Fault-Tolerance for Communicating Multidatabase Transactions*

Eva Kühn

Institute of Computer Languages, University of Technology Vienna
Argentinierstr. 8, 1040 Vienna, Austria, eva@mips.complang.tuwien.ac.at

Abstract

In this paper we propose a framework that can be used for the implementation of reliable multi database system communication protocols. The framework provides transactions on shared write-once objects, termed communication objects, that can be recovered after failures. Also concurrency is supported by recoverable processes. The realization of this framework that can be provided by means of a distributed coordination kernel, is the focus of this paper. Strategies for communication objects that belong to different reliability classes are presented. The application of this framework for MDBS transaction processing is shown and the failure behavior of a MDBS communication protocol supporting global serializability through a global two-phase-commit is discussed.

1 Introduction

A multi database system (MDBS) [5, 21] coordinates access to distributed data repositories maintained by autonomous institutions. The consistency of MDBS transactions must not be compromised by site nor by communication failures. MDBS transaction rollbacks should be avoided, since MDBS transactions tend to be long-lived and costly.

Function replication improves the availability of MDBS transactions. The Flex Transaction Model [9, 17] can be used for the specification of alternative services. However, a more fundamental necessity for MDBS is a communication concept that tolerates failures. In this paper we propose the use of a distributed coordination kernel that maintains communication objects which can be shared between different sites, in particular between the MDBS and its participating

local systems. These communication objects have the following properties: they can be assigned a value only once (rigorous single assignment) so that sites can rely on them; they are persistent and thus recoverable after system failures; they are written within transactions, providing atomic write of distributed data; and they support anonymous communication by avoiding a global name space. To improve their reliability, they can be replicated: their communication behavior can be specified. The kernel architecture is a general concept: its primitives can be embedded into any language paradigm. At each site, the kernel maintains copies of those communication objects that are relevant to this particular site. Communication between the MDBS and the MDI (multi database interface) of a local system consists of writing a communication object by calling the appropriate kernel primitives. For example, global commitment protocols depend upon critical messages not being lost: if a local system fails during a period of uncertainty about the global decision, global consistency may be violated [4]. This may be solved, assuming that the local systems provide a prepared to commit phase, by atomic global commitment (which is achievable by atomic write of a group of communication objects).

In previous work [7, 16] we presented a Prolog based language (VPL Vienna Parallel Logic) integrating communication objects in the form of communication variables. This paper focuses on the implementation issues of a language and paradigm independent kernel.

In Section 2 we analyze communication requirements in MDBSs. In Section 3 a framework for the reliable implementation of MDBS communication protocols is presented that bases on communication objects. In Section 4 we present strategies which treat communication objects belonging to different reliability classes. In Section 5 a MDBS communication protocol is shown, employing the proposed framework. The reliability of the different strategies is discussed.

2 Communication in a MDBS

The MDBS problem is a very general one: the integration of autonomous and distributed (database) systems comprises database aspects as well as networking and distributed processing. In the most general case a MDBS integrates arbitrary autonomous local software systems (LSYSs), not necessarily database systems.

If we look at a MDBS from the database point of view, we have to integrate (relational) databases and thus have to deal with the logic integration of schemas and data models [13]. The approaches here deal with the aspect of *naming autonomy* [11] (or *design autonomy*) and treat aspects of how to handle differences concerning data names and structures, semantic differences, scaling problems, and redundancy and inconsistency. In particular, two approaches can be found: (1) The global integration approach provides totally hidden integration and is a rather static approach that gives the user the feeling of one universal interface that hides all differences between different LSYSs. The deficiencies are that local schema changes cannot be reflected easily, and that the fully automatic generation of an integration schema is not possible in general. (2) The more dynamic approach of MDBS languages, such as MSQL [20], shows the advantage of flexibility: in an environment of hundreds and thousands of public databases full integration is not feasible. However, the dynamic approach shifts the burden of data integration to the user, who must provide the knowledge of how to access the data in all the different LSYSs (in their respective local language). The logic integration problem can be studied fully independently from the physical distribution.

The next step to look at the MDBS problem is to consider the *execution* and *communication autonomies* of LSYSs. Each LSYS is autonomous concerning the used concurrency control mechanism, how and when to execute transactions (if at all), and when to communicate.

We assume the architecture as shown in Figure 1. MDIs (MDBS Interfaces) are used to pass queries to a LSYS via its usual interfaces and are responsible in a self-reliant way for the communication with the LSYS. An MDI is a gateway (server process) that most likely will run at the same site as the LSYS. Each LSYS that wants to participate in a federation must support its own MDI, because otherwise we violate local autonomy. We must not assume that all LSYSs speak a language and provide a communication mechanism that is understood and used by the MDBS. A MDBS coordinates the traffic between the MDBS and the MDIs of the LSYSs. The main task of a MDBS is to minimize
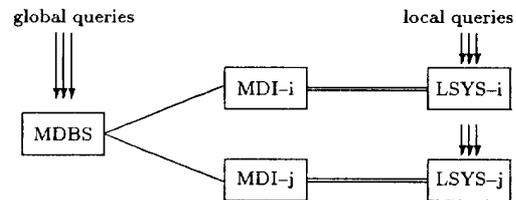

Figure 1: Communication in a MDBS.

the time a LSYS is blocked by a global transaction and to maximize the success probability of a global transaction—in particular if it is a long-running (e.g., cooperating transaction) or possibly everlasting one. Additionally, global correctness must be maintained. Communication protocols for the execution of MDBS transactions are required that guarantee that no global constraints are violated and that in case of failures, either the global transaction can complete by substituting the task to be done at the failing site by another task or at least guaranteeing that all local locks are released after some time.

