Time Service Specification

Version 1.0
New Edition: May 2000

Contents

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and X/Open Co Ltd, this document is a candidate for endorsement by X/Open, initially as a Preliminary Specification and later as a full CAE Specification. The collaboration between OMG and X/Open Co Ltd ensures joint review and cohesive support for emerging object-based specifications.

X/Open Preliminary Specifications undergo close scrutiny through a review process at X/Open before publication and are inherently stable specifications. Upgrade to full CAE Specification, after a reasonable interval, takes place following further review by X/Open. This further review considers the implementation experience of members and the full implications of conformance and branding.

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

X/Open

X/Open is an independent, worldwide, open systems organization supported by most of the world's largest information system suppliers, user organizations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

Intended Audience

The specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for object services; the benefits of compliance are outlined in the following section, "Need for Object Services."

Need for Object Services

To understand how Object Services benefit all computer vendors and users, it is helpful to understand their context within OMG's vision of object management. The key to understanding the structure of the architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in *CORBA: Common Object Request Broker Architecture and Specification.*
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains.
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility.

The Object Request Broker, then, is the core of the Reference Model. Nevertheless, an Object Request Broker alone cannot enable interoperability at the application semantic level. An ORB is like a telephone exchange: it provides the basic mechanism for making and receiving calls but does not ensure meaningful communication between subscribers. Meaningful, productive communication depends on additional interfaces, protocols, and policies that are agreed upon outside the telephone system, such as telephones, modems and directory services. This is equivalent to the role of Object Services.

What Is an Object Service Specification?

A specification of an Object Service usually consists of a set of interfaces and a description of the service's behavior. The syntax used to specify the interfaces is the OMG Interface Definition Language (OMG IDL). The semantics that specify a services's behavior are, in general, expressed in terms of the OMG Object Model. The OMG Object Model is based on objects, operations, types, and subtyping. It provides a standard, commonly understood set of terms with which to describe a service's behavior.

(For detailed information about the OMG Reference Model and the OMG Object Model, refer to the *Object Management Architecture Guide).*

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- CORBA Platform Technologies
  - *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
  - *CORBA Languages*, a collection of language mapping specifications. See the individual language mapping specifications.
  - *CORBA Services*, a collection of specifications for OMG's Object Services. See the individual service specifications.
  - *CORBA Facilities*, a collection of specifications for OMG's Common Facilities. See the individual facility specifications.
- CORBA Domain Technologies
  - *CORBA Manufacturing*, a collection of specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
  - *CORBA Med*, a collection of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.

- *CORBA Finance*, a collection of specifications that target a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*, a collection of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Service Design Principles

Build on CORBA Concepts

The design of each Object Service uses and builds on CORBA concepts:

- Separation of interface and implementation
- Object references are typed by interfaces
- Clients depend on interfaces, not implementations
- Use of multiple inheritance of interfaces
- Use of subtyping to extend, evolve and specialize functionality

Other related principles that the designs adhere to include:

- Assume good ORB and Object Services implementations. Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to “fine-grain” objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.
- Do not build non-type properties into interfaces

A discussion and rationale for the design of object services was included in the HP-SunSoft response to the OMG Object Services RFI (OMG TC Document 92.2.10).

Basic, Flexible Services

The services are designed to do one thing well and are only as complicated as they need to be. Individual services are by themselves relatively simple yet they can, by virtue of their structuring as objects, be combined together in interesting and powerful ways.

For example, the event and life cycle services, plus a future relationship service, may play together to support graphs of objects. Object graphs commonly occur in the real world and must be supported in many applications. A functionally-rich Folder compound object, for example, may be constructed using the life cycle, naming, events, and future relationship services as “building blocks.”

Generic Services

Services are designed to be generic in that they do not depend on the type of the client object nor, in general, on the type of data passed in requests. For example, the event channel interfaces accept event data of any type. Clients of the service can dynamically determine the actual data type and handle it appropriately.

Allow Local and Remote Implementations

In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility as regards the location of participating objects. So, for example, if the performance requirements of a particular application dictate it, objects can be implemented to work with a Library Object Adapter that enables their execution in the same process as the client.

Quality of Service is an Implementation Characteristic

Service interfaces are designed to allow a wide range of implementation approaches depending on the quality of service required in a particular environment. For example, in the Event Service, an event channel can be implemented to provide fast but unreliable delivery of events or slower but guaranteed delivery. However, the interfaces to the event channel are the same for all implementations and all clients. Because rules are not wired into a complex type hierarchy, developers can select particular implementations as building blocks and easily combine them with other components.

Objects Often Conspire in a Service

Services are typically decomposed into several distinct interfaces that provide different views for different kinds of clients of the service. For example, the Event Service is composed of *PushConsumer, PullSupplier* and *EventChannel* interfaces. This simplifies the way in which a particular client uses a service.

A particular service implementation can support the constituent interfaces as a single CORBA object or as a collection of distinct objects. This allows considerable implementation flexibility. A client of a service may use a different object reference to communicate with each distinct service function. Conceptually, these “internal” objects *conspire* to provide the complete service.

As an example, in the Event Service an event channel can provide both *PushConsumer* and *EventChannel* interfaces for use by different kinds of client. A particular client sends a request not to a single “event channel” object but to an object that implements either the *PushConsumer* and *EventChannel* interface. Hidden to all the clients, these objects interact to support the service.

The service designs also use distinct objects that implement specific service interfaces as the means to distinguish and coordinate different clients without relying on the existence of an object equality test or some special way of identifying clients. Using the event service again as an example, when an event consumer is connected with an event channel, a new object is created that supports the *PullSupplier* interface. An object reference to this object is returned to the event consumer which can then request events by invoking the appropriate operation on the new “supplier” object. Because each client uses a different object reference to interact with the event channel, the event channel can keep track of and manage multiple simultaneous clients. An event channel as a collection of objects conspiring to manage multiple simultaneous consumer clients.

Use of Callback Interfaces

Services often employ callback interfaces. Callback interfaces are interfaces that a client object is required to support to enable a service to *call back* to it to invoke some operation. The callback may be, for example, to pass back data asynchronously to a client.

Callback interfaces have two major benefits:

- They clearly define how a client object participates in a service.
- They allow the use of the standard interface definition (OMG IDL) and operation invocation (object reference) mechanisms.

Assume No Global Identifier Spaces

Several services employ identifiers to label and distinguish various elements. The service designs do not assume or rely on any global identifier service or global id spaces in order to function. The scope of identifiers is always limited to some context. For example, in the naming service, the scope of names is the particular naming context object.

In the case where a service generates ids, clients can assume that an id is unique within its scope but should not make any other assumption.

Finding a Service is Orthogonal to Using It

Finding a service is at a higher level and orthogonal to using a service. These services do not dictate a particular approach. They do not, for example, mandate that all services must be found via the naming service. Because services are structured as objects there does not need to be a special way of finding objects associated with services - general purpose finding services can be used. Solutions are anticipated to be application and policy specific.

