O_2 Notification User Manual

Release 5.0 - April 1998

Who should read this manual

The notification service available in O2 allows an o2 client to inform other clients connected to the same O2 server that an event has occured. The notification consists of a message sending for events regarding persistent objects and client connections / disconnections. The manual also describes the functional interface of the notification service with the ODMG C++ binding. User-defined events are also supported.

Other documents available are outlined, click below.

See O2 Documentation set.

TABLE OF CONTENTS

This manual is divided into the following chapters:

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1 Introduction

GENERAL OVERVIEW OF THE O_2 NOTIFICATION SERVICE

Congratulations! You are now a user of the O_2 notification service!

This document presents the notification service of O_2. It describes, in the second chapter, the functional interface of the classes of the notification services for the ODMG C++ binding and, in the third chapter, the notification services for O_2C.

This chapter introduces the O_2 notification service. It is divided into the following sections :

- System overview
- Notification overview
- Installing the O2 Notification schema

1 Introduction

1.1 System overview

The system architecture of O_2 is illustrated in Figure 1.1.

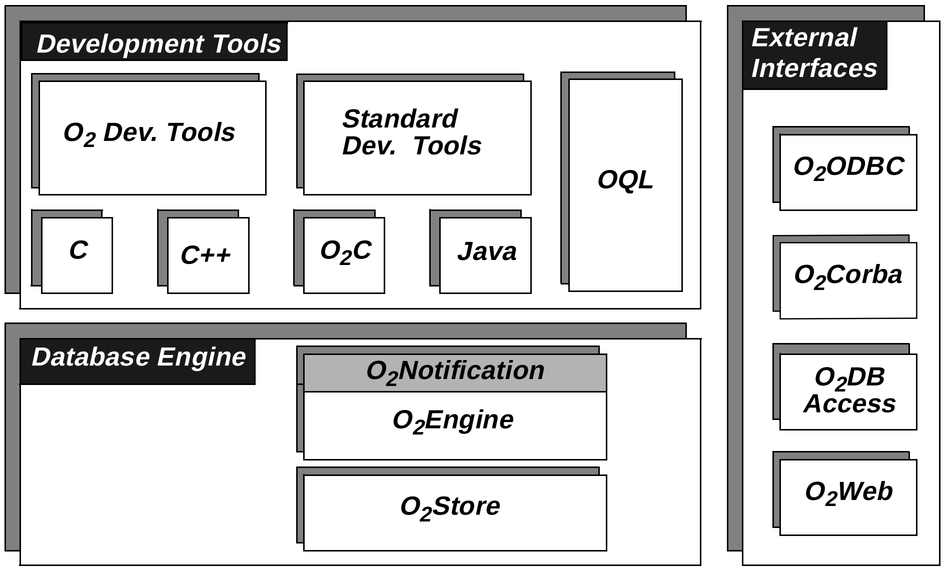

Figure 1.1: O_2 System Architecture

The O_2 system can be viewed as consisting of three components. The *Database Engine* provides all the features of a Database system and an object-oriented system. This engine is accessed with *Development Tools*, such as various programming languages, O_2 development tools and any standard development tool. Numerous *External Interfaces* are provided. All encompassing, O_2 is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

• O_2Store	The database management system provides low level facilities, through O_2Store API, to access and manage a database: disk volumes, files, records, indices and transactions.
• O_2Engine	The object database engine provides direct control of schemas, classes, objects and transactions, through O_2Engine API. It provides full text indexing and search capabilities with O_2Search and spatial indexing and retrieval capabilities with O_2Spatial. It includes a Notification manager for informing other clients connected to the same O_2 server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O2 system.

Programming Languages:

O_2 objects may be created and managed using the following programming languages, utilizing all the features available with O_2 (persistence, collection management, transaction management, OQL queries, etc.)

• C	O_2 functions can be invoked by C programs.
• C++	ODMG compliant C++ binding.
• Java	ODMG compliant Java binding.
• O_2C	A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
• OQL	ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O_2 objects and methods.

O_2 Development Tools:

• O_2Graph	Create, modify and edit any type of object graph.
• O_2Look	Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
• O_2Kit	Library of predefined classes and methods for faster development of user applications.
• O_2Tools	Complete graphical programming environment to design and develop O_2 database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

• O_2Corba	Create an O_2/ Orbix server to access an O_2 database with CORBA.
• O_2DBAccess	Connect O_2 applications to relational databases on remote hosts and invoke SQL statements.
• O_2ODBC	Connect remote ODBC client applications to O_2 databases.
• O_2Web	Create an O_2 World Wide Web server to access an O_2 database through the internet network.

1.2 Notification overview

The notification service allows an O_2 client to inform other clients connected to the same O_2 server that an event occurred. The notification consists in message sending on events regarding persistent objects.

The communication is asynchronous and the propagation of an event may be either provoked by the user or automatically launched by the system:

- *at transaction validation time* for events regarding persistent objects,
- at connection/ disconnection time for events regarding clients,
- either immediately or at transaction validation time for user-defined events, depending on the application choice.

Notifiable objects are objects whose updates or deletion are automatically notified by the system.

- An update event is provoked by any update operation.
- A deletion event is provoked by the delete_object method. The application thus must call this ODMG method if it wants the events to be notified.

Notifiable objects have to register explicitly with the notification service on a per event basis (update or deletion). Registration (Resp. disabling) of a notifiable object may be performed by any client, the object is then marked as notifiable (Resp. not notifiable).

Emitters of notifications are O_2 clients that

- either modify or delete notifiable objects,
- or explicitly send user events,
- or connect/ disconnect,

O_2 clients may register a symbolic name with the notification service. The symbolic name of the emitter of a notification, when it exists, is part of the notification information provided to the recipient(s) of the notification.

An emitter only interacts with the notification service for the notification of user-defined events.

The recipient of a notification is an O_2 client. It also has to register with the notification service on a per event basis. Registration (Resp. disabling) of a recipient is performed locally by the client, then forwarded to the server with internal information such as event type(s) and internal object identifier.

Notification processing is distributed between the O_2 clients and server as shown in Figure 1.

When receiving a notification, the server immediately propagates it to the recipients, i.e. to clients that have registered for the notified event, and put the notification in their notification queue.

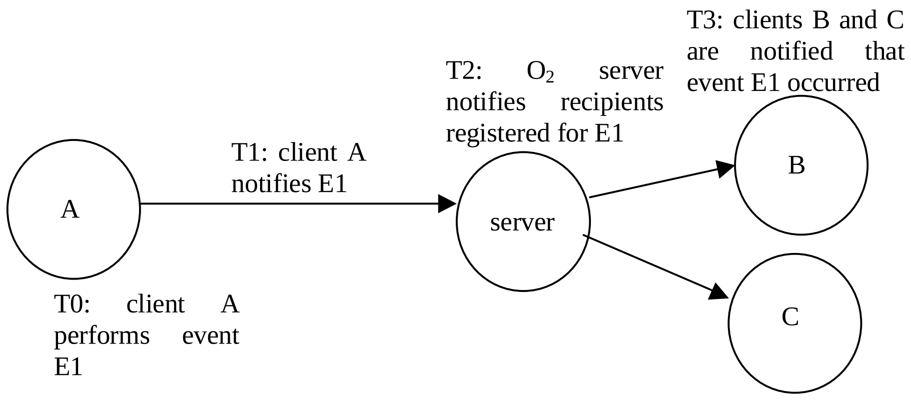
T3: clients B and C are notified that event E1 occurred
T2: O_2 server notifies recipients registered for E1
T1: client A notifies E1
A
server
B
C
T0: client A performs event E1

Figure 1: Distributed processing of a notification

The recipient may control, on a per event basis, the identity of the objects or clients whose notifications it is interested in by providing a filter at registration time. Filters associate an event type (for example, user-defined or connection), an object reference or a client name depending on the event type (client names in filters are related to connection/ disconnection events) and an optional label provided by the application. A label is a logical identifier that is returned to the recipient in the information part of a filtered notification. It allows the application to associate different processing to the notification of a given event. It is a discriminant whose validity spans transactions boundaries[1] while object references' validity does not. Filters may also apply to any object or client related to a given event. Examples of filters and associated labels are given in section 2.2.

