A TINA-like Computational Model and its Implementation within an Object-Oriented Distributed Transactional Platform

Pier Giorgio Bosco, Ennio Grasso, Giovanni Martini, Corrado Moiso

CSELT
via Reiss Romoli 274 - 10148 Torino (Italy)
e-mail: {bosco,grasso,martini,moiso}@cselt.stet.it

Abstract

This paper describes ODIN (Object Distributed INterfaces), a prototype implementation of MODA (a Model for an Object Distributed Architecture), which is the CSELT model aligned with ISO/ODP Reference Model. This work presents a prototype implementation based on OSF/DCE and extends the results of a previous activity by implementing an extended transaction model.

1. Introduction

The design and deployment of distributed applications in telecom network systems have led to the need for introducing modelling concepts to master the software complexity imposed by the rapidly growing, highly diversified market demands and to specify components that are to be installed and administrated in an independent way.

To achieve such an ambitious, though strategic, objective, it is under investigation [4] an approach based on a single conceptual model that is rich enough to express the descriptive, computational and deployment needs of different areas like Intelligent Networks, TMN and Operation Systems. Within this framework, network and service applications are designed in terms of interactions between components and formally specified using computational modelling concepts.

MODA is the computational model proposed by CSELT and one of the inputs to the TINA-C computational model [3]. Due to the variety and complexity of the application requirements that the architecture intends to support, the computational model needs to be powerful and adaptable. In particular, the transaction model offered by MODA allows the exploitation of a high degree of concurrency, combining the capability of handling open and closed nested transactions as a result of invoking transactional operations.

ODIN [2] is the engineering counterpart of MODA and has been implemented over OSF/DCE [11] and Transarc/Encina [12]; though an implementation of ODIN over a CORBA system [10] is under development .

The first part of the paper outlines the main features of MODA and in particular its transactional capabilities. The remainder of the paper describes ODIN in some details and then its technology implementation.

2. The Computational Model

MODA is based on the object-oriented paradigm and provides a uniform description of the structure of different entities (programs, data, network resources,...) involved in a telecom application. The model addresses both structural and behavioural aspects by the definition of templates and primitives.

2.1. Structural aspects

The structure of a distributed system is expressed in terms of objects interacting through services offered by interfaces. The objects are clustered into software units, called building-blocks (BBs), which are units of modularity, installation, administration, failure and replication. A BB introduces the following visibility rules over object interfaces:

- the interfaces visible outside the BB are called contracts;
- an object can interact with all interfaces provided by the objects in the same BB and with contracts provided by objects in other BBs.

A distributed application will consist of one or more BBs, each installed on a single node (Fig. 1). A BB (instance) is the instantiation of a parametric BB template whose internal state and behaviour are described by a set of object templates, one of which will be the BB-manager, whose main goal is to initialize the BB itself. During run-time, a BB forms a scope for creation and execution of the objects whose templates are included in the BB template.

The structure and behaviour of the objects in a BB are described by parametric object templates. Objects are

a unit of functionality and encapsulation and can interact through services offered by a static set of multiple interfaces; an object template declares:

- the internal state variables;
- the interfaces provided by the object;
- the methods that implement the operations offered at the interfaces, and the initialization and the termination phases of the object.

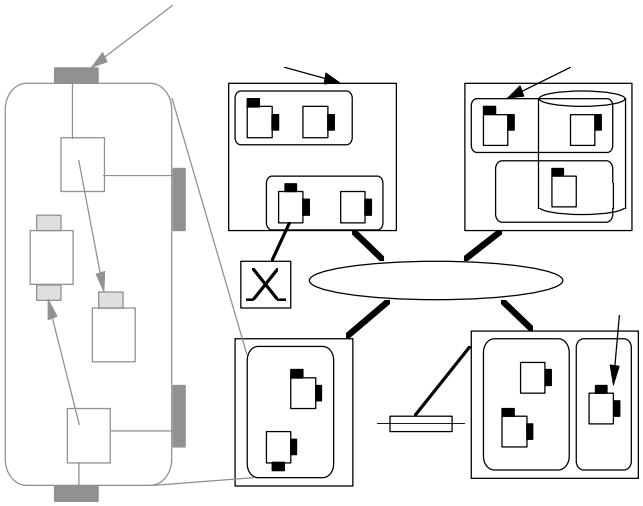

Fig. 1: distributed application with multiple building blocks

In MODA, an object interface clusters the computational entities pertaining to a particular service and provides the same kind of services defined by OSI managed objects [5], namely:

- operations to invoke methods of the object;
- attributes to read/write the state variables of the object;
- notifications to be notified of something that happens within the object.

