Language Constructs for Reliable Distributed Real-Time Transaction Processing

Yong-Ik Yoon and Ju-Hyun Cho

Real-Time OS Section (DB Project)
Electronics and Telecommunication Research Institute
YuSong P.O. Box 106, Daeduk Science Town, Daejeon, 305-600, South Korea
Tel: +82-42-860-5239, Fax: +82-42-860-6224

E-mail: yiyoon@nice.etri.re.kr

Abstract

Distributed transaction processing requires many message exchanges over the networks, which can hinder the timely completion. It is necessary to reduce the number of intersite communications for real-time processing. To ensure the timeliness of transactions, the time constraints are specified and reflected in the transaction processing time. In addition, it must guarantee the correct completion of the transactions. So, we must satisfy both correct completion and timely completion in a distributed real-time transaction processing environment. In this paper, we suggest new language constructs for the distributed real-time transactions. The language constructs consists of three features as follows: the specification of the time constraints, the asynchronous remote processing, and handling the missed time constraints. These features will reduce the number of message exchanging, reflects the time constraints for the timeliness of transactions, and ensure the correctness of distributed transactions.

Keywords Real-Time Transaction Processing, Real_time Database, Commit Protocol

1 Introduction

In real-time systems, all applications have stringent timing constraints. The primary goal of real-time systems is to meet the timing requirement of each application [9, 12]. A growing importance of real-time systems in many applications such as telecommunication systems, nuclear reactor control, and military systems stimulated database research toward real-time database systems [10]. They require timely completion so that some data manipulation operations should be executed within a specified time constraint [11]. The real-time database systems need to operate in distributed environment due to fault tolerance and/or inherent distributed nature of some applications [3, 6].

Proceedings of the Fifth International Conferennce on Database Systems for Advanced Applications, Melbourne, Australia, April 1-4, 1997.

In a distributed environment, a distributed program requires a group of cooperating processes distributed over the whole system. It is natural for the system to require correct completion so that a group of processes behaves consistently in the presence of failures [4]. A key property on the correct completion is failure atomicity which means that either a program is executed successfully, thus producing the intended results, or it has no results at all. The concept of transaction is applied to the processes for the reliable distributed processing since a transaction is the atomic unit for consistency and recovery [4]. These processing is called as the distributed transaction processing [1, 2].

There are several problems when the distributed transaction processing is adopted for real-time applications. The distributed transaction processing requires many message exchanges over the networks, which can hinder the timely completion. Because the speed of communication network limits the performance of distributed systems, it is necessary to reduce the number of intersite communications for real-time processing. Other important aspects for the timeliness are how to specify the time constraints and how to execute the transactions within the time constraints. In addition, it must guarantee the correct completion of the transactions. To summarize, we must satisfy both correct completion and timely completion in a distributed real-time transaction processing environment. In this paper, we suggest new language constructs for the distributed real-time transactions.

2 Necessity of Language constructs

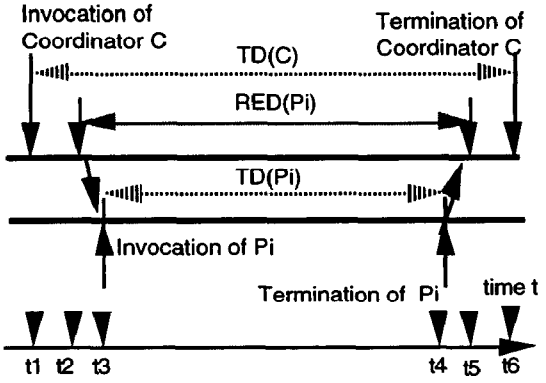
For the reliable distributed real-time transactions, three features must be considered in language constructs: the specification of time constraints that have to be satisfied for the timeliness of transactions, asynchronous remote processing between distributed transactions, and exception handling for supporting the reliable processing of faulty transactions. In this section, we discuss the necessity of these three features.

2.1 Necessity of The Specification of Time Constraints

In conventional real-time systems, transaction is characterized by several time constraints: invocation time, execution time, and deadline. In distributed real-time transactions, a coordinating transaction (called coordinator) requires one more time constraint, which specifies how long the transaction is willing to wait for some results of participating transaction (called participant) invoked by the transaction's request.

Considering of these time constraints, we define four kinds of time constraints for the real-time transactions: invocation time (IT), execution time (ET), transaction deadline (TD), and remote execution deadline (RED). The first two time constraints are required to support a scheduling policy to guarantee the timely execution of transactions. The invocation time is defined to be a time when a transaction is created. The execution time is an estimated computation time for each transaction. The transaction deadline is required to ensure the completion of a transaction, whereas the remote execution deadline is required to ensure the completion of a remote request.

Figure 1 shows some relationships between the time constraints. When a coordinator C in site Sl need a remote request to a distributed other site Sm, the coordinator generates a remote execution deadline RED(Pi) for a participant Pi which processes the remote request in Sm and sends the remote request message with RED(Pi) to Sm. When a remote request from a coordinator C arrived in the site Sm, newly participant Pi is created with its invocation time IT(Pi), it is time point t3. At the same time, a transaction deadline TD(Pi) for Pi is assigned the participant Pi. The participant must be terminated within TD(Pi), it is time point t4. Here, we get a timing relationship between RED(Pi) and TD(Pi), which are related to the participant Pi. The relationship is that RED(Pi), which is a time interval between time point t2 and t5, includes TD(Pi) which is a time interval between time point t3 and t4. That is, the execution results of participant Pi may be transferred to its coordinator C within RED(Pi).