The recipient interacts with all the objects of the notification service: the notification server for registration and for the creation of a notification queue, the notification queue for polling notified events and event objects that are stored in the notification queue. The relationship between these different objects is shown in Figure 2.

1. A transaction boundary is a `commit` or `abort` operation. A `validate` operation preserves the validity of object references.

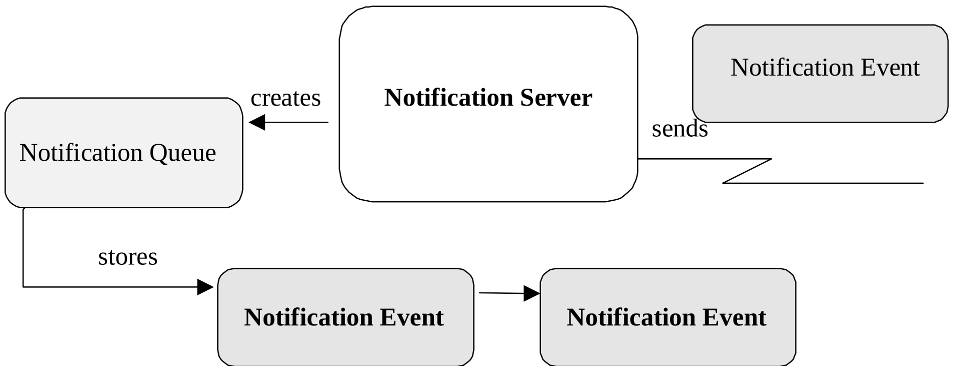

Figure 2: Objects of the Notification Service

The class hierarchy of events objects is presented in Figure 3.

The o2_Object_Event class is related to update and deletion events that are automatically notified by the system.

The class for user-defined events inherits the o2_Object_Event one. It does not mean that user-defined events must be related to an object update. It only means that a user event may be related to an object and, if so, filtering on object identity applies to user-defined events.

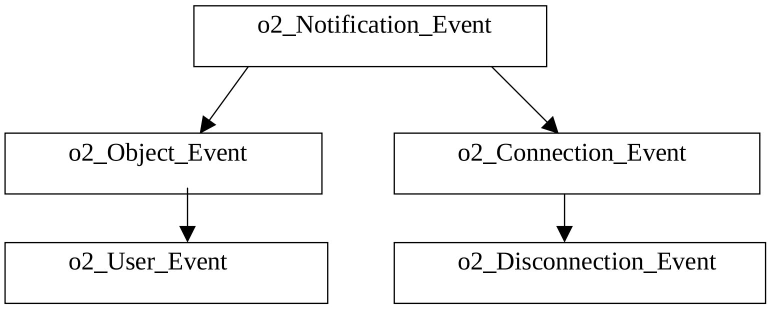

Figure 3: Event classes hierarchy

In O_2C, the o2_User_Event class and its subclasses only may be used.

1.3 Installing the O_2 Notification schema

In order to use the notification service, you have to install the $o_2$notification schema in your system. After running o2_dba_init, you use the o2dba_schema_load tool (see the *O_2 System Administration Reference Manual)* to load the schemas from the $O2HOME/o2schemas directory :

```
o2dba_schema_load -file
 $O2HOME/o2schemas/o2notification.dump -system my_system
-sources -verbose
```

o2dba_schema_load asks you for the name of the volume for the schema to install. Using the volume CatalVol for this purpose is recommended.

1 Introduction

2 C++ Interface to the Notification Service

This chapter is divided into the following sections :

2 C++ Interface to the Notification Service

2.1 Initializing your schema

When building your own schema, you have to import from the o2notification schema the event classes you will use in your application.

Example of o2dsa_shell commands to initialize your schema:

```
schema appli_s;

import schema o2notification class
o2_Notification_Event,o2_Object_Event,o2_User_Event, o2_Connection_Event,
o2_Disconnection_Event;
```

Include file

To make use of the notification package in a C++ program, you must use the following include directive :

```
  #include " o2_Notification.hxx "
```

2.2 How to use the Notification service in C++

Many scenarios are possible, according to the events you want to be notified.

If you want to ask a service to the notification system you must call a method of the O2_Notification class. But you must first create one object of this class.

(1) You want to be notified of the updates/deletions of some objects :

In order to be notifiable, an object must firstly be declared to the notification service. Any O_2 client can declare an object by calling the **`o2_Notification::register_notifiable_object`** method as many times as necessary. This property is made persistent at commit time.

<u>On the emitter side :</u> Any O_2 client can then start working as usual and update and delete notifiable objects in a transparent way.

<u>On the recipient side :</u> An O_2 client must define precisely the actual objects it is interested in.

It declares them by calling **`o2_Notification::register_notification_client`** as many times as necessary.

Now it is ready to receive updates/ deletions events. They are all stored in a queue, which is obtained by calling once **`o2_Notification::get_queue`**.

`o2_Notification_Queue::get_event` has just to be called to enable the user to wait for the next event.

When an event exists the user receives it as an instance of the **`o2_Notification_Event`** class or as one of its subclasses.

It can thus analyze the event to get more detailed information, as for instance the object which has been updated. It can then access this object, inside a transaction.

(2) You want to be notified of connections/disconnections of other clients:

On the emitter side : to make itself known, an O_2 client can give its name by calling **o2_Notification::register_client_name**. The connection/ disconnection will then be notified automatically.

On the recipient side : an O_2 client, interested in this service, must inform the system by calling **o2_Notification::register_notification_client**.

This method has many parameters including the type of checked events (which are then of the **O2_CONNECT_EVENT** or **O2_DISCONNECT_EVENT** type).

The filter, which is another parameter, may tell more precisely which O_2 client name you are interested in.

(3) You want to send and receive user built-in messages :

On the emitter side : you obviously have to build an object of the **O2_User_Event** class or one of its subclasses you can define.

This object may contain any information you want and particularly a reference to a persistent object for instance.

To send the object you just apply the **o2_User_Event::notify** method.

Such an event can be identified by an **event_user_id** which allows the recipient to recognize it.

On the recipient side : Again you use the **o2_Notification::register_notification_client** method to indicate which user events you are interested in. The parameters enable to define the nature of the event (**O2_USER_EVENT**) and through a filter the value of the user id. You can also filter on the value of a reference to a persistent object contained in the message.

As usual you get the message by calling **o2_Notification_Queue::get_event**.

You get an object of the **o2_User_Event** class in which you can find more information. You may use a virtual method call when the actual class of the object is a subclass of **o2_User_Event.** For example, the **o2_User_Event::execute method** is virtual and thus can be redefined by your own subclasses.

The rest of the chapter gives details about these functionalities.

2.3 Initialization

Initialization of emitters

A client which wants to have a name associated to the events it notifies, either explicitly or implicitly, must register its symbolic name with the notification service. This may be useful, for example, for notifying other clients at connect time, so that they may now be notified of some specific events emitted by the connecting client.

This can be done by the constructor of the notification server or by calling the following method:

```
void o2_Notification::register_client_name (char * clientName);
```

Any further registration of a name cancels the previous one. The registration of names is not notified by the system. If the application wants to notify name changes, it must define and notify the associated user event.

Several clients may register the same name with the notification server.

The name registration must be done before a connection to O_2, otherwise it is not taken into account.

Initialization of recipients

A client that wants to receive notifications may retrieve the notification queue from the notification server.

The queue that will receive the notification messages must be explicitly polled by the recipient. Its interface is described in section 5.2. It is retrieved by calling the following method of the notification server:

```
class o2_Notification_Queue;
 o2_Notification_Queue * o2_Notification::get_queue ( );
```

2.4 Registration of notifiable objects

Registration of a notifiable object is only necessary for the UPDATE and DELETION event types. It must occur in the scope of a transaction, if this transaction aborts the object will no longer be notifiable. The registration method marks an object as notifiable (it therefore acquires an exclusive lock on it during the transaction). It becomes immediately notifiable for the calling client and the notifiable property is visible at transaction validation time by other clients.