Interface Style Consistency

Use of Exceptions and Return Codes

Throughout the services, exceptions are used exclusively for handling exceptional conditions such as error returns. Normal return codes are passed back via output parameters. An example of this is the use of a DONE return code to indicate iteration completion.

Explicit Versus Implicit Operations

Operations are always explicit rather than implied (e.g., by a flag passed as a parameter value to some "umbrella" operation). In other words, there is always a distinct operation corresponding to each distinct function of a service.

Use of Interface Inheritance

Interface inheritance (subtyping) is used whenever one can imagine that client code should depend on less functionality than the full interface. Services are often partitioned into several unrelated interfaces when it is possible to partition the clients into different roles. For example, an administrative interface is often unrelated and distinct in the type system from the interface used by "normal" clients.

Acknowledgments

The following companies submitted and/or supported parts of the *Time Service* specification:

- AT&T Global Information Solutions Co.
- Digital Equipment Corporation
- Expersoft Corporation
- Groupe Bull
- Hewlett-Packard Company
- International Business Machines Corporation (in collaboration with Taligent, Inc.)
- International Computers Limited
- Novell, Inc.
- Siemens Nixdorf Informationssysteme AG

- SunSoft, Inc.
- Tandem Computers, Inc. (in collaboration with Odyssey Research Assoc., Inc.)
- Tivoli Systems, Inc.

Service Description 1

Contents

This chapter contains the following topics.

1.1 Overview

1.1.1 Time Service Requirements

The requirements explicitly stated in the RFP ask for a service that enables the user to obtain current time together with an error estimate associated with it.

Additionally, the RFP suggests that the service also provide the following facilities:

- Ascertain the order in which "events" occurred.
- Generate time-based events based on timers and alarms.
- Compute the interval between two events.

Although the RFP mentions specification of a synchronization mechanism, the submitters deemed it inappropriate to specify a single mechanism such as discussed in Section 1.1.3, "Source of Time," on page 1-2.

1.1.2 Representation of Time

Time is represented many ways in programs. For example the *X/Open DCE Time Service* [1] defines three binary representations of absolute time, while the UNIX SVID defines a different representation of time. Other systems use time represented in myriads of different ways. It is not a goal of the service defined in this submission to deal with all these different representations of time or to propose a new unifying representation of time.

To satisfy the set of requirements that are addressed, we have chosen to use only the Universal Time Coordinated (UTC) representation from the *X/Open DCE Time Service*. Global clock synchronization time sources, such as the UTC signals broadcast by the WWV radio station of the National Bureau of Standards, deliver time, which is relatively easy to handle in this representation. UTC time is defined as follows.

`Time units`	`100 nanoseconds (10`$^{-7}$` seconds)`
`Base time`	`15 October 1582 00:00:00.`
`Approximate range`	`AD 30,000`

UTC time in this service specification always refers to time in Greenwich Time Zone. The corresponding binary representations of relative time is the same one as for absolute time, and hence with similar characteristics:

`Time units`	`100 nanoseconds    (10`$^{-7}$` seconds)`
`Approximate range`	`+/- 30,000 years`

In order to ease implementation on existing systems, migration from them and interoperation with them, care has been taken to ensure that the representation of time used interoperates with *X/Open DCE Time Service* [1], and that the operation for getting current time is easy to implement on *X/Open DCE Time Service*, *NTP* [2] (and for that matter any other reasonable distributed time synchronization algorithm that one might come up with. For example, ones presented in [3]) with appropriate values for inaccuracies.

1.1.3 Source of Time

The services defined in this chapter depend on the availability of an underlying *Time Service* that obtains and synchronizes time as required to provide a reasonable approximation of the current time to these services. The following assumptions are made about the underlying time synchronization service:

- The *Time Service* is able to return current time with an associated error parameter.
- Within reasonable interpretation of the terms, the *Time Service* is available and reliable. The time provided by the underlying service can be trusted to be within the inaccuracy window provided by the underlying system.
- The time returned by the *Time Service* is from a monotonically increasing series.

Additionally, if the underlying *Time Service* meets the criteria to be followed for secure time presented in Appendix A, Implementation Guidelines, then the *Time Service* object is able to provide trusted time.

No additional assumptions are made about how the underlying service obtains the time that it delivers to this service. For example it could utilize a range of techniques whether it be using a Cesium clock attached to each node or some hardware/software time synchronization method. It is assumed that the underlying service may fail occasionally. This is accounted for by providing an appropriate exception as part of the interface. The availability and accuracy of trusted time depends on what is provided by the underlying *Time Service*.

1.2 General Object Model

The general architectural pattern used is that a service object manages objects of a specific category as shown in Figure 1-1.

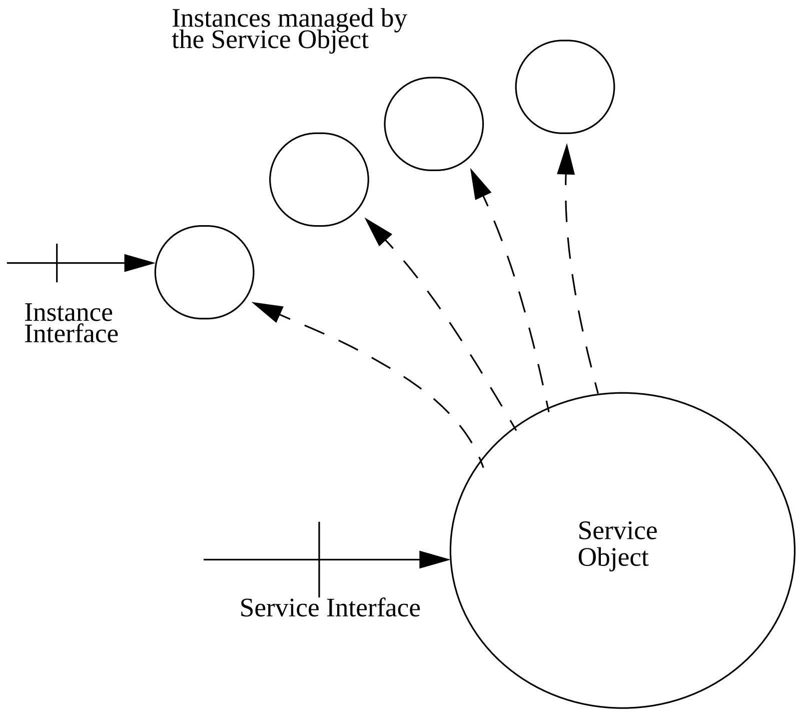

Figure 1-1 General Object Model for Service

The service interface provides operations for creating the objects that the service manages and, if appropriate, also provides operations for getting rid of them.

