X/Open CAE Specification

X/Open DCE: Time Services

X/Open Company Ltd.

© *October 1994, X/Open Company Limited*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior permission of the copyright owners.

Published by X/Open Company Limited under license from the Open Software Foundation (OSF). Portions of this document include text excerpted and/or derived from the Open Software Foundation Application Environment Specification for Distributed Computing (AES/DC) with the permission of OSF. However, the text appearing herein does not represent the official OSF version of the AES/DC, which is copyright © 1992, 1993 Open Software Foundation, Inc. This document and the software to which it relates are derived in part from materials which are copyright © 1990, 1991 Digital Equipment Corporation and copyright © 1990, 1991 Hewlett-Packard Company.

X/Open CAE Specification

X/Open DCE: Time Services

ISBN: 1-85912-067-9
X/Open Document Number: C310

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

Part	**1**	**Time Services and Protocols**	**1**
Chapter	**1**	**Introduction**	**3**
	1.1	Time in Distributed Systems	3
	1.2	Time and Clocks	4
	1.2.1	Time	4
	1.2.2	Clocks	4
	1.2.3	Correct Time	5
	1.3	Time Service Rationale	6
	1.4	Architectural Overview	7
	1.5	Conformance Requirements	8
Chapter	**2**	**Time Service Functional Overview**	**9**
	2.1	Obtaining a Time Value From a Server	9
	2.2	Obtaining Time Values from Time Providers	12
	2.3	Computing a Correct Time	13
	2.4	Adjusting the Clock	16
	2.5	Determining the Inaccuracy	17
	2.6	Leap Seconds	19
	2.6.1	Leap Seconds and Inaccuracy	19
	2.6.2	Leap Seconds and Obtaining Time	22
	2.7	Time Zones	23
	2.8	Local Faults	23
	2.9	Primitive Procedures	24
	2.9.1	The EstimateServerTime Procedure	25
	2.9.2	The ComputedTimeMinimum Procedure	26
	2.9.3	The AdjustClock Procedure	28
	2.9.4	The AdjustClkEnd Procedure	30
	2.9.5	The SetClock Procedure	31
	2.9.6	The CalcInaccuracy Procedure	32
Chapter	**3**	**Time Service Configuration**	**33**
	3.1	Configuration	34
	3.2	Local Set Import and Export	35
	3.3	Global Set Import and Export	36
	3.4	Couriers	37
Chapter	**4**	**Time Service Clerk Specification**	**39**
	4.1	Initialising the System Clock	39
	4.2	Synchronisation	40
	4.3	Determining the Next Synchronisation	42
	4.4	Maintaining the Server Lists	43

Chapter 5 Time Service Server Specification **45**
5.1 Initialisation 45
5.2 Epochs 46
5.3 Synchronisation 47
5.3.1 Synchronising with a TP 47
5.3.2 Synchronising with Other Servers 47
5.4 Determining the Next Synchronisation 49
5.5 Checking For Faulty Servers 49
5.6 Maintaining the Server Lists 50

Chapter 6 Time Service IDL Declarations **51**
6.1 Data Types and Ranges 51
6.1.1 The utc Structure 51
6.1.2 Parameter Ranges 51
6.2 Local Set Time Service Interface 52
6.2.1 ClerkRequestTime 52
6.2.2 ServerRequestTime 53
6.3 Global Set Time Service Interface 54
6.3.1 ClerkRequestGlobalTime 54
6.3.2 ServerRequestGlobalTime 55
6.4 Time Provider Interface 56
6.4.1 Data Types 57
6.4.2 ContactProvider 58
6.4.3 ServerRequestProviderTime 58

Part 2 Time API **59**

Chapter 7 Time API **61**
7.1 Timestamps 62
7.1.1 The utc_t Type 62
7.2 Non-opaque Time Representations 63
7.2.1 Character Representations of Time 63
7.2.2 Character-relative Time Type 64
7.2.3 The tm Structure 65
7.2.4 The timespec Structure 65
7.2.5 The reltimespec Structure 65
7.2.6 Conversion Rules 66
7.3 Time Service API Taxonomy 67
7.3.1 Time Conversions 68

Chapter 8 Time API Manual Pages **71**
utc_abstime( ) 72
utc_addtime( ) 73
utc_anytime( ) 74
utc_anyzone( ) 75
utc_ascanytime( ) 76
utc_ascgmtime( ) 77
utc_asclocaltime( ) 78

utc_ascreltime( ) 79
utc_binreltime( ) 80
utc_bintime( ) 81
utc_boundtime( ) 82
utc_cmpintervaltime( ) 83
utc_cmpmidtime( ) 84
utc_gettime( ) 85
utc_getusertime( ) 86
utc_gmtime( ) 87
utc_gmtzone( ) 88
utc_localtime( ) 89
utc_localzone( ) 90
utc_mkanytime( ) 91
utc_mkascreltime( ) 92
utc_mkasctime( ) 93
utc_mkbinreltime( ) 94
utc_mkbintime( ) 95
utc_mkgmtime( ) 96
utc_mklocaltime( ) 97
utc_mkreltime( ) 98
utc_mulftime( ) 99
utc_multime( ) 100
utc_pointtime( ) 101
utc_reltime( ) 102
utc_spantime( ) 103
utc_subtime( ) 104

Appendix A **Time Representation** **105**

Appendix B **Parameters, Constants and Names** **107**
B.1 Parameters 107
B.2 Constants 107
B.3 Names 108

Appendix C **LAN Services Interface Definition** **109**

Appendix D **Time Interval Arithmetic** **111**

Index **113**

List of Figures

1-1 Properties of Clocks 4
2-1 Components of Delay 10
2-2 Computing the Best Correct Time by Taking an Intersection 14
2-3 Computing the Best Correct Time Assuming One Faulty Server 14
2-4 How a Negative Leap Second Causes a Clock to Be Incorrect 20
2-5 How a Positive Leap Second Causes a Clock to Be Incorrect 21

7-1 DTS API Routines by Functional Grouping 67

List of Tables

B-1 Default Parameter Values 107
B-2 Architectural Constant Values 107

Preface

X/Open

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

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This document is a CAE Specification (see above). It specifies the Distributed Time Service (DTS)[1], time representations, RPC interfaces to the DTS and application programmers' interfaces to the DTS.

The purpose of this document is to provide a portability guide for DTS application programs and a conformance specification for DTS implementations.

Structure

This document is organised into two parts.

Part 1, Time Services and Protocols specifies the DTS and the RPC interfaces to the DTS. It contains material mainly relevant to implementors. However, Chapter 1 also contains material relevant to application programmers. Chapter 1 introduces the DCE Time Service specification. The remaining chapters in Part 1 give a detailed specification of the time service. Chapter 2 specifies a set of time related services that implementations must provide. Chapter 3 specifies Time Service configuration. Chapter 4 and Chapter 5 specify clerk and server entities, respectively. Chapter 6 gives the RPC interfaces to Time Server and Time Provider services.

Part 2, Time API specifies a portable DTS Application Programmers' Interface (API). It contains material relevant both to application programmers and implementors.

This volume also includes a series of appendices containing a specification of time representation, a specification of algorithms for the calculation of time intervals, and a specification of default values for several parameters used in the DTS specification. It contains material mainly relevant to implementors.

---

1. The terms ''Distributed Time Service (DTS)'' and ''Time Services'' are used interchangeably throughout this document.

Intended Audience

This document is written for applications programmers who need to make use of the DTS API and developers of DTS implementations.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members. The procedures defined in this specification also use **Bold**.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — variable names, for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — utility names
  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*().
- Normal font is used for the names of constants and literals.
- The notation <**file.h**> indicates a header file.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font.
- Variables within syntax statements are shown in *`italic fixed width font`*.

Trade Marks

X/Open™ and the ‘‘X’’ device are trade marks of X/Open Company Limited.

Referenced Documents

The following standards are referenced in this specification:

ISO 8601
: ISO 8601: 1988, Data Elements and Interchange Formats — Information Interchange — Representation of Dates and Times.

ISO 8859-1
: ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

The following X/Open document is referenced in this specification:

DCE RPC
: X/Open CAE Specification, August 1994, X/Open DCE: Remote Procedure Call (ISBN: 1-85912-041-5, C309).

The following non-X/Open documents provide background information to this specification:

- Byron E. Blair, Ed., *Time and Frequency: Theory and Fundamentals*, U.S. Department of Commerce, National Bureau of Standards, May 1974.
- International Radio Consultative Committee (CCIR), *Detailed Instructions by Study Group 7 for Implementation of Recommendation 460 Concerning the Improved Coordinated Universal Time (UTC) System, Valid from 1 January 1972*, XII Plenary Assembly CCIR, New Delhi, 1970.
  Reprinted in Byron E. Blair, Ed., *Time and Frequency: Theory and Fundamentals*, U.S. Department of Commerce, National Bureau of Standards, May 1974.
- K.A. Marzullo, *Maintaining Time in a Distributed System: An Example of a Loosely-coupled Distributed Service*, Ph.D. dissertation, Stanford University, February, 1984.

X/Open CAE Specification

Part 1

Time Services and Protocols

X/Open Company Ltd.

Chapter 1

Introduction

This chapter introduces the DCE Time Service specification. It includes a brief discussion of the problem of time in distributed systems, definitions of a set of basic concepts related to time and clocks, a rationale for the time service, and an overview of the specified architecture.

The remaining chapters in Part 1 give a detailed specification of the time service.

1.1 Time in Distributed Systems

The DCE Time Service is designed to provide a consistent measure of time in a distributed system. In centralised systems, it is relatively easy to provide a consistent measure of time by means of a single, central clock. In distributed systems, however, where each node has its own clock, this consistent measure of time is not available. This is problematic since a distributed system actually has an additional need for a consistent measure of time: coordinating the activities of distributed applications.

Providing a consistent measure of time in a distributed system is complicated by a number of factors, including:

- differences between time sources used by various components of the distributed system
- indeterminate processing and propagation delays for network time requests
- partial failure modes resulting from failure of some components in the system.

Nevertheless, with a well-defined architecture and fault-tolerant algorithms, a reliable measure of time can be provided in a distributed system. This specification presents such an architecture together with algorithms.

1.2 Time and Clocks

This section defines a set of basic concepts related to time and clocks that are used throughout the remainder of the specification.

1.2.1 Time

The abstract notion of time can be defined by a frame of reference to which all values of time are related. In this specification, the reference time standard is Coordinated Universal Time (UTC), an international standard maintained by the International Time Bureau (BIH).

Political representations of UTC (such as Eastern Daylight Time (EDT), Pacific Daylight Time (PDT), and so on) are functionally equivalent. They are derived from UTC by adding a Time Differential Factor (TDF) to a UTC value.

1.2.2 Clocks

For our purposes, a clock is a device that provides a measure of UTC. In this specification, we denote the value of some clock *j* at the instant when UTC = *t* by $T_j(t)$. (Throughout this specification, times that are specified by the lower case letter *t* refer to UTC.) In order to quantify how well a clock measures UTC, we introduce several properties of clocks. These are depicted in Figure 1-1.

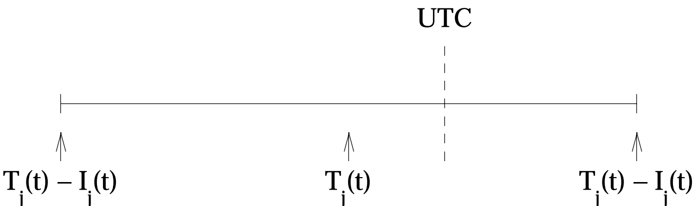

Figure 1-1 Properties of Clocks

The properties are:

inaccuracy — In a perfect clock, $T_j(t) = t$ over all time. Real clocks, however, always exhibit some deviation from UTC, called *error*. The error of a clock can never be known exactly, but it can be bounded. We call this bound *inaccuracy* and denote it by $I_j(t)$.

Since a clock always has some error, bounded by its inaccuracy, it should not be thought of as reporting a scalar value for UTC, but rather a range of values, any one of which could be UTC. More precisely, suppose that at the instant when UTC is *t*, the time on clock *j* is $T_j(t)$, and the inaccuracy is $I_j(t)$. Then according to clock *j*, UTC at this instant is between $T_j(t)-I_j(t)$ and $T_j(t)+I_j(t)$. These two values are the endpoints of a time interval in which UTC lies. This notion is illustrated in Figure 1-1.

drift — The error of a clock is not constant but changes with time. The rate of change of error is called *drift*. Like error, it cannot be known exactly but can be bounded. However, while error can become arbitrarily large — so that its bound must vary with time — drift can be bounded by a constant. We denote the bound on the drift of clock *j* by δ_j.

skew — Error and its bound, inaccuracy, measure the difference between a clock and UTC. We define *skew* as the difference between two clocks. The skew between

clocks *j* and *k* is given by:

$$\sigma_{jk}(t) = T_j(t) - T_k(t)$$

resolution — In general, clocks of computer systems are digital; they increment at discrete points in time. The elapsed time between these points, or ticks, is called the *resolution* of the clock and is denoted by ρ; the clock increments by ρ at each tick.

1.2.3 Correct Time

For a clock to be useful, it must not only supply the time but the inaccuracy as well. In this specification, a clock supplies its current inaccuracy $I_j(t)$ together with the time $T_j(t)$. The clock is *correct* if UTC lies within the interval given by the time and inaccuracy, so that $T_j(t) - I_j(t) \leq t \leq T_j(t) + I_j(t)$. A faulty clock is one that does not meet this correctness criterion. A time interval that satisfies the correctness condition is known as the *correct time*.

1.3 Time Service Rationale

Three canonical uses of time are identified:

Timing and Ordering of Events
: Given two events and the times at which they occurred, ascertain the order in which they occurred. Clearly, if the time intervals do not intersect, this can be accomplished. If the intervals intersect, the order cannot be ascertained. For this canonical use of the time, the inaccuracy must be known. Examples are event logs and decisions based on causality.

Timing Intervals
: Given two values of time, calculate the period of time that elapsed between the two readings. This period can only be computed to within a known error, which is a function of the inaccuracies of the time values. Examples are packet lifetimes, password expiration, and events external to the system interfacing through some I/O channel.

Scheduling Events
: Schedule some event to occur either before or after a specified time. To perform this operation correctly, the inaccuracy must be known. (It is not possible to schedule an event for an exact time.)

In developing a Time Service architecture that meets these needs, the following goals were considered:

Correctness
: This is of greatest importance. The specification is designed to minimise the probability of a client obtaining an incorrect value for the time. Unlike other services where faults or errors can usually be detected immediately, faulty time values are difficult (if not impossible) to detect and may manifest themselves in ways that lead to undetected, incorrect operation of the system.

Client/Server Model
: The specification conforms to the client/server model of distributed systems. Hence, clients query the service for the correct time. Clients do not have to perform any synchronisation among themselves.