Recently advanced transaction management models [8] have been proposed that recognized that the ACID properties of classical transactions are too strict. In particular, if cooperating transactions want to exchange intermediate results, the isolation property must be relaxed. Compensate actions may be used to semantically undo effects of subtransactions that have committed and made their effects visible, before the global transaction has committed. Another feature, introduced by advanced transaction models, is function replication. This idea bears some aspects of software fault-tolerance: a task can be substituted by another one. The specification of alternatives allows for indeterminism. The alternatives may be tried either sequentially or in parallel in which case those that are not needed are either aborted (if the local systems provide a 2-phase-commit protocol [3]), or compensated. However, function replication is by far not sufficient to guarantee reliable MDBS transaction management: it can only mask failures, where the LSYS and the MDBS have the same view about whether the task has been accomplished or not (e.g., the LSYS cannot execute the local subtransaction, because local constraints are violated; or the MDBS rejects the result produced by the LSYS, because global constraints are not fulfilled). But if for example a failure occurs at one site, then global consistency may be violated. A network failure may cause the MDBS to assume that the LSYS did not fulfill its task, issue a global abort, while the LSYS either waits in its prepared state forever, or at least a compensation at the LSYS should
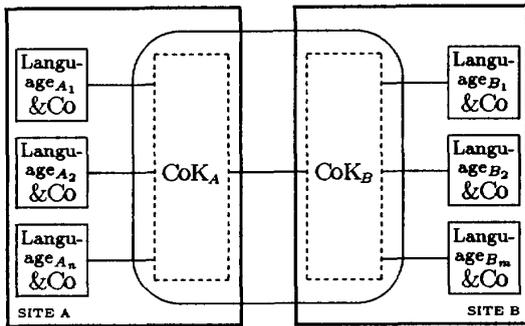
Figure 2: Coordination package for communication between different languages at two sites.

have been issued. The maintenance of global atomic commitment is a critical issue in MDBS transaction management [4].

Execution of MDBS transactions requires a communication framework with the following properties:

• order preserving of messages so that internal dependencies like failure or success dependencies [9] between subtransactions can be controlled,

• atomic write of distributed objects to implement a global atomic commitment,

• reliability of the communication so that neither site nor network failures can compromise the correctness of a global transaction execution,

• support of reliably shared data structures for either passing results between communicating transactions (in advanced transaction models), or to maintain global data structures like serializability graphs, if employed by the global transaction manager.

3 A Distributed Coordination Kernel

We propose to use a framework [18] called the coordination kernel (CoK), that offers an advanced communication mechanism based on shared data that can be written in *transactions*, called TRANSs. In general, this architecture allows languages (belonging to different paradigms) at different sites to communicate in a reliable way (see Figure 2). We term a language that calls CoK primitives "language&Co", which means language + coordination. For example the extension to C is called C&Co [10]. With the implementation of MDIs and MDBSs in &Co–languages, (or by purely calling the primitives of the CoK), different MDBS communication protocols, for either traditional or advanced transaction processing can be implemented and very easily fine-tuned to changing environments (see Section 5). Fail-stop processors do not lead to inconsistencies of MDBS transactions.

Coordination comprises *concurrency* and *communication*: The CoK supports concurrency through the explicit creation of processes and—in a limited form—through a primitive that allows the concurrent waiting for CoK events. Communication is provided by the shared data paradigm. The CoK maintains *shared write-once objects*, termed communication objects, that are written in TRANSs. All participating processes see one globally shared space of communication objects. Each process may only see those communication objects to which it possesses a reference that is passed to the process via its parameter list.

3.1 Primitives for Coordination

The primitives of the CoK (see Table 1) are language independent and can be embedded into any host programming language.

The start of a TRANS is defined by TRANS_BEGIN, where it is associated with a unique communication object, the tid (TRANS identification). To inform the CoK about the nesting structure, the tid of the father is also passed. Every COBJ_WRITE call belongs to a TRANS and is a *request* to assign the communication object, uniquely identified by oid (object identification), a value (data). Write requests are collected, but not performed. On execution of the corresponding TRANS_END the CoK tries to perform all write requests in an atomic step. If one write cannot be performed, because the communication object has already a value, thus violating the write-once property of communication objects, TRANS_END fails. No writes of communication objects are performed (*all-or-nothing property*). A completed TRANS specifies a compensate action which is called by the CoK should the enclosing TRANS be later aborted. These extended semantics of classical transaction processing have been chosen because two different kinds of concurrency are supported: processes that execute a TRANS and may perform writing of communication objects, and processes that simply define an entry point (like in the classical nested transaction model), but that do not produce isolated side-effects.

