Zeitschrift: Comtec : Informations- und Telekommunikationstechnologie = information and telecommunication technology

Herausgeber: Swisscom

Band: 75 (1997)

Heft: 8

Artikel: Fault-tolerant Cobra : using checkpoint and recovery

Autor: Zweiacker, Marc

DOI: https://doi.org/10.5169/seals-876955

FAULT TOLERANCE

FAULT-TOLERANT CORBA, USING CHECKPOINTING AND RECOVERY

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 article 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 noncoordinated 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 using CORBA as a standard distributed systems technology.

MARC ZWEIACKER, BERN

An approach to work around the failure of a distributed application is to use *checkpointing and recovery* techniques. Swiss Telecom R&D has launched a research project to investigate the applicability of the approach to distributed systems that are built using standard architectures and platforms like the OMG's Common Object Request Broker Architecture (CORBA) [3].

Applying checkpointing to distributed objects forces to survey the checkpointing activities of the objects in order to maintain *checkpoint consistency.* Consistency of checkpoints is a prerequisite to using them for recovery. Inconsistency among the individual checkpoints does not affect the application's progress, but it will certainly impair the usage of these checkpoints for recovery. The *Persistent Object Group Service (POGS)* keeps track of the state of a distributed application, as far as checkpointing is concerned, and is used by the application objects to prevent checkpoint inconsistency. The POGS does not take responsibility for checkpointing the objects, but rather acts as a guide to decide *when* checkpoints need to be taken.

Normally, communication in CORBA is of a RPC style, i.e., with every request message, there is an associated reply; however, the fault tolerance approach presented in this article applies to *messaging systems* that do not imply response messages, as far as the underlying communication protocol is concerned. Taking RPC-like interaction into account is not a trivial task in checkpointing. The reasons are twofold: Firstly, applications tend to make use of *threads,* a very efficient way of handling concurrent requests at the server side. It turns out in practice that it is very difficult, not to say impossible, to recover an object from a checkpoint that incorporates threads. A second difficulty is the taking of a checkpoint in the presence of open communication connections. In practice, an object communicating with another uses some run-time library module to enable networking. It is not sensible to incorporate the state of a run-time library for checkpointing, as the library

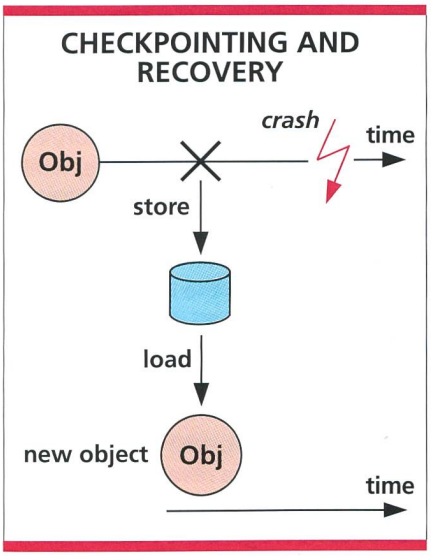


Fig. 1. Checkpointing and recovery of an object.

would certainly not recover the way we wanted: Time-outs would lead to the breakdown of pending communication links; thus, channel identifiers stored in a checkpoint would most probably be invalid upon recovery of the object. For these reasons, the scope of applications has been narrowed to those that use messaging as the communication paradigm. In a CORBA context, this is equivalent to having *one-way operations* in the application interfaces (operations with a void return type).

Distributed checkpointing

Fault tolerance through persistence

The idea behind checkpointing and recovery is to regularly store an object's state on stable storage, i.e. on a device that is considered safe from durable data loss. The state information 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 and then initialized with the latest state of 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 (Fig. 1).

As a checkpoint represents the object's state, it also preserves the object's history; hence, it is the only work lost is the activity that took place after the last checkpoint had been taken and after 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 serious happen to the computer, like a system crash, there is at least a large portion of the document stored, if not all of it.