The registration method has the following interface:

```
// throws d_Error_RefInvalid, d_Error_EventInvalid,
d_Error_TransactionNotOpen

void o2_Notification::register_notifiable_object (
            o2_Notification_Event_type eventType,
            const o2_pointer &objectReference);
```

The eventType must be either **O2_UPDATE_EVENT** or **O2_DELETE_EVENT**. The objectReference[1] argument must be filled with a persistent capable reference to the persistent object that will be notifiable, such as in the following example :

```
  d_Ref<my_Class> objRef;
  ...

notification_server->register_notifiable_object(O2_UPDATE_EVENT, objRef);
```

The following method allows to test if a given object is notifiable. It returns 1 if the object is notifiable, 0 elsewhere.

```
int o2_Notification::is_notifiable_object (
         o2_Notification_Event_type eventType, const o2_pointer &objectReference);
```

1. The class o2_pointer is a superclass of all d_Ref<T>.

As for the register_notifiable_object method, the objectReference argument must be filled with a persistent capable reference to a persistent object.

Cancelling a registration

The disabling method marks an object as not notifiable and must also occur in the scope of a transaction. The notifiable property is lost immediately for the calling client, at transaction validation time for the other clients. If the current transaction aborts, the object will still be notifiable. It has the following interface:

```
// throws d_Error_RefInvalid,  d_Error_EventInvalid,
d_Error_TransactionNotOpen

int o2_Notification::forget_notifiable_object (
         o2_Notification_Event_type eventType, const o2_pointer &objectReference);
```

As for the register_notifiable_object method, the objectReference argument must be filled with a persistent capable reference to the notifiable object.

Disabling is effective for the requesting client if there are no recipients registered for that eventType and that objectReference, in which case a status equal to 1 is returned. If some recipients are registered, a status equal to 0 is returned.

Disabling is effective for all other O_2 clients at requesting client’s transaction validation time.

2.5 Registration of recipients

Registration of a recipient for events regarding notifiable objects must occur in the scope of a transaction in order to synchronize with the register_notifiable_object and forget_notifiable_object methods that are described above. Registration of a recipient for other events is effective at registration time and may occur outside the scope of a transaction. The registration operation for a notification recipient has the following interface:

```
typedef int o2_Notification_Reg_Id;

/* throws d_Error_RefInvalid, d_Error_RefNotNotifiable,
d_Error_RegistryConflicting, d_Error_TransactionNotOpen */

o2_Notification_Reg_Id
o2_Notification::register_notification_client (
            o2_Notification_Event_type eventType, o2_Notification_Filter &filter,
            o2_Notification_Event_Id userEventId= 0,
            d_Boolean updateLabelFlag= 0);
```

The register_notification_client method returns a registry identifier that will be used if the registration is explicitly cancelled.

All Event types are defined in the Events Type paragraph below.

The userEventId parameter is provided by the application, it allows to select the different user-defined events which can be received. It corresponds to the userId attribute of class o2_User_Event (see section 6). Its type is unsigned short.

A filter has to be specified by a recipient. It enables to sharpen the description of the events this registration deals with. The filter type is defined in the *Filter* paragraph below. If a further registration *for the same event*, i.e. events with identical eventTypes and userEventIds, is performed, the union of filters will be made.

1. If a previous registration has been performed with *exactly* the same parameters, the notification service increments the count associated to the previous registration and returns the same registry identifier.

2. If a previous registration has been performed with the same parameters except the label field of the filter, if the updateLabelFlag is true, the previous registration is canceled and the new one is taken into account, else the exception d_Error_RegistryConflicting is raised.

3. If a previous registration has been performed with a scope O2_ONE_OBJECT, a further registration with the scope O2_ANY_OBJECT will not be taken into account and the exception d_Error_RegistryConflicting is raised.

4. if a previous registration has been performed with a scope O2_ANY_OBJECT, no further registration will be taken into account with the scope O2_ONE_OBJECT and the exception d_Error_RegistryConflicting will be raised.

Events type

Pre-defined events types are associated to event objects whose class hierarchy is presented in the introduction. They are provided by the enumeration o2_Notification_Event_Type.

User events:

- user-defined event, **O2_USER_EVENT*.***

Object events:

- deletion of a persistent object, **O2_DELETE_EVENT**,
- update of a persistent object, **O2_UPDATE_EVENT*.***

Client events:

- connection of a client, **O2_CONNECT_EVENT,**
- disconnection of a client**, O2_DISCONNECT_EVENT*.***

Some combinations of these basic events:

- the union of deletion and update events, **O2_OBJECT_EVENTS,**
- the union of connection and disconnection events, **O2_CLIENT_EVENTS*.***

These events are subdivided in two categories:

User-defined events are not interpreted by the notification service and they are raised explicitly by the application whereas client and object events are automatically detected and raised by the database system.

User-defined events are of the O2_USER_EVENT type and have an additional application-dependent identifier which allows to discriminate them:

```
typedef unsigned short o2_Notification_Event_Id;
```

User-defined events allow the user to control the granularity of notifications. For example, in a cooperative application such as book editing, when modifying a paragraph of chapter 10, one may want to notify that chapter 10 has changed rather than to notify changes of paragraph N of this chapter. The notification service does not perform any control on the validity of user-defined events, i.e. it will not check if the object for chapter 10 has really been modified.

Event classes corresponding to these basic event types, which all inherit the o2_Notification_Event base class, are defined in section 6.

Filtering

Supplying filters allows the recipient to control the identity of the objects or clients about which it wants to receive notifications: either all notifiable objects or one given object, either all clients or one given client. Filters are set by the recipient at registration time. A filter has 3 significant fields, a scope, a label and either an object reference or a client name depending upon the event type to which it applies. The whole interface of the **o2_Notification_Filter** class is given in section 2.11.

```
class o2_Notification_Filter  {
        d_Ref_Any                               reference;
        char                                    *name;
        o2_Notification_Scope                   scope;
        o2_Notification_Label                   label;
public:
        // constructors, destructor and access methods
        // ...
};
```

A label is a logical identifier (an integer) that is returned to the recipient in the information part of a filtered notification. It allows the application to associate different processing to the notification of a given event. It is a discriminant whose validity spans transactions boundaries while object references' validity doesn't.

There are four scopes of filters for recipients:

```
enum o2_Notification_Scope {
     O2_ANY_OBJECT,
     O2_ONE_OBJECT,
     O2_ANY_CLIENT,
     O2_ONE_CLIENT
};
```

O2_ANY_OBJECT and O2_ONE_OBJECT scopes apply to events related to object updates and deletion and to user-defined events. The O2_ANY_OBJECT scope means that the filter applies to all objects for a given event. The client name is not taken into account in filters with scopes regarding objects.

O2_ANY_CLIENT and O2_ONE_CLIENT scopes apply to events related to connections and disconnections. The O2_ANY_CLIENT scope means that the filter applies to all clients that connect and/ or disconnect. The reference is not taken into account in filters with scopes regarding clients.

The default filter has the O2_ANY_OBJECT scope, a null reference, an empty name and a null label.

For a filter with the O2_ONE_OBJECT scope, the reference of the corresponding notifiable object must be provided to the notification service.

Objects references are no longer valid after a transaction has been validated with commit or aborted. In order to discriminate an event and the associated object(s) across transactions, the filter is tagged with a label, local to the recipient. For example,

- a monitoring application will register for the **O2_UPDATE_EVENT** event with the default filter to be notified of all changes of a given database,
- to be notified of growth and decreasing of the population, a census application will register for the **O2_USER_EVENT** events with the " grow " and " decrease " user event identifiers and will supply the following filter:

```
const o2_Notification_Label POPULATION = 10;
o2_Notification_Filter f (ParisCollectionRef, POPULATION);
```

Cancelling a registration

Registration may be explicitly disabled. If not explicitly disabled, a notification recipient is automatically disabled at the end of client session.

The disabling operation has the following interface:

```
// throws d_Error_RegIdInvalid

void o2_Notification::forget_notification_client (o2_Notification_Reg_Id
registryId);
```

If several registrations have been performed with the same registryId, the registration will only be canceled when the registry count falls to zero.

Some registration scenarios for notifiable objects

At least two registration approaches are possible for registration of notifiable objects.

1. A notifiable object is registered once and for all. If either the application needs to disable the notifiable property, it loops in a transaction until the forget_notifiable_object method returns the success status.