The *Time Service* object consists of two services, and hence defines two service interfaces:

- *Time Service* manages *Universal Time Objects* (UTOs) and *Time Interval Objects* (TIOs), and is represented by the **TimeService** interface.
- *Timer Event Service* manages *Timer Event Handler* objects, and is represented by the **TimerEventService** interface.

The underlying facility that delivers time is associated with the **UniversalTime** and **SecureUniversalTime** operation of the **TimeService** interface as described in Section 1.3, "Basic Time Service," on page 1-4.

1.2.1 Conformance Points

There are two conformance points for this service.

- *Basic Time Service*. This service consists of all data types and interfaces defined in the TimeBase and CosTime modules in Section 1.3, "Basic Time Service," on page 1-4. It provides operations for getting time and manipulating time. A complete implementation of the **TimeBase** and the **CosTime** modules is necessary and sufficient to conform to the Time Service object standard. An implementation of the **CosTime** module in which the **universal_time** operation always raises the TimeUnavailable exception is not acceptable for satisfying this conformance point.
- *Timer Event Service*. This service consists of all data types and interfaces defined in the **CosTimerEvent** module in Section 14.3, Timer Event Service. It provides operations for managing time-triggered event handlers and the events that they handle. A complete implementation of this module is necessary to conform to the optional Timer Event Service component of the Time Service object. Since the **CosTimerEvent** module depends on the **CosTime** module, it is not possible to conform just to the Timer Event Service without conforming to Basic Time Service. To claim conformance to Timer Event Service, both Timer Event Service and Time Service must be provided.

1.3 Basic Time Service

All data structures pertaining to the basic Time Service, Universal Time Object, and Time Interval Object are defined in the **TimeBase** module so that other services can make use of these data structures without requiring the interface definitions. The interface definitions and associated enums and exceptions are encapsulated in the **CosTime** module.

1.3.1 Object Model

The object model of this service is depicted in Figure 1-2. The Time Service object manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs). It does so by providing methods for creating UTOs and TIOs. Each UTO represents a time, and each TIO represents a time interval, and reference to each can be freely passed around, subject to the caveats discussed in Appendix A, Implementation Guidelines.

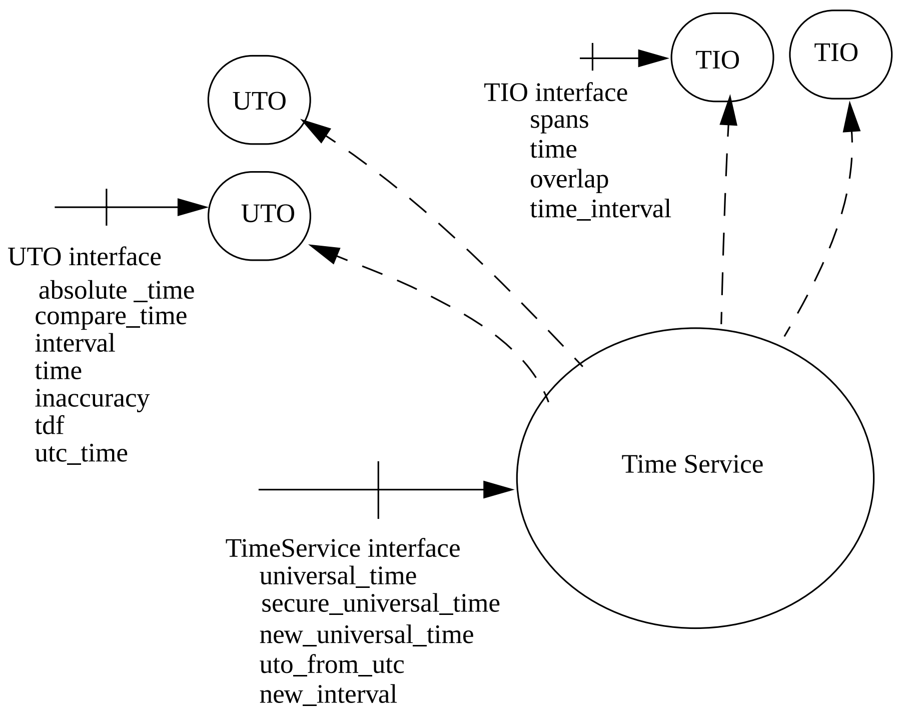

Figure 1-2 Object Model for Time Service

1.3.2 Data Types

A number of types and interfaces are defined and used by this service. All definitions of data structures are placed in the **TimeBase** module. All interfaces, and associated enum and exception declarations are placed in the **CosTime** module. This separation of basic data type definitions from interface related definitions allows other services to use the time data types without explicitly incorporating the interfaces, while allowing clients of those services to use the interfaces provided by the Time Service to manipulate the data used by those services.

```
module TimeBase {

    typedef unsigned long long    TimeT;
    typedef TimeT                 InaccuracyT;
    typedef short                 TdfT;
    struct UtcT {
        TimeT                     time;       // 8 octets
        unsigned long             inacclo;    // 4 octets
        unsigned short            inacchi;    // 2 octets
        TdfT                      tdf;        // 2 octets
                                              // total 16 octets.
    };
```

```
                struct IntervalT {
                    TimeT                lower_bound;
                TimeT                upper_bound;
            };
        };
```

1.3.2.1 Type TimeT

TimeT represents a single time value, which is 64 bits in size, and holds the number of 100 nanoseconds that have passed since the base time. For absolute time the base is 15 October 1582 00:00 of the Gregorian Calendar. All absolute time shall be computed using dates from the Gregorian Calendar.

1.3.2.2 Type InaccuracyT

InaccuracyT represents the value of inaccuracy in time in units of 100 nanoseconds. As per the definition of the inaccuracy field in the *X/Open DCE Time Service* [1], 48 bits is sufficient to hold this value.

1.3.2.3 Type TdfT

TdfT is of size 16 bits short type and holds the time displacement factor in the form of minutes of displacement from the Greenwich Meridian. Displacements East of the meridian are positive, while those to the West are negative.

1.3.2.4 Type UtcT

UtcT defines the structure of the time value that is used universally in this service. The basic value of time is of type **TimeT** that is held in the time field. Whether a **UtcT** structure is holding a relative or absolute time is determined by its history. There is no explicit flag within the object holding that state information. The iacclo and inacchi fields together hold a 48-bit estimate of inaccuracy in the time field. These two fields together hold a value of type **InaccuracyT** packed into 48 bits. The tdf field holds time zone information. Implementation must place the time displacement factor for the local time zone in this field whenever they create a UTO.

The contents of this structure are intended to be opaque, but in order to be able to marshal it correctly, at least the types of fields need to be identified.