The PROCESS primitive introduces concurrency. It serves to start a process—if supported by the corresponding software system it starts a thread—at another site. We assume that "entry_name(args)" is specified analogous to a function definition, or can be retrieved from a CoK-library at that site. Communication objects may be passed in "args" via their unique oid, so that they become shared between the site calling PROCESS and the site where PROCESS is executed.

There are two ways to define an entry name: either as ENTRY or as TRANS_ENTRY:

ENTRY entry_name(args)
IN (tid$_{father}$) USES (cobjs)

A process that calls an ENTRY is part of the calling TRANS. Its termination state influences the success of the TRANS_END of its father TRANS: the father TRANS must wait until all ENTRY type processes it has called have terminated successfully (*completed*). The tid of the father is passed to the call via the IN clause. USES serves to define which communication objects are created by this process (see Section 3.2).

TRANS_ENTRY entry_name(args)
WITH (tid) USES (cobjs)

A TRANS_ENTRY type process starts a new autonomous TRANS that runs decoupled from the calling TRANS. The WITH clause specifies a tid that is created when the TRANS_ENTRY is called and USES specifies all communication objects created by this process (see Section 3.2).

COBJ_NEW creates a new communication object dynamically. COBJ_READ is a blocking primitive that tries to read the value of a communication object. If the communication object is still *undefined*, COBJ_READ waits until another process completes a TRANS by which the communication object is written (*synchronization*). Another blocking primitive is ALT_WAIT, which waits until one of the conditions specified in its argument list, fires, returning the number of this alternative (compare with [14]). The conditions have the form "[k] condition-expression", where k is the number of the alternative and condition-expression contains COBJ_READ calls.

TRANS_ABORT aborts the TRANS to which it refers. This causes write requests to be ignored, all ENTRY type processes called by this TRANS to be sent a signal ABORT, all compensate actions of completed TRANS called by this TRANS to be started, and the exit of the TRANS (i.e., if possible, a jump to the TRANS_END; otherwise simply all other CoK calls that refer to this TRANS are ignored).

tid, and pid serve to report the execution state of a TRANS and a PROCESS, respectively. If a TRANS succeeds, SUCCEEDED is written into the tid communication object and this write is immediately performed, otherwise the token ABORTED is written into tid. Analogous for pid: if the PROCESS reaches its end pid is assigned SUCCEEDED, otherwise ABORTED. If the PROCESS is of type TRANS_ENTRY, the state of tid of the called TRANS is written into the pid.

3.2 Failure Behavior

Communication objects are persistent: as soon as they have been written in a TRANS their values are

primitive name	1st arg	2nd arg	last arg	result value
TRANS_BEGIN	tid	tid$_{father}$		INT
TRANS_END	tid	compensate-action		INT
TRANS_ABORT	tid			
COBJ_NEW	TYPE			TYPE OID
COBJ_WRITE	tid	oid	val	
COBJ_READ	oid			TYPE
ALT_WAIT	cond$_1$	...	cond$_n$	i
PROCESS	system @site	entry_name (args) {IN (tid)}	pid	
SIGNAL	pid	{ABORT| STOP|...}		

Table 1: CoK Primitives

type	parameter
STAT OID	tid, tid$_{father}$, pid
TRANS_ENTRY	compensate_action
TYPE OID	oid
TYPE	val
CONDITION	cond$_1$,...,cond$_n$
1,...,n	i
ADDRESS	system@site
val$_1$,...,val$_k$	args
OID$_1$,...,OID$_l$	cobjs
ENTRY|TRANS_ENTRY	entry_name(args) {...}

Table 2: Argument Types

reliably stored on a non-volatile storage medium. We allow the existence of fail-stop processors [22, 23]. If a site fails, its CoK is automatically re-booted after the site recovers. The CoK recovers all communication objects, and remembers all PROCESS requests of type TRANS_ENTRY that have been called at its site and not yet terminated, and re-starts them.

A re-started process starts to execute its task again, thus we have to explain what happens, if CoK calls are executed a second time (see Table 3) and how the CoK may recognize this. We do not assume that the host language itself, from which the CoK primitives are called, is able to recover with the statement where it was interrupted by the failure—this would be the ideal situation.