2. A notifiable object is registered by a recipient which wants to receive notifications about its updates. The registrations of the notifiable object and the recipient may be made in the scope of the same transaction. When the recipient does no longer need to receive notifications, it cancels the notifiable object registration within a transaction as shown in the following example:

```
d_Session session;
d_Database database;
d_Transaction trans;
o2_Notification notif_svr (" my_name ");
o2_Notification_Label my_obj_update =  1;
o2_Notification_Reg_Id* reg_ids;
```

```
main( ) {
       d_Ref<my_object > obj;
       int i = 0;
       // some initialization actions such as session beginning, etc...
       trans.begin(  );
       //my_coll is the list of objects about which the client wants to receive
       notifications
       d_List<d_Ref<my_object >  >my_coll(" my_collection ");
       d_Iterator<d_Ref<my_object> > my_iterator =
                                          my_coll.create_iterator( );
       reg_ids = new o2_Notification_Reg_Id[my_coll.cardinality( )];
       my_iterator.next(obj);
       do {
            // associates the same label to all the elements of my_coll
            o2_Notification_Filter my_filter(obj, my_obj_update);
            if (!notif_svr. is_notifiable_object(O2_UPDATE_EVENT, obj))
                 notif_svr.register_notifiable_object(O2_UPDATE_EVENT,obj);

            reg_ids[i++] =
                 notif_svr.register_notification_client(O2_UPDATE_EVENT,
                                                              my_filter);
       } while (my_iterator.next(obj));
       trans.validate( );
       my_iterator.reset( );
       // ...
       // before ending, cancel registrations
       trans.begin( );
       my_iterator.next(obj);
       i = 0;
       do {
            notif_svr.forget_notification_client(reg_ids[i++]);
            if (notif_svr. is_notifiable_object(O2_UPDATE_EVENT, obj))
                 notif_svr.forget_notifiable_object(O2_UPDATE_EVENT, obj);
       } while (my_iterator.next(obj));
       trans.commit( );
       // ...
}
```

2.6 Communication

Emission and propagation

The emission of update and delete events is implicitly performed at transaction validation time. The emission of user-defined events must be explicitly performed with a propagation flag: O2_IMMEDIATE_PROPAGATION or O2_VALIDATED_PROPAGATION .

The Notification Service provides generic methods for the propagation of any kind of user events and associated data. The interface of the notification method is the following:

```
// throws d_Error_RefInvalid, d_Error_EventTooBig,
d_Error_EventInvalid,

// d_Error_Memory_Exhausted

void o2_Notification::notify_user_event(
            o2_User_Event * event, o2_Propagation_Flag raise_time);
```

The caller of **o2_Notification::notify_user_event** must provide the address of a d-Ref pointer to a user-defined event as shown in the following example.

Events emitted with the O2_IMMEDIATE_PROPAGATION flag are immediately propagated to the recipient(s). Else, propagation is performed at commit time to clients of other transactions.

An example of emission of a user event in a census application is the birth of a child in Paris:

```
extern o2_Notification_Event_Id grow_id;
void child_birth(char * firstName, d_Ref<Person> father,
              d_Ref<Person> mother, d_Date date,
              d_List<d_Ref<Person> > ParisPopulation )  {
      child = new Person(firstName, father, mother, date);
       // registers the birth and inserts the new child in
       ParisPopulation
      //...
       d-Ref<o2_User_Event> evt = new o2_User_Event(grow_id,
                                  (d-Ref-Any *)(&ParisPopulation));
      notif_svr.notify_user_event(&evt, O2_VALIDATED_PROPAGATION);

}
```

Reception

The client has to explicitly poll and consume the notification from the notification queue.

The interface of the o2_Notification_Queue class is the following:

```
typedef enum {
  O2_EVENT_SUCCESS,
  O2_QUEUE_EMPTY,
  O2_WAIT_INTERRUPTED,
  O2_MEMORY_EXHAUSTED,
  O2_SESSION_NOT_OPEN
} o2_Notification_Report;

class o2_Notification_Queue {
...
public:
  // returns the number of events that are still to be consumed
  int cardinality( ) const;
  // consumes the first event of the queue, throws d_Error_MemoryExhausted
  o2_Notification_Report get_event (o2_pointer &event, int timeout = -1);
  // returns next event without consuming it, throws
d_Error_MemoryExhausted
  o2_Notification_Report peek_event (o2_pointer &event,int timeout = -1);
  /* removes the last previously peeked event from the queue, if the
  scan has not been reset and the event has not already been consumed
  by a get, in which cases the d_Error_NotificationQueueEmpty is thrown */
  void remove ( );
  // resets the scan, there is no longer a previously peeked event
  void reset  ( );
  // append an event at the end of the queue
  // throws d_Error_MemoryExhausted
  void append (const o2_pointer &event);
};
```

Default timeout makes the get_event and peek_event methods wait until an event is notified. A positive timeout tells the maximal number of seconds those methods have to wait for the notification of an event before returning.

If an event has been notified during the timeout delay, the get_event and peek_event methods return O2_EVENT_SUCCESS and the event argument contains a persistent capable reference to an event. The caller of the method must provide a typed persistent capable pointer:

```
d_Ref<o2_Notification_Event> evt ;
status = queue->get_event(evt);
```

If no event has been notified during the timeout delay, a status of the o2_Notification_Report type is returned.

Event objects may contain references to persistent objects, which must be accessed within the scope of a transaction. As usual for persistent d-Refs, they are valid until a commit/abort is performed. Indeed references embedded inside C++ objects are no longer valid after a commit or an abort. Events that are not consumed and still lay in the notification queue may still be consumed or peeked after a commit or an abort.

Destruction of events returned by the get_event and peek_event methods are under the user's responsibility. Returned events are of the base class o2_Notication_Event, depending on the event basic type, the application has to cast it into the right event class.

Event objects with the O2_DELETE_EVENT event type, contains a nil object reference since object references of deleted objects are no longer valid. Recipients of such events should have been given individual discriminant labels at registration time as shown in the following example:

```
d_Session session;
d_Database database;
d_Transaction trans;
o2_Notification notif_svr (" my_name ");
o2_Notification_Label my_obj_delete =  0;
o2_Notification_Reg_Id* reg_ids;

main( ) {
     d_Ref<my_object > obj, nilref;
     d_Ref <o2_Notification_Event>  evt;
     o2_Notification_Queue *  queue;
     int status;

     // some initialization actions such as session beginning,etc.
     // ...
     trans.begin(  );
     // my_coll is the list of objects about which the client wants
     // to receive notifications of deletions
     d_List<d_Ref<my_object >  >my_coll(" my_collection ");
     d_Iterator<d_Ref<my_object> > my_iterator =
                                  my_coll.create_iterator( );
     reg_ids = new o2_Notification_Event_Id[my_coll.cardinality()];
     my_iterator.next(obj);
     do {
        if (obj != nilref) {
             // associates their range as label to the different
             // elements of my_coll
             o2_Notification_Filter my_filter(obj, my_obj_delete);
             if (!notif_svr. is_notifiable_object(O2_DELETE_EVENT,obj))
               notif_svr.register_notifiable_object(O2_DELETE_EVENT,obj);
             reg_ids[i++] =
             notif_svr.register_notification_client(O2_DELETE_EVENT,
              my_filter);
        }
        my_obj_delete++;
} while (my_iterator.next(obj));
     trans.validate( );
     my_iterator.reset( );
     queue = notif_svr.get_queue( );
     while (1) {
        status = queue.get_event( evt);
        if ((status == 0) &&
              (evt->get_event_type( ) == O2_DELETE_EVENT)) {
              // remove the deleted element from the list
              my_coll.replace_element_at(nilref, evt->get_label( ));
        }
        // else...
        // ...
     }
}
```

2.7 Event objects

Generic classes of events o2_Object_Event, o2_Connection_Event, o2_Disconnection_Event and o2_User_Event corresponding respectively to object and connection event types propagated by the system and to user-defined event types, are provided. Event objects of these classes are returned to a recipient client by the notification queue operations. They all inherit the base class o2_Notification_Event. Their interface is given below. All user-defined event classes should inherit the o2_User_Event class as explained latter in this section.