1.3.2.5 Type IntervalT

This type holds a time interval represented as two **TimeT** values corresponding to the lower and upper bound of the interval. An **IntervalT** structure containing a lower bound greater than the upper bound is invalid. For the interval to be meaningful, the time base used for the lower and upper bound must be the same, and the time base itself must not be spanned by the interval.

```
module CosTime {
    enum TimeComparison {
        TCEqualTo,
        TCLessThan,
        TCGreaterThan,
        TCIndeterminate
    };

    enum ComparisonType {
        IntervalC,
        MidC
    };

    enum OverlapType {
        OTContainer,
        OTContained,
        OTOverlap,
        OTNoOverlap
        };
};
```

1.3.2.6 Enum ComparisonType

ComparisonType defines the two types of time comparison that are supported. **IntervalC** comparison does the comparison taking into account the error envelope. **MidC** comparison just compares the base times. A **MidC** comparison can never return **TCIndeterminate**.

1.3.2.7 Enum TimeComparison

TimeComparison defines the possible values that can be returned as a result of comparing two UTOs. The values are self-explanatory. In an **IntervalC** comparison, **TCIndeterminate** value is returned if the error envelopes around the two times being compared overlap. For this purpose the error envelope is assumed to be symmetrically placed around the base time covering time-inaccuracy to time+inaccuracy. For **IntervalC** comparison, two UTOs are deemed to contain the same time only if the Time attribute of the two objects are equal and the Inaccuracy attributes of both the objects are zero.

1.3.2.8 Enum OverlapType

OverlapType specifies the type of overlap between two time intervals. Figure 1-3 depicts the meaning of the four values of this enum. When interval A wholly contains interval B, then it is an OTContainer of interval B and the overlap interval is the same as the interval B. When interval B wholly contains interval A, then interval A is OTContained in interval B and the overlap region is the same as interval A. When neither interval is wholly contained in the other but they overlap, then the OTOverlap case applies and the overlap region is the length of interval that overlaps. Finally, when the two intervals do not overlap, the OTNoOverlap case applies.

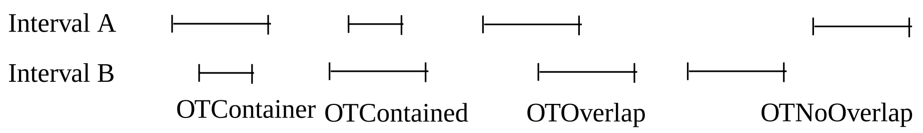

Figure 1-3 Illustration of Interval Overlap

1.3.3 *Exceptions*

This service returns standard CORBA exceptions where specified in addition to the service-specific exception described in this section.

```
module CosTime {
    exception TimeUnavailable {};
}
```

1.3.3.1 *TimeUnavailable*

This exception is raised when the underlying trusted time service fails, or is unable to provide time that meets the required security assurance.

1.3.4 *Universal Time Object (UTO)*

The UTO provides various operations on basic time. These include the following groups of operations:

- Construction of a UTO from piece parts, and extraction of piece parts from a UTO (as read only attributes).
- Comparison of time.
- Conversion from relative to absolute time, and conversion to an interval.

Of these, the first operation is required for completeness, since in its absence it would be difficult to provide a time input to the timer event handler, for example. The second operation is required by the RFP, and the third is required for completeness and usability.

```
module CosTime {
    interface TIO;          // forward declaration
    interface UTO {
        readonly attribute TimeBase::TimeT                  time;
        readonly attribute TimeBase::InaccuracyT     inaccuracy;
        readonly attribute TimeBase::TdfT     tdf;
        readonly attribute TimeBase::UtcT     utc_time;

        UTO absolute_time();
```

```
            TimeComparison compare_time(
                in        ComparisonType  comparison_type,
                in        UTO             uto
            );

            TIO time_to_interval(
                in        UTO             uto
            );

            TIO interval();
        };
};
```

The **UTO** interface corresponds to an object that contains utc time, and is the means for manipulating the time contained in the object. This interface has operations for getting a **UtcT** type data structure containing the current value of time in the object, as well as operations for getting the values of individual fields of utc time, getting absolute time from relative time, and comparing and doing bounds operations on UTOs. The **UTO** interface does not provide any operation for modifying the time in the object. It is intended that UTOs are immutable.

1.3.4.1 Readonly attribute time

This is the time attribute of a UTO represented as a value of type **TimeT**.

1.3.4.2 Readonly attribute inaccuracy

This is the inaccuracy attribute of a UTO represented as a value of type **InaccuracyT**.

1.3.4.3 Readonly attribute tdf

This is the time displacement factor attribute tdf of a UTO represented as a value of type **TdfT**.

1.3.4.4 Readonly attribute utc_time

This attribute returns a properly populated **UtcT** structure with data corresponding to the contents of the UTO.

1.3.4.5 Operation absolute_time

This attribute returns a UTO containing the absolute time corresponding to the relative time in object. Absolute time = current time + time in the object. Raises the CORBA::DATA_CONVERSION exception if the attempt to obtain absolute time causes an overflow.

1.3.4.6 Operation compare_time

Compares the time contained in the object with the time given in the input parameter uto using the comparison type specified in the in parameter **comparison_type**, and returns the result. See the description of **TimeComparison** in Section 1.3.2, "Data Types," on page 1-5 for an explanation of the result. See the explanation of **ComparisonType** in

Section 14.2.2 for an explanation of comparison types. Note that the time in the object is always used as the first parameter in the comparison. The time in the utc parameter is used as the second parameter in the comparison.

1.3.4.7 Operation time_to_interval

Returns a TIO representing the time interval between the time in the object and the time in the UTO passed in the parameter uto. The interval returned is the interval between the midpoints of the two UTOs and the inaccuracies in the UTOs are not taken into consideration. The result is meaningless if the time base used by the two UTOs are different.

1.3.4.8 Operation interval

Returns a TIO representing the error interval around the time value in the UTO as a time interval. **TIO.upper_bound = UTO.time+UTO.inaccuracy**. **TIO.lower_bound = UTO.time - UTO.inaccuracy**.

1.3.5 Time Interval Object (TIO)

The TIO represents a time interval and contains operations relevant to time intervals.

```
module CosTime {
        interface TIO {
            readonly attribute TimeBase::IntervalT time_interval;

            OverlapType spans (
                in   UTO                          time,
                out TIO                           overlap
            );
            OverlapType overlaps (
                in   TIO                          interval,
                out TIO                           overlap
            );

            UTO time ();
        }
}
```

1.3.5.1 *Readonly attribute time_interval*

This attribute returns an **IntervalT** structure with the values of its fields filled in with the corresponding values from the TIO.

1.3.5.2 *Operation spans*

This operation returns a value of type **OverlapType** depending on how the interval in the object and the time range represented by the parameter UTO overlap. See the definition of **OverlapType** in Section 1.3.2, "Data Types," on page 1-5. The interval in the object is interval A and the interval in the parameter UTO is interval B. If **OverlapType** is not **OTNoOverlap**, then the out parameter overlap contains the overlap interval, otherwise the out parameter contains the gap between the two intervals. The exception CORBA::BAD_PARAM is raised if the UTO passed in is invalid.