A definition of a PROCESS entry (its header) consists of three parts: entry name and arguments, an IN or WITH clause, and a USES clause. If the process is called, the CoK stores the whole header and associates it with the unique process pid. After a failure, any non-terminated PROCESS can be recovered by the CoK with its original image. By this, the references to locally created communication objects (USES and WITH clauses) are not lost, and the arguments of the entry and the IN clause assure that the references to passed communication objects are not lost. Thus, if a process is recovered, it can continue to use all com-

primitive name	behavior if executed on a restart
TRANS_BEGIN	if TRANS had not yet terminated: this informs the CoK that a failure has occurred; the CoK forgets about all actions done by this TRANS, except of compensate actions of completed TRANSs; otherwise the TRANS_BEGIN returns an error-code and we assume that the TRANS block is skipped
PROCESS	if the PROCESS is of type ENTRY: it is now re-started with its original image; if the PROCESS is of type TRANS_ENTRY: the CoK simply ignores the call, because such processes are automatically re-started when the CoK recovers

Table 3: Re-Execution Behavior after a Failure

munication objects that appeared in its entry header definition. By using oids, all processes see the same most recent state of all communication objects they may see.

All other CoK calls perform in their usual way, after a failure.

3.3 Properties and Maintenance of Communication Objects

A communication object is uniquely identified by its oid. This identification is used by the host language to refer to communication objects, and does not require the naming of communication objects and thus in contrast to the Linda model [12, 6] can avoid a global name space problem. Once created, the oid remains unique forever.

Communication objects are typed, depending into which language they are embedded. The CoK accesses objects only via methods that are allowed on them. The system where the communication object was created determines the type of the communication object. If the communication object is passed to a different system/site, methods to convert this type into a neutral representation, and from a neutral representation into a suitable type at the other system/site are applied. If such methods do not exist, the corresponding accessing CoK call fails (possibly returning an error code). This guarantees that everyone *understands* communicated data.

Communication objects may be structured and contain other communication objects as members (subterms, etc.). Thus, infinite data structures can be created, e.g., tuple spaces can be maintained this way as linked lists. Sub-communication objects may be written at another site than the site of the communication object to which they belong. If the communication object is written, the CoK maintains also its

OID
Type Description (XDR)
Data
Administration Information (Remote Pointers, Distributed Object Tree)
Reliability (Communication Behavior)
Access Methods

Table 4: Internal Structure of Communication Object

value, possibly being structured and containing sub-communication objects.

To implement communication objects shared between different sites in a network we use replication techniques [19], as explained in Section 4. All administration information concerning the implementation of the global view is also part of a communication object.

The reliability of an object may be specified as part of its type description, concerning the role of the object during communication. An object may be more or less reliable, as explained in the following section. Generally speaking, this refers to how many replicas of such an object are maintained and which strategy is used by the CoK to implement this communication object.

4 Realization of the Kernel

All CoK primitives that access communication objects must employ a strategy to maintain the correctness of the global view of the shared communication objects. At any time, all processes must have the same consistent view of them, message order must be preserved, and reliability guaranteed. A strategy for the atomic write of many distributed communication objects, without causing deadlocks, is required. The strategy to implement communication objects depends mainly on the underlying hardware. For the demonstration of MDBS transaction processing, we assume workstations in local and wide area networks with unreliable physical communication links. Every such site may fail independently from the others.

Without a realization in mind, let us first assume that there exists conceptually one *main copy* of each communication object which represents the communication object and must be consulted on reading and writing. Which CoK primitives access communication objects, causing communication messages to be sent between different CoKs?

TRANS_BEGIN, TRANS_ABORT, SIGNAL do not involve communication between CoKs.

ALT_WAIT calls COBJ_READ. If several such calls are joined in one expression, where all of them must be fulfilled, they can be executed one after the other, because once a COBJ_READ is fulfilled, the read value will not change. The task of ALT_WAIT is to check in a fair way which conditions fire—the interaction with other CoKs is done by the COBJ_READs.

COBJ_NEW simply creates a new communication object with a new unique oid, and can be executed locally by the CoK without interaction with other CoKs. The implementation of COBJ_NEW simply requires that the CoK generates and stores the new oid and the type information of the new communication object.

COBJ_WRITE also can be performed locally: the CoK collects and stores all requests and the corresponding tid.

COBJ_READ must access the main copy of the communication object it wants to read to find out whether is has a value or not, involving communication with the CoK that is responsible for the main copy. If the value of a communication object has already been accessed by one site, the CoK may obtain a copy of its value locally and can avoid communication messages if further COBJ_READs are performed on the same communication object.

TRANS_END is the most complex CoK primitive. It performs the following steps in an atomic action:
1. Find out which communication objects have been addressed in write requests ($\mathrm{cobj}_{w_1}, \ldots, \mathrm{cobj}_{w_n}$).
2. If it can be decided locally that a cobj_{w_i} ($i = 1, \ldots, n$) is already defined, return an error code.
3. *Request* access to the main copy of each cobj_{w_i}. If during this step another TRANS requests the main copy, then *grant* this request only if the object identification of its tid (which can be considered as a logical timestamp [23]) associated with the other TRANS is less then the own tid. Thus, deadlocks are avoided.
4. If during the request-phase it is detected that one cobj_{w_i} is already defined, abort the transaction and return an error code.
5. After all main copies have been acquired, lock them all. Even if a older TRANS sends a request, keep the locks until the end of the TRANS_END.
6. Write all main copies of all cobj_{w_i} (*binding*-phase).
7. Write SUCCEEDED into the tid of the TRANS [1].
8. Release all locks.