Notification

The o2_Notification_Event class is the base class for all event objects. It contains the type of the notified event and the name of the emitter of the notification. It also contains a registry identifier that was allocated by the notification service when the recipient has registered for that event. Finally, it contains a label that was provided by the recipient when it has registered for that event.

```
typedef int o2_Notification_Reg_Id;
class o2_Notification_Event {
     friend class o2_Notification;
protected:
     // type of the notified event
     o2_Notification_Event_Type                event;
     // name of the client that emitted the notification
     char                                      *emitterName;
     // label attached to the reception of that notification
     o2_Notification_Label                     label;
     // registry identifier of the recipient
     o2_Notification_Reg_Id                     registryId;
public:
     o2_Notification_Event ( );
     ~ o2_Notification_Event ( );
     void set_event_type (o2_Notification_Event_type event);
     o2_Notification_Event_Type get_event_type ( ) const;
     o2_Notification_Label get_label ( ) const;
     char * get_name ( ) const;
     o2_Notification_Reg_Id registryId get_registry_id ( ) const;
};
```

The C++ string returned by the get_name method is freed by the destructor of o2_Notification_Event.

Object

The **o2_Object_Event** class is the class for the update and delete events, and the base class for user-defined events. It contains the reference of the object whose update is notified.

```
class o2_Object_Event : public o2_Notification_Event {
     friend class o2_Notification;
     // reference of object on which event occurred
     d_Ref_Any                                      object;
public:
     o2_Object_Event (o2_Notification_Event_Type  event,
                          d_Ref_Any objectReference);
     void set_reference (d_Ref_Any objectReference);
     d_Ref_Any get_reference( ) const;
};
```

Connection

The **o2_Connection_Event** class is the class for the connection event, and the base class for the disconnection one. It contains the host identifier and the processus identifier of the client that has connected.

```
class o2_Connection_Event : public o2_Notification_Event {
     friend class o2_Notification;
protected:
     char hostName[MAXHOSTNAMELEN];
     int pid;
public:
     o2_Connection_Event (char * host, int proc, char *

client = 0);
     o2_Connection_Event ( );
     char * get_host_name( ) const;
     int get_pid( ) const;
};
typedef enum o2_Disconnect_Status = {O2_CLNT_DISCONNECT, O2_CLNT_CRASH,
O2_CLNT_MONITORING};
```

The C++ string returned by the get_host_name method is freed by the destructor of o2_Connection_Event.

Disconnection

The **o2_Disconnection_Event** class is the class for the disconnection event. It contains the status of the disconnection. The O2_CLNT_MONITORING event means that the client has been killed by the O2 monitoring tool.

```
class o2_Disconnection_Event : public o2_Connection_Event {
     int status;
public:
     o2_Disconnection_Event (char * host, int proc, int
                                                     status, char * client = 0);
     ~o2_Disconnection_Event ( );
     int get_status ( ) const;
};
```

User

All user-defined event classes should inherit the o2_User_Event class. A user-defined event class that inherits the o2_User_Event class should also be imported in the application schema in order to be known by the system that must transfer its instances over the network.

Some important restrictions on contents of user-defined event classes are:

1. they should not contain any d_Array field, d_Bits, C++ arrays or long string fields (long means > 4 K-bytes),

2. they should not contain any transient field,

3. it is under the application responsibility to ensure the durability of the persistent objects that they reference.

Restriction on the type of the field is due to the fact that the event object's size must not be bigger than the message size. The size of the message is system dependent[2]. It is thus a bad idea to put large values of variable size in that object.

2. For example, on DEC Alpha, it is limited to 32 Kbytes.

```
typedef enum o2_Notification_Area = {O2_LOCAL_NOTIFICATION,
                                             O2_GLOBAL_NOTIFICATION};
class o2_User_Event : public o2_Object_Event {
  o2_Notification_Event_Id userId;
public:
  o2_User_Event ( );
  o2_User_Event (o2_Notification_Event_Id id, d_Ref_Any*=0);
  ~o2_User_Event ( );
  void set_user_event_id (o2_Notification_Event_Id id);
  o2_Notification_Event_Id get_user_event_id ( );
  static
  o2_Notification_Event_Id get_class_event_id (const char * class_name)
                                                         const;
  static void get_sub_classes (const char * class_name,d_List <char*>& result);
  // throws d_Error_RefInvalid, d_Error_EventTooBig, d_Error_MemoryExhausted,
  //d_Error_EventInvalid
  void notify(o2_Notification * server, o2_Propagation_Flag raise);
  /* virtual method that has to be implemented by subclasses. It is intended to
  implement the treatment associated to the received event */
  virtual void execute() {};
  // throws d_Error_RefInvalid, d_Error_EventTooBig,
  //d_Error_MemoryExhausted,d_Error_EventInvalid
  o2_Notification_Reg_Id
       register_for_time_notification (o2_Notification * server,
                                               const d_Time* first_notif_time,
                                               const d_Interval*the_period =  0,
                                               const int nbTimes = 1,
                                               const o2_Notification_Area area =
                                               O2_LOCAL_NOTIFICATION);
};
```

The get_class_event_id method returns a unique identifier associated to a given class name. The identifier associated to a subclass of o2_User_Event may be used as the identifier of events of that latter class as shown in the next example.

The get_sub_classes method fills the result argument with the list of subclasses of the given class name. This allows to register for events of a given class and all its subclasses. The programmer must take care of freeing the C++ strings of class names that it retrieves from the list as shown in the next example.

The notify method call the notify_user_event method of the notification service (see section 5.1). The server argument is the address of the notification server instantiated by the calling client. The raise flags tells if the notification has to be propagated immediately or at transaction validation time.

The register_for_time_notification method allows to notify the current event several times after a given delay. This periodic notification method may be interrupted by calling the o2_Notification::forget_notification_client method.

Time notifications are propagated immediately after first_notif_time delay and if nbTimes is greater than 1, it is propagated again nbTimes - 1 times at the_period interval. If the area argument is O2_LOCAL_NOTIFICATION, the notification is inserted at the end of the local notification queue, else it is propagated to recipients that have registered for the associated event type.

Example

The following example shows how to execute events which are of the class **My_Class_Event** or events which are sub classes of this class. **My_Class_Event** is a sub class of **o2_User_Event**.

```
d_Session session;
d_Database database;
d_Transaction trans;
o2_Notification notif_svr (" my_name ");
o2_Notification_Reg_Id* reg_ids;
int max_registries =  0;
o2_Notification_Event_Id classEvtId;

main( ) {

       d_Ref<o2_Notification_Event>  evt;
       d_Ref<o2_User_Event> user_evt ;
       o2_Notification_Queue *  queue;
      int status;
       Some initialization actions such as session beginning,etc...to receive
       notifications of events  "My_Class_Event" and its subclasses. A default
       filter is associated to the registration
o2_Notification_Filter my_filter;
        char * cl_name;
        d_List<char * >class_col;
        My_Class_Event::get_sub_classes("My_Class_Event", class_col);
        // max_registries = number of subclasses + "My_Class_Event"
        max_registries= class_col.cardinality() + 1;
        reg_ids = new o2_Notification_Reg_Id[max_registries];
        // register the event Id associated with the class
        classEvtId =My_Class_Event::get_class_event_id(" My_Class_Event ");
        reg_ids(0) = notif_svr.register_notification_client(O2_USER_EVENT,
                                                              my_filter,
                                                              classEvtId);
        for  (register i = 1; i < max_registries; i++ ) {
           /* extract the  event Id associated with the name of the class and
           register it */
           cl_name = class_col[i-1]
           classEvtId=My_Class_Event::get_class_event_id(cl_name);
           delete cl_name ;
           reg_ids[i] = notif_svr.register_notification_client(O2_USER_EVENT,
                                                              my_filter,
                                                              classEvtId);
        }
           queue = notif_svr.get_queue( );
           while (1) {
                   status = queue.get_event(evt );
                   if ((status == 0) && (evt->get_event_type( ) == O2_USER_EVENT))
                   {
                        user_evt = evt ;
                        user_evt-> execute();
                   }
                   // else...
                   // ...
                   evt.destroy();
        }
        // forget now the notifications
        for (i = 0; i < max_registries; i++)
           notif_svr.forget_notification_client(reg_ids [i]);
        // ...
}
```

2.8 Statistics of Notifications

Statistics are interesting to know the dynamic evolution of the flow of events which are emitted by notifiers or received by recipients.