Simplicity/Consistency
: The specification provides a simple and conventional view of time to consumers of time (users and application software). Only a single time standard is considered. This service provides UTC and synchronises clients to UTC.

Quality
: Every value of time has associated with it a quality, expressed quantitatively as its inaccuracy. This specification places no requirements on the quality of time values. Quality is implementation-specific. Furthermore, the specification does not impose any restrictions that would limit the quality that specific implementations can achieve.

Fault Tolerance
: The specification is designed to withstand arbitrary failures of a small number of servers.

Scale
: The specification is designed to operate in a network of arbitrary size and can gracefully accommodate network growth.

Monotonicity
: Since time always advances, clocks too must always advance. This specification ensures that a non-faulty clock never runs backward. The specification also provides for gradual adjustment of clocks to synchronise them with provided time values.

1.4 Architectural Overview

The Time Service is a network application entity. The Time Service entity on a system is responsible for synchronising the clock on that system to Coordinated Universal Time (UTC). It also provides estimated UTC and inaccuracy (which may vary) to clients on that system.

The specification distinguishes two types of Time Service entities: *servers* and *clerks*. Only one of these two entities exists on a given system. The term ‘‘entity’’ is used to refer to clerks and servers alike. When it is necessary to distinguish one or the other, we use the terms ‘‘clerk’’ and ‘‘server’’.

Most systems contain clerk entities. To synchronise its clock, a clerk periodically obtains the time from some servers, computes a correct time from these time values, and uses the result to adjust its local clock by advancing or retarding it. The purpose of the adjustment is to compensate for the drift of the clerk’s clock and to reduce its inaccuracy. Otherwise, inaccuracy increases continually to account for the drift of the clock.

In order for clerks to perform this procedure, the Time Service requires that several systems, scattered throughout the distributed system, contain server entities that listen for and respond to time requests.

It is desirable for a server to have a device on its system that acquires UTC from some external source. We call such a device an *External Time Provider* (ETP). There are several means by which an ETP can acquire UTC including telephone, radio and satellite. The AES specifies only the *Time Provider* (TP) interface that must be exported to servers. The specification of an ETP itself is outside the scope of the AES and is implementation-specific.

If there is a TP on its system, a server synchronises its clock by periodically requesting time from the TP. If there is no TP, the server synchronises with other servers in a similar fashion to clerks.

In addition to synchronising and answering time requests, servers running TPs also obtain the time from other servers periodically in order to check for and report faulty servers.

Note that, without any TPs, the inaccuracies of all servers increase because of drift, eventually becoming unreasonably large. To prevent this, a manager must mimic an ETP periodically by providing servers with accurate time. To reduce the likelihood of a manager entering an erroneous time, servers validate management-supplied time intervals by comparing them with the time intervals of their clocks.

The estimated time and inaccuracy of the local clock is made available to clients by means of a subroutine call.

1.5 Conformance Requirements

To conform to this document, implementations must meet the following requirements:

- Implementations must conform to the algorithms specified in Section 2.9 on page 24 and its subsections.
- Implementations must conform to the algorithms specified in Section 3.2 on page 35, Section 3.3 on page 36 and Section 3.4 on page 37.
- Implementations must conform to the algorithms specified in Chapter 4 and its subsections.
- Server implementations must conform to the algorithms specified in Chapter 5 and its subsections.
- Implementations must conform to and support the rules, values and ranges, types and structures, and semantics specified in Chapter 6 and its subsections.
- Implementations must support and conform to the conversion rules, types and representations, as well as the API naming, syntax and semantics specified in Chapter 7 and its subsections.
- Implementations must conform to the binary representation of time specified in Appendix A.
- Implementations must conform to the architectural constant values specified in Section B.2 on page 107.
- Implementations must use the value specified in Appendix C as the interface identifier for the LAN Services interface.
- Implementations must conform to the algorithms specified in Appendix D on page 111 for time interval arithmetic.

Chapter 2

Time Service Functional Overview

This chapter describes a set of procedures that can be used to satisfy the time service goals specified in Chapter 1. These procedures apply both to DTS clerks and servers. Chapter 4 and Chapter 5 specify clerk and server operation in terms of these procedures.

From the point of view of the RPC client/server model, both DTS clerks and DTS servers act in both client roles and server roles. A DTS clerk acts as a server for RPC applications that request time from the time service, but the clerk acts as a client when it requests time from a DTS server. Similarly, a DTS server acts as a client when it requests time from another DTS server or a Time Provider. In order to avoid ambiguity, this chapter uses the term *server acting as client* to refer to a DTS server that is acting as the client of some other DTS server.

2.1 Obtaining a Time Value From a Server

This section describes how a DTS clerk or DTS server obtains time from some DTS server by performing a Remote Procedure Call (RPC).

Figure 2-1 on page 10 represents a time-space diagram of the messages that implement the clerk or server's remote procedure call to the server.

The times t_k, where k=1, 2 . . . 8, correspond to values of UTC, which can never be known exactly. A double arrow denotes a message transmission and a single arrow denotes a message reception.

The procedure begins with the clerk or server acting as client reading its clock in preparation for sending the request. The value of UTC at this instant is t_1 and the clock reads $T_c(t_1)$. Immediately thereafter, the clerk or server acting as client sends its request. In most systems, the request incurs some sending delay s_c as the operating system transfers it to the network adapter and as the adapter queues it for transmission. Although some component of s_c is deterministic (some minimum number of instructions must be executed in transferring the request from the clerk or server acting as client to the network), the remainder is random, depending on other system and network activities at that time. In Figure 2-1 on page 10, the request is actually transmitted at time t_2 and received at the server system at time t_3 after a random propagation delay denoted by τ_{cs}.

The first action the server takes upon the arrival of a request is to note the time. As with sending, some random receiving delay r_s is incurred. Consequently, the server notes the arrival time of the request at time t_4, which is measured as $T_s(t_4)$ by the server's clock. The server then processes the request and sends a response datagram at time t_5. The clerk or server acting as client receives and time stamps this response at time t_8. The response incurs similar delays to those of the request, namely s_s, τ_{sc} and r_c. We do not assume that the corresponding delays of the request and response messages are equal; that is, in general: $s_s \neq s_c$, $\tau_{sc} \neq \tau_{cs}$ and $r_c \neq r_s$.

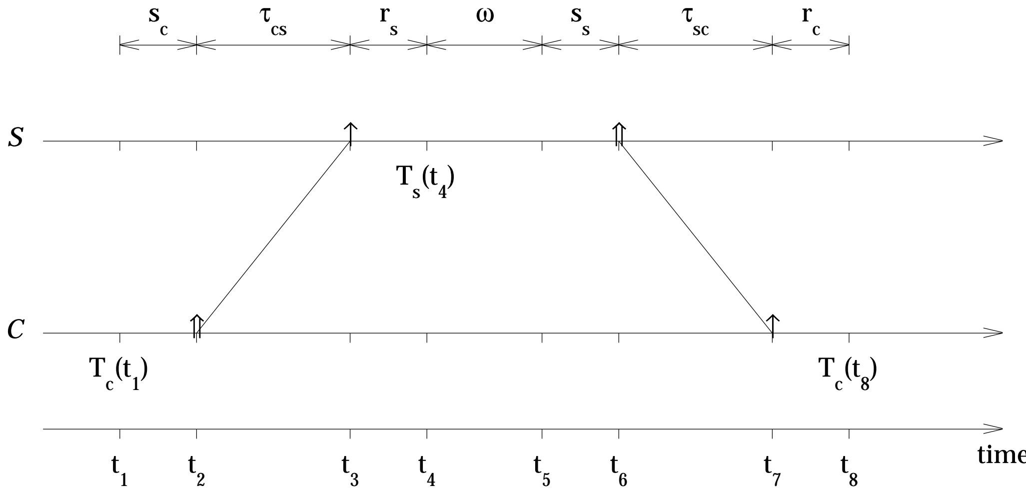

Figure 2-1 Components of Delay

The processing delay at the server is explicitly accounted for in Figure 2-1. It illustrates the components of the delay incurred when a clerk or server acting as a client obtains the time from a server. Although you might expect the delay to be small, this may not be the case. For example, at a highly loaded server, a time request may be queued for service rather than processed immediately; in a secure system, a server may undertake the time-consuming task of signing the response message.

The processing delay is returned in the response message together with the value of the server's clock at t_4, $T_s(t_4)$, and the clock's inaccuracy at that instant, $I_s(t_4)$. This eliminates any need for servers to respond to requests with a high degree of timeliness. Instead, the clerk or server acting as client can compensate for processing and scheduling delays at the server as shown in the following calculations.

The value of the server's clock at t_4 is not useful to the clerk or server acting as client as it does not know the value of its own clock at that instant. So, the clerk or server acting as client must calculate the value of the server's clock at an instant for which it does know the value of its own clock. Let this instant be t_1 (however, t_8 or any other instant could equally have been chosen). The result of this calculation is effectively the reading of the server's clock at the instant when the clerk's clock read $T_c(t_1)$.

Assuming the server is non-faulty, the clerk or server acting as client knows that:

$$T_s(t_4) - I_s(t_4) \le t_4 \le T_s(t_4) + I_s(t_4)$$

Therefore, t_1 is in the range:

$$T_s(t_4) - I_s(t_4) - x \le t_1 \le T_s(t_4) + I_s(t_4) - x \qquad (2.1)$$

where, from Figure 2-1:

$$x = s_c + \tau_{cs} + r_s$$

Although *x* is unknown, it is in the range:

$$0 \le x \le t_8 - t_1 - w$$

Now, $t_8 - t_1$ is given by:

$$t_8 - t_1 \le (T_c(t_8) + \rho - T_c(t_1))(1 + \delta_c) \qquad (2.2)$$ (See footnote[1].)

The clock resolution ρ, in the preceding formula, accounts for the discrete nature of the clerk's clock and the factor $(1 + \delta_c)$ accounts for its drift over the period $[t_1, t_8]$. Consequently:

$$0 \le x \le (T_c(t_8) + \rho - T_c(t_1))(1 + \delta_c) - w \qquad (2.3)$$

Combining the inequalities of (2.1) and (2.3), the clerk or server acting as client ascertains that, when its clock read $T_c(t_1)$, the server believed that UTC was in the range:

$$T_s(t_4) - I_s(t_4) - (T_c(t_8) + \rho - T_c(t_1))(1 + \delta_c) + w \le t_1 \le T_s(t_4) + I_s(t_4) \qquad (2.4)$$

This estimate of the server's clock at time t_1 can be represented as a time with inaccuracy by:

$$T_s^{(c)}(t_1) = T_s(t_4) - (T_c(t_8) + \rho - T_c(t_1))(1 + \delta_c)/2 + w/2$$

$$I_s^{(c)}(t_1) = I_s(t_4) + (T_c(t_8) + \rho - T_c(t_1))(1 + \delta_c)/2 - w/2. \qquad (2.5)$$ (See footnote[2].)

A sophisticated implementation of a server would include all known components of delay in *w* so as to reduce the estimated inaccuracy in $I_s^{(c)}(t_1)$. For example, it could include any known components of s_s and r_s; but any component of r_s included in *w* requires that $T_s(t_4)$ be decremented by that amount.

Similarly, a sophisticated clerk or server acting as client could reduce $I_s^{(c)}(t_1)$ by compensating for known components of s_c, r_c, τ_{sc} or τ_{cs}. To prevent double compensation, servers are prohibited from compensating for known components of τ_{sc} or τ_{cs}.

---

1. This inequality is complicated further if you consider that a leap second might occur between t_8 and t_1. We describe the appropriate modifications to account for this in Section 2.6 on page 19. Section 2.9.1 on page 25 incorporates this modification.
2. The notation $T_i^{(j)}(t)$ and $I_i^{(j)}(t)$ is used to indicate that the time and inaccuracy is an estimate of clock *i* obtained by clerk or server acting as client *j* as described in this section.

2.2 Obtaining Time Values from Time Providers

A server on a system with a Time Provider obtains time from the TP rather than from other servers. This is also achieved by a Remote Procedure Call. The description of this interface is presented in Chapter 6. An estimate of the TP's clock at time t_1 may be calculated as described in Section 2.1 on page 9.

2.3 Computing a Correct Time

This section describes how a clerk or server acting as client computes a correct time from time values obtained from several servers or from the TP, even if some are faulty. The description is presented as if the time values were obtained from other servers. However, the procedure is the same if the time values are obtained from the TP interface.[3]

Consider a clerk or server acting as client which has obtained *M* time values. For each server S_j with j = 1, 2 . . .M, the clerk or server acting as client has computed $T_j^{(c)}(t_j)$ and $I_j^{(c)}(t_j)$ where t_j corresponds to the instant t_1 in Figure 2-1 on page 10, but for the request sent to S_j.

Calculating a correct time is feasible only if all the time values pertain to the same instant. So the first task is to translate these values to correspond to one synchronisation instant, which is denoted by t_s. Any choice of t_s is adequate, the only requirement being that the value of the clerk or server acting as client's clock $T_c(t_s)$ is known. Least inaccuracy is achieved if t_s is close to the current time. All calculations that the clerk or server acting as client performs are in terms of $T_c(t_s)$ since t_s itself is never known. Translating the S_j time value to t_s is achieved by the following equations:

$$T_j^{(c)}(t_s) = T_j^{(c)}(t_j) + T_c(t_s) - T_c(t_j)$$
$$I_j^{(c)}(t_s) = I_j^{(c)}(t_j) + (T_c(t_s) - T_c(t_j))\delta_c \qquad (2.6)$$ (See footnote[4].)

Note: The inaccuracy of each time value increases to compensate for the maximum possible drift of the clerk's clock over the period $[t_j, t_s]$.

The basis of the calculation is now described. Assume for now that all servers are correct. Therefore, all the *M* time intervals contain UTC. The narrowest correct time that the clerk or server acting as client can compute is simply the intersection of these *M* time intervals. An example is shown in Figure 2-2 on page 14. We point out that this intersection is the smallest interval containing UTC that the clerk or server acting as client could possibly compute from the given information (the *M* time values).

If some servers are faulty, there may be no intersection, or even worse, the intersection may not contain UTC. The actual algorithm[5] embellishes the idea of a simple intersection to accommodate the possibility of faulty servers. The rationale for this is as follows.

The narrowest correct time that the clerk or server acting as client could possibly compute is the intersection of all the correct time intervals; any point on the real line contained in all the correct time values could potentially be the true value of UTC at the instant when the clerk's clock reads $T_c(t_s)$. However, the clerk or server acting as client does not know which servers are correct and which (if any) are faulty.