1.3.5.3 *Operation overlaps*

This operation returns a value of type **OverlapType** depending on how the interval in the object and interval in the parameter TIO overlap. See the definition of **OverlapType** in Section 1.3.2, "Data Types," on page 1-5. The interval in the object is interval A and the interval in the parameter TIO is interval B. If **OverlapType** is not **OTNoOverlap**, then the out parameter overlap contains the overlap interval, otherwise the out parameter contains the gap between the two intervals. The exception CORBA::BAD_PARAM is raised if the TIO passed in is invalid.

1.3.5.4 *Operation time*

Returns a UTO in which the inaccuracy interval is equal to the time interval in the ITO and time value is the midpoint of the interval.

Time Service Interfaces 2

Contents

This chapter contains the following topics.

2.1 Time Service Interface

The **TimeService** interface provides operations for obtaining the current time, constructing a UTO with specified values for each attribute, and constructing a TIO with specified upper and lower bounds.

```
module CosTime {
    interface TimeService {
        UTO universal_time()
                raises(TimeUnavailable
        );
        UTO secure_universal_time()
                raises(TimeUnavailable
        );
        UTO new_universal_time(
                in TimeBase::TimeT          time,
                in TimeBase::InaccuracyT inaccuracy,
                in TimeBase::TdfT           tdf
```

```
            );
            UTO uto_from_utc(
                    in TimeBase::UtcT           utc
            );

            TIO new_interval(
                    in TimeBase::TimeT          lower,
                    in TimeBase::TimeT          upper
            );
        };
    };
```

2.1.1 Operation universal_time

The **universal_time** operation returns the current time and an estimate of inaccuracy in a UTO. It raises TimeUnavailable exceptions to indicate failure of an underlying time provider. The time returned in the UTO by this operation is not guaranteed to be secure or trusted. If any time is available at all, that time is returned by this operation.

2.1.2 Operation secure_universal_time

The **secure_universal_time** operation returns the current time in a UTO only if the time can be guaranteed to have been obtained securely. In order to make such a guarantee, the underlying Time Service must meet the criteria to be followed for secure time, presented in Appendix A, Implementation Guidelines. If there is any uncertainty at all about meeting any aspect of these criteria, then this operation must return the TimeUnavailable exception. Thus, time obtained through this operation can always be trusted.

2.1.2.1 Operation new_universal_time

The **new_universal_time** operation is used for constructing a new UTO. The parameters passed in are the time of type TimeT and inaccuracy of type **InaccuracyT**. This is the only way to create a UTO with an arbitrary time from its components. This is expected to be used for building UTOs that can be passed as the various time arguments to the Timer Event Service, for example. CORBA::BAD_PARAM is raised in the case of an out-of-range parameter value for inaccuracy.

2.1.2.2 Operation uto_from_utc

The **uto_from_utc** operation is used to create a UTO given a time in the **UtcT** form. This has a single in parameter UTC, which contains a time together with inaccuracy and tdf. The UTO returned is initialized with the values from the UTC parameter. This operation is used to convert a UTC received over the wire into a UTO.

2.1.2.3 Operation new_interval

The **new_interval** operation is used to construct a new TIO. The parameters are lower and upper, both of type TimeT, holding the lower and upper bounds of the interval. If the value of the lower parameter is greater than the value of the upper parameter, then a CORBA::BAD_PARAM exception is raised.

2.2 Timer Event Service

The module **CosTimerEvent** encapsulates all data type and interface definitions pertaining to the Timer Event Service.

2.2.1 Object Model

The **TimerEventService** object manages Timer Event Handlers represented by Timer Event Handler objects as shown in Figure 2-1. Each Timer Event Handler is immutably associated with a specific event channel at the time of its creation. The Timer Event Handler can be passed around as any other object. It can be used to program the time and content of the events that will be generated on the channel associated with it. The user of a Timer Event Handler is expected to notify the Timer Event Service when it has no further use for the handler.

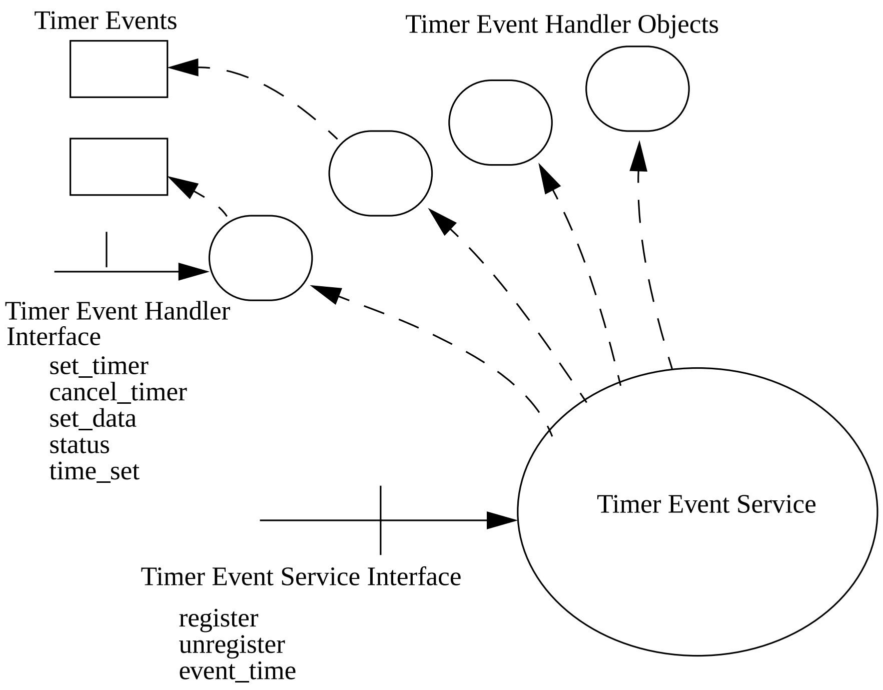

Figure 2-1 Object Model of Timer Event Service

2.2.2 Usage

In a typical usage scenario of this service, the user must first create an event channel of the “push” type (see the *Event Service Specification*). The user must then register this event channel as the sink for events generated by the timer event handler that is returned by the registration operation. The user can then use the timer event handler object to set up timer events as desired. The service will cause events to be pushed through the event channel within a reasonable interval around the requested event time. The implementor of the service will document what the expected interval is for their implementation. The data associated with the event includes a timestamp of the actual event time with the error envelope including the requested event time.