Any client may know the number of events it has sent or received. The notification service also maintains general information about the total number of events which are emitted and the total number of events which are lost (emitted events without recipient). Global statistics on clients reception of notifications may be built by periodic notifications (see section 6) of user-defined statistic events.

The returned counter of events is classified into the following categories : user-defined events , deletion or update of objects, connection and disconnection.

The notification service allows to reset these counters after reading.

Statistic are gathered in objects of the following class[3] :

```
class o2_Notification_Stat {
public:
     int user_event_count;
     int deleted_object_count;
     int updated_object_count;
     int connection_count;
     int disconnection_count;
};
```

The notification service provides the following methods for statistics:

```
int o2_Notification::stat_received_event( o2_Notification_Stat *stat_event,
                                          d_Boolean reinitialize = 0);
```

returns the total number of events by category received on its queue, since last local statistic reset for received events. If reinitialize is set to TRUE, the total number of events by category is reset.

3. This class is subject to evolution if the number of events increases but will remain compatible in future versions of the product.

```
int o2_Notification::stat_emitted_event(o2_Notification_Stat *stat_event,
                                        d_Boolean reinitialize = 0);
```

returns the total number of events by category emitted by the client itself, since last local statistic reset for emitted events. If reinitialize is set to TRUE, the total number of events by category is reset.

```
int o2_Notification::global_stat_emitted_event(o2_Notification_Stat *stat_event,
                                               int reinitialize = 0);
```

returns the total number of events by category emitted on the notification service by each client, since last global statistic reset. If reinitialize is set to TRUE, the total number of events by category is reset.

```
int o2_Notification::global_stat_lost_event(o2_Notification_Stat *stat_event,
                                            int reinitialize = 0);
```

returns the total number of events by category emitted on the notification service by each client which are lost (i.e. which have no recipient), since last global statistic reset. If reinitialize is set to TRUE, the total number of events by category is reset.

2.9 Class o2_Notification

```
class o2_Notification {
public:
  o2_Notification (char * clientName = 0);
  ~o2_Notification( );

  // disable default error management i.e.exception handling
  void set_errno_mode();
  //returns an O2 error code from the message if some error has occured
  int get_errno();
  // re_enable default error management i.e.exception handling
  // void set_exception_mode();
  register_client_name (char * clientName);
  o2_Notification_Queue * get_queue( );
  //throws d_Error_RefInvalid,d_Error_EventInvalid,
  //d_Error_TransactionNotOpen
  void register_notifiable_object(o2_Notification_Event_type eventType,
                                      const o2_pointer &objectReference);
  // throws d_Error_RefInvalid, d_Error_EventInvalid
  int is_notifiable_object(o2_Notification_Event_type eventType,
                              const o2_pointer &objectReference);
  // throws d_Error_RefInvalid, d_Error_EventInvalid,
  d_Error_TransactionNotOpen
  int forget_notifiable_object (o2_Notification_Event_type eventType
                              const o2_pointer &objectReference);
  // throws d_Error_RefInvalid, d_Error_RefNotNotifiable,
  // d_ErrorConflictingRegistry, d_Error_TransactionNotOpen
  o2_Notification_Reg_Id
  register_notification_client (o2_Notification_Event_type eventType,
                              o2_Notification_Filter &filter ,
                              o2_Notification_Event_Id userEventId= 0,
                              d_Boolean updateLabelFlag =0);
  // throws d_Error_RegIdInvalid
  void forget_notification_client (o2_Notification_Reg_Id registryId);
  // throws d_Error_RefInvalid, d_Error_EventTooBig, d_Error_EventInvalid
```

```
    void notify_user_event(o2_User_Event * event,
                                o2_Propagation_Flag raise_time);
    int o2_Notification::stat_received_event (o2_Notification_Stat *stat_event,
                                           d_Boolean reinitialize = 0);
    int o2_Notification::stat_emitted_event (o2_Notification_Stat *stat_event,
                                           d_Boolean reinitialize = 0);
    int o2_Notification::global_stat_emitted_event (o2_Notification_Stat
                                                           *stat_event,
                                                      int reinitialize = 0);
    int o2_Notification::global_stat_lost_event (o2_Notification_Stat *stat_event,
                                             int reinitialize = 0);
};
```

2.10 Class o2_Notification _Queue

```
 class o2_Notification_Queue {
...
public:
     // disable default error management i.e.exception handling

     void set_errno_mode();
     // returns an O2 error code from the message if some error has occured

     int get_errno();
     // re_enable default error management i.e.exception handling
     // void set_exception_mode();
     // returns the number of events that are still to be consumed

     int cardinality( ) const;
     // consumes the first event of the queue, throws d_Error_MemoryExhausted

     o2_Notification_Report get_event (o2_pointer &event, int timeout = -1);
     //returns next event without consuming it, throws d_Error_MemoryExhausted

     o2_Notification_Report peek_event (o2_pointer &event, int timeout = -1);
     /* removes the last previously peeked event from the queue, if the scan
     has not been reset and the event has not already been consumed by a
     get, in which cases the d_Error_NotificationQueueEmpty is thrown */

     void remove ( );
     // resets the scan, there is no longer a previously peeked event

     void reset  ( );
     // append an event at the end of the queue
     // throws d_Error_MemoryExhausted

     void append (const o2_pointer &event);
};
```

2.11 Class o2_Notification_Filter

```
typedef unsigned int o2_Notification_Label;
enum o2_Notification_Scope {
                                 O2_ANY_OBJECT,
                                 O2_ONE_OBJECT,
                                 O2_ANY_CLIENT,
                                 O2_ONE_CLIENT
};
class o2_Notification_Filter  {
       d_Ref_Any                                                reference;
       char                                                     *name ;
       o2_Notification_Scope                                    scope;
       o2_Notification_Label                                    label;
public:
       // filter with scope O2_ANY_OBJECT, default filter when label is null
       o2_Notification_Filter(o2_Notification_Label label = 0);
       // filter constructor for scope O2_ANY_CLIENT, for (dis)connection
     // events only
       o2_Notification_Filter(o2_Notification_Scope scope,
                              o2_Notification_Label label = 0);
       // filter with scope O2_ONE_OBJECT
       o2_Notification_Filter(d_Ref_Any objectReference,
                              o2_Notification_Label label);
       // filter with scope O2_ONE_CLIENT, for (dis)connection
       // events only
       o2_Notification_Filter(char * clientName,
       o2_Notification_Label label);
       ~o2_Notification_Filter( );
       void set_reference(d_Ref_Any objectReference);
       d_Ref_Any get_reference( ) const;
       void set_name (char* clientName);
       char * get_name( ) const;
       void set_scope(o2_Notification_Scope scope);
       o2_Notification_Scope get_scope( ) const;
       void set_label(o2_Notification_Label label);
       o2_Notification_Label get_label( ) const;
} ;
```

2 C++ Interface to the Notification Service

3 O_2C Interface to the Notification Service

This chapter is divided into the following sections :

3 O2C Interface to the Notification

3.1 Introduction

Initializing your schema

To use the notification service in O_2C, you must import the following classes of the o2notification schema :

```
import schema o2notification
class o2_User_Event, o2_Notification,o2_Notification_Queue;
```

You also have to start o2shell (or o2tools) with the libraries **libo2notification.so** and **libo2cppruntime.so** located in **$O2HOME/lib**.

Example:

```
$O2HOME/bin/o2shell -libpath $O2HOME/lib -libs o2cppruntime:o2notification
   -system ...
```

Notification service

The O_2C interface to the notification service enables O_2 clients to exchange messages asynchronously. This kind of message may contain a reference to any persistent object.

The service is provided through an object of the **o2_Notification** class. A message is built as an object of the **o2_User_Event** class. The service stores the events which are not yet consumed by a recipient in a queue instance of the **o2_Notification_Queue** class.