Consistency criteria for distributed checkpoints

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 reloaded 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 and only if all pairs of checkpoints are mutually consistent [4, 5, 8]. 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. 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 recognized in the recipient's checkpoint C_2 (Fig. 2, left-hand). 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 are rolled back to checkpoints 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 that are 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 recognized in the recipient's checkpoint (Fig. 2, right-hand). As almost every distributed application is of an nondeterministic nature, we are very uncertain about the resending of message m'. In particular, we cannot tell whether the content of m' will be the same in a second run and if it would be resent, after all. We conclude that orphan messages make two checkpoints useless in their combination; therefore, they are termed inconsistent. *Using checkpointing and recovery for fault tolerance in a distributed system means not to 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 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* [4, 8]. 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 or-

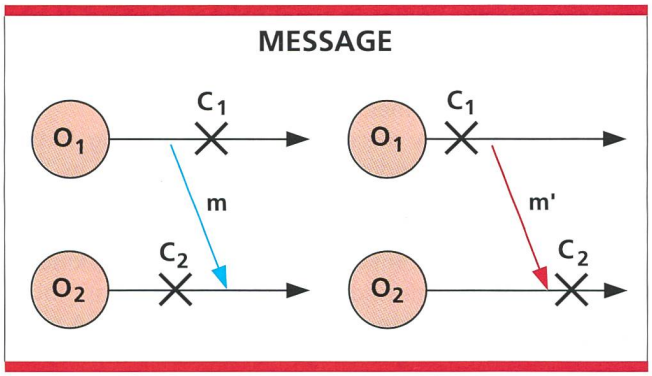


Fig. 2. Missing and orphan message.

phans. 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 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 concurrently, with no system-related messages passing between the objects meanwhile.
- 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.

In the first approach, checkpoint instructions 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,* as the entire distributed application needs to be stopped during checkpointing, a situation which is deemed unacceptable in many cases. The second solution allows interrupt-free operation of the application, but is more complex to realize: Each object requires a logging mechanism, and the message flow has to be traced. Yet, it is our preferred approach to realize the 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 inconsistency.

There exist algorithms that solve consistency 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 [5], 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. 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 our R&D laboratories.

The Persistent Object Group Service

Message tracing and checkpoint reporting

We can learn from distributed checkpointing theory that orphan messages are detectable, if the relationship between existing checkpoints and system messages is known. Thus, the POGS must be given the opportunity to *trace the entire message flow* between the application objects. More precisely, it must know the identity of the sending and the receiving object of a message and indicate to the receiving object that it needs to take a checkpoint prior to processing the message which – if no checkpoint were taken – would introduce inconsistency. It is the receiving object's responsibility to inform the POGS each time a new message arrives. It would provide the sender's identification (object A in Fig. 3) as well as its own (object B) and ask for advice about checkpointing (the ? request). Based on this information, the POGS updates state knowledge, while forecasting an orphan, and correspondingly replies to the object that it must or must not take a checkpoint prior to processing the message (using the ! reply).

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. As the POGS ensures checkpoint consistency, the application objects are in theory allowed to only take checkpoints that were forced by it; however, programmers would normally include their own, orthogonal checkpointing schedule for the objects in order to save the application's achievements (after a period of heavy computing, for instance). But 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.

Performance degradation

There is no question about the fact that message tracing leads to a performance degradation, as every application message between the objects induces an extra conversation between the receiving object and the POGS. This makes the total number of system messages twice the number of application messages (the system being comprised of application objects and the POGS). It is the price that we pay for having consistency control separated from the objects and to free programmers from having to implement the consistency algorithm in the objects. Note that it is possible to distribute the checkpointing algorithm into the objects, hence giving the POGS the appearance of being obso-

lete; however, this would mean that checkpointing is part of the application's design from the very beginning and that implementers need to know about checkpoint consistency programming. Moreover, recovery coordination is not covered in checkpoint consistency algorithms, and a means to find a (most recent) consistent set of checkpoints (so-called recovery vector) would have to be programmed in a distributed manner. Conversely, the POGS keeps all relevant information in one place; therefore, it is able to orchestrate not only checkpointing but recovery as well. Furthermore, the POGS allows programmers to include fault tolerance measures to applications even after they have been designed and programmed. This latter property was one of the driving forces when the POGS idea popped up: to have an independent entity watch over the consistency of a group of checkpointed objects and to easily integrate it as a programming component.