Let us assume that at most *f* servers are faulty. In this case, any point on the real line contained in at least *M–f* of the time intervals could be a point in all the correct intervals, and hence could be UTC. While this is the smallest set of points guaranteed to contain UTC, it does not necessarily constitute a single interval as shown by the example in Figure 2-3 on page 14. Rather than considering the time to be multiple intervals, the clerk or server acting as client takes the correct time to be the smallest single interval containing all points in at least *M–f* of the intervals.

---

3. The algorithm is from the referenced Stanford University dissertation, slightly modified for the case where not all servers intersect.
4. This inequality is complicated further if you consider that a leap second might occur between t_8 and t_1. We describe the appropriate modifications to account for this in Section 2.6 on page 19. Section 2.9.1 on page 25 incorporates this modification.
5. See the referenced Stanford University dissertation.

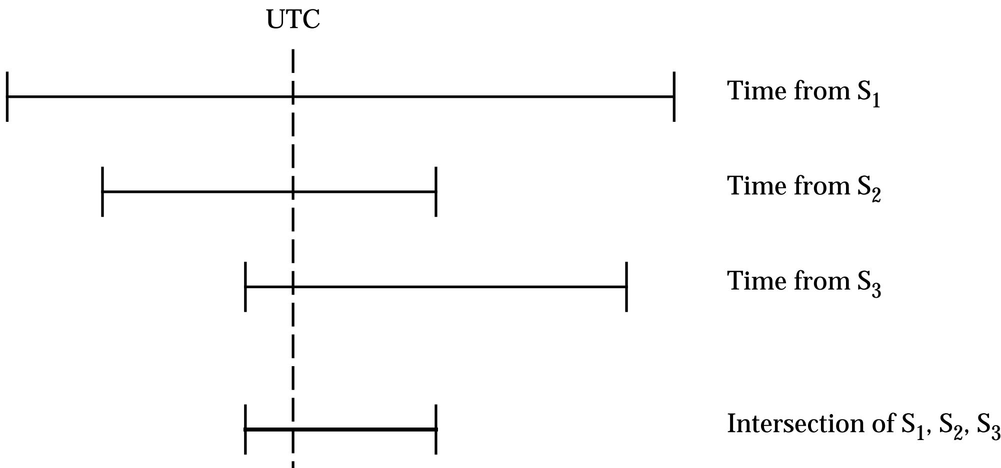

Figure 2-2 Computing the Best Correct Time by Taking an Intersection

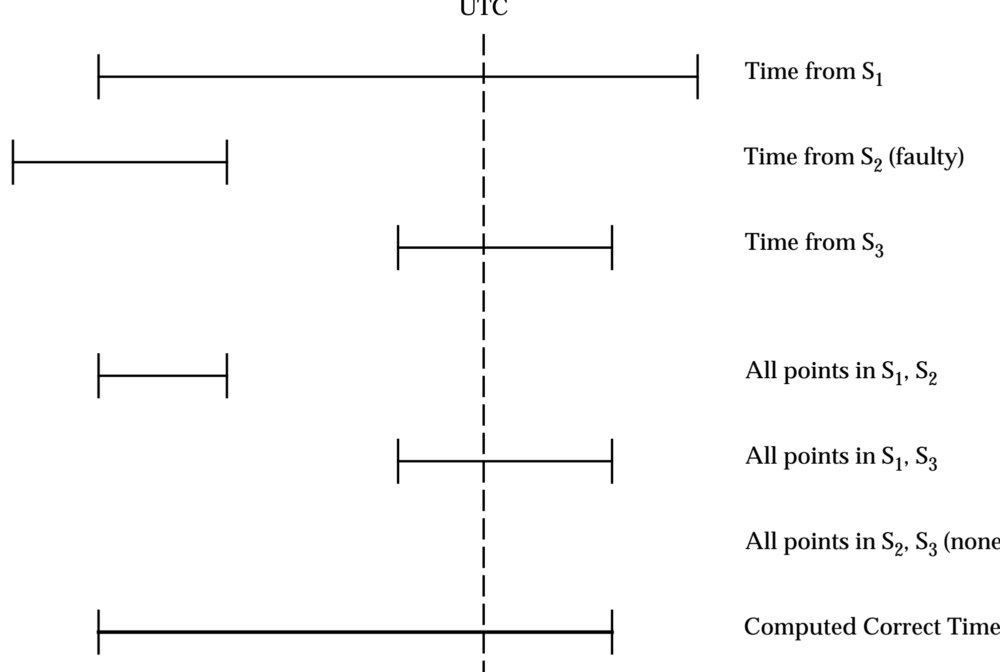

Figure 2-3 Computing the Best Correct Time Assuming One Faulty Server

The complexity of computing the narrowest correct time may at first appear to be high. However, it is simple and fast with a suitable data structure and algorithm. We now describe this computation algorithmically:

1. Arrange the endpoints of the *M* time values into a list. The list is of length 2*M* as each time value contributes two elements to the list: $T_j^{(c)}(t_s) - I_j^{(c)}(t_s)$ and $T_j^{(c)}(t_s) + I_j^{(c)}(t_s)$.
2. Mark each end point to indicate if it is a minimum or maximum endpoint.

3. Sort the list according to the values of the endpoints in ascending order. In the case where two or more endpoints take on the same value, those corresponding to lower bounds must precede those corresponding to upper bounds in the list; that is, if:

   $T_j^{(c)}(t_s) - I_j^{(c)}(t_s) = T_k^{(c)}(t_s) + I_k^{(c)}(t_s)$

   then:

   $T_j^{(c)}(t_s) - I_j^{(c)}(t_s)$ must precede $T_k^{(c)}(t_s) + I_k^{(c)}(t_s)$ in the list.

4. Set the initial value of *f*.

5. Scan the list in ascending order to find the first endpoint that is contained in at least *M–f* intervals. This point corresponds to the minimum value of the correct time interval.

6. If no such point is found, then there are more than *f* faulty servers. Increase *f* by one and go back to step 5. If a minimum value has been found, then continue with the current value of *f*.

7. Scan the list in descending order to find the first endpoint that is contained in at least *M–f* intervals. This point corresponds to the maximum value of the correct time interval.

Computing the minimum and maximum endpoints of the narrowest correct time interval is equivalent to computing a time and inaccuracy. We denote this computed time and inaccuracy by $CT_c(t_s)$ and $CI_c(t_s)$, respectively.

2.4 Adjusting the Clock

Once a clerk or server acting as client or server has computed a correct time $CT_c(t_s)$ and inaccuracy $CI_c(t_s)$, it adjusts its clock in accordance with the computed time. This section describes how that adjustment is made so that the clock never jumps forward or backward.

Recall that, at the synchronisation point t_s, the clerk or server acting as client's clock reads $T_c(t_s)$. Consequently, the clerk or server acting as client must adjust the clock by $CT_c(t_s) - T_c(t_s)$.

Since time always advances, the clock too must always advance; that is, increase monotonically. A clerk or server acting as client should not set its clock backward to adjust a fast clock. When the clock is slow, it is desirable, but not strictly necessary, for the clerk or server acting as client to adjust the clock gradually so that users do not experience a sudden forward jump in the time. To satisfy these requirements, a clerk or server acting as client adjusts a fast clock by slowing it down so that UTC catches up, and a slow clock by speeding it up so that it catches up to UTC.

The procedure by which gradual monotonic adjustments are made depends on the underlying hardware and software constituting the clock. We describe this procedure for one particular realisation of a clock, one common to many computer systems. Other realisations of clocks may use different procedures as long as they guarantee that a non-faulty clock always contains UTC in its interval.

A clock consists of memory containing its current measure of UTC and a hardware timer that periodically interrupts the processor. Normally, the routine servicing these tick interrupts increments the clock's memory by the resolution ρ, causing the contents of the memory to increase at (approximately) the same rate as UTC.

To adjust the clock, the clerk or server acting as client changes the amount by which the clock increments at each tick, increasing the amount if $CT_c(t_s) > T_c(t_s)$; otherwise, decreasing it. We denote the amount by which the nominal tick increment ρ is adjusted by ε. Suppose that the clerk or server acting as client increases the tick increment to $\rho+\varepsilon$ for some number of ticks, say n. If ε is positive, the clock gains $n\varepsilon$ seconds, while if it is negative, the clock loses that amount. So to gain $CT_c(t_s) - T_c(t_s)$, the clerk or server acting as client modifies the tick increment by ε for:

$$N = \lfloor (CT_c(t_s) - T_c(t_s))/\varepsilon \rfloor \qquad (2.7)$$ (See footnote[6].)

ticks. (If $CT_c(t_s) - T_c(t_s) < 0$, then ε is negative and the clock loses rather than gains.)

The value of ε is an implementation constant. However, the specification imposes two restrictions: to ensure that the clock is monotonic, ε must be greater than or equal to $-\rho$; and because the adjustment compensates for drift, the adjustment rate must be greater than the drift, or, $|\varepsilon| > \rho\delta_c$. If, in addition, $|\varepsilon|$ is small compared with ρ, users barely perceive the adjustment, if at all.

---

6. The notation $\lfloor x \rfloor$ denotes the greatest integer less than or equal to *x*.

2.5 Determining the Inaccuracy

Our realisation of the clock does not automatically measure inaccuracy as it does time. Instead, the clerk or server calculates the inaccuracy whenever the clock is read. This section presents the formula for making that calculation.

The inaccuracy of a clock consists of four components:

base inaccuracy
: This is the inaccuracy at the synchronisation point t_s, which is given by $I_c(t_s) = CI_c(t_s) + |CT_c(t_s) - T_c(t_s)|$.

drift increase
: This is the most the base inaccuracy has increased due to drift. At time *t*, the drift increase is $(T_c(t) - T_c(t_s))\delta_c$.

adjustment decrease
: This is the amount by which the inaccuracy is decreased by adjusting the clock. For every tick for which the clock is adjusted by ε, inaccuracy is decreased by $|\varepsilon|$. Assuming adjustment begins at synchronisation point t_s, the adjustment decrease is $|(T_c(t) - T_c(t_s))\varepsilon/(\rho + \varepsilon)|$ but no more than the total of $|N\varepsilon|$.

clock resolution
: The resolution ρ is included in the inaccuracy to reflect the amount by which UTC and the clock diverge between two consecutive ticks. It is scaled to account for drift.

Combining these four components gives the formula for calculating the inaccuracy at any time after t_s as:

$$I_c(t) = CI_c(t_s) + |CT_c(t_s) - T_c(t_s)| + (T_c(t) - T_c(t_s))\delta_c - \min\{(T_c(t) - T_c(t_s))/(\rho + \varepsilon), N\}|\varepsilon| + (1 + \delta_c)\rho \quad (2.8)$$

where *N* is given by equation (2.7).

Note: Implementations can reduce the contribution of resolution to $0.5(1 + \delta_c)\rho$ if the time reported by the clock is increased by $0.5(1 + \delta_c)\rho$ over its actual value.

It may not be possible to schedule the beginning of the adjustment at precisely t_s as required by the formula for adjustment decrease in the preceding list. If this is the case, the formula for calculating the inaccuracy must take into account the number of ticks between the synchronisation instant and the instant at which adjustment actually begins. There is now one way to do this.

Call the instant at which adjustment begins the base time t_b. The clock reads $T_c(t_b)$ at this instant. The inaccuracy at t_b, which we call the base inaccuracy, is the inaccuracy at t_s, increased to account for the drift over the time t_s to t_b. So:

$$I_c(t_b) = CI_c(t_s) + |CT_c(t_s) - T_c(t_s)| + (T_c(t_b) - T_c(t_s))\delta_c$$

(Higher order terms are ignored in this equation.) As the base inaccuracy now includes drift from t_s to t_b, the drift increase need only include drift thereafter. So at time *t*, it is given by $(T_c(t) - T_c(t_b))\delta_c$.

The adjustment decrease is calculated by measuring the number of ticks that have elapsed since adjustment began as long as adjustment began at t_b. So, it is given by $|(T_c(t) - T_c(t_b))\varepsilon/(\rho + \varepsilon)|$ but still no more than the total of $|N\varepsilon|$.

Thus, equation (2.8) can be expressed in more general terms by:

$$I_c(t) = CI_c(t_s) + |CT_c(t_s) - T_c(t_s)| + (T_c(t_b) - T_c(t_s))\delta_c +$$
$$(T_c(t) - T_c(t_b))\delta_c - \min\{(T_c(t) - T_c(t_b))/(\rho + \varepsilon), N\}|\varepsilon| + (1 + \delta_c)\rho$$

2.6 Leap Seconds

The UTC time standard runs at a rate that is almost constant since it is based on ultrastable atomic clocks. However, some users of time signals need time that is referenced to the rotation of the earth. This time standard is known as UT1 and is inferred from astronomical observations. To keep UTC and UT1 approximately equal, occasional corrections of exactly 1 second, called *leap seconds*, are inserted into the UTC time scale when necessary. These corrections are determined and announced by the International Time Bureau (BIH)[7], the international body responsible for the UTC time standard. When they occur, leap seconds cause 1 second to be inserted or removed in the last minute of a specified month.

Leap seconds can be thought of as discontinuities in the UTC timescale; when a negative leap second occurs, UTC effectively jumps forward 1 second; when a positive one occurs, UTC effectively stops for 1 second (allowing UT to catch up). As clocks generally do not mimic these discontinuities in UTC, leap seconds may compromise the correct operation of the Time Service unless we take explicit action to accommodate them.

The following sections describe how the architecture compensates for leap seconds. The architecture does not require any explicit notification of leap seconds. Instead, clerks and servers take appropriate action whenever a leap second might occur. According to BIH rules, leap seconds can be scheduled at the end of any calendar month.[8]

2.6.1 Leap Seconds and Inaccuracy

When a leap second occurs, the interval represented by the time and inaccuracy of a clock may no longer contain UTC; that is, the clock becomes incorrect.

To see why this is the case, consider Figure 2-4 on page 20, which shows the value of a clock as a function of UTC in the vicinity of a negative leap second. The dashed line represents points in the plane where the UTC scale and the clock scale are equal. The two solid lines represent the endpoints of the clock's interval as a function of UTC.

According to our definition of correctness, the line $T_c(t) - I_c(t)$ must always be below the dashed line, which it is in the example, and the line $T_c(t) + I_c(t)$ must always be above the dashed line, which it is not in the example because of the discontinuity in UTC caused by the leap second. Consequently, there is a period when the clock is incorrect. This period depends on the value of the clock at the time of the leap second and the difference between its actual drift and the bound on the drift δ_c. Figure 2-5 on page 21 shows a similar situation for positive leap seconds.

To prevent a clock from becoming incorrect when a negative leap second occurs, the clerk or server must adjust it so that the line $T_c(t) + I_c(t)$ moves above the dashed line for all *t*. Similarly, for a positive leap second, the clerk or server acting as client must adjust the clock so that the line $T_c(t) - I_c(t)$ moves below the dashed line. One suitable adjustment — which is not the most optimal, but is simple, adjusts for both positive and negative leap seconds, and has no effect on correctness if a leap second does not occur — is to increment the inaccuracy by 1 second when $T_c(t) + I_c(t)$ reaches the time of a potential leap second.