2.2.3 Data Types

All declarations pertaining to this service is encapsulated in the **CosTimerEvent** module.

```
module CosTimerEvent{
    enum TimeType {
        TTAbsolute,
        TTRelative,
        TTPeriodic
    };

    enum EventStatus {
        ESTimeSet,
        ESTimeCleared,
        ESTriggered,
        ESFailedTrigger
    };

    struct TimerEventT{
        TimeBase::UtcT    utc;
        any               event_data;
    };
};
```

2.2.3.1 Enum TimeType

TimeType is used to specify whether a time is TTRelative, TTAbsolute, or TTPeriodic in operations for setting timer intervals for the event-triggering mechanism. The TTRelative value is used to specify that the time provided is relative to current time, TTAbsolute is used to specify that the time provided is absolute, and TTPeriodic is used to specify that the time provided is a period (and hence a relative time) between successive events. If TTPeriodic is used, then the same event continues to be triggered repeatedly at the completion of the time interval specified, until the timer is reset.

2.2.3.2 Enum EventStatus

EventStatus defines the state of a **TimerEventHandler** object. The state ESTimeSet means that the event has been set with a time in the future, and will be triggered when that time arrives. ESTimeCleared means that the event is not set to go off, and the time was cleared before the previously set triggering time arrived. ESTriggered means that the event has already triggered and the appropriate data has been sent the event channel. ESFailedTrigger means that the event did trigger, but data could not be delivered over the event channel.

In case of TTPeriodic events, the status ESTriggered never occurs. Upon successful triggering of a TTPeriodic event, the status is set to ESTimeSet.

2.2.3.3 Type TimerEventT

This is the structure that is returned to the event requester by the time-driven event-triggering mechanism. It has two fields. The first field, utc, contains the actual time at which the event was triggered. This value is set in the time field of utc. The inaccuracy fields inacclo and inacchi of utc are set to the difference between the requested event time and the actual event time.

The second field, **event_data**, contains the data that the requester of the event had asked to be sent when the event was triggered.

2.2.4 Exceptions

Timer Event Service raises standard CORBA exceptions as specified in OMG IDL for the service. It does not have any service-specific exceptions.

2.3 Timer Event Handler

Timer Event Handlers are created and managed by the Timer Event Service. A **TimerEventHandler** object holds information about an event that is to be triggered at a specific time and action that is to be taken when the event is triggered. It provides operations for setting, resetting, and canceling the timer event associated with it, as well as for changing the event data that is sent back as a part of a TimeEventT structure on the event channel upon the triggering of the event. The only thing that cannot be changed is the event channel associated with that event handler. An attribute named status holds the current status of the event handler.

```
module CosTimerEvent {
    interface TimerEventHandler {
        readonly attribute EventStatus      status;
        boolean time_set(
            out CosTime::UTO                uto
        );
        void set_timer(
            in TimeType                     time_type,
            in CosTime::UTO                 trigger_time
```

```
                );
                    boolean cancel_timer();
                    void set_data(
                        in any                              event_data
                    );
                };
            };
```

2.3.1 Attribute status

status is a readonly attribute that reflects the current state of the **TimerEventHandler**. See the definition of **EventStatus** enumerator in Section 2.2.1, “Object Model,” on page 2-3 for details.

2.3.1.1 Operation time_set

Returns TRUE if the time has been set for an event that is yet to be triggered, FALSE otherwise. In addition, it always returns the current value of the timer in the event handler as the out uto parameter.

2.3.1.2 Operation set_timer

Sets the triggering time for the event to the time specified by the uto parameter, which may contain TTRelative, TTAbsolute or TTPeriodic time. The **time_type** parameter specifies what type of time is contained in the uto parameter. The previous trigger, if any, is canceled and a new trigger is enabled at the time specified if absolute, or at current time + time specified if relative. If a relative time value of zero is specified (i.e., the time attribute of utc = 0LL), then the last relative time that was specified is reused. If no relative time was previously specified, then a CORBA::BAD_PARAM exception is raised. If a periodic time is specified (time_type == periodic), then the time parameter is interpreted as a relative time and the time trigger is set at the periodicity defined by the time (i.e., at current time + time or current time + 2 * time).

2.3.1.3 Operation cancel_timer

Cancels the trigger if one had been set and had not gone off yet. Returns TRUE if an event is actually canceled, FALSE otherwise.

2.3.1.4 Operation set_data

The data that will be passed back through the event channel in a **TimerEventT** structure for all future triggering of the event handler is set to **event_data**.

2.4 Timer Event Service

The Timer Event Service provides operations for registering and unregistering events.

```
module CosTimerEvent {
    interface TimerEventService {

        TimerEventHandler register(
            in CosEventComm::PushConsumer event_interface,
            in any                        data
        );
        void unregister(
            in TimerEventhandler          timer_event_handler
        );
        CosTime::UTO event_time(
            in TimerEventT                timer_event
        );
    };
};
```

2.4.1 Operation register

The **register** operation registers the event handler specified by the data and the **event_interface** parameters. When the event handler is triggered, the data is delivered using the **push** operation (of the *PushConsumer* interface in the *Event Service Specification*, *CosEventComm Module*) specified in the **event_interface** parameter. Only the **Push Model** is supported for timer event delivery. Note that the event handler needs to be primed with a triggering time using the **set_time** operation of the **TimerEventHandler** interface in order for an actual event to be triggered. At initialization, the time in the handler is set to current time and its state is set to ESTimeCleared, and no event is scheduled. Raises CORBA::NO_RESOURCE exception if lack of resources causes it to fail to register the event handler.

2.4.2 Operation unregister

The **unregister** operation notifies the service that the timer_event_handler will not be used any more and all resources associated with it can be destroyed. Subsequent attempts to use that object reference will raise CORBA::INV_OBJREF.

2.4.3 Operation event_time

The **event_time** operation returns a UTO containing the time at which the event contained in the **timer_event** structure was triggered.

2.5 Conformance

It is sufficient to provide just the Time Service (module **TimeBase** and **CosTime**) to claim conformance with the Time Service object. To claim conformance with the Timer Event Service, both Time Service and Timer Event Service (module **CosTimerEvent**) must be provided.

In order to conform to the Basic Time Service, the semantics of the **secure_universal_time** operation must be strictly adhered to. In order to return a valid time from this operation, the vendor must provide a statement about how the security assurance criteria specified in Appendix A, Implementation Guidelines, are met in their product. To conform to the object Time Service, in all other cases (i.e., when the security assurance criteria are not satisfied, the **secure_universal_time** operation must raise the TimeUnavailable exception).

Implementation Guidelines A

This appendix contains advice to implementors. Appropriate documented handling of the criteria presented here is mandatory for conformance to the Basic Time Service conformance point.

A.1 Criteria to Be Followed for Secure Time

The following criteria must be followed in order to assure that the time returned by the **secure_universal_time** operation is in fact secure time. If these criteria are not satisfactorily addressed in an ORB, then it must return the TimeUnavailable exception upon invocation of the **secure_universal_time** operation of the **TimeService** interface.