2.2. Typing aspects

MODA relies on a type system that guarantees the correctness of the interactions among objects. In accordance with the ODP Reference Model [1], the type system introduces a subtype relation between object interface types: an interface T1 is subtype of T2 if it provides more operations, attributes, and notifications.

2.3. Behavioural aspects

The behavioural aspects considered by the model can be grouped in constructs to:

- perform interactions between objects;
- begin and complete transactions;
- create and delete instances of objects and BB;
- manage concurrent activities;
- access architectural services (e.g. the Trader).

Interaction constructs

A client object can interact with a server object by means of an interface offered by the latter. MODA supports two kinds of interactions:

- interrogations, with a request phase and a result phase;
- announcements, with only the request phase.

An interrogation, in turn, can be performed in either of two modalities:

- blocking: after having sent the request the client suspends and waits for the results;
- deferred blocking: after having sent the request the client continues its internal processing up to a point where it explicitly blocks and waits for the results.

An invocation permits the client to:

- request the execution of an operation; when the server object receives the invocation, it executes the method implementing the behaviour of the operation;
- request the access of an attribute; when the server object receives the invocation, it reads/writes the corresponding state variable;
- express the interest in receiving notifications; a client object that wishes to receive a particular kind of notification emitted by a server object must subscribe to the server by specifying the notification handler that will be executed when notifications are received. A notification emitted by the server object will be delivered to all subscribed objects.

Transactional behaviour

Like the TINA-C computational model, MODA supports the nested transaction model [6] on a per object basis. Nested transactions provide full isolation on the global level but permit increased modularity and finer granularity of failure handling than traditional flat transactions. Initiation of transactions is specified on the basis of tags associated with object operations. During the execution of a transactional operation an object can invoke other transactional operations. This forces an implicit nesting on the transaction itself leading to a tree of subtransactions one for each invoked operation.

If the invocation is issued from inside a transaction, the call spawns a nested transaction in the invoked object. On the other hand, if the invocation is issued from outside transactional boundaries, the call starts a top-level transaction in the invoked object.

Besides nested transactions with full ACID properties, MODA extends the transactional model by

allowing the definition of "open" nested transactions [7] so as to relax the ACID paradigm according to the application requirements.

In the classic nested model a subtransaction transfers all its locks to the parent at commit. This rule is mandatory to guarantee atomicity and isolation since a committed subtransaction may later be rolled back; but the very same rule may detriment concurrency. Conversely, an open nested transaction releases its locks at commit, thereby increasing the parallelism of the system.

Since the locks acquired by an open subtransaction are released at commit, abort recovery cannot rely on traditional rollback: rolling back an open subtransaction may unintentionally undo the effects of a second possibly interleaved transaction. Hence, abort recovery is performed by executing a compensating transaction that semantically reverses the effects of the subtransaction to be compensated. This relaxes atomicity since the compensating transaction need not necessarily undo all effects of the transaction compensated for, but can result in a semantic specific repair action.

In MODA, the declaration of a transactional operation with open semantics must specify two implementing methods: the *primary* and the *inverse* methods. The primary method implements the actual behaviour of the operation, whereas the inverse method will be automatically executed in the event that the operation has to be compensated. The inverse method executes as a separate transaction and is supposed to undo the effects of the whole tree whose root is the subtransaction compensated for (no nested compensation is required). Note that the specification of the inverse methods is not as big a problem as it could seem at first glance: object interfaces usually provide inverse operations (it would be of little value if the object did not allow some means to undo operations).

Concurrency control

MODA addresses both inter-object and intra-object concurrency (i.e. more execution threads in the same object). The model introduces two mechanisms to control and synchronize concurrent threads:

- guards: predicates associated with the methods of the object; the execution of a method will be delayed until the corresponding guard evaluates true;
- locks: allow a fine-grain control on thread and transaction synchronization.