---

7. See the referenced CCIR document.
8. See the referenced CCIR document. Preference is given to the months of June and December.

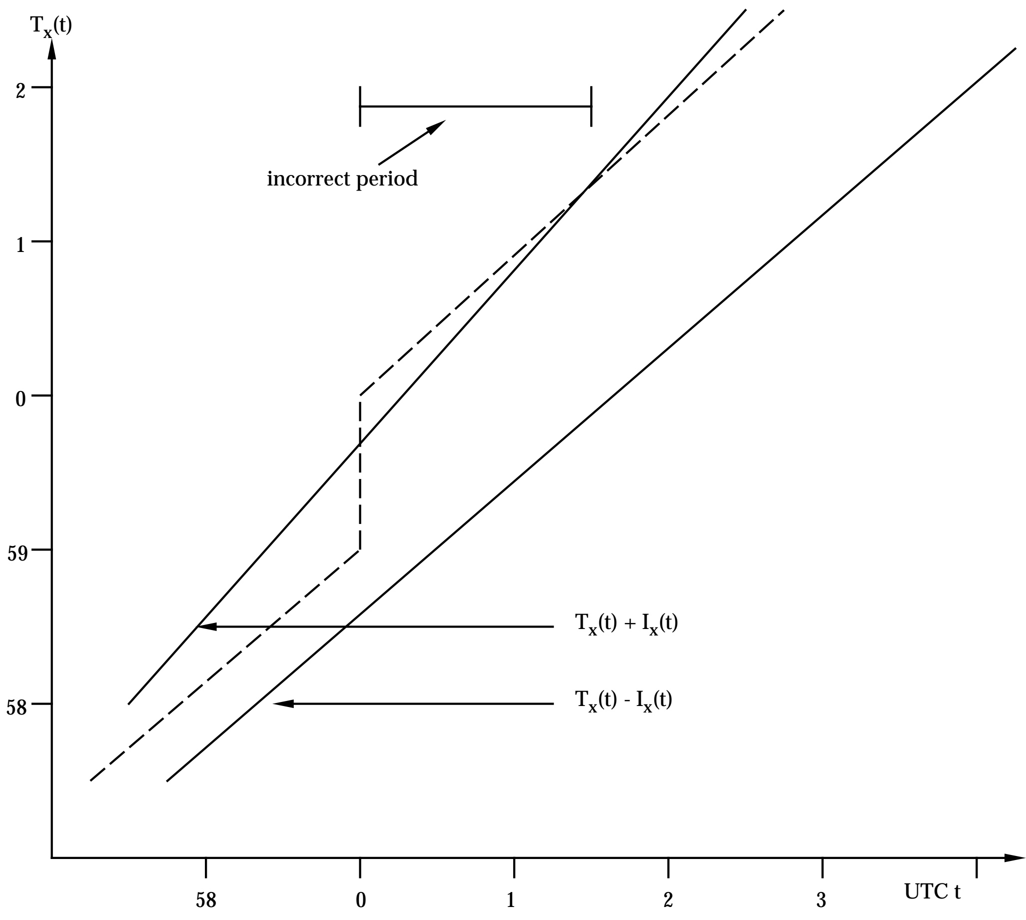

Figure 2-4 How a Negative Leap Second Causes a Clock to Be Incorrect

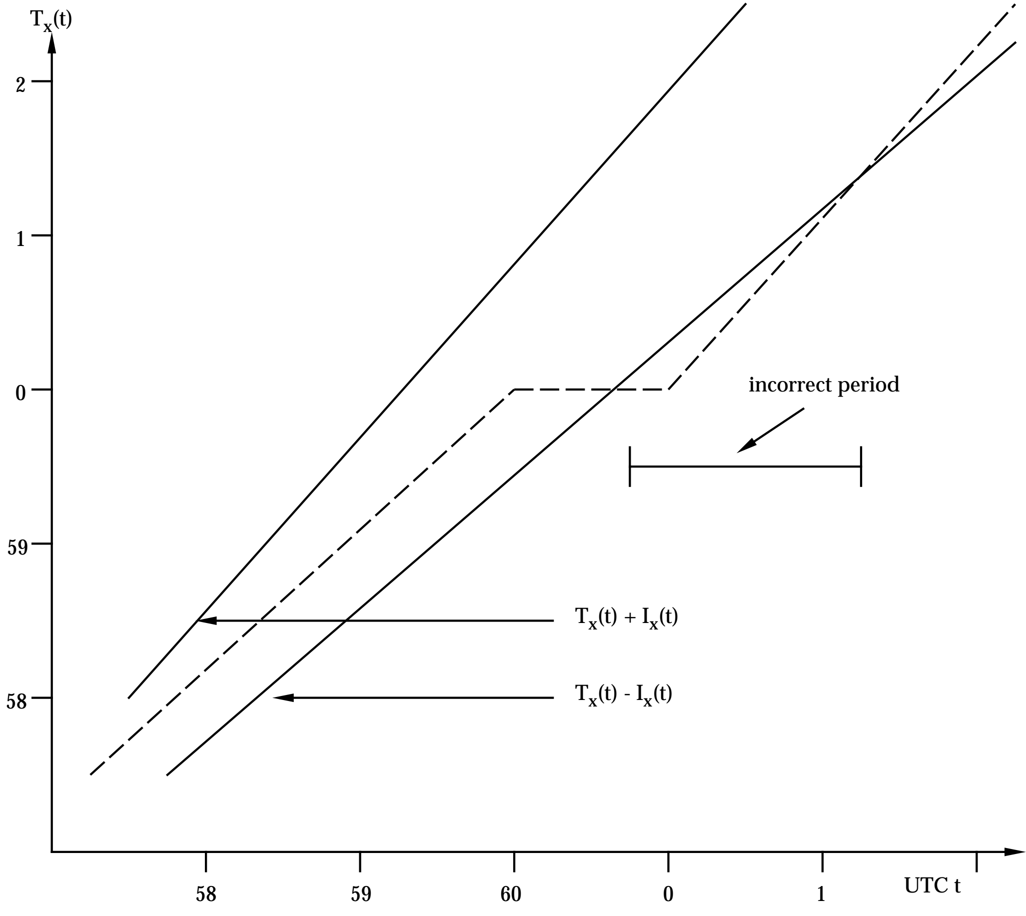

Figure 2-5 How a Positive Leap Second Causes a Clock to Be Incorrect

Incrementing the inaccuracy unnecessarily is of little consequence as the effect is undone at the first subsequent synchronisation.

The operation to compensate for leap seconds is described algorithmically as follows. Denote the current time and inaccuracy of the clock by $T_c(t)$ and $I_c(t)$:

1. Compute the time of the next possible leap second after $T_c(t) + I_c(t)$. Denote this by t_l.
2. Since a leap second can occur at the end of any calendar month, t_l is the beginning of the 60th second of the last minute in the month; that is, 23:59:59.0 on the last day of the month.
3. When $T_c(t) + I_c(t) = t_l$, increment I(t) by 1 second.
4. Go to step 1.

Section 2.9.6 on page 32 shows one possible implementation of this algorithm.

2.6.2 Leap Seconds and Obtaining Time

If a leap second occurs during the synchronisation procedure, the estimate of a server's time and inaccuracy, which is used to calculate the narrowest correct time, may be incorrect.

To see why this is the case, we recall the steps of the synchronisation procedure. At time t_1 in Figure 2-1 on page 10, a clerk or server acting as client sends a time request to a server. The server returns the value of its clock at t_4. This is denoted by $T_s(t_4)$ and $I_s(t_4)$. The clerk or server acting as client then computes a correct time based on an estimate of the server's clock at the synchronisation instant t_s. This estimate is denoted by $T_s^{(c)}(t_s)$ and $I_s^{(c)}(t_s)$. It is given by equation (2.6).

Now consider the case where a leap second occurs between t_1 and t_s. If t_l is before t_4, then the server would have already made the appropriate adjustment to its inaccuracy as described in the previous section. However, if t_l is after t_4, then the server makes the adjustment after returning its time to the clerk or server acting as client. So the clerk or server acting as client must make a similar adjustment to the time it gets back from the server, namely, it must increase the estimate of the server's inaccuracy by 1 second.

As UTC itself is never known, the clerk or server acting as client must make its decision about what to do based on the values of the server's clock and its own. More precisely, if $T_s(t_4) + I_s(t_4) < t_l$, then the server has not made any adjustment for the leap second. So the clerk or server acting as client must make the adjustment on the server's behalf, but only if the leap second is before the synchronisation instant; that is, only if $t_l < T_c(t_s) + I_c(t_s)$.

2.7 Time Zones

Although it in no way affects the determination of the correct time, a client of the Time Service may wish to obtain time zone information with the time. Consequently, this specification requires clerks to provide this information in the form of a numeric *Time Differential Factor* (TDF) and a character *Time Differential Label* (TDL). The TDF is the difference between the local time of the clerk and UTC in units of minutes; the TDL can be used to provide a character representation of the TDF (for example, Pacific Standard Time (PST), Atlantic Standard Time (AST), British Standard Time (BST) or Greenwich Mean Time (GMT) + 1:00). Since there is no standard governing the assignment of labels to TDFs (and, in fact, the commonly used ones are ambiguous), the TDL, if used, has only local significance.

Time zone information can be examined by the management attribute **TDF**.

2.8 Local Faults

Despite periodic synchronisation, an error (either transient or permanent) on a system may cause the clock on that system to become faulty. A faulty clock may be detected at synchronisation by comparing the interval of the clock to the computed interval. If they do not intersect, the local clock is faulty.[9]

The usual mechanism for adjusting the clock and computing the inaccuracy corrects a faulty clock. However, if the amount by which the clock is in error is large (say, several days), it may be undesirable to adjust the clock gradually to correct the error; instead, it may be preferable to set the clock to the computed time.

To facilitate this, there is a management attribute to control whether a faulty clock is set to the correct time or adjusted monotonically. This parameter is called *errorTolerance.* A faulty clock is set rather than monotonically adjusted if the separation between its interval and the computed interval is greater than *errorTolerance.* This is expressed quantitatively by:

$$| CT_c(t_s) - T_c(t_s) | - CI_c(t_s) - I_c(t_s) > \text{errorTolerance}$$

By specifying the error tolerance of a faulty clock in this way, it is easy to accommodate the boundary conditions, namely, never to reset a faulty clock (errorTolerance = ∞ or, to immediately reset a faulty clock, errorTolerance = 0).

9. Albeit unlikely, it may be that most of the servers are faulty but there is no means for the Time Service entity to ascertain that. Consequently, we declare the local clock faulty.

2.9 Primitive Procedures

This document specifies a set of primitive procedures for clerks and servers. The clerk and server specifications in Chapter 4 and Chapter 5 refer to these procedures.

Note: The following specifications include pseudo-code. This is a notational convenience for describing the algorithms involved. This document does not specify the implementation of these procedures except that they must obey the algorithms described.

2.9.1 The EstimateServerTime Procedure

The **EstimateServerTime** procedure performs the calculations to estimate a server's time at the synchronisation instant from the response to a time request. The calculations are in essence equations (2.5) and (2.6) combined with some additional checks to take into account the possibility of a leap second occurring during synchronisation.

The following pseudo-code describes the procedure:

```
IMPORT
   (* Globals *)

   resolution: RelTime;                (* clock resolution *)
   maxDrift: real                      (* bound on clock drift *)
   CalcNextLS(): UTCValue;             (* returns next possible leap
                                          sec after time given in
                                          argument *)

CONST
   100nsInSec = 10000000;              (* # 100ns units in sec *)

PROCEDURE EstimateServerTime  (
   (* Input parameters *)

   Tsend: UTCValue;                    (* time req. sent *)
   Tresp: UTCValue;                    (* time in response *)
   Iresp: Inaccuracy;                  (* inaccuracy in response *)
   procDel: Inaccuracy;                (* server processing delay *)
   Trec: UTCValue;                     (* time response received *)
   Tsync: UTCValue;                    (* clock time at sync instant *)
   Isync: Inaccuracy;                  (* at sync instant *)

   (* Output parameters *)

   VAR Tserv: UTCValue;                (* time result *)
   VAR Iserv: Inaccuracy;              (* inacc result *)
);

BEGIN
   Tserv := Tresp -
           (Trec + resolution - Tsend)*(1 + maxDrift)/2 + procDel/2
            - Tsend + Tsync;

   Iserv := Iresp  +
           (Trec + resolution - Tsend)*(1 + maxDrift)/2 - procDel/2
           + (Tsync - Tsend)*maxDrift;

   (* Maybe make an adjustment for a leap second *)
   IF (Tsync + Isync >= CalcNextLS( Tresp + Iresp ) ) THEN
      Iserv := Iserv + 100nsInSec;
   END (* if *);
END EstimateServerTime;
```

2.9.2 The ComputedTimeMinimum Procedure

The **ComputedTimeMinimum** procedure finds the minimum endpoint of the best correct time interval given M input time values and assuming up to f faulty servers. It returns the actual number of intersecting intervals in the computed time. The return value should be $M - f$, but may be smaller if there happen to be more than f faulty servers.

A similar algorithm can be used to find the maximum endpoint of the best correct time.

The following pseudo-code describes the procedure:

```
TYPE
   EndPointType = (min, max);         (* distinguish min and max points *)
   Element = RECORD
      value: UTCValue;                (* the value of the endpoint *)
      type: EndPointType;             (* min or max *)
   END (* record *);

PROCEDURE ComputedTimeMinimum (
   (* Input parameters *)

   M: integer;                        (* number of time values in list *)
   f: integer;                        (* max faulty ones tolerated *)
   list: array[1..2M] of Element;     (* end points already sorted *)

   (* Output parameter *)

   VAR result: UTCValue;              (* the result *)
   ) : INTEGER;

VAR
   intersectingIntervals: integer;
   (* the number intersecting at current point in list *)
   mostIntersections: integer;
   (* the most intersecting seen so far in list *)
   i: integer;

BEGIN
   (* start with minimum end point in list as minimum computed time *)
   intersectingIntervals := 1;
   mostIntersections := 1;
   i := 1;
   result := list[i].value;

   (* we now loop until either found point in M-f intervals or done
      with list *)
   WHILE (mostIntersections<M-f) AND (i<2M) DO
      i := i + 1;                     (* look at next element *)
      IF list[i].type = min THEN

         (* the number of intersecting intervals is
            one larger than before this point *)
         intersectingIntervals := intersectingIntervals + 1;

         (* if the number intersecting is more than we have seen
            so far, make this end point the new computed minimum *)
         IF intersectingIntervals > mostIntersections THEN
           mostIntersections := intersectingIntervals;
           result := list[i].value;
```

```
            END (* if *);
         ELSE
            (* this element is a max end point so number intersecting
               intervals decreases by 1 *)
            intersectingIntervals := intersectingIntervals - 1;
         END (* if *);
      END (* while *);

      RETURN(mostIntersections);
   END ComputedTimeMinimum;
```

2.9.3 The AdjustClock Procedure

The **AdjustClock** procedure initiates an adjustment of the clock. It also calculates all the variables needed for future inaccuracy calculations. It assumes that the correct time is greater than or equal to the synchronisation instant t_s.