PROCESS causes the pid of the PROCESS and its im-

---

[1] If the CoK calls are called correctly, e.g., if they are generated by the compiler of the host language, the CoK has the main copy of the tid, otherwise also the main copy of the tid must be requested, analogous to the cobj_{w_i}s.

age to be stored by the CoK. Then it is either called locally, or a new (only internally, to the CoK visible, communication object) is created, the image of the PROCESS is *transmitted* to the CoK at the remote site by this communication object. The arguments of the entry that are or contain communication objects are transmitted to the CoK at the remote site. A CoK that receives a transmit message, containing a PROCESS image to be executed at its site, starts the PROCESS upon receipt of this message, as if the PROCESS were called locally. After termination of the process, its termination state is written within an (internally issued) TRANS.

4.1 Passive Replication/Deep DT

We start with a simple strategy PR_{deep}, where every site that may access a communication object owns its own copy (replica), with one copy, the main copy, distinguished as the *primary copy*. All other copies are called *secondary copies*. If a communication object is created, a primary copy is created. If the communication object is transmitted to another site, a secondary copy is created and transmitted to the CoK at the remote site. Thus, for each communication object, a distributed tree (DT_i) builds up, where each node knows the nodes to which is has transmitted a copy (sons), and each son knows the node (father) from which it has received a new copy. The root of a DT_i is the primary copy of communication object$_i$.

As motivated above, there are different classes of messages that are sent between CoKs: request, grant, binding, and transmit messages. Messages that modify the DT_i are called critical messages. They must not be lost and the CoK must assure that it can recover the DT_i, by storing the changes to the DT_i either into a LOG-file or directly into the DT_i (i.e., into the administration information of the internal representation of the communication object, see Table 4) on non-volatile memory, before sending a critical message.

An *unlink* message will be motivated below for the case that a CoK already knows a communication object that is retransmitted to it. Additionally, to achieve reliability, the receipt of certain messages must be confirmed by an *acknowledge* message.

The **migration-request-message** requests exclusive access to the primary copy and is sent to the father node. This is not a critical message and can periodically be resubmitted. If the father owns the primary copy, and if the communication object is still undefined and is not currently locked by a TRANS_END (step 3) the father sends a migration-grant-message. If the father does not own the primary copy, the father in

turn sends a migration-request-message to its father. If, however, the father detects that the communication object is already defined, it sends a binding-message.

The **migration-grant-message** is a critical message that must not be lost and that involves a modification of the DT_i. The sender of this message (which owns the primary copy) must store the intended changes to the DT_i (i.e., that its son becomes its father), before it sends the message. Thus, after a failure, when the acknowledge-message eventually is sent by the son, the DT_i can be recovered.

The **binding-message** is sent from the primary copy owner to its sons to inform them about the value that is written to a communication object$_i$. This is a critical message and before it is sent, the father must store the value in the internal representation of the communication object. If a binding-message is not acknowledged within a certain time-limit, it is resubmitted, periodically. After all sons have sent an acknowledge-message, the father deletes its part of the DT_i, because a defined communication object can never be written again and no other CoK will request the primary copy.

The **transmit-message** informs a CoK about a new communication object$_i$ that it has not seen before. This critical message requires the modified link structure of DT_i, i.e., that a new son has to be added, to be written to the LOG, before it is sent. The son either responds with an unlink-message, if the communication object is already known there—then the father clears its LOG—or it responds with an acknowledge-message, upon which the father applies the changes of the LOG to the DT_i. Also the transmit-message is periodically repeated, until an answer arrives.

The **unlink-message** is a non-critical message, that is repeated periodically, until an acknowledgement is received.

The **acknowledge-message** is a critical message. If sent in return to a: ($\star$) migration-grant-message, the CoK must store the changes to the DT_i (i.e., that the father, the sender of the migration-grant-message, becomes a new son, and that this CoK now owns the primary copy) before sending it; ($\star$) binding-message, the CoK must store the value of the communication object$_i$ in the internal representation of the communication object$_i$, before sending it. Afterwards, the CoK sends binding-messages to all sons of communication object$_i$; ($\star$) transmit-message, the secondary copy of the communication object$_i$ must be created and stored, before sending it; ($\star$) unlink-message, no state storing is required. Acknowledge-messages are not repeated and do not expect an answer. If an acknowledge-message is lost, the CoK waiting for this message eventually will resubmit the initiating message. If an acknowledge-message is reinforced this way, and it has already stored its state, then it is simply submitted again.