Architecture

The architecture of the POGS is depicted in Figure 4. Each of the objects has its own mechanism to log messages and store checkpoints. The objects use the `POGScheck` interface to notify the arrival of a new message and to report checkpoints. Another major task of the POGS is recovery coordination; thus, it must be able to issue instructions to the objects to rollback (using the `POGSrecover` interface). The `POGSadmin` interface is used to administer the object group, such as the registration of an object. Having distinct names for object groups allows the POGS to control many groups independently. 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 checkpointed objects (registration)
- report every checkpoint taken apart from those decided by the POGS
- consult and obey the checkpoint decision each time a message arrives

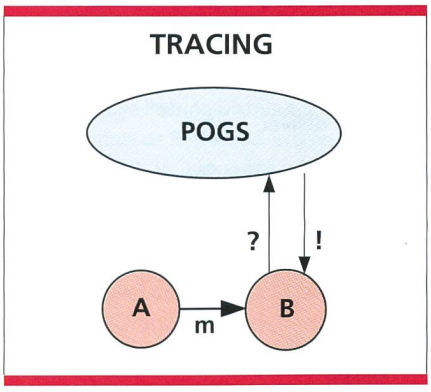


Fig. 3. Message tracing.

Checkpointing

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. 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 a service that would allow cooperating objects to rely on a third-party decision about checkpointing and not care about the algorithm that implements the decision.

Robustness of the approach

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. 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 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 is 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) as soon as the POGS is found to be unavailable, and then report the checkpointing activities that occurred during this period of nonguidance to the POGS, when it has recovered.

Specification

The following is a list of requirements that 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 checkpointing schedule.
- *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.
- *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 to 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 the code into the objects that integrates the POGS.

In order to describe the interactions at the boundary between the POGS and the objects, the `POGSadmin`, the `POGScheck`, and the `POGSrecover` interfaces need to be specified.

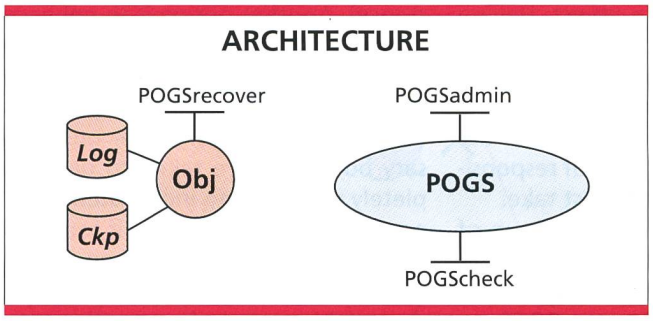


Fig. 4. POGS architecture.

```
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 identifiers (or names) of the objects that belong to a group that is subject to checkpoint consistency control. The ContextCreate operation allows to set up a new context that is then referable by 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. rec_id denotes the callback interface to be used by the POGS in case the application has to recover. It represents the POGSrecover interface identifier of the 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 interpreted as the checkpoint decision, which possibly forces a checkpoint before the message is processed. Using CheckpointNow, the object notifies the POGS of having taken a voluntary checkpoint, i.e., a checkpoint that has been taken as a result of the application's checkpointing schedule.

```
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 its message log all missing messages that correspond to the recovered system state.