. TD(C): Transaction Deadline of Coordinator C (t6-t1)
. TD(Pi): Transaction Deadline of Participant Pi (t3)
. RED(Pi): Remote Execution Deadline for Participant Pi (t5-t2)

Figure 1: Time constraints for distributed real-time transaction processing

Based on this relationship, we define a theorem that is very useful to decide the correct completion in the distributed transaction processing. The correct completion is achieved through a fact, which all the participants must be timely executed in the given time constraints, RED(Pi) and TD(Pi). So, to guarantee the correct completion between the coordinator and the participants, all the transactions must satisfy this theorem. This theorem plays a role of the criterion for timeliness in commit procedures.

[Theorem 1] The transaction deadline of participating transaction, TD(Pi) is always less than its remote execution deadline of coordinating transaction, RED(Pi).

[Proof] Since the coordinator and the participants in the distributed systems are on different sites, they may be subject to arbitrary communication delay. In real-time processing, such delay is undesirable and sometimes cannot be tolerated. The timeliness for distributed real-time systems depends on the execution time of all the participants in the remote sites, the traffic on the network, and the failures of the communication network. When a transaction requests a remote processing to other sites, the participants for the remote request may be executed in a given deadline, RED, which includes both the longest processing time among the processing times of participants and the communication delay time. If the specified deadline for a remote request, RED, cannot be satisfied by the participants, the results produced by the participants will be not useful. This is because RED indicates that the coordinator is willing to wait for the completion of all the participants and the result of participants may be used by the coordinator after RED. Therefore, a participant's deadline TD(Pi) must be less than the coordinator's remote execution deadline RED(Pi) as shown in Figure 1. ❑

The time constraints may be specified and reflected to guarantee the timeliness of a transaction. To accomplish this, we suggest a new mechanism to specify the time constraints in a transaction and to reflect the time constraints on the transaction processing.

2.2 Necessity of Asynchronous Processing

The second feature of the language constructs for real-time processing is the asynchronous processing to support the several characteristics for the distributed real-time processing. One characteristic is that some transaction wants to continue its local activities during a remote request generated by itself is in progress in the remote site. For example, when a transaction requests an initialization of a remote database, the transaction wants to execute its local jobs without interruption during the initialization. Consider another example to execute the queries over objects, which are distributed in several sites. The coordinator, which needs a remote request for the query processing, sends

the request messages to the distributed sites. In the sites, newly participants are invoked. The participants process the requested query and return its execution results to the coordinator. After then, the coordinator receives the participant's execution result which arrives asynchronously. Here, the coordinator need to keep the result messages arrived asynchronously until these are used by itself.

Based on the examples, we define two requirements for the asynchronous remote processing as follows; One is that the coordinator should be able to continue its execution while a remote request, which is generated by the coordinator, is in progress in the participating site. The other is that the coordinator should effectively manage the multiple remote requests. To meet these requirements, we suggest some syntax and semantics for the 'send-and-no-wait' communication primitives and a data structure to keep the asynchronously arrived messages.

2.3 Necessity of Exception Handling

The last feature of the language constructs for real-time processing is an exception handling mechanism to resolve a missed deadline or system failures. In the distributed systems, must be dealt with partial failures such as processor crash and communication delay. If the failures occurred, transactions related to the failures may miss their deadlines. The transactions will, then, generate an exception to handle the failure. So, we need an exception handling mechanism to deal with the failure situations. We suggest a mechanism to handle the exception at the language level. That is, the mechanism deals with the exception through a functional mapping which is achieved by attaching the exception to its handling routine when an exception is happened. For the functional mapping, each transaction need to define some exception types and handling routines for the exceptions. To do this, we describe the syntax and semantics of well defined exception handling.

3 Specification of Time Constraints

We defined four kinds of time constraints so far: invocation time (IT), execution time (ET), transaction deadline (TD), and remote execution deadline (RED). In real-time systems, TD and RED among these time constraints typically take the form of deadline to guarantee the timely execution of transactions as shown in Figure 1. The time constraints must be specified and reflected so that each transaction could be timely executed to satisfy the timely correctness. To specify the time constraints, we firstly describe how to decide the value of time constraints. And then, we describe how the values are assigned to the time constraints for each transaction.

3.1 Decision Rules on Time Constraints

To decide proper values for time constraints, we have to consider the timing predictability of a transaction. The predictability is an important requirement of real-time systems; specifically, it is an indication to predict whether or not a set of transactions will meet their timing requirements. To ensure predictable executions, the ability to determine the execution time of transaction's code segment is a prerequisite. For example, bounded loops and bounded recursions in execution code segment should be pre-determined, but an arbitrary delay for unbounded time periods should be disallowed. In real-time transaction processing, however, includes the unbounded time periods which are generated by several unpredictability sources as the following:

. Dependence on remote transaction execution
. Blocking due to data and resource conflicts
. Swapping and I/O processing
. Transaction abortion, rollback, and restart

