach to fault-tolerance in distributed systems. *IEEE*, 1984

[22] XU J., NETZER R.H.B., Adaptative independent checkpointing for reducing rollback propagation. In *Proceedings of the 5th IEEE int. Symposium on Parallel and Distributed Processing*, Dallas, Dec. 1993, pp. 754-761

[23] ZWEIACKER M., The Persistent Object Group Service (POGS). Project Proposal for the ObjectLab initiative at APM Ltd., *Swiss Telecom R&D internal report*, June 1996, unpublished, available on request.

[24] ZWEIACKER M., *The POGS Consistency Algorithm: Description of the consistency algorithm for distributed checkpointing services, Swiss Telecom R&D internal report*, Aug. 1996, unpublished, available on request.