MODA provides a mechanism to improve concurrency by exploiting the semantic knowledge of object operations, and in particular *commutativity* of operations [8]. In the simplest case the only two transaction operations one may consider are *read* and *write* object attributes. In general, object operations are not simply *read* and *write* but may be more abstract; if the semantics of object operations is taken into account it may be possible to permit much more concurrency and waits will be caused by conflicts based on the application semantics and will occur less frequently. For each MODA object template, the programmer can specify a conflict relationships between the object operations; two conflicting operations will be serialized, while two compatible operation will be admitted concurrently.

The model relies on a programmable locking tool as concurrency control protocol. In addition to the usual *shared* and *exclusive* lock-modes, dealing with abstract operations requires specific lock-modes such that two lock-modes are compatible iff their corresponding operations do not conflict.

Creation and deletion of instances

MODA allows a distributed system to deploy several instances of the same BB template, possibly on the same node. A request to instantiate a new BB specifies:

- the name of the BB template;
- the list of actual parameters used to configure the instance;
- the name of the target node.

The actions performed when a BB is instantiated on a node are:

- retrieving and loading the executable code on the node;
- creating the BB-manager, whose initialization phase performs the configuration of the BB;
- returning the interface reference of the BB-manager.

An object in a BB may create new objects in the same BB; object instantiation is specified by:

- the name of the object template;
- the list of the actual parameters used to initialize the instance.

The instantiation of an object template consists of:

- the internal configuration of the object and activation of its interfaces, and
- the execution of the initialization phase specified in the template.

An object enters an idle state and waits for requests arriving at its interfaces, after having completed its initialization phase. When a request is received, it is served by the object according to the described behaviour and the synchronization constraints.

MODA addresses also the paramount issue of creation and deletion of objects within transactional context following these rules:

- if the creating transaction aborts, the object will be deleted;
- if the deleting transaction aborts, the object will be re-instanced and its state restored as if it had never been deleted.
- the object must only serve requests coming on behalf of the creating transaction until commit.

This behaviour prevents transactions (other than the creating one) from accessing an object that may be deleted if the creating transaction aborts.

The transactional behaviour of creation and deletion stems from the transactional semantics of the methods implementing the initialization and termination phases of the object. The implicit constraint is that the initialization and termination methods conflict with all other transactional methods and, as such, they are protected by an exclusive-lock.

When a transaction creates an object it acquires the exclusive-lock associated with the initialization method; in this way no transaction other than the creating one can access the transactional services of the object. Then:

- if the creating transaction commits, it releases the exclusive-lock thereby allowing other transactions to access transactional operations;
- if the creating transaction aborts, the object is destroyed and the waiting transactional requests are sent back with a failure code.

Access to architectural services

The model can be enriched with additional architectural services. One of these is the Trader which realizes the same functionalities of the Trader in ANSA [9].

3. The ODIN platform

ODIN is a set of constructs to program "in-the-large" a distributed system according to MODA's concepts. ODIN consists of two parts:

- a language to declare interface, object and BB templates;
- a C++ library, which forms an API for the behavioural primitives of the platform.

The definition of an ODIN application consists of:

- a set of interface template;
- a set of object templates, which are servers and clients of services offered by interface templates;
- a set of BB templates, which group object templates in software packages.

3.1. Building Block

A BB is specified by a named template; the BB template lists the object templates whose instances can be created and executed in the BB, as indicated in Fig. 2.

Fig. 2: building block template

A BB template may have several instantiations, possibly on the same node, each identified by a different logical name. The following construct instantiates a BB identified by the logical name `L`, from the template `T`, on the node `N`, with optional parameters to be passed to the BB-manager:

```
ret = ODIN_CreateBB_T("N","L",a1,.,an,&r1,.,&rm,&BBRef);
```

The reference `BBRef` to the main interface of the BB-manager is automatically exported to the Trader along with the properties: name = L; node = N; template = T.

The following construct terminates the BB whose manager interface is referenced by `BBRef`:

```
ret = ODIN_DestroyBB(BBRef);
```

The above constructs have a synchronous blocking semantics.

3.2 Interface types

An interface type defines the services provided by an interface (Fig. 3):

- the operations: the name, the signature and the invocation mode (synchronous/ asynchronous);
- the attributes: the name, the type and the access mode (read or read & write);
- possible transactional behaviour of operations;
- the notifications: the name and the argument types.