For the predictability, we need to know the worst case execution time under both the bounded time and unbounded time. There are two aspects to determine the worst-case execution time of a transaction: its execution must be bounded and the variance in the unbounded execution should not be large. The latter need arises from the fact that to guarantee the timeliness of a transaction, resources (including time) needed for it must be found with respect to worst case needs. The worst case execution time becomes a deadline which is needed to guarantee a transaction's timeliness. That is because a deadline means a time boundary that a transaction must be executed within the given time requirements. Based on this fact, we can find some rules to assign the proper values for the time constraints as follows:

```
ET  := estimated_excution_time;
RED := estimated_remote_execution_time +
       (2 * communication_delay) +
       slack_factor_for_remote_request;
TD  := ET + (summation of RED) +
       slack_factor_for_completion;
```

The execution time of a transaction, ET, is the estimated computation time of a transaction under the predictability, but it includes only the bounded time of code segments. The remote execution deadline, RED, includes the estimated computation time of remote transactions, communication delay, and a slack factor for the remote request. The slack_factor_for_remote_request is a time of how long a transaction can wait to use the result of a remote request. It is decided by considering the transaction's status, how many remote requests will be generated and when the result of remote request will be used. The transaction deadline, TD, is decided through follows: the estimated execution time, the summation of RED, and slack_factor_for_completion, which is a time how long a transaction is allowed to delay to complete its execution.

In these rules, the estimated execution times are the bounded time. Communication delay and slack factors are the unbounded time for distributed real-time processing. The estimated execution time cannot be changed since the deadline is decided at the specification time. At some time, however, we need to change the

deadline to support the flexibility and to increase the possibility for the timeliness of transactions via considering the current status of systems. To this effect, the rules should have the slack factor which is a parameter to control the tightness or looseness of the deadline. In other words, the rules should support the flexibility on the deadline decision which is dynamically evaluated and specified at the execution time by using the slack factor. Thus, the deadlines can be decided fixedly at specification time and flexibly at execution time. The time decision rules may be pre-defined in the language constructs for transaction definition.

3.2 Assignment Policy of Time Constraints

Considering the deadline decision rules as previously explained, there are two different approaches of assignment policies with which the time constraints can be specified in language constructs and be reflected during the distributed real-time transaction processing. These are static assignment and dynamic assignment. The static assignment is made at the transaction definition time. That is, the time constraints are explicitly assigned in the transaction definition when the values are decided at the specification time. The dynamic assignment is made at the invocation time of a transaction. That is, the deadlines are decided by using the slack factor and the input parameters, one of which is a remote execution deadline passed from coordinating transaction. The decided values are implicitly assigned to the time constraints at run-time. The assignment policies are distinguished by whether the value of slack factor is fixed or not. Let us see how the policies work.

Static Assignment Policy

The values of the time constraints are decided by using a slack factor that was already fixed at the transaction definition time. That is, during the real-time distributed processing, the values cannot be exchanged.

Dynamic Assignment Policy

At transaction invocation time, the values of the time constraints are decided with the given requirements, i.e., a predictable execution time which is bound and described at the transaction definition time and the input arguments indicating current status of calling transaction and timeliness of called transaction. So, under this policy, the values of slack factor are evaluated in relation to the input arguments. The deadline is dynamically decided by using the flexible slack factor and given requirements. That is, the value can be changed for the real-time processing through the slack factor.

Although we allowed two approaches for the time specification, the dynamic assignment is more desirable because we can reflect a system status during run-time. That is, the flexible specification method increases the timeliness of transactions because each transaction's deadline is assigned dynamically by the current status of transactions. Example 1 shows the flow of dynamic assignment policy that the slack time of remote request are evaluated in the coordinating transaction and are used in participating transactions.

Example 1 Suppose that a coordinator Tc requests a remote processing and a participant Tp is invoked for the remote request. The remote execution deadline of Tp, RED(Tp), is decided before transmitting the remote request to Tp and RED(Tp) is passed to the participating site at the time of transmission. At the invocation time of Tp, the passed RED(Tp) are used to decide the transaction deadline for Tp, TD(Tp), as shown in Figure 2. ❑

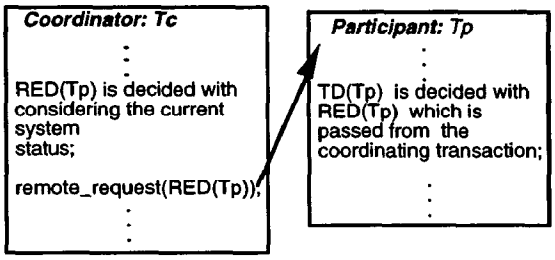


Figure 2: The flow of dynamic assignment of TD

4 Asynchronous Remote Processing

4.1 Design Criteria for Asynchronous Remote Processing

In the literature, several asynchronous remote processing mechanisms have been designed and implemented to achieve high parallelism while retaining the simplicity of synchronous remote processing mechanisms. There are several criteria used for the design of asynchronous remote processing. Firstly, an asynchronous remote processing must have the look and feel of a synchronous remote processing, except that the coordinator does not wait for a reply after making an asynchronous remote processing call. In this case, the coordinator may or may not be able to defer receipt of return replies. Secondly, asynchronous remote processing must be designed to improve throughput and response time. To do this, a multicast of asynchronous remote request may be needed. Lastly, asynchronous remote processing must keep the replies from the participants. This is because asynchronous remote processing can generate several remote request concurrently as multicasting of remote request and the coordinator must defer the using of replies on the request until itself needs the replies. So, a mechanism to defer the receipt of results must be supported.