The procedure assumes the existence of a system service to modify the tick length of the clock for some specified number of ticks; the system is expected to return the tick length to the nominal value ρ after the specified number of ticks have elapsed. If this is not the case, the entity has to invoke the **AdjustClkEnd** procedure in a timely manner before the number of ticks to complete adjustment have occurred.

The procedure also assumes that the system provides procedures to start and end critical sections. These procedures must inhibit clock ticks. If system services like these are not available, the example procedure specified in this section must be modified to read the clock before and after the critical section and make an appropriate adjustment to the inaccuracy.

The following pseudo-code describes the procedure:

```
IMPORT
   (* Operating system services *)

   AdjClk();                    (* modifies ticklength for N ticks *)
   ReadClk(): UTCValue;         (* returns current value of clock *)
   BeginAtomic();               (* start a critical section *)
   EndAtomic();                 (* end a critical section *)

   (* Globals *)

   Tbase: UTCValue;             (* "base" time for inacc calc. *)
   Ibase: Inaccuracy;           (* "base" inacc for inacc calc. *)
   resolution: RelTime;         (* clock resolution *)
   maxDrift: real               (* bound on clock drift *)
   tickAdj: RelTime;            (* adjustment to nominal tick.
                                   This is epsilon in text *)
   tickAdjCount: INTEGER;       (* number of ticks for which adjustment
                                   applies. Used by inacc calculation.
                                   This is N in the text. *)
   CalcNextLS(): UTCValue;      (* returns next possible leap second
                                   after time given in argument *)

CONST
   100nsInSec = 10000000;       (* # 100ns units in sec *)

PROCEDURE AdjustClock (
   (* Input parameters *)

   Tsync: UTCValue;             (* value of clock at sync inst. *)
   Isync: Inaccuracy:           (* ditto for inacc *)
   Tcomp: UTCValue;             (* computed time at Tsync *)
   Icomp: Inaccuracy;           (* ditto for inacc *)
   );

VAR
   newTick: Inaccuracy;
   (* the temporary new tick length. we add or subtract this depending
      on whether clock is slow or fast *)

BEGIN
```

```
   (* First determine new tick length *)
   IF Tcomp > Tsync THEN
      (* clock is slow: increase tick value *)
      newTick := resolution + tickAdj;
   ELSE
      (* clock is fast: decrease tick value *)
      newTick := resolution - tickAdj;
   END (* if *);

   (* calc number of ticks for which new tick applies *)
   tickAdjCount := ABS(Tsync - Tcomp) DIV tickAdj;

   (* Now enter a critical section to read clock and set tick
      length to new value. Reading clock gives new base time *)
   BeginAtomic();
      Tbase := ReadClk();
      AdjClk(newTick, tickAdjCount);
   EndAtomic();                 (* clock can tick again *)

   (* Calc new base inaccuracy: new inaccuracy at sync instant plus
      drift between Tsync and Tbase *)
   Ibase := Icomp + ABS(Tcomp - Tsync) + (Tbase - Tsync)*maxDrift;

   (* check whether a leap second possibly occurred between
      Tsync and Tbase *)
   IF (Tbase + Ibase >= CalcNextLS( Tsync + Isync )) THEN
      Ibase := Ibase + 100nsInSec;

   END (* if *);

END AdjustClock;
```

Note: Only the time of the first possible leap second is computed after synchronisation. This implicitly assumes that at most one leap second actually occurs between any two synchronisations. As leap seconds occur approximately once every eighteen months, this seems a reasonable assumption.

2.9.4 The AdjustClkEnd Procedure

Sometimes, such as when a manager issues a command to synchronise, it may be necessary to terminate the adjustment phase prematurely. The **AdjustClkEnd** procedure does this in a way that keeps future inaccuracy calculations correct.

The following pseudo-code describes the procedure:

```
IMPORT
   (* Operating system services *)

   RestoreTick();               (* restores tick length to nominal value *)
   ReadClk(): UTCValue;         (* returns current value of clock *)
   BeginAtomic();               (* start a critical section *)
   EndAtomic();                 (* end a critical section *)

   (* Globals *)

   Tbase: UTCValue;             (* "base" time for inacc calc. *)
   Ibase: Inaccuracy;           (* "base" inacc for inacc calc. *)
   tickAdjCount: INTEGER;       (* number of ticks for which adjustment
                                   applies. Used by inacc calculation.
                                   This is N in the text *)
   CalcNextLS(): UTCValue;      (* returns next possible leap sec
                                   after time given in argument *)
   CalcInaccuracy(): Inaccuracy; (* retuns inacc at time specified by arg *)

PROCEDURE AdjustClkEnd ();

VAR
   T: UTCValue;                 (* temporary variable *)

BEGIN
   (* get current time and abort adjustment *)
   BeginAtomic();
      T := ReadClk();
      RestoreTick();
   EndAtomic();

   (* set base values to current inaccuracy and time for future
      inaccuracy calculations *)
   Ibase := CalcInaccuracy(T);
   Tbase := T;
   tickAdjCount := 0;
   END (* if *);

END AdjustClkEnd;
```

2.9.5 The SetClock Procedure

Sometimes, such as when **errorTolerance** is exceeded or when the clock is first initialised, it may be necessary to set the clock rather than adjust it monotonically. The **SetClock** procedure does this in a way that keeps future inaccuracy calculations correct. This routine assumes that the tick increment is at its nominal value of ρ.

The following pseudo-code describes the procedure:

```
IMPORT
   (* Operating system services *)

   ReadClk(): UTCValue;          (* returns current value of clock *)
   LoadClk();                    (* loads the argument into the clock *)
   BeginAtomic();                (* start a critical section *)
   EndAtomic();                  (* end a critical section *)

   (* Globals *)

   Tbase: UTCValue;              (* "base" time for inacc calc. *)
   Ibase: RelTime;               (* "base" inacc for inacc calc. *)
   CalcNextLS(): UTCValue;       (* returns next possible leap sec
                                    after time given in argument *)
PROCEDURE SetClock (

   (* Input parameters *)

   Tcomp: UTCValue;              (* new time *)
   Icomp: RelTime;               (* new inaccuracy at Tcomp *)
   Tsync: UTCValue;              (* value of clock at which Tcomp applies *)
   );

VAR
   T: UTCValue;                  (* temporary variable *)

BEGIN
   (* read clock, set to Tcomp adjusted for elapsed time since Tsync *)
   BeginAtomic();
      T := ReadClk();
      LoadClk(Tcomp + T - Tsync);
   EndAtomic();

   (* set base values for future inaccuracy calculations *)
   Ibase := Icomp;
   Tbase := Tcomp;
   tickAdjCount := 0;

END SetClock;
```

2.9.6 The CalcInaccuracy Procedure

The **CalcInaccuracy** procedure takes a UTC time argument and returns the inaccuracy of the clock at that time. It is equation (2.9) with a check for leap seconds. This procedure assumes that the current time is greater than or equal to the base time $T_c(t_b)$.

The following pseudo-code describes the procedure:

```
IMPORT
   (* Globals *)

   Tbase: UTCValue;              (* "base" time for inacc calc. *)
   Ibase: Inaccuracy;            (* "base" inacc for inacc calc. *)
   resolution: Inaccuracy;       (* clock resolution *)
   maxDrift: real                (* bound on clock drift *)
   tickAdj: Inaccuracy;          (* adjustment to nominal tick *)
   tickAdjCount: INTEGER;        (* number of ticks for which adjustment
                                    applies. Used by inacc calculation *)
   CalcNextLS(): UTCValue;       (* returns next possible leap sec after
                                    time given in argument *)
CONST
   100nsInSec = 10000000;        (* # 100ns units in sec *)

PROCEDURE CalcInaccuracy (
   (* Input parameter *)

   T: UTCValue;                  (* time. Must be greater than Tbase *)
   ) : Inaccuracy;

VAR
   I: Inaccuracy;                (* the result *)
   n: Integer                    (* ticks since Tbase *)

BEGIN
   (* set to base inacc plus drift since Tbase plus resolution *)
   I := Ibase + (T-Tbase)*maxDrift  + (1+maxDrift)*resolution;

   (* Now subtract out the adjustment. Figure out the number of ticks
      since Tbase. We need to divide by resolution+tickAdj if
      clock was slow and by resolution-tickAdj if clock was fast.
      We don't know which it is so we take the conservative value of
      resolution+tickAdj. This just means it takes a few more ticks
      to subtract out the full adjustment but the end result is the same
      and the clock is always correct. *)
   n := (T-Tbase) DIV (resolution+tickAdjust);
   IF n < tickAdjCount THEN
      (* at least n ticks have occurred since Tbase *)
      I := I - n*tickAdj;
   ELSE
      (* at least tickAdjCount (N in the text) ticks have ticked *)
      I := I - tickAdjCount*tickAdj;
   END (* if *);

   (* maybe add a leap second *)
   IF T+I >= CalcNextLS( Tbase + Ibase ) THEN
      I := I + 100nsInSec;
   END (* if *);

   RETURN(I);
END CalcInaccuracy;
```

Chapter 3

Time Service Configuration

The distributed system that the Time Service targets consists of an arbitrarily large number of autonomous systems, interconnected by a packet-switching communication network. Most of these systems are connected to a Local Area Network (LAN), such as Ethernet, and LANs are interconnected by bridges and routers to form a distributed computing environment cell. This cell could be implemented as a Wide Area Network (WAN) of potentially worldwide extent. Systems may also be connected to the cell by point-to-point links, but this is considered the exception rather than the rule, and the algorithms and protocols are not optimised for this case.

This chapter specifies how systems are configured as servers or clerks and how the Time Service entities find out about the current configuration.

3.1 Configuration

The number of servers from whom a clerk requests the time during each synchronisation is determined by a management attribute called *minServers*. The Time Service must be configured so that there are enough servers to satisfy the needs of every clerk. In addition, it is desirable for servers to be located close to the clerks to whom they provide the time. This minimises communication delay which contributes to inaccuracy.

While small cells can satisfy these criteria with a single set of servers serving all clerks, large cells must be configured with many more servers than required for any one clerk. Consequently, the architecture partitions servers into sets with each set serving a subset of clerks.

To simplify management partitioning, we exploit the assumption that most systems are connected to LANs. Each LAN contains a (possibly empty) set of servers called the *local set*. Normally, there are sufficient servers in a local set to satisfy the needs of all clerks on the LAN of that set. If this is the case, clerks obtain the time from servers in their respective local sets. They discover these servers through RPC service profiles using the algorithm described in Section 3.2 on page 35. This algorithm makes it possible to configure automatically the Time Service local sets.

While this arrangement is well suited for configuring local sets, it suffers two shortcomings:

- If none of the servers in a local set have a TP, an operator must periodically reset the time at f servers in this set.
- Clerks whose local sets contain insufficient servers have no mechanism to discover additional servers.

To overcome these shortcomings, we provide an additional set of servers that are available throughout the cell. We call this set the *global set* and the servers of this set we call *global servers*. A global server is usually a member of some local set, but this is not required.

A clerk accesses global servers only if there are fewer servers in its local set than the number it requires for synchronisation. A server accesses global servers if it does not have a TP and either of the following two conditions holds:

- There are fewer servers in the local set than the number it requires for synchronisation.
- The server is a courier.

Couriers are servers that import time from the global set into the local set. This is useful when none of the servers in the local set have TPs but some servers in the global set do. The mechanism by which a server becomes a courier is described in Section 3.4 on page 37.

Note that global servers do not synchronise with each other explicitly. A global server synchronises with another global server only if the two are in different local sets and the first does not have a TP.

3.2 Local Set Import and Export

Configuration of a local set is by means of RPC profiles. The format and use of RPC profiles is specified in **X/Open DCE: Remote Procedure Call**. Servers and clerks locate members of their local set by periodically performing an RPC import of the Time Service local set interface. The profile search begins with the LAN's profile.[10] The LAN profile is located using the algorithm in **Locating the LAN Profile**. The import operation results in a collection of binding information, which corresponds to servers in the local set.

Servers announce themselves as members of a local set by exporting binding information to a name space entry, and adding an entry to the LAN profile. The first step is achieved by performing an RPC export to the name space entry. The name of the entry is computed by appending the management parameter *serverEntryName* to the system's cell relative name. (See Appendix B for the default server entry name.) The second step is achieved by the server performing an RPC add entry to the LAN profile. When an entity imports bindings, it is possible that insufficient servers are obtained, either because the LAN profile could not be located or insufficient servers are available in the LAN profile. The specifications of clerks and servers in Chapter 4 and Chapter 5 discuss what the entities do in this case.

Locating the LAN Profile

Entities locate the LAN profile by examining profiles starting with the system's initial profile. (See Appendix B for default well-known cell profile and LAN profile names.) Basically, this profile is searched for an entry whose interface UUID matches the LAN service interface. (See Chapter 6.) If an entry for the interface is found, the member associated with this profile entry is used as the LAN profile. If no such entry is found, the profile's default entry is in turn examined. If the member of this entry refers to another profile, the above steps are repeated until one of the following: an entry with the LAN service interface is located, a profile loop is detected, a profile lacking a default entry is found, or a profile is found whose default entry does not refer to another profile. In the last three cases, this algorithm reports failure.

10. There is no requirement that the LAN profile actually represents a physical Local Area Network. It is simply a set of systems that are connected by relatively low latency paths and that share sets of basic services servers.

3.3 Global Set Import and Export

Configuration of the global set is by manual registration. The network manager selects some servers to be members of the global set and instructs the servers to export bindings of the Time Service global set interface to the cell profile. The cell profile is specified by a cell relative well-known name.

Any server can be made a global server simply by instructing it to export the Time Service global set interface. To minimise inaccuracy, however, network managers should give preference to servers that have TPs, but it is generally sufficient for a global server to be on the same LAN as a TP. Also, the global servers should be selected to maximise availability and minimise communication delays.

Security

Each server executes under the identity of the host security principal. The DCE default host principal name is **/.:/***hostname***/self**. All servers are members of a unique security group, specified by the management parameter *groupName*. (See Appendix B for the default value.) Server security principals use generated passwords that are automatically updated on a regular basis.

Entities obtaining the time from servers perform authenticated remote procedure calls and verify that the server satisfying their request is an authentic member of the security group. Servers do not verify authentication when responding to a request. Upon initialisation and at each server cache refresh, servers verify that they are members of the security group. Clients use their host's principal identity for performing the authenticated RPC.