In addition in an interface type it is possible to define structured data types. The interface type can be defined as an extension/subtype of other interface types.

Fig. 3: interface template

3.3. Objects

Objects are created by instantiating parametric object templates (Fig. 4) which specify:

- the formal parameters used to initialize instances;

- the state variables;
- the behaviour of the creation and termination phases;
- the interfaces templates offered by the object. For each interface template the object must specify the methods that implement the operations offered by the interface. In the event that an operation has "open" transactional semantics, the specification includes both the primary and the inverse methods;
- the notification handlers to deal with notifications received upon request. The declaration specifies the type of the notification and the method to be executed when the notification is received;
- the lock conflict table of object methods;
- the behaviour of the object in C language:
  - the initialization and termination phases;
  - the methods implementing the operations provided by the interfaces;
  - the notification handlers;
  - the guards associated with the methods.

An object can create other objects in the same BB to which it belongs. The construct for creation specifies the actual parameters and the placeholders for the results of the initialization phase and for the main interface reference of the created object:

```
ret=ODIN_Create_ObjTemplate(a 1,.,an,&r1,.,&rm,&ObjIfref);
```

Objects cannot destroy other objects, though an object can terminate itself by the construct:

```
ret = ODIN_Terminate();
```

After the completion of the ongoing activities, the termination phase is executed and the object space is freed.

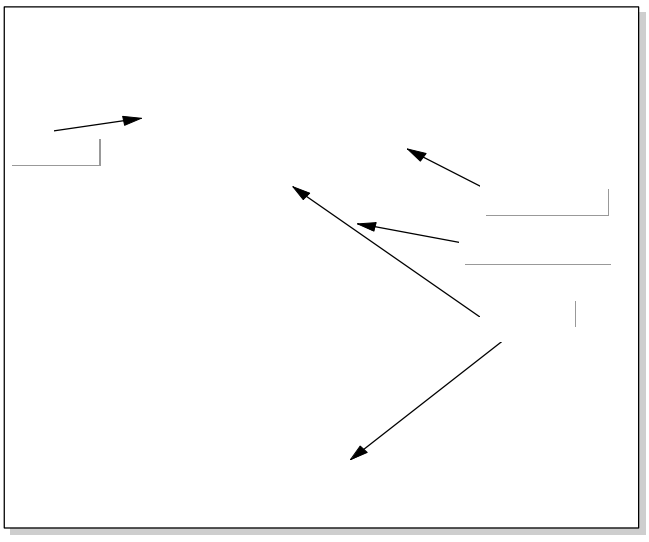

Fig. 4: object template

Invocation of operations

ODIN supports both interrogations and announcements. The following construct invokes the operation `OpName` on the interface referenced by `Ifref` of type `TypeName`:

```
ret = TypeName_OpName(Ifref,a 1,...,an,&r1,...,&rm);
```

If the operation is an interrogation it will be invoked in blocking mode. In order to invoke an interrogation in deferred-blocking mode the following two constructs are related by a variable `v` of type `ODIN_voucher`:

```
v = TypeName_OpName_Fork(Ifref,a 1,...,an);
.....
ret = TypeName_OpName_Join(v,&r 1,...,&rm);
```

The constructs above are akin to the ones adopted in the static access mode in CORBA .

The following constructs allow the client to read and write the value of the attribute `AttrName` visible at the interface referenced by `Ifref` of type `TypeName`:

```
ret = TypeName_Get_AttrName(Ifref,&curr_value);
ret = TypeName_Set_AttrName(Ifref,new_value);
```

A client can register its interest to the notification `Name` emitted by the interface referenced by `Ifref` of type `TypeName` through the construct:

```
ret = TypeName_ODIN_Request(Ifref,"Name",HandlerName);
```

where `HandlerName` is the name of the handler declared in the client template. The client can retract its interest through the construct:

```
ret = TypeName_ODIN_UndoRequest(Ifref,"NotifName");
```

An object can emit a notification `NotifName` declared in the interface `IfName` of type `TypeName`, through the construct:

```
ODIN_Emit_TypeName_NotifName(IfName,a 1,...,an);
```

the infrastructure will send the notification `NotifName(a 1,...,an)` to all subscribed objects.