For this effect, we need 'send-and-no-wait' communication primitives, which allow that coordinator proceeds locally in parallel with its participants. Also, we have to define data structures to manage and keep the asynchronously arrived messages in the coordinator. We first describe the data structures to keep the return messages and then describe the primitives for 'send-and-no-wait' semantics.

4.2 Data Structure for The Asynchronous Processing

Participants, which are invoked to support the remote request, return their executed results to its coordinator at the termination time. The return messages arrive asynchronously in the coordinating site. The coordinator need to keep the return messages until they are used to guarantee the correctness of the participants.

To do these, we define two data structures as shown in Procedure 1: multi_cast and return_array. These data structures are defined in the coordinator. The flag field in multi_cast means that some remote processing must perform in the indicated site. The destination_index field is an index to identify remote requests because many requests can occur concurrently. The status field keeps the status of participants and has a value between 'OK' and 'NOK'. To check the correctness of remote processing, there are three count fields in the return_array: the first for all participants (Pn), the second for commit participants (Pc), and the last for aborted participants (Pa). Pn is updated when coordinator issues a remote processing request. Pc and Pa are updated when the results arrive. The contents in status filed and three count fields are used in the commit procedure to guarantee the failure atomicity [13].

Procedure 1 Data structures for the asynchronous remote processing

```
struct multi_cast {
        int       flag; };
        struct    return_array {
        unsigned destination_index[MAX_SITES];
        int         status[MAX_SITES];
        int       Pn; /* number_of_participants */
        int       Pc; /* number_of_commits */
        int       Pa; /* number_of_aborts */ };
```

4.3 Communication Primitives

To support 'send-and-no-wait' semantics, we define three kinds of communication primitives: request_rpc to transmit request messages, reply_rpc to return result messages, and receive_case_rpc to receive the request message. A detailed format of these primitives is summarized in Procedure 2. The sending primitives are distinguished by the notion of transmitted message because request_rpc invokes a new participant for the request and reply_rpc only transmits the executed result messages to its coordinator. For the receipt of messages, we only has one primitive like receive_case_rpc, which checks the type of request message and invokes a new participant.

Procedure 2 Syntax of the extended remote procedure call primitives

```
request_rpc (request_type,
                    remote_execution_deadline)
        unsigned request_type;
        unsigned remote_execution_deadline;

reply_rpc(status)
        unsigned status;

receive_case_rpc (request_type) {
        case request_type: /* statements for getting
                              the returned values */
                           :
        default: };
        unsigned request_type;
```

Example 2 shows how the asynchronous remote processing works through both the suggested primitives and the data structures.

Example 2 Suppose that there are N different sites; a coordinator T1 in S1 makes a remote processing request to Si and Sj among sites; each site has a daemon to handle the request; and the daemon generates participants for the requests. As shown in Figure 3, T1 creates two data areas as follows: named MC for multicasting of a remote request and RA for keeping the received results of participants. Before sending the request message, T1 assigns 'SET' value in ith slot and jth slot of the named MC as shown in arrow (A) of Figure 3. It means that this coordinator has to request a remote request to these two sites. Also, T1 decides the remote execution deadline to be guaranteed the timely correctness. After then, T1 requests simultaneously a remote processing to both Si and Sj according to the flag in MC. As shown in arrow (B) of Figure 3, the count field Pn in RA is increased and the 1st slot and 2nd slot of index field has the value for site Si and Sj, respectively. Then, the daemons in Si and Sj generate participants T1i and T1j for the remote request as shown in arrow (C). When the participants T1i and T1j terminate, they takes unilaterally a commit or abort action and reply the taken status to T1 as shown Figure 4. The result messages arrive asynchronously to T1 and are stored in the status field of return_array, named RM. In this example, because the participants taken the commit action, the status field is marked by 'OK' and the count filed for the commit, Pc, has '2'. ❑