In [16] we have analyzed how to implement communication variables on different hardware architectures (shared/distributed memory; un/reliable physical communication links) and have proposed the passive replication strategy for the implementation of communication variables in VPL where communication links must be assumed to be unreliable. Although the state storing of communication variables is mentioned there, the fact that communication variables are maintained by the VPL programming language complicated the state storing, because this cannot be accomplished separately from the computations on the communication variables. As a VPL system consists of a large number of concurrently executing threads, the stacks of which always should be stored by each completing TRANS, so that computation can also be recovered, is a time-consuming task. In fact, the first VPL implementation (which was done in Prolog) implemented the strategy without state storing so that reliability could not be guaranteed. The current VPL prototype (which is implemented in C) employs the kernel calls. Also a C&Co prototype is developed at our department that uses the kernel calls.

In the here proposed kernel based architecture, the maintenance of communication objects is separated from the processes executing the programs that access communication objects, leading to a feasible implementation of state storing phases that are necessary to guarantee reliability. For a cost evaluation of sent messages we refer to [15].

A CoK that wants to access a communication object$_i$ depends on all sites of its fathers in DT_i. A failure of such a site is crucial, because the CoK must wait until the site recovers. We show improvements of PR_{deep} by providing alternative primary copies (see Section 4.2). Also other strategies can be designed. However, in contrast to the protocol used for a distributed Orca [1, 2] implementation, all sites below the current node in the DT_i may fail without compromising the access of this CoK. A further advantage of communication objects is that as soon as a communication object$_i$ is written, the DT_i can be resolved. All copies turn into read-only ones that can be maintained locally.

4.2 Different Reliability Classes

An essential aspect of communication objects is that for each communication object a different strat-

egy may be used—defined as part of its type description. Even if a communication object appears as the sub-communication object of another one, both may employ different strategies, because for each communication object a separate DT must be maintained. Each communication object belongs to a certain reliability class.

Also important is that the migration of communication objects serves to realize the locking of a set of communication objects during the TRANS_END. In strategies without migration, where for example a locking request is sent to the primary copy, deadlocks cannot be avoided.

We now analyze some variants for strategies with the intention to increase the fault-tolerant behavior.

4.2.1 Active+Passive Replication/Deep DT

The most obvious attempt is to try to improve reliability of PR_{deep} through replication of the primary copy. This leads to a combination of active and passive replication: both primary copies are allowed to answer independently.

Obviously APR_{deep} leads to an unjustifyably increased communication overhead. Also new messages must be introduced (the direct sons of the replicated primaries must be aware that there are two fathers right now).

4.2.2 Passive Replication/Flat DT

Also an obvious attempt is to introduce a flat tree: if a communication object is shared between n sites, one site is the father and all others are its sons. This idea is motivated by the InterNet addressing mechanism, where conceptually each two sites may communicate directly. Thus, every site knows where the primary is located. If now a migration-request-message occurs, the primary copy site must either immediately inform all (n-1) sons about this migration or must maintain a reference to the site to which it sent the migration-grant-message, and so forth. The first approach has the main deficiency that a migration-request-message must wait for the response of *all* sons. This was not the case in all other strategies as presented so far and decreases availability. The reference solution is preferable: A reference chain is build up that can be entered at the original primary copy site to find out the current primary copy owner.

So far, the improvement of PR_{flat} in comparison to PR_{deep} is that a node only depends on the primary copy site. Failures of all other sites do not compromise the access to the primary copy.

As a further improvement, read-request-message could be introduced, that expresses that a copy is
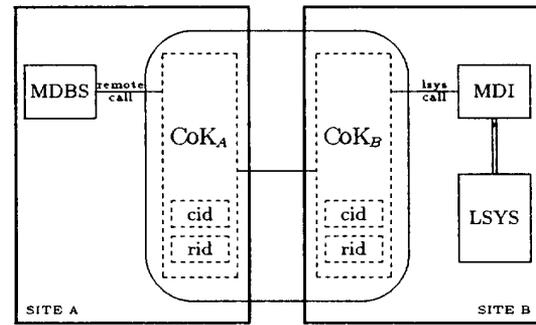
CoK_A/CoK_B = coordination kernels at MDBS/MDI site
MDBS and MDI are implemented in &Co-languages

Figure 3: Coordination package used for MDBS.

needed only for reading. This message prevents a primary copy migration for read access. Note that in the PR_{deep} read-request-messages are not possible, because such a message must be passed through several stages of the DT so that during its submission the communication object may become defined. In a flat DT there is direct communication with the primary copy owner.

4.2.3 Active+Passive Replication/Flat DT

Strategy $APR_{flat}(n)$ bases on PR_{flat} where the primary copy is replicated n times (n > 1) and the sites of all primary copy replicas are known to each site. This solution improves reliability, depending on the number of primary copy replicas.

The strategy is as follows: Every migration-request-message is sent to all primary copy replicas and must be granted by at least (n/2 + 1) primary replicas. If the requesting site (i.e., a secondary copy owner) decides about the failure of some primary replica sites, it must inform at least (n/2 + 1) primary replicas about that so that the majority of primary copy replicas knows that this primary replica is disabled. Also, repeatedly messages are sent to the supposed failed site so that it becomes informed about this decision, even if it has not really failed.

If a failed site that owns a primary replica, recovers, it asks a majority of other primary copy replicas to find out the current state, updates its state, and informs them about its reexistence asking to be enabled again.