We first present the User Event class which defines the container of the messages to send and to receive. Then we define the Notification class, which enables to initialize the service and to register clients to the service. The last class presents the Queue, from which the messages are received. We finally end this presentation with a commented example.

3.2 User Event

An **o2_User_Event** object is built by an emitter and posted to the notification service by calling **notify_user_event**. It is received by recipients who have previously registered for it by calling the service **register_notification_client** of the **o2_Notification** class.

The emitter includes in the object:

- a **userEventId**

- a reference to a PERSISTENT object (nil by default), which means that the event is somehow related to this object.

The notification service adds the name of the emitter ("" by default).

The recipient gets a User Event by polling the queue with the **get_event** method.

Along with the information posted by the emitter, a label and a registry id are returned (in the **o2_User_Event object**) to the recipient according to the registration made previously with the method **register_notification_client**.

Subclasses of **o2_User_Event** can be used. The virtual method **execute** can be redefined of a subclass to do actions after receiving a user event.

A user event may contain references to persistent objects. The recipient of such an event must access referenced object inside a transaction.

Class o2_User_Event

```
method public init(userEventId: integer, reference: Object)
```

This method initializes an **o2_User_Event**.

userEventId: an integer to identify this event.

reference: if not nil, it must be a persistent object.

```
method public set_reference(reference: Object)
```

This method changes the reference of the object the event refers to.

reference: if not nil, it must be a persistent object.

```
method public get_reference: Object
```

This method returns the reference of the object the event refers to. Make sure that you access the referenced object inside a transaction.

```
method public set_user_event_id(userEventId: integer)
```

This method changes the identifier of the event.

userEventId: an integer to identify this event.

```
method public get_user_event_id: integer
```

This method returns the identifier of the event.

```
method public get_label: integer
```

This method returns the label of the filter matching this event (see the **register_notification_client** method). Meaningful only for the recipient.

```
method public get_name: string
```

This method returns the name of the emitter of the event. Meaningful only for the recipient.

```
method public set_name(emitter_name: string)
```

This method sets the name of the emitter of the event. By default, this is the name given when the **o2_Notification** object has been created.

emitter_name : the name of the O_2 client

```
method public get_registry_id: integer
```

This method returns the registration identifier matching this event (see the **register_notification_client** method). Meaningful only for the recipient.

```
method public execute: integer
```

This method is virtual and can be redefined on subclasses. This method returns 0.

3.3 Notification service

The **o2_Notification** class enables to initialize the notification service, to get the message queue, to register events the client is interested in, and to notify an event.

Class o2_Notification

```
method public init(clientName: string)
```

This method initializes the Notification service. The notication service cannot be used until the **o2_Notification** object is created.

clientName: is the logical name which identifies the emitter of messages.

```
method public register_client_name(clientName: string)
```

This method changes the clientName.

clientName: is the logical name which identifies the emitter of messages.

```
method public get_queue: o2_Notification_Queue
```

This method returns the message queue object associated to the service. The received messages are put in this queue from which the recipient extracts them.

```
  method public notify_user_event(event: o2_User_Event,
                                  immediate: boolean)
```

This method sends a message to O_2 clients which are interested in this message. "interested" means that the client has registered for this User Event by calling the **register_notification_client** method (see below). The message is sent immediately if "immediate" is true or else at commit (or validate) time only.

event : the event to send

immediate : true or false. If false, it will be sent at commit time only.

```
method public register_notification_client(
      filter: tuple(reference: Object, label: integer),
      userEventId: integer): integer
```

This method must be called by an O_2 client who wants to receive some messages sent by another O_2 client which calls the **notify_user_event** method. The messages this recipient is interested in are characterized by a **userEventId**. Moreover, a "filter" can be given to be more selective. If "reference" is not nil, only the messages related to this object are selected.

Because an O_2 Client can register more than one time (for different user events) a "label" can be given at registration time to identify this event. This label will be part of the received message. It helps the application to sort the different received messages.

The method returns an integer which is a "registry Id". This number can be used in the **forget_notification_client** method to cancel this particular registration.

To define the user events, in which it is interested, the O_2 client must call the **register_notification_client** method as many times as necessary.

filter : a tuple value

reference : a particular object the client is interested in. If nil it means any object.

```
method public forget_notification_client(registryId: integer)
```

This method cancels the registration whose number is "registryId". This number has been returned by a previous call of **register_notification_client.**

registryId : the id of the registry to cancel.

```
method public close
```

This method must be called when the notification service is no longer used.

3.4 Notification Queue

The **o2_Notification_Queue** class allows to get received events.

Class o2_Notification_Queue

```
  method public cardinality: integer
```

This method returns the number of events that are still to be consumed.

```
method public get_event(timeout: integer):
                   tuple(report: string, event:o2_User_Event)
```

This method consumes the first event of the queue.

timeout:	- if equal -1, the method waits until a message exists in the queue. - if > 0, the method waits maximum "timeout" seconds until a message exists in the queue.
report:	- equal "success", if a message has been extracted from the queue. In this case, **event** contains this message. - equal "queue_empty" when there exists no message. - equal "error" in all other cases (memory exhausted or communication broken)

```
  method public append(event:o2_User_Event)
```

This method appends an event at the end of the queue. This can be useful when an O_2 client sends a message to itself.

3.5 Statistics

It is possible to get statistics about the notification service through two C functions.

```
o2_notification_global_stat_emitted(o2_Notification_stat*, int reset);
o2_notification_global_stat_lost(o2_Notification_stat*, int reset);
```

The first function returns in the struct the number of emitted user event.

The second returns the number of messages which were emitted, but which the recipient did not receive.

If reset = 1, then statistics are reset to initial count.

The o2_Notification_Stat type is :

```
typedef struct {
      int user_event_count, int void1; int void2; int void3; int void4}
      o2_Notification_Stat
```

3.6 Commented example

In this example we have 2 applications which run in parallel as 2 different O_2 clients.

- "creator" create objects and notifies the creations of objects
- "insertor" receives the new created objects and insert them in a persistent collection. This insertion is notified back to the creator.
- The creator receives back the objects and checks in the end that the set of emitted objects is equal to the set of received objects.

Comments are written in italics inside the code.

```
schema test;

To use the Notification service: import the classes

import schema o2notification
     class o2_User_Event, o2_Notification,o2_Notification_Queue;

The user classes are as follows

class Person public type tuple(
     name: string)
method
     public init(name: string)
end;

class o2_set_Person public type
     unique set(Person)
method
     public insert(p: Person)
end;

constant name people: o2_set_Person;
```

We want to notify two events: creation of a person and insertion into people. We use **o2_User_Event** for this creation and define a subclass for the insertion: **Insertion_Event**.

We also have an event of the **End_Communication** subclass, whose unique role is to indicate the end of the creation stream.

```
method private notify_creation in class Person;

method private notify_insertion(p: Person) in class o2_set_Person;

method body notify_creation in class Person{
     o2 extern o2_Notification notification_service;
#define CREATION 1
     o2 o2_User_Event e = new o2_User_Event(CREATION, self);
     notification_service->notify_user_event(e, false);
};
```

The class Insertion_Event adds information to user event : the inserted object

```
class Insertion_Event inherit o2_User_Event public type tuple(
     inserted_elem: Person )
method
     public init(p: Person)
end;

method body init(p: Person) in class Insertion_Event{
#define INSERTION 2
     self->inserted_elem = p;
     self->o2_User_Event@init(INSERTION, people);
};

method body notify_insertion (p: Person) in class o2_set_Person{
     o2 extern o2_Notification notification_service;

     o2 Insertion_Event e = new Insertion_Event(p);
     notification_service->notify_user_event(e, false);
};
```

To indicate that the emission is over we just redefine execute *on a subclass of* UserEvent*. By convention a returned* 1 *would mean that it is over.*

```
class End_Communication inherit o2_User_Event
method public execute:integer
end;

method body execute:integer in class End_Communication{
     return 1; Meaning the communication is over
};

function notify_end;

function body notify_end{
#define CREATION 1
     o2 extern o2_Notification notification_service;
     o2 End_Communication e = new End_Communication;
     e-> o2_User_Event@init(CREATION, nil);
     notification_service->notify_user_event(e, true);
};

Note that the notification END is immediate, whereas the creation and
insertion are notified at validation time only.
```