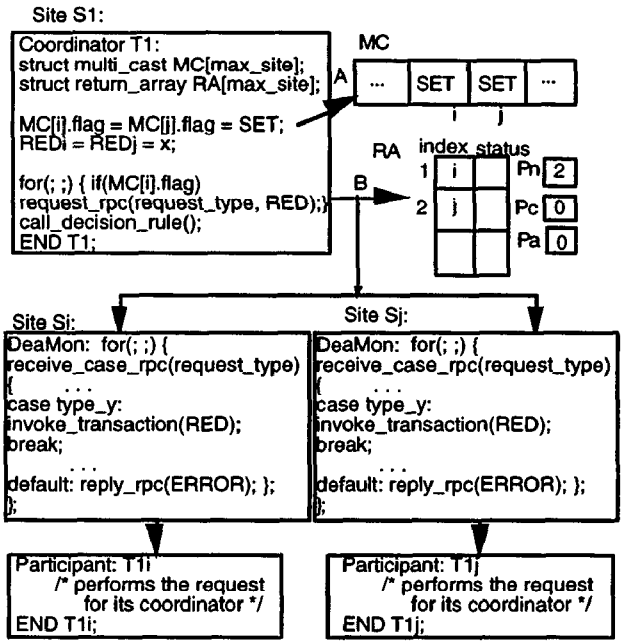


Figure 3: Flow of asynchronous remote processing - I

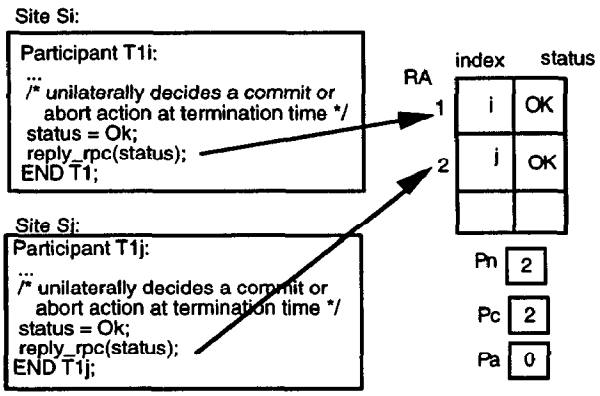
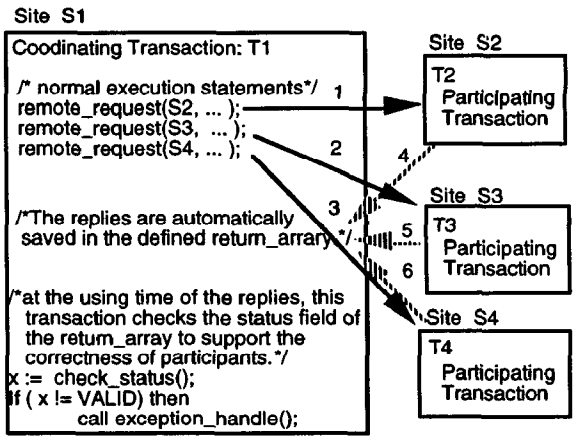


Figure 4: Flow of asynchronous remote processing - II



1,2,3 : request_message, respectively
4 : result_message with OK (It means normal completion)
5 : result_message with NOK (It means abnormal completion)
6 : no response from S4 (It means a failure was occurred in site S4)

Figure 5: Flow of asynchronous transaction processing

5 Exception Handling

5.1 Exceptions in Real-Time Transaction Processing

In the distributed real-time transaction processing, there are several kinds of faults that may result from software or hardware faults due to timing errors or component failures. The faults are directly related to the timeliness of transaction processing in the distributed environment. When a fault is detected, the transaction that detected the fault generates an abort action to guarantee the atomicity between transactions. The atomicity requires that an aborted transaction have no effect on the state of the database. The most common method of performing the abort is to provide the undo operation which restores the data updated by a transaction to the value they had just prior to the execution of the transaction.

Here, we need a scheme to resolve the exception cases which are caused by faults. Example 3 shows how an exception can happen during remote transaction processing.

Example 3 Suppose that there are four sites; S1, S2, S3, and S4. In S1, a coordinator T1 sends a remote_request message to other sites; S2, S3, and S4. In the remote sites, its participants T2, T3, and T4 are invoked, respectively, as shown in Figure 5. When the remote processing of the participants is completed, the executed results are committed and retransmitted to T1. The replied messages are automatically saved in the defined return_array. The coordinator will use the replied messages to decide the correctness of the distributed transaction processing. Suppose that T2 normally terminates its execution and transmits its executed results with OK signal to T1 as shown in arrow (4) in Figure 5. T3 abnormally terminates its execution and transmits its results with NOK signal to T1, as shown in arrow (5). And T4 stops its execution because S4 is failed as shown in arrow (6). At this time, T1 decides abort action to cancel the remote request because T1 knows that T3 has abnormally terminated and T1 could not receive the return message from T4. Then, an exception handling routine is invoked to support the decided action. ❑

5.2 Handling of Exceptions

As shown in Example 3, a coordinator cannot receive the reply messages from the participant by either a failure of remote site or a failure of participant. In this case, the coordinator generates an exception to resolve the situation. To handle the exceptions for reliable processing, the exception handling routines are defined in the application programs. The routines contain the operations to support the forward recovery and to resolve the inconsistency caused by exceptions or failures. The forward recovery is very useful method in real-time systems because this will reflect the advanced status of other world which is the current status of this other world, but the backward recovery cannot reflect the external world which has progressed to some other state. Thus, we must do the forward recovery to reflect the advanced status of other world.

To resolve the problems, we use the compensation concept that conforms to consistency constraints. The compensative routines undo an abortable transaction's effects in a semantic manner, rather than by physically restoring a prior state. That is, the compensative routines guarantee that a consistent state is established based on semantic information. To do this, each transaction has several exception handling routines which consist of compensative routines for time-out handling, error recovery, and undoing the updates. Procedure 3 shows the syntax for the exception handling. In the syntax, the event_type means the types of exception. Each event_type has a handling routine which consists of the statements for compensative routines or other routines to support the exception. The exception handling routines are declared in the transaction at the transaction definition time.

Procedure 3 Syntax of exception handling

```
EXCEPTION (event_type){
        case 1 : /* handling routines */
                 . . .
        case n:  /* handling routines */
        default:
} END_OF_EXCEPTION;
```

The routines for the exception handling behave as interrupt service routine, in order to handle the exceptions immediately. Considering Example 3, the coordinator generates an exception to handle the inconsistent remote processing. The exception handling is achieved by generating an event which are declared in the transaction, and then, executing the compensative routines defined in the exception block. Figure 6 shows how to proceed the exception handling routines in real environments. If an exception occurs, a handling routine defined in the transaction is executed to handle the exception cases.