3.4 Couriers

It is likely that some local sets are configured without any servers that have TPs. With this configuration, an operator must periodically mimic a TP to prevent inaccuracies in such a local set from becoming excessively large.

However, there may be some global servers in the cell with close proximity to TPs from which the servers in a local set could obtain accurate time. The architecture provides a mechanism for this with a design that limits the load placed on global servers and is easy to manage at the expense of reduced fault tolerance. With this mechanism, only those servers designated as couriers synchronise with global servers rather than all servers without TPs.

A server becomes a courier by a combination of an election mechanism and an attribute called *courierRole* which takes one of the three values: courier, non-courier or backup-courier. A server with *courierRole* set to non-courier never becomes a courier; one with *courierRole* set to courier is always a courier; a server with *courierRole* set to backup-courier is, in general, not a courier but becomes one if:

1. There are no servers in the local set with *courierRole* set to courier.
2. It is the server whose security UUID (the UUID associated with the server's security principal) precedes all others with *courierRole* set to backup-courier.

Servers exchange their *courierRole* values in time request RPCs. Servers with *courierRole* set to *backup-courier* must redetermine whether or not they are couriers whenever they add an entry to, or remove one from, their lists of local servers.

Chapter 4

Time Service Clerk Specification

This chapter describes aspects of the Time Service architecture specifically applicable to clerk entities.

4.1 Initialising the System Clock

On startup, a clerk must initialise the system clock. The procedure is the same for both clerk and server Time Service entities. To facilitate initialisation, it is desirable but not required for Time Service entities to store their base inaccuracy $I_c(t_b)$ and the time of the clock corresponding to this base, $T_c(t_b)$, in non-volatile memory after each synchronisation.

The initialisation procedure depends on whether the system knows its base inaccuracy at start up and whether the clock continues to run in the absence of the Time Service entity. The sequence is as follows:

1. The base time and base inaccuracy are restored, and the time of the first leap second after the base time is computed if the following conditions are met:

   — The entity has saved a base time and inaccuracy.

   — The clock has run uninterrupted since the base time and inaccuracy were last saved.

   — The upper bound on clock drift is independent of whether the Time Service entity is present.

   Otherwise, the base time is set to the current time (determined arbitrarily) and the base inaccuracy is set to infinity.

2. The entity schedules a synchronisation to occur immediately and asynchronously.

3. If the base inaccuracy before synchronisation is infinite, then after synchronisation, the entity sets the clock (rather than adjusts it monotonically); otherwise, the entity adjusts the clock in the usual way.

4.2 Synchronisation

This section describes a model algorithm for synchronisation. It includes importing local server and global server bindings, if necessary.

Note: Other algorithms may be used only if they determine a correct time that is at least as accurate, do not reduce fault tolerance, and consume no more network resources under all conditions.

1. Invoke the **AdjustClkEnd** procedure to abort any ongoing clock adjustment. This ensures that the rate of the clock is within δ_c of the rate of UTC. (If a synchronisation is initiated by management action, a clock adjustment may be in progress. This step ensures that the procedure is properly terminated in this case.)

2. Obtain the time from the maximum of *minServers* and *minLocalServers* servers. The *minServers* management attribute specifies the number of servers required for synchronisation to occur. The *minLocalServers* architectural constant specifies the minimum number of local servers to use for synchronisation if they are available. This is done according to the following substeps:

   a. Select a server at random from the list of local servers. The selected server must be one not yet selected for this synchronisation. If there are none, go to substep 2.e. Otherwise, continue with the next substep.

   b. Query the server until a response is received but no more than *repetitions* times. The next step depends on the status of the RPC call. If a valid response is received, continue with the next substep. If an invalid response is received, remove the server from the list of servers and go to substep 2.a. A response is invalid if the server time representation version does not match that of the client; the server is not a member of the security group given by the *groupName* management parameter; or the RPC fails for reasons other than a timeout. If no response is received, (the RPCs timeout) remove the server from the list of servers and go to substep 2.a.

   c. If *minLocalServers* servers have not been successfully queried and there are still servers in the list of local servers that have not been selected for this synchronisation, then go to substep 2.a. Otherwise, continue with the next step.

   d. If *minServers* servers have been successfully queried, the requisite number of time values have been obtained. Go to step 3. Otherwise, go to substep 2.a.

   e. There are less than *minServer* servers in the local list. If the clerk has not yet performed an RPC import operation for the local set continue with the next substep. Otherwise, go to substep 2.g.

   f. Perform an RPC import (using the algorithm specified in Section 3.2 on page 35). Add any newly discovered servers to the list of local servers, then go to substep 2.a.

   g. All local servers have been tried and an RPC import of the local set profile has been done. Try the global servers. Select a global server at random from the list of global servers. The selected server must be one not yet selected for this synchronisation. If there are none, go to substep 2.j. Otherwise, continue with the next substep.

   h. Query the global server. If a valid response is received, continue with the next substep. If an invalid response is received, remove the server from the list of servers and go to substep 2g. If no response is received, remove the server from the list of servers and go to substep 2.g.

   i. If a total of *minServers* servers have been successfully queried, the requisite number of time values have been obtained. Go to step 3. Otherwise, go to substep 2.g.

   j. If the clerk has not yet performed an RPC import operation for the global set, continue with the next substep. Otherwise, go to substep 2.l.

   k. Perform an RPC import (using the algorithm in Section 3.2 on page 35). Add any newly discovered servers to the list of global servers.

   l. There are less than *minServers* servers (local and global). Abort the synchronisation and go to step 8

3. At least *minServers* time values have been obtained. Read the clock to determine the synchronisation instant $T_c(t_s)$ and $I_c(t_s)$.
4. Translate each server's time to the synchronisation instant using the **EstimateServerTime** procedure.
5. Compute the best correct time from the time values obtained from the servers. Use a value of $f = \lfloor minServers/2 \rfloor$. This protects the clerk from up to $\lfloor minServers/2 \rfloor$ faulty servers. Use the **ComputedTimeMinimum** procedure.
6. Compare the time interval of the local clock, which is given by $T_c(t_s)$ and $I_c(t_s)$ against the computed interval. If they intersect, initiate clock adjustment and go to step 8. Otherwise, go to the next step.
7. The local clock is faulty. If the error tolerance is exceeded, set the clock to the computed time. Otherwise, initiate clock adjustment.
8. Schedule the next synchronisation as described in Section 4.3 on page 42.

Note: Although, in this description of the procedure, clerks query servers one at a time, they may query servers in parallel if they desire. However, the performance effects of returning responses to earlier requests competing for processor resources with subsequent requests may add to inaccuracy.

4.3 Determining the Next Synchronisation

A clerk determines when to synchronise its clock by attempting to bound its inaccuracy. The desired bound on the inaccuracy is specified by the management attribute *maxInacc*.

From the computed time and inaccuracy $CT_c(t_s)$ and $CI_c(t_s)$, the clerk can calculate the time at which its inaccuracy will reach the value of *maxInacc*. This is given by:

$$T = CT_c(t_s) + (maxInacc - CI_c(t_s))/\delta_c$$

To keep $I_c(t) < maxInacc$, the clerk must synchronise before its clock reads this time.

Note that the Time Service does not guarantee any bound on $CI_c(t_s)$. Consequently, it is possible that $maxInacc - CI_c(t_s)$ is small or even negative. To prevent a clerk from synchronising continuously, we require that the time between synchronisations be more than a minimum value which is specified by the management attribute *syncHold*.

Unfortunately there are likely conditions where, using the approach described in the previous paragraph, all clerks will choose the same instant at which to synchronise, causing bursty loads on the network and servers. To avoid these, some randomness is introduced into the times at which clerks synchronise.

The following steps describe how to schedule the next synchronisation based on the computed time and the two parameters *maxInacc* and *syncHold*.

1. Compute the time for the inaccuracy to grow to *maxInacc*. This is given by:

   $$D = (maxInacc - CI_c(t_s))/\delta_c$$

2. If D < syncHold, go to step 4. Otherwise, continue with the next step.

3. Draw a random number, *R*, uniformly distributed over the interval:

   $[D/2, D]$

   Go to step 5.

4. Draw a random number, *R*, uniformly distributed over:

   $[3(syncHold)/4, 5(syncHold)/4]$

5. Schedule the next synchronisation so that it can complete before the clock reads:

   $$CT_c(t_s) + R$$

4.4 Maintaining the Server Lists

To learn of new local servers, clerks must periodically perform an RPC import of the local set. To learn of new global servers, those clerks using global servers must periodically perform RPC imports of the global set.

The mechanism by which a clerk is forced to perform these imports is for it periodically to flush its entire lists of global and local servers. As can be seen from the synchronisation procedure in Section 4.2 on page 40, this causes the clerk to import the local set and, if necessary, the global set the next time it synchronises.

Flushing is done periodically. The period between flushes is specified by the management attribute *cacheRefresh*.

Chapter 5

Time Service Server Specification

This chapter describes aspects of the Time Service architecture specifically applicable to server entities.

5.1 Initialisation

When a server entity starts up it initialises the system clock using the same procedure as a clerk entity, as specified in Section 4.1 on page 39. The server epoch number is initialised according to the procedure given in Section 5.2 on page 46. The server also exports its bindings to the local set using the algorithm specified in Section 3.2 on page 35.

5.2 Epochs

Although it is unlikely, we cannot exclude the possibility that the time of the Time Service is incorrect even if the hardware and software of the servers are not faulty. This may be due to some catastrophic failure or to an operator entering the incorrect time to an uninitialised service.

If most servers in a local set have TPs, that local set recovers without human intervention. However, if most servers in a local set do not have TPs, the incorrect time persists.

To correct this situation, a manager would have to shut down the servers and restart them with a new, correct time. A manager must shut down enough servers so that there are more with correct time than with incorrect time. However, the procedure could not be carried out if some faulty remote servers fail to shut down. To recover from this condition, the specification includes the notion of an epoch number that allows groups of servers to be decoupled, making it possible to introduce a new value of time into the distributed system by introducing it into one of the decoupled groups.

Each server is assigned an epoch number. This number is denoted by the attribute *epochNumber* in this specification. Servers supply their epoch numbers when responding to requests for the time to other servers. When a server synchronises, it ignores servers with epoch numbers different from its own, thereby decoupling it from servers of a different epoch. (See Section 4.2 on page 40.)

To introduce a new time into the distributed system, the manager changes the epoch number at a single server and supplies it with the correct time. Then, the manager migrates the other servers to the new (correct) time by changing their epoch numbers, one at a time, to correspond to that of the new epoch.

Servers that have been given a new epoch number drop the minimum number of servers requirement from their synchronisation algorithm until after their first synchronisation in the new epoch. If the minservers requirement were not temporarily suspended, the second and subsequent servers changed to the new epoch would never find enough servers in the new epoch to satisfy the minservers requirement for synchronisation. When a server's epoch is updated and no new time is set, the server sets its inaccuracy to unknown (infinity). Servers that have been given a new epoch number refrain from responding to any time requests until after they have synchronised in the new epoch.

Initialising the Epoch

The epoch number is saved in nonvolatile memory, if available, after every epoch change. During initialisation, the epoch is either set to the value stored in the non-volatile memory or, if there is no non-volatile memory, in some system-dependent manner. If neither of these options is feasible, the server requests a time from all servers and initialises to the epoch used most often by the other servers.

5.3 Synchronisation

This section describes the steps involved in synchronisation. A server synchronises with the TP if it is available. Otherwise, the server synchronises with other servers.

1. Invoke the **AdjustClkEnd** procedure to abort any ongoing clock adjustment. This ensures that the rate of the clock is within δ_c of the rate of UTC. (If a synchronisation is initiated by management action, the clock adjustment procedure may be in progress. This step ensures that the procedure is properly terminated in this case.)
2. Call the TP interface *ContactProvider*( ) to obtain the provider control message. Next, call the TP interface *ServerRequestProviderTime*( ) to obtain the time. If either call fails, synchronise with other servers using the procedure described in Section 5.3.2. Otherwise, synchronise with the time obtained from the TP as described in Section 5.3.1.

5.3.1 Synchronising with a TP

This section presents the remainder of the synchronisation procedure for the case that a response is received from the TP.

1. Read the clock to determine the synchronisation instant $T_c(t_s)$ and $I_c(t_s)$.
2. Translate each time value in the set returned by the TP to the synchronisation instant.
3. Compute the best correct time from these times. Also include the time of the server's clock (that is, time $T_c(t_s)$ and inaccuracy $I_c(t_s)$). Use the **ComputedTimeMinimum** procedure with $f = 0$.
4. Compare the time interval of the local clock, which is given by $T_c(t_s)$ and $I_c(t_s)$, against the computed interval. If they intersect, initiate clock adjustment and go to step 6. Otherwise, go to the next step.
5. The local clock is faulty. If the error tolerance is exceeded, set the clock to the computed time. Otherwise, initiate clock adjustment.
6. Schedule the next synchronisation as described in Section 5.4 on page 49.

5.3.2 Synchronising with Other Servers

This section presents the remainder of the synchronisation procedure for the case that no response is received from the TP.

1. Obtain the time from local and global servers in the same way that clerks do except that:
   a. All servers in the local set are queried.
   b. For synchronisation to proceed, at least *minServers* – 1 (rather than *minServers*) other servers must be successfully queried. The server's own time is included to make up a total of *minServers* time values.
   c. If the server is a courier, it must obtain the time from at least one global server, selected at random. This may already have been done in the process of obtaining *minServers* – 1 time values.
   d. The server rejects any time response messages with epoch numbers different from its own.

   If, after trying all local and global servers, less than *minServers* – 1 servers are successfully queried, go to step 8. Otherwise, continue with the next step.

2. Read the clock to determine the synchronisation instant $T_c(t_s)$ and $I_c(t_s)$.
3. Translate each server's time to the synchronisation instant.
4. Compute the best correct time using the **ComputedTimeMinimum** procedure with $f = 0$. Include the server's time in the computation. However, when a server's epoch has been updated and its local clock has unknown (infinite) inaccuracy, the server need not include the infinite time interval corresponding to its local clock in the synchronisation algorithm since this does not affect the results.
5. Check for faulty servers. If all servers' time values do not intersect, go to the next step. Otherwise, initiate clock adjustment and go to step 8.
6. A faulty server exists. First check if it is the local server: compare the time interval of the local clock, which is given by $T_c(t_s)$ and $I_c(t_s)$, against the computed interval. If they intersect, initiate clock adjustment and go to step 8. Otherwise, go to the next step.
7. The local clock is faulty. If the error tolerance is exceeded, set the clock to the computed time. Otherwise, initiate clock adjustment.
8. Schedule the next synchronisation as described in the following section.