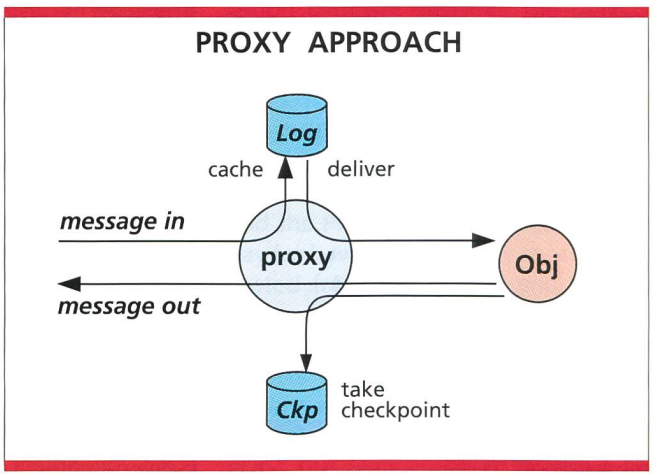


Fig. 5. The proxy shields the object from the POGS and from other objects.

A note on realization

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.
Looking at the implementation, message logging and 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.

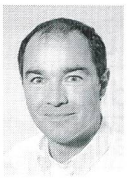
Discussion

The practical problems of applying checkpointing and recovery techniques 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 want to do is shield the programmer of a distributed application from too much detail that comes from using checkpointing and recovery as a failure transparency mechanism.
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 whether it is 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 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 between the POGS and existing *CORBAservices* still needs to be studied. 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. 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 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 [2]. 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. 2



Marc Zweiacker ist nach Abschluss der Studienzeit an der HTL Burgdorf (Elektrotechnik) und der ETH Zürich (Informatik) im Frühjahr 1991 der Gruppe «Netzwerke und Kommunikation» der Direktion Forschung und Entwicklung beigetreten. Sein Spezialgebiet sind die verteilten Systeme in der Normierung wie auch in der praktischen Handhabung sowie im Management. Von 1993 bis 1994 hat er aktiv an der Definition des ITU-T-Standards X900 «Reference Model for Open Distributed Prosessing» mitgewirkt. Seit 1996 befasst sich Marc Zweiacker vornehmlich mit Fragen der Fehlertoleranz von verteilten Systemen und speziell von CORBA-basierten Applikationen.

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 10746–3 / ITU-T Recommendation X.903, Reference Model of Open Distributed Processing, Part 3: Architecture, 1995.
[3] *The Common Object Request Broker: Architecture and Specification,* The Object Management Group, Revision 2.0, July 1995.
[4] 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.
[5] 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.
[6] Chandi K. M., Lamport L., Distributed snapshots: determining global states of distributed systems. *ACM TOCS,* 3 (1):63–75, July 1985.
[7] 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, August 1988.
[8] Koo R., Toueg S., Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Eng.,* Vol. SE-13, No. 1, January 1987.

Zusammenfassung

Fehlertoleranz mittels Checkpointing und Recovery

Fehlertoleranz ist ein wichtiger Aspekt im Zusammenhang mit verteilten Diensten und Anwendungen. Diese Tatsache kommt unter anderem im ISO/ITU-Referenzmodell für ODP zum Ausdruck, wo die sogenannte Fehlertransparenz definiert worden ist. Der hier vorgestellte Persistent Object Group Service (POGS) unterstützt die Fehlertoleranz mittels Checkpointing und Recovery, indem er die Konsistenz von Checkpoints in einem verteilten System sicherstellt. Die Objekte einer Applikation benutzen diesen Dienst, um auf mögliche Inkonsistenzen aufmerksam zu werden und die daraus notwendige Konsequenz zu ziehen, das heisst, um einen zusätzlichen, durch die Applikation nicht unbedingt vorgegebenen Checkpoint durchzuführen. Dieser Ansatz birgt den Vorteil, dass die Konsistenzsicherung für verteilte Checkpoints von den Applikationsobjekten losgelöst wird. Damit wird erreicht, dass Fehlertoleranz als modularer Dienst zu einem späten Zeitpunkt der Applikationsentwicklung integriert werden kann. Ein POGS-Prototyp wurde mittels CORBA realisiert.