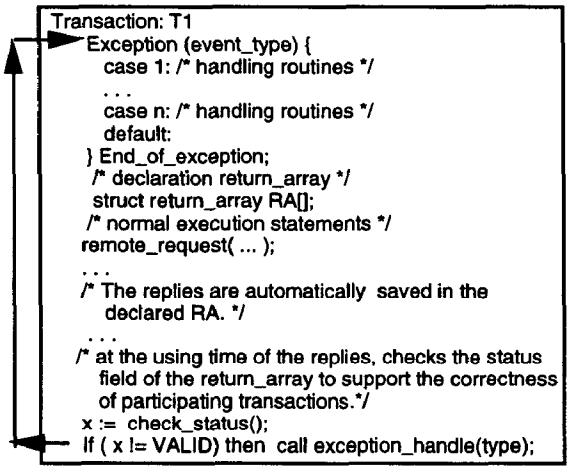


Figure 6: Flow of Exception Handling

6 Implementation of Language Constructs

To implement the language constructs in the real-time environment, we need a real-time operating systems. Since we only have non real-time operating systems named UNIX, we developed a simulation environment, called Real-Time Executive (RTE) on UNIX. Under RTE, we simulate a distributed processing through mapping object files to remote sites. RTE has several facilities to support multiprocessing in a single object file and to communicate between processes in each object file. RTE consists of several functions as follows: process manager to support process creation and termination function, communication manager to support the message exchange, scheduler to control the execution order between processes. Here, based on RTE, we describe an implementation method to support the asynchronous remote processing through a shared memory in tightly coupled system and the exception handling.

6.1 Implementation of Asynchronous Remote Processing

6.1.1 Logical Inter-connection among Sites

UNIX has a facility, called shared memory, this is a useful way for the message exchange between distributed transactions. The shared memory is a logical interconnection between object files, through which a message is transmitted between transactions in each other object files. Hereafter, we call an object file site. When a transaction transmits a message to another in a remote site, the transmission is achieved by a pre-defined shared memory for receiving transaction. There are lots of shared memory for the message exchanges between sites. That is, a coordinator defines a shared memory to transmit a request message to its participants and the participants also transmit their result messages to the coordinator via the shared memory, which is defined by the coordinator. The shared memories are defined in main memory at the initialization time of RTE as shown in Figure 7.

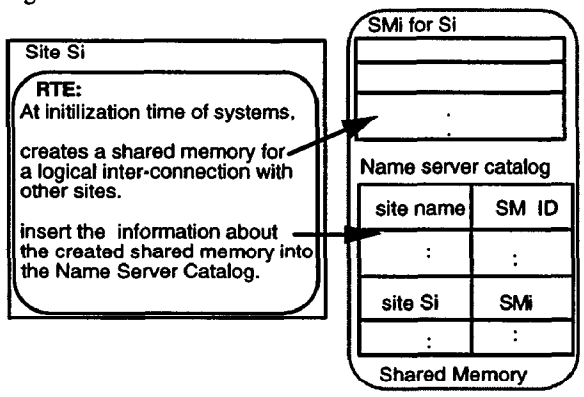


Figure 7: Flow of the creation of shared memories

In Figure 7, the shared memories consist of a fixed size space for the message exchange. The information of these shared memories are preserved in name server catalog by RTE. The name server keeps site name and shared memory ID, these indicate a way to connect between sites. Figure 7 shows how the shared memories are created. At initialization time of RTE for coordinating site, RTE creates shared memories for inter-site communication. Then, RTE records the information about the shared memories in the name server catalog that preserves all information for the communication.

6.1.2 Format of The Transmission Message

The shared memory has a format for transmission messages as shown in Figure 8. The format consists of five fields as follows:

. CallerT_ID field: the coordinator's identifier.
. TransactionType field: a participant's type to execute the remote request.
. MessageType field: a type of message, i.e., INVOKE, ABORT or COMMIT.
. MessageSize field: the size of sending message.

. Pointer field: the address of message contents.

Caller's T_ID (4 bytes)	Transaction Type (4 bytes)	Message Type(4 bytes)	Message Size(4 bytes)	Pointer (4 bytes)

Figure 8: Slot's format in channel

6.1.3 Language Interface

We have defined two kinds of sending primitives, request_rpc and reply_rpc, and one receipt primitive, receive_rpc. When a transaction wants to communicate with other transactions, a communication link is made through the library routines, which are defined in the application program interface (API). Procedure 4 shows the library routines to process the asynchronous remote requests.

Procedure 4 Library routine for asynchronous remote processing

```
request_rpc:
        message_ptr:=
        attach_shared_memory(site_id);
                encode_message(message_ptr,
                coordinator_id, transaction_type,
                message_type,
        remote_execution_deadline,
        message_size, message_list);
                set_flag(site_id);
        return;

receive_rpc:
        message_ptr:=
        attach_shared_memory(site_id);
        decode_message(message_ptr);
        switch_by_message_type();
        return;

reply_rpc:
        message_ptr:=
        encode_message(participant_id,
                size_of_result_messages,
                result_messages);
        return_array :=
        attach_shared_memory(return_array_id);
        return_array->status := result_status;
        return_array->message_ptr := message_ptr;
        return;
```

6.1.4 Example of Asynchronous Processing

When a coordinator transmits a message by using request_rpc, the message is transmitted through the library routine that places the request message into the shared memory for receiving site. After then, the coordinator does not wait a return message from the participant, but continuously executes its further progression. When a request message arrive in a participating site, daemon process invokes a participant to execute the remote request by using receive_rpc primitive. Example 4 shows the progress of the asynchronous remote processing.