Additionally, read access may be differentiated by a new message, thus also improving the access speed, because unnecessary migrations can be avoided.

5 Reliable MDBS Transactions

The use of the CoK leads to the architecture shown in Figure 3. The remote_call is the protocol that

330

is executed by the MDBS and is in charge of the communication between MDBS and MDI. The MDI runs the lsys_call protocol that handles the communication between MDBS and MDI at the MDI site, and controls the communication between MDI and LSYS. gta serves to start several global MDBS transactions concurrently.

The following two communication objects are shared between MDBS and MDI: cid is a communication object that controls the execution of the lsys_call. It can be used to send controlling information to the MDI, e.g., to implement a global 2PC, and to send the termination state of lsys_call to the MDBS. rid serves for the reliable transportation of result data from MDI to MDBS. It signals the MDBS that result data are available.

In [18] several protocols including proofs of their correctness are presented. The most realistic assumption turned out to be that local databases provide rigorous schedules (for example by using strict two-phase-locking), and that the MDBS uses a time-out mechanism to decide about failures. Then, also a co-existence of replicated and distributed MDBSs can be allowed, leading to open MDBS architectures. The MDBS communication protocol for these assumptions is shown in Figure 4.

All remote_calls are started concurrently by gta, which causes n lsys_calls to be executed by the corresponding MDIs. Before the remote_call starts the lsys_call, it executes a TRANS to define a compensate action, which writes the token ABORT into the communication object cid, which is executed should the TRANS (tid$_{father}$) in which the remote_call was called be aborted. After the lsys_call, which is of type TRANS_ENTRY, has been started with PROCESS it runs to completion. It executes the query at the local system: if the execution is successful there, the lsys_call reports its prepared state to the remote_call by writing the communication object rid. Then it waits for the cid to be written by the remote_call and depending on this value, sends db_abort or db_commit to the LSYS. If the global decision is to commit the global transaction G, then the COBJ_WRITE of the remote_call, which requested to set cid to COMMIT, is performed when the TRANS_END of the gta is executed. Otherwise, if the decision is to abort G, then the remote_call is aborted, which causes the activation of all compensate actions in its scope and thus the execution of compensate.

The communication protocol is resistant against temporary site-failures and network failures, because any interrupted process eventually is be revoked with its original image, independently whether after the failure (interrupt) the decision is to continue or to abort the global transaction. All locks at LSYSs are released after some time less than the specified time limit + possible failure times.

All LSYSs and the MDBS always have the same view about the commitment of G (there is no time where there are inconsistencies concerning global commitment [4]), because the information about the global abort is reliably passed to all LSYSs and as soon as the LSYS reads this variable, it will abort. Should G be aborted at any prior time, the execution of the compensations defined in the remote_call guarantees the abort of the local transactions. A decision about global commitment is also passed reliably by the cids that are all written in one atomic step at the TRANS_END of the gta.

An important feature of this architecture and the proposed protocol is that MDBSs can be replicated and distributed, leading to open architectures.

6 Conclusions

The MDBS problem is a general one that has a lot in common with distributed processing. An MDBS controls the access to different local software systems, and performs coordination tasks, recently also termed work-flow management. Communication between the MDBS and the participating local systems is an essential part of the coordination task. An MDBS must deal with concurrent executions.

We propose a general framework that provides an advanced transaction mechanism based on shared write-once objects, and concurrency. The framework is provided as a distributed coordination kernel (CoK). The primitives of the CoK are language and paradigm independent and can be embedded into any host programming languages. Languages that support CoK calls are termed &Co–languages.

&Co–languages can be used to implement reliable communication protocols for MDBS transactions. If MDBS transactions have to communicate, for example, because a common global data structure like a serializability graph must be maintained, this distributed structure can reliably be administrated by the CoK. Atomic global commitment is supported by the atomic write of (several) distributed communication objects.

The idea of the CoK leads to open architectures, where also the reliability of communicated objects can be fine-tuned. Depending on the application requirements, different strategies can be employed that are realized by different replication techniques.