Administration of Time B

Only administrators authorized by the system security policy may set the time and specify the source of time for time synchronization purposes.

B.1 Protection of Operations and Mandatory Audits

The following types of operations must be protected against unauthorized invocation. They must also be mandatorily audited:

- Operations that set or reset the current time
- Operations that designate a time source as authoritative
- Operations that modify the accuracy of the time service or the uncertainty interval of generated timestamps

B.1.1 Synchronization of Time

Synchronization of time must be transmitted over the network. This presents an opportunity for unauthorized tampering with time, which must be adequately guarded against. Time Service implementors must state how time values used for time synchronization are protected while they are in transit over the network.

Time Service implementors must state whether or not their implementation is secure. Implementors of secure time services must state how their system is secured against threats documented in Chapter 15, Security Service Specification. They must also document how the issues mentioned in this section are addressed adequately.

B.2 Proxies and Time Uncertainty

The Time Service object returns a timestamp, which contains both a time and an associated uncertainty interval. These values are considered valid at the instant they are returned by the Time Service object; however, if these values are not delivered to the caller immediately, they may no longer be reliable by the time the caller receives them.

In a CORBA system, the use of proxy objects can render time values unreliable by introducing unpredictable and uncorrected latency between the time the time server object generates a timestamp and the time the caller's time server proxy receives the timestamp and returns it to the caller.

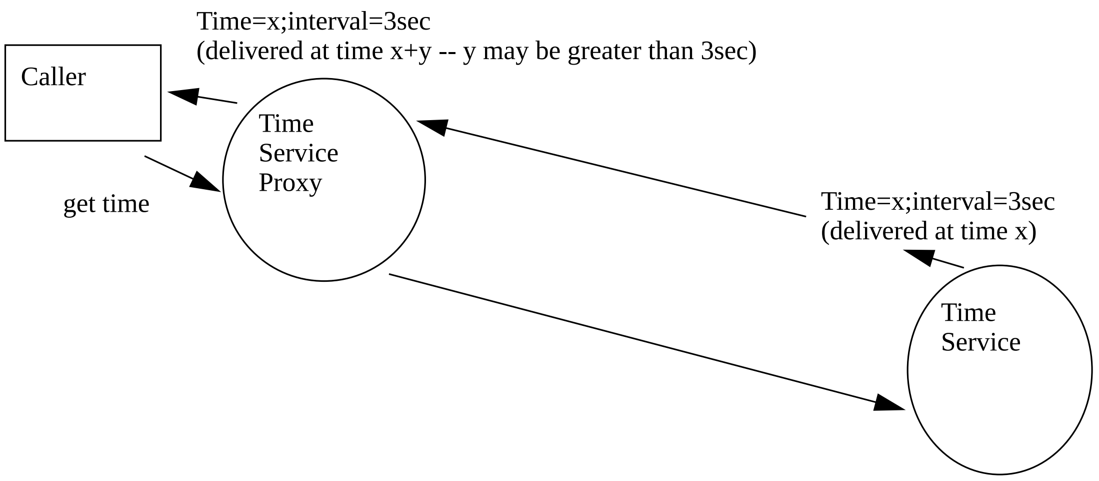

Figure B-1 Time Service and Proxies

Implementors of the Time Service must prevent this problem from occurring. Two possible ways of preventing proxy latency are:

- Prohibit proxies of the time server object (i.e., require a Time Service implementation in every address space that will need to make Time Service calls).
- Create a special time server proxy, which measures latency between the Time Service object and the proxy, recalculates the time interval's uncertainty, and adjusts the interval value before returning the timestamp to the caller.

Other approaches probably exist; the two above are intended as examples only.

Consolidated OMG IDL C

This appendix contains a summary of the OMG IDL defined in this document.

C.1 Time Service

This section contains the OMG IDL definitions pertaining to the Time Service, which is encapsulated in the **TimeBase** and **CosTime** modules. The **TimeBase** module contains the basic data type declarations that can be used by others without pulling in the Time Service interfaces. The **Time Service** interface and associated enums and exceptions are declared in the **CosTime** module.

```
module TimeBase {
    typedef unsigned long long    TimeT;
    typedef TimeT                 InaccuracyT;
    typedef short                 TdfT;
    struct UtcT {
        TimeT                     time;      // 8 octets
        unsigned long             inacclo;   // 4 octets
        unsigned short            inacchi;   // 2 octets
        TdfT                      tdf;       // 2 octets
                                             // total 16 octets.
    };

    struct IntervalT {
        TimeT             lower_bound;
        TimeT             upper_bound;
    };
};


module CosTime {

    enum TimeComparison {
        TCEqualTo,
        TCLessThan,
        TCGreaterThan,
        TCIndeterminate
};

enum ComparisonType{
        IntervalC,
        MidC
};

enum OverlapType {
        OTContainer,
        OTContained,
        OTOverlap,
        OTNoOverlap
};

exception TimeUnavailable {};
interface TIO;              // forward declaration

interface UTO {
        readonly attribute TimeBase::TimeTtime;
        readonly attribute TimeBase::InaccuracyTinaccuracy;
        readonly attribute TimeBase::TdfT       tdf;
        readonly attribute TimeBase::UtcT       utc_time;
        UTO absolute_time();
        TimeComparison compare_time(
                in      ComparisonTypecomparison_type,
                in      UTO             uto
        );
        TIO time_to_interval(
                in      UTO             uto
        );
        TIO interval();
};

interface TIO {
        readonly attribute TimeBase::IntervalT time_interval;
        OverlapType spans (
                        in    UTO       time,
                        out TIO         overlap
        );
        OverlapType overlaps (
                        in    TIO       interval,
                        out TIO         overlap
        );
        UTO time ();
};

interface TimeService {
        UTO universal_time()
                raises(TimeUnavailable
        );
        UTO secure_universal_time()
                raises(TimeUnavailable
        );
        UTO new_universal_time(
                in TimeBase::TimeT          time,
                in TimeBase::InaccuracyT inaccuracy,
                in TimeBase::TdfT           tdf
        );
        UTO uto_from_utc(
                in TimeBase::UtcT           utc
        );
        TIO new_interval(
                in TimeBase::TimeT          lower,
                in TimeBase::TimeT          upper
        );
    };
};
```

C.2 Timer Event Service

This section contains all the OMG IDL definitions pertaining to the Timer Event Service, which are encapsulated in the **CosTimerEvent** module. This module depends on **TimeBase**, **CosTime**, **CosEventComm**, and **CORBA**.

```
module CosTimerEvent{
    enum TimeType {
                TTAbsolute,
                TTRelative,
                TTPeriodic
    };

    enum EventStatus {
                ESTimeSet,
                ESTimeCleared,
                ESTriggered,
                ESFailedTrigger
    };

    struct TimerEventT {
        TimeBase::UtcT                      utc;
        any                                 event_data;
    };

    interface TimerEventHandler {
        readonly attribute EventStatus status;
        boolean time_set(
            out CosTime::UTO                uto
```

```
		);
			void SetTimer(
				in TimeType					time_type,
				in CosTime::UTO				trigger_time
			);
			boolean cancel_timer();
			void set_data(
				in any						event_data
			);
		};

		interface TimerEventService {
			TimerEventHandler register(
				in CosEventComm::PushConsumer event_interface,
				in any						data
			);
			void unregister(
				in TimerEventHandler		timer_event_handler
			);
			CosTime::UTO event_time(
				in TimerEventT			timer_event
			);
		};
	};
```

Notes for Users — D

This appendix contains notes covering the following matters:

- Guarding against proxy-related inaccuracies in time contained in UTO.
- How to transmit time and time intervals across the network and recover the corresponding UTO and TIO at the other end.