We encapsulate the notifications in the creation and insert methods.

```
method body init in class Person{
     self->name = name;
     self->notify_creation;
};

method body insert in class o2_set_Person{
     *self += unique set(p);
     self->notify_insertion(p);
};

commit;
```

Two applications communicating through the notification service

You will find below the creator application. We define as application variables the notification service and its queue. In fact, these two entities have a session life and must be created only once in the **init** program. The **restart** program enables to export the notification object as an external variable which can then be referred to from inside methods like **notify_creation** of class Person for instance.

```
application creator
variable
     sent_set: set(Person),
     received_set: set(Person),

     notif_service: o2_Notification,
     queue: o2_Notification_Queue,
     registryId: integer

program
     init(trace: boolean),
     restart(why: integer, notification_service: o2_Notification),
     exit,

     wait_acknowledge,

public create
end;
```

In the **init** program, the notification service is created, and one event is registered the INSERTION whatever the nature of the inserted. The creator wants to receive INSERTION events which are emitted by the insertor.

```
set application creator;

transaction body init{
#define INSERTION 2

     *people = unique set();
     stat = trace;
```

Initialize the notification service :

```
notif_service = new  o2_Notification("Creator");
queue = notif_service->get_queue;
```

I am interested in INSERTION events :

```
     registryId = notif_service->register_notification_client(
       tuple(reference: (o2 Object) nil, label: (o2 integer) INSERTION)
       , INSERTION);
     commit;
};

program body restart(why: integer, notification_service:
o2_Notification){
#include "o2_event.h"
     switch(why){
     case O2_ERROR:
       printf("Error\n");
       exit();
     case O2_COMMIT:
       notification_service = notif_service;
       break;
     case O2_ABORT:
       printf("Abort %d\n");
       exit();
     case O2_DEADLOCK:
       printf("Deadlock %d\n");
       exit();
     }
};
```

The **exit** program cancels the registration done by the creator and closes the service before logout.

```
transaction body exit{
     if(received_set != sent_set){
       display(tuple(E: "error", Receive: received_set, sent:sent_set));
     }
     notif_service->forget_notification_client( registryId);
     notif_service->close;
     display("Bye!");
};
```

The **create** program builds objects in several transactions. After two creations a transaction is committed and the creation is thus notified. Caution : a reference must refer to a persistent object. After a transaction the creator waits for acknowledgement. It finally notifies that the process is over.

```
program body create{
     o2 Person p;
     char NAME[100];
     o2 string name;
     int i, j;
     o2 string go;

     input(go); Just to wait

     for(i = 1; i <= 5; i++){
       transaction;
         for(j = 1; j <= 2; j++){
              sprintf(NAME, "n_%d_%d", i, j);
              strcpy(name, NAME);
              p = new Person(name);
              sent_set += set(p); Make it persistent
              }
       validate;
       wait_acknowledge();
     }
     notify_end();
};
```

In this program the creator waits until an INSERTION event occurs. It just checks (in this example) that everything is consistent.

```
program body wait_acknowledge{
#define INSERTION 2
     o2 tuple(report: string, event: o2_User_Event) result;
     int i;

     for(i = 1; i <=2; i++){
       result = queue->get_event(60);
       if(result.report == "success"){
          o2 Insertion_Event event;
            o2 Person p; o2 o2_set_Person s;
            if(result.event->get_label() != INSERTION ||
               result.event->get_user_event_id() != INSERTION){
                   display(tuple(E: "error",
                           label: result.event->get_label()
                           id: result.event->get_user_event_id()));
                   exit();
            }
            transaction;
            s = (o2 o2_set_Person) result.event->get_reference();
            if(s != people){
                display("Error reference");
                exit();
            }

            event = (o2 Insertion_Event) result.event;
            p = event->inserted_elem;

            printf("Received from %s\n", result.event->get_name());
            if(p != nil){
                printf(" .... Person %s\n", p->name);
            }
            received_set += set(p);
            validate;

          }else{
              display(result.report);
              exit();
          }
        }
    };

    commit;
```

Application insertor

In this application, you will find the **init, restart, exit** programs, which are similar to those of the **creator** application.

```
application insertor
variable
     notif_service: o2_Notification,
     queue: o2_Notification_Queue,
     registryId: integer

program
     init,
     restart(why: integer, notification_service: o2_Notification),
     exit,

public insert
end;

set application insertor;

transaction body init{
#define CREATION 1
     notif_service = new  o2_Notification("Insertor");
     queue = notif_service->get_queue;
     //I am interested in creation  events :
     registryId = notif_service->register_notification_client(
        tuple(reference: (o2 Object) nil, label: (o2 integer) CREATION)
        , CREATION);
     commit;
};
program body restart(why: integer, notification_service:
o2_Notification){
#include "o2_event.h"
switch(why){
     case O2_ERROR:
        printf("Error\n");
```

```
          exit();
     case O2_COMMIT:
          notification_service = notif_service;
          break;
     case O2_ABORT:
          printf("Abort %d\n");
          exit();
     case O2_DEADLOCK:
          printf("Deadlock %d\n");
          exit();
     }
};
transaction body exit{
     notif_service->forget_notification_client( registryId);
     notif_service->close;
     display("Bye!");
};
```

In the **insert** program we wait until CREATION events occur. We check consistency (in this example) and when inserting the received object into the collection "people", we notify back to the creator that the object has now been inserted in this collection.

```
program body insert{
#define CREATION 1
     o2 tuple(report: string, event: o2_User_Event) result;
     o2 string go;

     input(go);  Just to wait

     do{
       result = queue->get_event(60);
       if(result.report == "success"){
         o2 Insertion_Event event;
         o2 Person p;
```

```
          if(result.event->execute()){ exit(); } End Message

            if(result.event->get_label() != CREATION ||
               result.event->get_user_event_id() != CREATION){
                   display(tuple(E: "error",
                                 label: result.event->get_label(),
                                 id: result.event->get_user_event_id()));
                   exit();
            }

            transaction;
            p = (o2 Person) result.event->get_reference();
            printf("Received from emitter %s\n", result.event->get_name());
            if(p != nil){
                printf(" .... Person %s\n",  p->name);
            }

            people->insert(p);
            validate;
       }else{
            display(result.report);
            exit();
       }

       }while(1);
   };

   confirm classes;

   base test_base;

   quit;
```

Running the application

The **creator** application is launched.

```
$O2HOME/bin/o2shell -v -system ... -server ...
-libpath $O2HOME/lib
-libs o2cppruntime:o2notification

set base test_base;

run program create in application creator(true);
```

The **insertor** application is launched meanwhile.

```
$O2HOME/bin/o2shell -v -system ... -server ...
-libpath $O2HOME/lib
-libs o2cppruntime:o2notification */

set base test_base;

run program insert in application insertor;
```

4 Appendix

This chapter gives the list of error messages for the d_Class with their explanations.

EventInvalid	current operation does not apply to the given eventtype

Some operations of the notification service only apply to specific event types. For example, the **register_notifiable_object** only applies to events of the **o2_Object_Event** class and the **notify_user_event** method only applies to events of the **o2_User_Event** class and subclasses.

EventTooBig	size of user event too is too big

The event provided to the **notify_user_event** operation does not fit in a message.

NotificationNotAvailable	session not opened or notification not available on the current platform.

You called a function that requires your session to be open. Notification either lacks a valid license or requires a multithread platform.

NotificationQueueEmpty	empty notification queue.

The remove method of **o2_Notification_Queue** is called on an empty queue.

RefNotNotifiable	object is either temporary or not notifiable.

The **register_notification_client** operation is applied to an event of the **o2_Object_Event** class, but provides a filter with the reference of an object that has not been registered as a notifiable object or that has become temporary after its registration.

RegIdInvalid	invalid registry identifier

The registry identifier provided to the **forget_notifiable_client** operation is not a valid identifier. The caller of the append method of **o2_Notification_Queue** has not registered itself as a client for the event it wants to append.

RegistryConflicting	conflicting client registry

Filter provided for the current registry conflicts with the filter of a previous registry performed by the same client.

INDEX