Transactional primitives

An object can transactionally update its internal state through the call:

```
ret = ODIN_TransactionalCopy(StateVar,NewStateValue);
```

which must be executed on behalf of a transaction; if the transaction aborts the state of the variable will be restored to the old value (before-image recovery).

A lock of mode `Mode` (which may be either shared or exclusive) on the variable `VarName` can be requested through the call

```
ret = ODIN_LockWait(VarName,Mode);
```

if the lock is requested on behalf of a transaction it will automatically be released at its completion, otherwise it has to be explicitly released by:

```
ret = ODIN_LockRelease(VarName);
```

The code of a primary method implementig an open transactional operation may specify the values of the parameters `par1,..,parN` of the inverse method in the event that the operations has to be compensated:

```
ret = ODIN_LogCompensate(par1,...,parN);
```

4. Implementation of ODIN over DCE/C++

The current implementation of ODIN is based on DCE/C^{++}. The main rationale at the basis of DCE's choice is that OSF/DCE is a widely accepted industry product and may support transactional features by adding Transarc/Encina. To manage open nested transactions we realized an eXtended Transaction Manager (XTM) laid over Encina/TRAN; since Encina/LOCK provides a static set of lock-modes, we developed a Programmable Lock Manager (PLM) that allows the programmer of each object template to define a conflict table of lock-modes.

The usage of C^{++} enables a smooth interfacing among the distributed object model and a variety of (available or under development) OO software, including OODBMS's.

The implementation is based on the following mappings:

- BBs are translated into DCE servers;
- object templates are translated into C^{++} classes, while an ODIN object becomes a C^{++} object running into a DCE server;
- ODIN interfaces are translated into DCE interfaces;
- references to ODIN interfaces are C structures that contain information on how the interface may be reached;

The following sections briefly describe the main aspects of this implementation emphasizing the transactional structures.

4.1. Coding of interface references

An interface reference (offered by an object *obj* running on a BB *bb*) contains information on how the interface can be reached. This information consists of three fields:

1) the logical name of the BB*bb*;
2) the logical pointer to the C^{++} object corresponding to *obj*;
3) an integer value that identifies the particular interface provided by the object.

The usage of logical names allows the decoupling of interface references from physical addresses so as to ease run-time migration and reallocation of BBs. The mapping from logical names to physical addresses is accomplished at runtime by architectural objects.

4.2. The structure of a DCE server

A BB template is transformed into the executable code of a DCE server. The DCE server is the environment to create, invoke and execute the C^{++} objects corresponding to the objects in the BB. The DCE server provides the union of the interfaces of the object templates supported by the corresponding BB, and its behaviour consists of a set of C^{++} classes corresponding to the objects templates in the BB.

The main components in a DCE server are (Fig. 5):

- the DCE stub routines for the interfaces provided by the server;
- the C^{++} classes corresponding to the object templates supported by the BB;
- ODIN stub routines to perform invocations of methods of C^{++} objects.

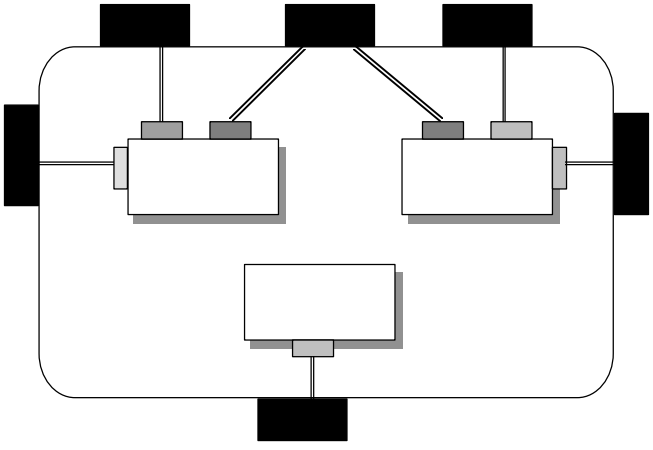

Fig. 5: internal structure of a building block

When the server receives an invocation on one of its interfaces, it must deliver the invocation to the correct C++ method. Indeed, an interface provided by a DCE server is shared by all objects that support the same interface template; this means that the server may have multiple implementations (called manager routines in DCE) of a single interface.