5.4 Determining the Next Synchronisation

If the server synchronised with the TP, it schedules the next synchronisation for the time specified by the TP (the **nextPoll** field of the **TPctlMsg** returned by a call to *ContactProvider*( ) of the Time Provider interface).

If the server synchronised with other servers, or if it aborted the synchronisation because there were too few servers, it determines the next synchronisation in the same way that clerks do as specified in Section 4.3 on page 42.

5.5 Checking For Faulty Servers

To detect and report faulty servers in a timely fashion, servers must periodically obtain time from all other servers in the local set and check that their intervals intersect.

The procedure is the one used for synchronising with other servers, as described in Section 5.3.2 on page 47, with the following changes:

- No global servers are queried.
- The procedure is not aborted if less than *minServers* – 1 are queried.
- The step to adjust (or set) the clock is not carried out.
- The step to schedule the next synchronisation is not carried out.

Checking for faulty servers is initiated by a periodic timer whose average period is specified by the management attribute *checkInt*. After initialisation and after completing the checking procedure, the timer is set by drawing a random number in the range [3(*checkInt*)/4, 5(*checkInt*)/4].

Note: Since checking is also done when a server synchronises with other servers, the timer is restarted (with a value that is a random number in the range [3(*checkInt*)/4, 5(*checkInt*)/4]) after successfully synchronising with other servers. So, explicit checking occurs if the server has a TP. (Explicit checking also occurs if the *checkInt* interval is smaller than the time between synchronisations. While this is not the normal case, it can happen because *syncHold* can be set independently.)

5.6 Maintaining the Server Lists

To learn of new local servers, servers must periodically perform an RPC import of the local set. To learn of new global servers, those servers using global servers must periodically perform RPC imports of the global set.

Servers must maintain their lists in the same way as clerks do: a server periodically flushes its entire list servers. Flushing is accomplished by a periodic timer whose period is specified by the value cacheRefresh.

Chapter 6

Time Service IDL Declarations

This chapter specifies the RPC interfaces used by DTS clerks and servers to get the time from other servers and by servers to get the time from time providers.

The RPC interfaces are specified in IDL. Part of the information in these IDL declarations is symbolic and need not be preserved identically. For example, the names of procedures and parameters are a local matter. The information that must be identical for a conforming implementation is the:

- interface identifier, composed of UUID and version
- order of procedures within the interface definition
- order and number of parameters in procedure signatures
- types of parameters and procedures
- attributes of parameters and procedures.

6.1 Data Types and Ranges

6.1.1 The utc Structure

The DTS RPC interfaces use the opaque **utc** data type to pass time values. This is declared as:

```
interface utctypes
{
  import "dce/nbase.idl";

typedef struct utc
{
        byte char_array[16];
} utc_t;
}
```

6.1.2 Parameter Ranges

Several DTS RPC interfaces declare integer parameters. These may have the following ranges of values:

courierRole	0	Courier
	1	Non-courier
	2	Backup Courier
epoch	0 to 255	
processingDelay	0 to $2^{32} - 1$	

6.2 Local Set Time Service Interface

The local set time service interface consists of two remote procedure calls that DTS clerk and server entities can use to request time from other local servers.

```
[uuid (019ee420-682d-11c9-a607-08002b0dea7a),
    version(1)
]

interface time_service
{
 import "dce/utctypes.idl";

void ClerkRequestTime
        (
        [in]   handle_t             bind_h,
        [out]  utc_t               *timeRequest,
        [out]  unsigned long       *processingDelay,
        [out]  error_status_t      *comStatus
        );

void ServerRequestTime
        (
        [in]   handle_t             bind_h,
        [out]  utc_t               *timeRequest,
        [out]  unsigned long       *processingDelay,
        [out]  long int            *epoch,
        [out]  long int            *courierRole,
        [out]  error_status_t      *comStatus
        );
}
```

6.2.1 ClerkRequestTime

```
void ClerkRequestTime
        (
        [in]   handle_t             bind_h,
        [out]  utc_t               *timeRequest,
        [out]  unsigned long       *processingDelay,
        [out]  error_status_t      *comStatus
        );
```

DTS clerk entities use **ClerkRequestTime** to request time from a DTS server. The server returns a timestamp with its current UTC time. If the implementation can provide a measure of the server's processing delay, the call returns the delay; otherwise, the delay returned is zero.

ClerkRequestTime parameters are:

bind_h RPC binding handle.

timeRequest Timestamp containing the server's current UTC time.

processingDelay Server's processing delay in nanoseconds.

comStatus Communications error status returned by the call.

6.2.2 ServerRequestTime

```
void ServerRequestTime
        (
        [in]   handle_t              bind_h,
        [out]  utc_t                *timeRequest,
        [out]  unsigned long        *processingDelay,
        [out]  long int             *epoch,
        [out]  long int             *courierRole,
        [out]  error_status_t       *comStatus
        );
```

DTS server entities use **ServerRequestTime** to request time from another DTS server. The server returns a timestamp with its current UTC time. If the implementation can provide a measure of the server's processing delay, the call returns the delay; otherwise, the delay returned is zero. The server also returns its epoch number and an indication of its courier role.

ServerRequestTime parameters are:

bind_h RPC binding handle.

timeRequest Timestamp containing the server's current UTC time.

processingDelay Server's processing delay in nanoseconds.

epoch Server's epoch number.

courierRole Server's courier role.

comStatus Communications error status returned by the call.

6.3 Global Set Time Service Interface

The global set time service interface consists of two remote procedure calls that DTS clerk and server entities can use to request time from global servers.

```
[uuid (17579714-82c9-11c9-8a59-08002b0dc035),
    version(1)
]

interface gbl_time_service
{
 import "dce/utctypes.idl";

void ClerkRequestGlobalTime
        (
        [in]   handle_t              bind_h,
        [out]  utc_t                *timeRequest,
        [out]  unsigned long        *processingDelay,
        [out]  error_status_t       *comStatus
        );

void ServerRequestGlobalTime
        (
        [in]   handle_t              bind_h,
        [out]  utc_t                *timeRequest,
        [out]  unsigned long        *processingDelay,
        [out]  long int             *epoch,
        [out]  error_status_t       *comStatus
        );
}
```

6.3.1 ClerkRequestGlobalTime

```
void ClerkRequestGlobalTime
        (
        [in]   handle_t              bind_h,
        [out]  utc_t                *timeRequest,
        [out]  unsigned long        *processingDelay,
        [out]  error_status_t       *comStatus
        );
```

DTS clerk entities use **ClerkRequestGlobalTime** to request time from a DTS global server. The global server returns a timestamp with its current UTC time. If the implementation can provide a measure of the global server's processing delay, the call returns the delay; otherwise, the delay returned is zero.

ClerkRequestGlobalTime parameters are:

bind_h RPC binding handle.

timeRequest Timestamp containing the global server's current UTC time.

processingDelay Global server's processing delay in nanoseconds.

comStatus Communication error status returned by the call.

6.3.2 ServerRequestGlobalTime

```
void ServerRequestGlobalTime
        (
        [in]   handle_t             bind_h,
        [out]  utc_t               *timeRequest,
        [out]  unsigned long       *processingDelay,
        [out]  long int            *epoch,
        [out]  error_status_t      *comStatus
        );
```

DTS server entities use **ServerRequestGlobalTime** to request time from a DTS global server. The global server returns a timestamp with its current UTC time. If the implementation can provide a measure of the global server's processing delay, the call returns the delay; otherwise the returned delay is zero. The global server also returns its epoch number.

ServerRequestGlobalTime parameters are:

bind_h PRC binding handle.

timeRequest Timestamp containing the global server's current UTC time.

processingDelay Global server's processing delay in nanoseconds.

epoch Global server's epoch number.

comStatus Communications error status returned by the call.

6.4 Time Provider Interface

The Time Provider Interface consists of two RPC interfaces that a server can use to get time from Time Providers (TPs).

```
[uuid (bfca1238-628a-11c9-a073-08002b0dea7a),
    version(1)
]

interface time_provider
{
 import "dce/utctypes.idl";

const long K_MIN_TIMESTAMPS    = 1;
const long K_MAX_TIMESTAMPS    = 6;

const long K_TPI_FAILURE       = 0;
const long K_TPI_SUCCESS       = 1;

typedef struct TimeResponseType
{
   utc_t beforeTime;
   utc_t TPtime;
   utc_t afterTime;
} TimeResponseType;

typedef struct TPctlMsg
{
    unsigned long        status;
    unsigned long        nextPoll;
    unsigned long        timeout;
    unsigned long        noClockSet;
} TPctlMsg;

typedef struct TPtimeMsg
{
    unsigned long        status;
    unsigned long        timeStampCount;
    TimeResponseType     timeStampList[K_MAX_TIMESTAMPS];

} TPtimeMsg;

void ContactProvider
        (
        [in]   handle_t            bind_h,
        [out]  TPctlMsg           *ctrlRespMsg,
        [out]  error_status_t     *comStatus
        );

void ServerRequestProviderTime
        (
        [in]   handle_t            bind_h,
        [out]  TPtimeMsg          *timesRspMsg,
        [out]  error_status_t     *comStatus
        );
}
```

6.4.1 Data Types

TimeResponseType

```
typedef struct TimeResponseType
{
   utc_t beforeTime;
   utc_t TPtime;
   utc_t afterTime;
} TimeResponseType;
```

The **TimeResponseType** contains one reading of the TP wrapped in the time stamps of the local clock.

TPctlMsg

```
typedef struct TPctlMsg
{
    unsigned long       status;
    unsigned long       nextPoll;
    unsigned long       timeout;
    unsigned long       noClockSet;
} TPctlMsg;
```

The **TPctlMsg** data type holds a time provider control message. This is the response to a **ContactProvider** call. The fields are:

status The status of the operation. **K_TPI_SUCCESS** indicates success, **K_TPI_FAILURE** indicates failure.

nextPoll Tells the client how many seconds to wait before the next call to the TP.

timeout Tells the client how long to wait for a time response from the TP.

noClockSet Tells the client whether or not it is allowed alter the system clock after a synchronisation with the TP. The values of **noClockSet** are:

0 Clock set allowed

1 Clock set not allowed

TPtimeMsg

```
typedef struct TPtimeMsg
{
    unsigned long        status;
    unsigned long        timeStampCount;
    TimeResponseType     timeStampList[K_MAX_TIMESTAMPS];
} TPtimeMsg;
```

The **TPtimeMsg** data type holds a time provider time stamp message. This is a response to a **ServerRequestProviderTime** call. The fields are:

status The status of the operation. This may be **K_TPI_SUCCESS** or **K_TPI_FAILURE**.

timeStampCount Holds the number of time stamps being returned in this message. The range is from K_MIN_TIMESTAMPS to K_MAX_TIMESTAMPS.

timeStampList The array of time stamps being returned from the TP.

6.4.2 ContactProvider

```
void ContactProvider
        (
        [in]   handle_t            bind_h,
        [out]  TPctlMsg           *ctrlRespMsg,
        [out]  error_status_t     *comStatus
        );
```

DTS servers use **ContactProvider** to send an initial contact message to a Time Provider. The Time Provider responds with a control message.

ContactProvider parameters are:

bind_h RPC binding handle.

ctrlRespMsg Time Provider control message returned in response to time service request.

comStatus Error status returned by call.

6.4.3 ServerRequestProviderTime

```
void ServerRequestProviderTime
        (
        [in]   handle_t            bind_h,
        [out]  TPtimeMsg          *timesRspMsg,
        [out]  error_status_t     *comStatus
        );
```

DTS servers use **ServerRequestProviderTime** to request times from a Time Provider. The TP server responds with a time stamp message containing an array of timestamps obtained by querying the time provider hardware that it polls.

ServerRequestProviderTime parameters are:

bind_h RPC binding handle.

timesRespMsg Timestamp message returned in response to a Time Service request.

comStatus Error status returned by the call.

X/Open CAE Specification

Part 2

Time API

X/Open Company Ltd.

Chapter 7

Time API

This part specifies the application programmer's interface (API) to the Time Service. The API contains routines that applications can use to obtain the time, convert times from one representation to another, do arithmetic on time values, compare time values and manipulate time spans.

7.1 Timestamps

The Time Service API routines perform operations on opaque *binary timestamps*. A binary timestamp presents the time service's internal time representation as an opaque data type, declared as **utc_t**, that applications are not meant to interpret or modify directly.

Several API routines convert between binary timestamps and data representations that programs can interpret and modify. Portable applications must use the Time Service API for all operations that interpret, manipulate or modify binary timestamps.

As described in Chapter 1 and Appendix A, the Time Service's representation of time includes three components (along with other time information):

Time — UTC time in 100 nanosecond units.

Inaccuracy — Inaccuracy of the absolute time in 100 nanosecond units.

TDF — The difference between UTC and local time, in minutes east of the Greenwich meridian.

These components are represented abstractly in the Time Service API specifications as the time, inaccuracy and TDF *components* of a binary timestamp. To interpret or manipulate these components, applications must use Time Service API routines that convert between binary timestamps and non-opaque representations of time. Several routines provide access to concrete, non-opaque representations of these components.

The times represented by binary timestamps are of two types: absolute and relative, represented abstractly in the Time Service API as *absolute binary timestamps* and *relative binary timestamps*.

Note: The concrete representation of a binary timestamp is always **utc_t**. Therefore, there is no concrete difference between absolute and relative timestamp representation in the API specifications. However, the API routines typically specify timestamp parameters as either absolute or relative. In such cases, passing a parameter that represents the wrong kind of time either results in an error being returned or produces unspecified results.

Absolute time is represented by all three time components and refers to a time (with inaccuracy) adjusted to specific timezone. Relative time consists of a time period and inaccuracy only, and is used to measure time intervals.[11]

7.1.1 The utc_t Type

The opaque **utc_t** type is declared by including the file <**dce/utc.h**>. Portable applications must not interpret or modify the contents of any **utc_t** parameter. Several API routines specify pointers to **utc_t** types as output arguments. In every case, the application is responsible for allocating and freeing the pointed-to object.

11. There is no specific representation at the API level of *simple relative time*, as defined in Appendix A.

7.2 Non-opaque Time Representations

Several Time Service API routines convert between binary timestamps and non-opaque representations of time that may be interpreted and modified by programs: ASCII string representation, the **tm** structure, the **timespec** structure and the **reltimespec** structure. The following sections specify these representations.

7.2.1 Character Representations of Time

Character representations of time are provided for human use. The Time Service API includes routines to convert between binary timestamps and these display forms. The routines with names of the form **utc_mkasc...** accept character representations of time as input. The routines with names of the form **utc_asc...** produce character representations of time as output.

This section defines the character representations of time and is an extension of ISO 8601: 1988. The terminology and notation of the standard are used in this section.