Example 4 Suppose that there are two sites: one is a coordinating site Si and the other is its participating site Sj. Also, suppose that Sj has a shared memory SMj for communication and an event flag EFj on the shared memory SMj. Note that event flag has two states as follows: 'SET', which means that a message exist in its shared memory, and 'CLEAR', which means that a message does not exist in one. The event flag plays the role of informing one of two communication status to each transaction: either a transaction places a message in a shared memory or a transaction wants to receive a message via shared memory. Figure 9 shows the transmission flow of a message for remote processing. When a transaction Tc in Si transmits a request message Mj to its participating site Sj, the message is placed in SMj. SMj includes several items as follows: ID of Tc, transaction name for remote processing, INVOKE message for the invocation of participant, message size, and address of message contents. At the same time, event flag is changed to 'SET' to inform this request to Sj. That is, Tc posts 'SET' in EFj. After then, the coordinator returns to next statement of this request and resumes its works.

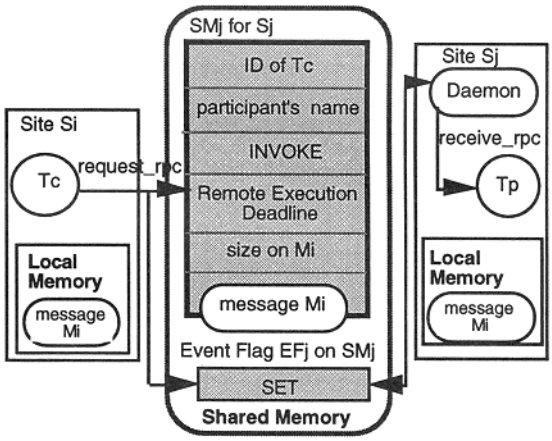


Figure 9: Processing flow for both request_rpc and receive_rpc

In site Sj, the daemon process periodically checks EFj for SMj. If the state of EFj is 'SET', the daemon process takes off the message Mj from SMj since it finds that a message arrives at this site. Then, by using receive_rpc primitive, the daemon process checks the message type in Mj and invokes a new participant Tp with the content in Mj. If it is 'CLEAR', then the daemon process pends in EFj until it becomes 'SET'.

When Tp completes its works, Tp transmits its result message to Tc by using reply_rpc primitive. Figure 10 shows the transmission flow of the result messages from participants to the coordinator. When Tp issues a reply_rpc primitive, RTE supports that the result message directly is placed in the return-array that is defined in the shared memory by Tc. Hence, Tc can use the result message immediately, which asynchronously arrives, without issuing the receive primitive intentionally for taking the result from its participant. ❑

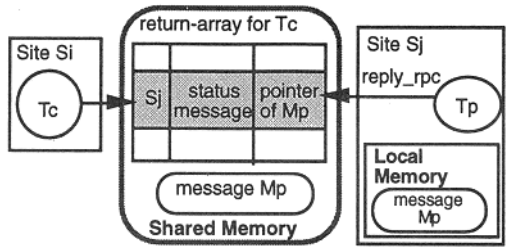


Figure 10: Transmission flow via reply_rpc primitive

6.2 Implementation of Exception Handling

6.2.1 Specification of Exceptions

Exception handling is defined as the activities performed by the processor in preparing to execute a handler routine for any condition that causes an exception. In particular, exception handling does not include execution of the handler routine itself. In our model, each transaction includes its exceptions and their routines in the transaction definition. When an exception occurs, the faulty transaction executes its handling routine for the exceptions. To support the exception handling, a basic framework for representing exceptions and their handlers are desired.

First, we show how the exceptions and their routines are placed in a transaction. Suppose that a transaction Ti contains declarations for the exceptions and also contains handling routines for them. When Ti is compiled in real environment, the compiler generates the codes about specification of Ti's exceptions and their routines as shown in arrow A in Figure 11. The code constructs a mapping table that consists of exception names and address of handling routines. When a transaction is invoked, they are placed in the transaction control block for Ti as shown in arrow B in Figure 11. That is, a filed of Ti's TCB keeps the address of mapping table. It is used to handle the exceptions which occur in Ti.

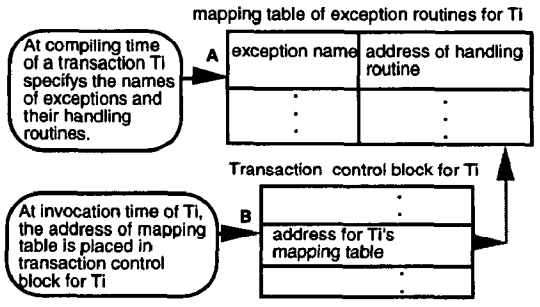


Figure 11: Specification of exceptions and handling routines for a transaction

To support the specification of exceptions in RTE, we define set-exception primitive. This primitive constructs a mapping table for exception handling on a transaction. When a transaction is declared, exception type and routines are also declared through the set-exception primitive. For example, if a transaction T1 defines two kinds of exception types, abort and timeout, and their handling routines, t1_abort and t1_timeout, T1 registers these information for its exception handling by using the set-exception primitive at transaction declaration time as shown in Figure 12.