331

```
TRANS_ENTRY gta(QUERY:L_1,...,L_n)WITH(tid)
    USES(STAT OID:pid_T,pid_1,...,pid_n)
    PROCESS(LOCAL,timeout(TIME),pid_T)
    PROCESS(LOCAL,(remote_call(MDI_1,L_1)IN(tid)),pid_1)
    ...
    PROCESS(LOCAL,(remote_call(MDI_n,L_n)IN(tid)),pid_n)
    IF ALT_WAIT(
        [1] COBJ_READ(pid_1)=SUCCEEDED AND ...
            ... AND COBJ_READ(pid_n)=SUCCEEDED,
        [2] COBJ_READ(pid_1)<>SUCCEEDED,
            ...
        [n+1] COBJ_READ(pid_n)<>SUCCEEDED,
        [n+2] COBJ_READ(pid_T)
        )<>1 THEN TRANS_ABORT(tid)
    TRANS_END gta(tid, TRUE)

ENTRY remote_call(ADDRESS:MDI;QUERY:q)IN(tid_f)
    USES(STAT OID:pid,tid;INT OID:cid;RES OID:rid)
    TRANS_BEGIN(tid,tid_f)
        TRANS_END(tid,compensate(cid))
    PROCESS(MDI,(lsys_call(q, cid, rid)),pid)
    IF ALT_WAIT(
        [1] COBJ_READ(pid),
        [2] COBJ_READ(rid)
        )=1 THEN TRANS_ABORT(tid_f)
    ELSE COBJ_WRITE(tid_f,cid,COMMIT)
    END remote_call

TRANS_ENTRY lsys_call(QUERY:q;INT OID:cid;
    RES OID:rid)WITH(tid)
    result ← db_execute(q)
    IF result.state<>PREPARED THEN TRANS_ABORT(tid)
    TRANS_BEGIN(tid ← COBJ_NEW(STAT),tid)
        COBJ_WRITE(tid_1,rid,result)
        TRANS_END(tid_1,TRUE)
    IF COBJ_READ(cid)=COMMIT THEN db_commit
    ELSE db_abort
    TRANS_END lsys_call(tid,TRUE)

TRANS_ENTRY compensate(INT OID:cid)WITH(tid)
    COBJ_WRITE(tid,cid,ABORT)
    TRANS_END compensate(tid,error)
```

Figure 4: 2PC MDBS Communication Protocol.

References

[1] H. Bal, *Programming Distributed Systems.* New York: Prentice Hall, 1990.

[2] H. Bal and S. Tanenbaum, "Distributed Programming with Shared Data," *Computer Languages Journal,* Vol. 16, No. 2, 1991.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] Y. Breitbart, A. Silberschatz, and G. R. Thompson, "Reliable Transaction Management in a Multidatabase System," in *Proceedings of the 19990 ACM SIGMOD International Conference on Management of Data* (H. Garcia-Molina and H. V. Jagadisch, eds.), May 1990.

[5] M. W. Bright, A. R. Hurson, and S. H. Pakzad, "A Taxonomy and Current Issues in Multidatabase Systems," *IEEE Computer,* March 1992.

[6] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys,* Vol. 21, September 1989.

[7] O. Bukhres, e. Kühn, and F. Puntigam, "A Language Multidatabase System Communication Protocol," in *Proceedings of the 9th International Conference on Data Engineering,* IEEE, April 1993.

[8] A. K. Elmagarmid, ed., *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1991.

[9] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, "A Multidatabase Transaction Model for InterBase," in *Proceedings of the 16th International Conference on Very Large Data Bases,* August 1990.

[10] A. Forst, e. Kühn, H. Pohlai, and K. Schwarz, "Logic Based and Imperative Coordination Languages" (submitted for publication).

[11] H. Garcia-Molina, and B. Kogan, "Node Autonomy in Distributed Systems," in *Proceedings of the Proceedings of the International Symposium on Databases in Parallel and Distributed Systems,* IEEE, December 1988.

[12] D. Gelernter and N. Carriero, "Coordination Languages and their Significance," *Communications of the ACM,* Vol. 35, February 1992.

[13] S. Heiler, M. Siegel, and S. Zdonik, "Heterogeneous Information Systems: Understanding Integration," in *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems,* IEEE, April 1991.

[14] C. Hoare, "Communicating Sequential Processes," *Communications of the ACM,* Vol. 21, August 1978.

[15] e. Kühn, H. Pohlai, and F. Puntigam, "Concurrency and Backtracking in Vienna Parallel Logic," *Computer Languages Journal,* Vol. 19, No. 3, March 1994 (to appear).

[16] e. Kühn and F. Puntigam, "Reliable Communication in VPL," in *Proceedings of the Parallel Architectures and Languages Europe (PARLE-92),* Springer Verlag, LNCS, June 1992.

[17] e. Kühn, F. Puntigam, and A. K. Elmagarmid, "An Execution Model for Distributed Database Transactions and its Implementation in VPL," in *Proceedings of the International Conference on Extending Database Technology, EDBT'92,* Springer Verlag, LNCS, March 1992.

[18] e. Kühn and K. Schwarz, "A Framework for Reliable Multidatabase Communication Protocols" (submitted for publication).

[19] M. C. Little, *Object Replication in a Distributed System.* PhD thesis, The University of Newcastle upon Tyne, September 1991.

[20] W. Litwin, A. Abdellatif, B. Nicolas, P. Vigier, and A. Zerounal, "MSQL: A Multidatabase Manipulation Language," *Information Science, An International Journal,* June 1987. Special Issue on DBS.

[21] W. Litwin, L. Mark, and N. Roussopoulos, "Interoperability of Multiple Autonomous Databases," *ACM Computing Surveys,* Vol. 22, No. 3, 1990.

[22] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems,* Vol. 1, August 1983.

[23] F. B. Schneider, "Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial," *ACM Computing Surveys,* Vol. 22, December 1990.