D.1 Proxies and Time

As explained in Appendix C, Consolidated OMG IDL, indiscriminate use of remote proxies to obtain value of current time can lead to obtaining values of time in which the inaccuracy is incorrect due to transmission delays. Consequently, care should be taken to ensure that the local Time Service is used to obtain the value of current time.

D.2 Sending Time Across the Network

When passing small objects such as UTO and TIO from one location to another, one should be aware that each time the passed object reference is used by the recipient it causes an object invocation to take place across the network and is inherently inefficient. The preferred way of dealing with this problem is to pass small objects by value instead of by reference. Unfortunately, due to various reasons, OMG IDL does not allow specification of passing of object parameters by value. Consequently, the user has to explicitly take action to avoid this problem.

The interfaces defined contain features that make it possible for the user to explicitly send the value of time, and time interval across from one location to another and then reconstruct the appropriate object at the receiving end. This is done as follows:

- The signature of the operation that passes time or time interval as a parameter across the network should specify that time is passed as the data type and not as an object reference. For example, for passing universal time, a signature such as

```
void foo(in TimeBase::UtcT);
```

should be used instead of

```
void foo(in CosTime::UTO);
```

- The invoker should use the data attribute of the UTO as the in parameter. In pseudo-code, something such as the following should be done by the invoker:

```
CosTime::UTO uto = CosTime::universal_time();
foo(uto.data);
```

- At the server end, the time data received can be converted to a UTO as follows:

```
foo(in TimeBase::UtcT utc) {
    CosTime::UTO uto = CosTime::TimeService::uto_from_utc(utc);

    .....

};
```

It would be nice to say in the definition of the **foo** operation something such as:

```
foo(in byvalue UTO uto);
```

and have the system take care of doing essentially what is described above. However, there are difficult model- and paradigm-related issues that need resolution before such a change can be coherently proposed.

Extension Examples E

The process of constructing the contents of a **TimeBase::TimeT** value can be quite tedious, involving many 64-bit multiplications and additions. The CORBA Facility for Time Representation is going to provide user-friendly ways of creating **TimeT** data and displaying them. However, if one is planning to use only the Time Service, it will be necessary to construct some rudimentary facility to build **TimeT** things. This appendix shows one way of doing this as an example of how to extend this service in useful ways.

E.1 Object Model

Following the design pattern used in the rest of this service definition, the basic extension is to define a TimeI object corresponding to the **TimeT** structure, and extend TimeService to provide an operation for creating such objects. The **TimeI** object has attributes corresponding to the user-friendly representation of time such as year, month, day, hour, minute, second and microsecond.

E.2 Summary of Extensions

The additions are encapsulated in the **FriendlyTime** module. The changes are as follows:

- Data type declaration for components of time.
- Definition of the **TimeI** interface, consisting mostly of attributes.
- Definition of the **FriendlyTime::TimeService** interface derived from the **CosTime::TimeService** interface, for adding the operation to create **TimeI** objects.

E.3 Data Types

The data types are self-explanatory for the purposes of setting up this example. A complete specification should state more specific properties of each of these data types.

```
module FriendlyTime {
        typedef unsigned short    YearT; // must be > 1581
        typedef unsigned short    MonthT; // 1 - 12
        typedef unsigned short    DayT; // 1 - 31
        typedef unsigned short    HourT; // 0 - 24
        typedef unsigned short    MinuteT; // 0 - 59
        typedef unsigned short    SecondT; // 0 - 59
        typedef unsigned short    MicrosecondT;
}
```

E.4 Exceptions

No exceptions are defined in this module.

E.5 Friendly Time Object

The time object provides a friendly interface to the various components usually used to represent time in normal human discourse. The set of attributes used in this example are by no means exhaustive, and is used only for illustrative purposes.

```
module FriendlyTime {
    interface TimeI {
        attribute    YearT                year;
        attribute    MonthT               month;
        attribute    DayT                 day;
        attribute    HourT                hour;
        attribute    MinuteT              minute;
        attribute    SecondT              second;
        attribute    MicrosecondT         microsecond;
        attribute    TimeBase::TimeT      time;
        void reset(); // set all attributes to zero
    };
};
```

The **TimeI** object can be viewed as a representation conversion object. The general technique for using it is to create one using the operation **CosFriendlyTime::TimeService::time** introduced in Section D.7, Extended Time Service. This creates a TimeI object with time set to zero in it. Then the **_set** operation can be used to set the values of the various attributes. Finally, the attribute time can be used to get the corresponding **TimeT** value.

Conversely, one can set any **TimeT** value in the time attribute and then get the year, month, etc. from the appropriate attributes.

The **reset** operation facilitates reuse of time objects.

E.6 Extended Time Service

CosTime::TimeService is extended by derivation to provide an operation for creating **TimeI** objects.

```
module FriendlyTime {
    interface TimeService : CosTime::TimeService {
        TimeI time();
    };
};
```

E.7 Epilogue

The extension provided in this appendix makes the Time Service defined in the normative part of the document more easily usable. This leads one to wonder why this extension is not part of the main body of this specification. The reason is that there is no agreement on what the most useful representative components of time are, and the feeling that in general this should be dealt with at the Common Facilities level in general. We still felt that it would be useful to illustrate how easy it is to extend the basic service to provide this ease-of-use facility, thus this appendix.

References F

- X/Open DCE Time Service, X/Open CAE Specification C310, November 1994.
- RFC 1119 Network Time Protocol, D. Mills, September 1989.
- Probabilistic Clock Synchronization, Flaviu Cristian, Distributed Computing (1989) 3: Pg. 146-158.
- OMG IDL type Extensions RFP, Andrew Watson Ed., OMG Doc. No. 95-1-35.
- CORBAServices: Common Object Service Specification, OMG Doc. No. 95-3-31, March 31 1995 revision, Chapter 4, Event Service Specification, Section 4.2 Pg. 4-6.
- CORBAServices: Common Object Service Specification, OMG Doc. No. 96-10-1, October 1996 revison, Chapter 15, Security Service Specification.