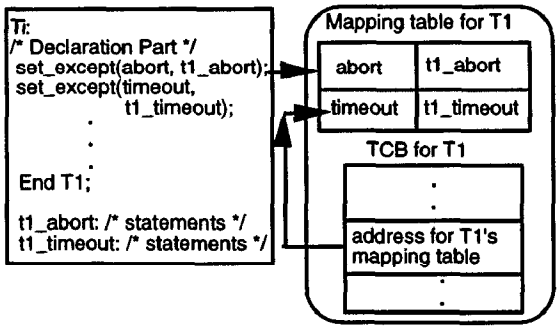


Figure 12: Flow of specification through the set-exception primitive

In Figure 12, when T1 issues set-exception primitives, RTE constructs a mapping table for these requests, like arrow A in Figure 11, and assign the pointer for the mapping table in address field in TCB for T1, like arrow B in Figure 11. Hence, this specification method has the same effect in case a complier is provided as shown in Figure 11.

6.2.2 Exception Handling

Regardless of how an exception is raised, it is handled by its associated exception handling routine. We describe how an exception is handled. In our systems, there is an exception handler that looks like interrupt service routine. The exception handler has a role of mapping between an exception and its handling routine. When an event like timeout occurs, the exception handler checks the type of event and the faulty transaction. Then, it generates the corresponding exception and takes the address of handling routine from the faulty transaction's TCB. The handling routine is conducted as shown in Figure 13.

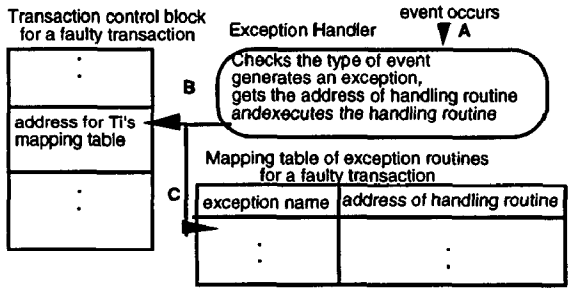


Figure 13 Flow of exception handling for an event

7 Conclusion and Futher Study

We so far have presented the language constructs for the distributed real-time transaction processing. The language constructs are composed of the three features as the following:

. The specification rules of time constraints to reflect the real-time requirements at run time.
. The asynchronous communication primitives and the management rule of the return messages generated in the course of the communication.
. The registration method of the exception handling routines that resolve faulty situations.

To implement the language constructs, we developed a simulation environment called Real-Time Executive (RTE). Under RTE, we implemented the language constructs. The transaction described by using the language constructs has its time constraints, the asynchronous remote procedure calls, an array to preserve the return messages for the remote request, and the exception handling routines to support the failures. Here, because the transaction specifies the time constraints to ensure its timeliness, the time requirements are reflected in the execution of the transaction. The scheduler in Real-Time Executive makes use of the time requirements so that the real-time transactions should be completed within the time constraints.

Furthermore, the communication between coordinator and participants is done very fast. It is because the coordinator receives the result message immediately from participants without issuing the receive primitive intentionally for taking the result from its participants.

References

[1] Stefano Ceri and Giuseppe Pelagatti, Distributed Database: Principles and Systems, McGraw-Hill, 1984.

[2] Ahmed K. Elmagarmid, Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers, 1992.

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

[4] S. K. Shrivastava, Resilient Computing Systems, Collins Professional and Technical Books, 1985.

[5] Susan Davidson, Insup Lee, and Victor Wolfe, "Timed Atomic Commitment," MS-CIS-88-80, GRASP Lab 156, Dept. of Computer and Information Science, University of Pennsylvania, October 1989.

[6] Hector Garcia-Molina, et. al, "Coordinating Multi-Transaction Activities," University of Maryland, Computer Science Technical Report, UMIACS-TR-90-24, February 1990.

[7] Kwei-Jay Lin, "Consistency Issues in Real-Time Database Systems," Proceedings of 22nd Hawaii International Conference on Systems Sciences, January 1989, pp. 654-661.

[8] Pui Ng, "A Commit Protocol for Resilient Transaction," Technical Report, UILU-ENG-87-1762 (TR-Illinois-57), University of Illinois, Urbana-Champaign, Illinois, USA, September 1987.

[9] K. G. Shin, "Introduction to the Special Issue on Real-Time Systems," IEEE Trans. on Computers, August 1987, pp. 901-902.

[10] Mukesh Singhal, "Issues and Applications to Design of Real-Time Database Systems," SIGMOD Record, Vol. 17, No. 1, March 1988, pp. 19-33.

[11] Sang H. Son, "Real-Time Database Systems: A New Challenge," IEEE 6th International Conference on Data Engineering, Vol. 13, No. 4, December 1990, pp. 51-57.

[12] John A. Stankovic, "Misconceptions About Real-Time Computing," IEEE Computer, October 1988, pp. 10-19.

[13] Yong-Ik Yoon and Song C. Moon, "Integrated Commitment Protocol for Parallel Transaction Processing in Real-Time Systems," Journal of Microprocessing and Microprogramming, Vol. 40 1994, pp. 151-166.