4.3. The eXtended Transaction Manager

Beginning a transaction amounts to creating a transaction object. A transaction object encapsulates the information pertaining the transaction service:

```
class transaction {
protected:
 trans_id tid;       // transaction identifier
 lock lock_list[ ];  // locks held by the transaction

 register_lock (lock *granted_lock) {....}
public:
 transaction () {tid = Begin_Transaction(.....); }

 commit () {....}
 abort () {....}
}
```

The protected method `register` is called by a lock to insert itself within the transaction's lock-list.

The class `open_transaction` is a subclass and allows the definition of open nested transactions by adding the method `log_compensation` to indicate the code to be invoked if the transaction needs compensation:

```
class open_transaction : transaction {
private:
 void (*comp_procedure)();

public:
 log_compensation (void (*proc)(), void par1,...., void
parN) {
  comp_procedure = proc;
  .....
  // save par1,..., parN
 }
}
```

The behaviour of the method `commit` depends upon the nature of the transaction. An open transaction is actually completed by triggering the commitment protocol among the participants; whereas a closed transaction only commits relative to its parent.

The method `abort` forces the abortion of the transaction causing all its children to be aborted or compensated:

- all active subtransactions are aborted;
- all committed closed subtransactions are rolled back;
- all committed open subtransactions are compensated by issuing the procedure registered with `log_compensation()`.

4.4. From DCE operation invocation to C++ method call

As mentioned before, a relevant task of the DCE server is to deliver an invocation to the correct C++ method. This selection (for a generic operation `Op` at an interface of type `Type`) relies on a four layer code structure:

- *server stub code*: this code is generated by compiling the DCE/IDL definition of an interface, and is in charge of unmarshalling the parameters of the invocation;
- *manager code*: this C code invokes the `bridge_Type_Op()`, forwarding the C++ logical pointer, the interface identifier, the arguments and the pointers to the results;
- *bridge code*: this C++ code performs a first switching among the possible C++ classes which have an interface `Type`; this code uses the C++ logical pointer to select the right object instance (by looking up an internal mapping table), and the corresponding class member `_ODIN_Type_Op()` function is invoked, passing the interface identifier, the arguments, and the pointers to the results;
- *object code*: the last switching concerns the selection of the interface (an object may have more than one interface of the same type). This function is in charge of concurrency and transaction management and calls the right C++ method, in accordance with the operation/method association in the object template. Below is a skeletal code of a generic `_ODIN_Type_Op()` method. For the sake of clarity, we will concentrate only on the way transaction facilities are integrated to support transaction transparency by abstracting out some details. Consider the declaration of a transactional operation `Op` with open semantics:

```
Intf_Type [ Op -> primary_method1 - inverse_method1];
```

The compilation of this interface will yield the following `_ODIN_Type_Op` method:

```
_ODIN_Type_Op(intf_discr, a 1,..., an, r1,..., rm)
{
 transaction *TID, *newTID;
 TID = get_thread_association();                    (1)
  obj_lock.wait(TID, primary_method);               (2)
  newTid = new  open_transaction(TID);              (3)
  put_thread_association(newTID);                   (4)
  switch  intf_discr  {
    case 1:  {
      status   =   primary_method1(a 1,...,an,&r1,...,&rm,
&l1,...,&lk);                                       (5)
      if (status == SUCCESS)  {
        newTid    ->    log_compensation(inverse_method1,
l1,..., lk);                                        (6)
```

```
        newTid -> commit();
      }  else
        newTid -> abort();                          (7)
      }
    case 2:  {
      .........
    }
}
```

(1) the current transaction is retrieved from the thread.

(2) each C++ object, derived from an ODIN object template, embodies a lock object to schedule the execution of the transactional methods. The call `obj_lock.wait` blocks the thread until the transaction is allowed to execute the method. Note that the lock is requested in a mode corresponding to the method's name.

(3) an open subtransaction is begun.

(4) the thread is associated with the new subtransaction.

(5) using the interface discriminator, the primary method implementing the behaviour of the operation is delivered, in this case `primary_method1`. The method will return the actual parameters $l_1, \ldots, l_k$ for the inverse method.

(6) if the call returns with `SUCCESS` (`return_with_commit` in the user method), the inverse method and its parameters are registered with the transaction object (this step is skipped if the subtransaction is not an open transaction). The subtransaction is committed.

(7) if the call returns with `FAILURE` (`return_with_abort` in the user method), the subtransaction is aborted.

The invocation of a generic operation `Op` of an interface of type `Type` at the client side is much simpler, through the following layers:

- *call layer*: it performs the mapping from the logical name of the BB address onto the real one (by asking the architectural objects to resolve the mapping), and saves the result into an internal cache; then it invokes the rpc operation `Op()`;
- *client stub code*: this DCE generated code marshals the arguments, invokes the rpc and then unmarshals the results.

If the server interface belongs to the same BB as the client does, the "call layer" performs a shortcut by calling the corresponding "bridge" routine within the same BB.

Different routines in the call layer implement different kinds of invocations (blocking synchronous, deferred/non-blocking synchronous and asynchronous):

- blocking synchronous: the calling thread performs a rpc which keeps it blocked until the reply from the server arrives;
- deferred-blocking synchronous: the calling thread spawns a new thread that performs the rpc; when the original thread wants to collect the results, "joins" the spawned one and acquires the results;
- asynchronous: the calling thread spawns a new thread that performs the rpc, and then cancels itself.

4.5. From ODIN object templates to C++ classes

A C++ class is generated for each ODIN object template. The private state variables of this C++ class can be grouped in the following sets:

- the state variables defined in the ODIN template;
- the reference variables which refer to the interfaces defined in the template;
- the internal data structure for concurrency control, registration of notifications, etc.

All the functions defined in the implementation part of the template become private methods of the C++ class. The public methods of the class are:

- the Init method invoked during the initialization phase;
- the methods invoked by the bridge code (the `_ODIN_<Type>_Op()` functions previously discussed);
- the class constructor that performs internal initializations;
- the class destructor that performs the termination phase.

5. Related works

The approach of integrating C++ and DCE has been carried out also by [13], where transactional aspects, subtyping and notifications are not covered, and [14], where a C++ encapsulation of the DCE functionalities is described.

6. Conclusions

Though the performance issue is very delicate and subtle, from our experiments we obtained performance results, from the communication point of view, comparable to the bare DCE platform. As a testbed application, it is being used as the distributed processing environment for a connection management application.

Some open issues still remains, among them the most important are:

- introduction of persistent objects by integrating an OODB into the existing platform;

- porting of the actual implementation over a CORBA platform.

Bibliography

[1] ISO/ODP, *Basic Reference Model of ODP-part 3: Prescriptive Model*, ISO/IEC JTC1/SC21 N8125 (June 1993).

[2] P.G. Bosco, G. Martini, C. Moiso, *A Distributed Object-Oriented Platform based on DCE and C++*, in Proc. ICODP '93 (Sept. 1993)

[3] N. Natarajan, F. Dupuy, N. Singer, *Computational Modelling Concepts*, TINA Consortium (Dec. 1993).

[4] P.G. Bosco, G. Giandonato and C. Moiso, *A distributed processing model for telecommunication services and operations software*, in Proc. TINA'93 (Sept. 1993).

[5] OSI/NM, *Information Technology - OSI - Management Information Services - Structure of Management Information*, ISO/IEC JTC1/SC21 N01165 (1991)

[6] E. Moss, *Nested Transactions*, MIT Press (1985)

[7] G. Weikum, *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*, in Database Transaction Models for Advanced Applications (1992)

[8] W. Weihl, *Commutativity-Based Concurrency Control for Abstract Data Types*, IEEE Trans. Comp. (Dec. 1988)

[9] APM Limited, *ANSAware 3.0 Implementation Manual*, (Jan. 1991).

[10] OMG, *The common object request broker: architecture and specification*, OMG Document Number 91.8.1 (Aug. 1991).

[11] OSF, *Introduction to OSF DCE*, Prentice Hall (1992).

[12] Transarc Corporation, *ENCINA*, Transarc Corporation (1991).

[13] M.U. Mock, DCE++: Distributing C++-Objects using OSF DCE, in Proc. International DCE Workshop (Oct. 1993).

[14] J. Dilley, *Object-Oriented Distributing Computing with C++ and OSF DCE*, in Proc. International DCE Workshop (Oct. 1993).