Character Absolute Time Type

The extended format of the complete representation of combined calendar date and time of the day in UTC is specified by ISO 8601: 1988 as follows:

```
CCYY-MM-DDThh:mm:ss,ffffZ
```

Where `ffff` indicates a variable length field that may be null.

In this representation, the `,` (comma) represents the preferred separator between the seconds and fraction of a second. ISO 8601: 1988, however, specifies that a `.` (dot) is acceptable. The `T` is the specified separator between the dates and times. To conform with customary usage, implementations are required to accept the use of `-` (minus) in this role for complete representations. Only the `T` separator is allowed in truncated representations.

If times other than UTC are to be expressed, the local time is followed by a `+` (plus) or `-` (minus) sign and the timezone differential, expressed in hours and minutes. Again, strictly following ISO 8601: 1988, the representation is as follows:

```
CCYY-MM-DDThh:mm:ss,ffff+hh:mm
CCYY-MM-DDThh:mm:ss,ffff-hh:mm
```

To obtain UTC (also, the Greenwich Mean Time) corresponding to such a display, the TDF is subtracted from the local time.

These representations do not support the display (or input) of inaccuracies. Thus, the representation must be extended. By analogy with the `T` separator, the inaccuracy is specified by using an `I` as a separator. Implementations must also accept the symbol ± (plus-minus), which is encoded in ASCII as 261 decimal, as a replacement for `I`. The inaccuracy value is expressed in seconds and fractions of a second. The complete representation of a combined calendar date and time of the day (UTC or local time) with inaccuracy in extended format is as follows:

```
CCYY-MM-DDThh:mm:ss,ffffZIsss,ffff
CCYY-MM-DDThh:mm:ss,ffff+hh:mmIssss,ffff
CCYY-MM-DDThh:mm:ss,ffff-hh:mmIssss,ffff
CCYY-MM-DDThh:mm:ss,ffff-hh:mmI- ----
```

For the inaccuracy, both `sss` and `ffff` indicate variable length fields that may be null.

In the example that follows, the character times represent the date January 18, 1991 at the time 23:00 GMT with an inaccuracy of 23 milliseconds.

```
1991-01-18T23:00:00,00ZI0,023
1991-01-18T17:00:00,00-06:00I00,023
```

The rules for truncation and reduced precision are similar to those specified in ISO 8601: 1988. The representation may be truncated by omitting one or more of the leftmost fields. The precision may be reduced by deleting one or more of the right-most fields.

Infinite inaccuracies may be specified by an inaccuracy designator (`I` or the symbol ± (plus-minus)) with no trailing inaccuracy or with the string `-----`. Examples of the above times with infinite inaccuracy include the following:

```
1991-01-18T23:00:00ZI
1991-01-18-27:00:00-0600
1991-01-18-23:00:00Z
1991-01-18-23:00:00
1991-01-18-17:00:00.000-06:00I- ----
1991-01-18-17:00:00,-06:00 ±-----
```

Input and Output Representations

Implementations shall accept for input parameters all the representations specified in the previous section. This includes all the extended format complete representations of combined calendar date and time of the day, as specified in ISO 8601: 1988, with extensions for inaccuracy and the substitution of – (minus) for `T`. Implementations shall also accept any valid truncated or reduced precision extended format representation. Implementations are not required to accept the basic format where extended format is applicable. Implementations are not required to accept either ordinal dates or dates identified by week and day numbers.

Implementations shall return for output only the extended format of the complete representation of combined calendar date and time of the day, as specified in ISO 8601: 1988, with extensions for inaccuracy. Either the `.` (dot) or `,` (comma) separator may be used for the fraction field in both the time and inaccuracy. However, it is required that the same separator be used for the fraction field in both the time and inaccuracy.

Implementations may select between the valid separators for output format consistent with local custom, but are required to accept all valid inputs.

7.2.2 Character-relative Time Type

Character-relative times are specified in ISO 8601: 1988 as *periods.* Fields within periods are separated by period identifiers, for example, the string

```
P3W4D2H7M
```

represents the period, three weeks, four days, two hours and seven minutes. Again, this is extended in this specification by the optional addition of an inaccuracy field. The inaccuracy is appended to this string using the same methods as specified for absolute time.

This specification also mandates use of the following representation for character relative times:

```
ddddThh:mm:ss,ffffIssss,ffff

25T02:07:00I.023
25-02:07:00I0,023
25T02:07:00I00.023
25-02:07:00,00I0,023
25-02:07:00.00I.023
```

These examples represent the same 25 days (3 weeks plus 4 days), two hours and seven minutes along with an inaccuracy of 23 milliseconds.

Again, it is required for output that the same separator be used for the fraction field in both the time and inaccuracy. Likewise, implementations should select an output format from those specified that is consistent with local custom. Extended format complete representation without truncation or reduced precision is always used for output. Implementations are required to accept all valid inputs as defined by ISO 8601: 1988 with the extensions for inaccuracy defined in this section, as well as the formats specified in this section and the substitution of – (minus) for `T`.

7.2.3 The tm Structure

The **tm** structure declaration is as follows:

```
struct tm {
    int tm_sec;       /* Seconds (0 - 59)                   */
    int tm_min;       /* Minutes (0 - 59)                   */
    int tm_hour;      /* Hours (0 - 23)                     */
    int tm_mday;      /* Day of Month (1 - 31)              */
    int tm_mon;       /* Month of Year (0 - 11)             */
    int tm_year;      /* Year - 1900                        */
    int tm_wday;      /* Day of Week (Sunday = 0)           */
    int tm_yday;      /* Day of Year (0 - 364)              */
    int tm_isdst;     /* Non-zero if Daylight Savings       */
                      /* Time is in effect                  */
};
```

7.2.4 The timespec Structure

The **timespec** structure declaration is as follows:

```
struct timespec {
    time_t tv_sec;    /* Seconds since 00:00:00 GMT,    */
                      /* 1 January 1970                 */
    long   tv_nsec;   /* Additional nanoseconds since   */
                      /* tv_sec                         */
} timespec_t;
```

7.2.5 The reltimespec Structure

The **reltimespec** structure declaration is as follows:

```
struct reltimespec {
    time_t tv_sec;    /* Seconds of relative time   */
    long   tv_nsec;    * Additional nanoseconds of  */
                      /* relative time              */
} reltimespec_t;
```

7.2.6 Conversion Rules

Rules for converting between representations of time must yield consistent results across multiple implementations. To ensure consistency, all API implementations must apply the following rules:

- Character time must be syntactically correct and each element must take on a value in the valid range. Otherwise, the conversion routine must indicate an error. Similarly, in structured time representations (such as the POSIX **tm** structure) all fields must contain values in the valid range.
- When converting dates after the Gregorian reform (1582-10-15 and later), Gregorian rules for leap years must be used, and the binary time returned (or supplied) must be positive. When converting dates before the reform (1582-10-4 and earlier), Julian rules for leap years must be used, and the binary time returned or supplied must be negative. (All dates are limited to the range of years from 1 to 9999, inclusive.)
- A character time or structured time representation corresponding to the time of a leap second must be converted to binary representation by applying the following algorithm. The supplied time is of the form '`23 : 59 : 60.fIi`' where `f` is the fractional part of the second and `i` is the inaccuracy. The time value is converted to the value corresponding to the beginning of the next day (that is 00:00:00.0) and the inaccuracy is replaced with the value `i+1−f`. Then the normal conversion process may be applied.
- The version number of a supplied binary time must be one that is supported. Otherwise an error must be indicated.
- Character absolute time may not specify any of the dates from 1582-10-5 to 1582-10-14 inclusive. If such a date is specified, an error must be indicated. Similarly, if such a date is presented in a structured time representation, an error must be indicated.

7.3 Time Service API Taxonomy

The Time Service API routines may be divided into several categories according to their functions, as shown in Figure 7-1.

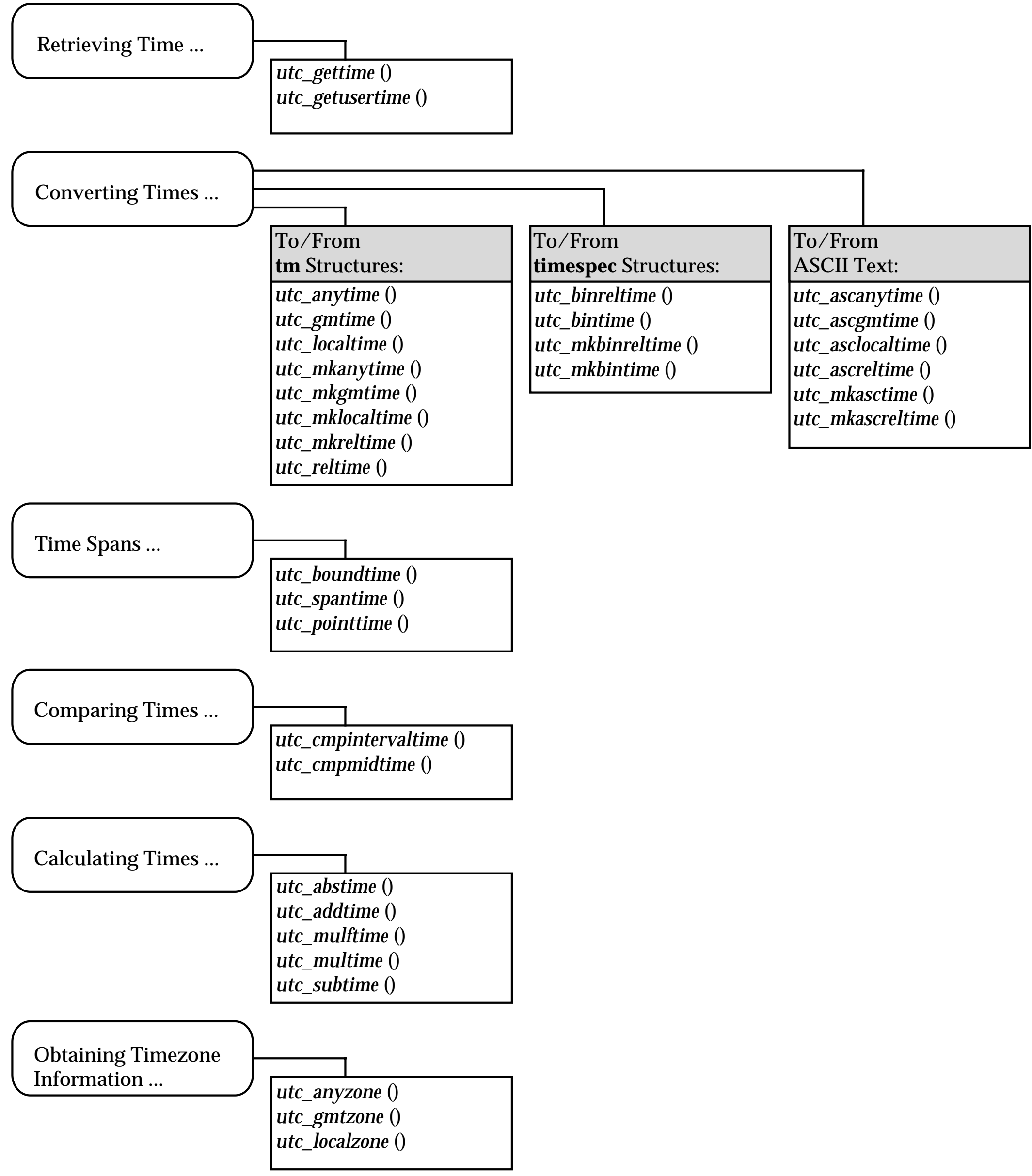

Figure 7-1 DTS API Routines by Functional Grouping

The general functions of the routines in each group are as follows:

Retrieving Time
Return a local time as provided by the Time Service.

Converting Times
Convert between opaque binary timestamps and time representations that may be interpreted and manipulated by programs.

Time Spans
Convert between a single timestamp and two or three timestamps, so that both the input and output timestamps span the same interval.

Comparing Times
Compare binary timestamps.

Calculating Times
Perform calculations on binary timestamps, returning the results as a binary timestamp.

Obtaining Timezone Information
Return timezone labels.

7.3.1 Time Conversions

When converting between time representations, the DTS API routines take account of the differing interpretations of time implied by each representation. The time service's binary representation of absolute time includes the UTC time and a TDF that represents the difference between UTC and local time in a given timezone. The ASCII representation of time also includes time and TDF components, but the time is interpreted as a *local time* in the timezone represented by the TDF. The **tm** and **timespec** structures explicitly include only a time component, and any TDF must be derived or supplied separately.

In general a **tm** structure is interpreted as containing a local time. When converting from a **tm** structure to a timestamp, this local time is adjusted to UTC using a TDF that is either user or system supplied. When converting from a timestamp to a **tm** structure, the TDF component of the timestamp is used to adjust the UTC time to a local time. In certain cases (see *utc_gmtime*( ) and *utc_mkgmtime*( )), a **tm** structure is interpreted as containing GMT, and no TDF adjustment is made in the conversion.

In general, the Time Service API interprets **timespec** structures as containing GMT.

Note: Aside from interpreting the timezone differently, each time representation also expresses time relative to a representation-specific base time using representation-specific units. When necessary, the time conversion API routines adjust time representations to the applicable base times and units.

The following list summarises the adjustments applied by the relevant API routines.

utc_gettime( )
: Returns a binary timestamp that represents local time (UTC and the local zone TDF). The TDF for the local timezone is derived in a system-dependent manner.

utc_getusertime( )
: Returns a timestamp that represents local time (UTC and the local zone TDF). The TDF for the local timezone is derived from the **TZ** environment variable.

utc_gmtime( )
: Converts a binary timestamp that represents time in an arbitrary zone (UTC and an arbitrary TDF) to a **tm** structure that represents the UTC component of the timestamp.

utc_anytime( )
: Converts a binary timestamp that represents time in an arbitrary zone (UTC and an arbitrary TDF) to a **tm** structure and a TDF parameter. The **tm** structure represents the time component of the timestamp, adjusted to the timezone of the TDF, and the TDF parameter represents the TDF component of the timestamp.

utc_localtime( )
: Converts a binary timestamp that represents time in an arbitrary zone (UTC and an arbitrary TDF) to a **tm** structure that represents the local

time. A system-derived TDF is applied to the timestamp UTC component to adjust it to the local time.

utc_ascgmtime( ) Converts a binary timestamp that represents time in an arbitrary zone (UTC and an arbitrary TDF) to an ASCII representation of GMT. The time component of the output string represents the UTC component of the timestamp, and the TDF component of the output string represents the GMT zone.

utc_ascanytime( ) Converts a binary timestamp that represents time in an arbitrary zone (UTC and an arbitrary TDF) to an ASCII representation. The time component of the output string represents the time component of the timestamp adjusted to the timezone of the TDF. The TDF component of the output string represents the TDF component of the timestamp.

utc_asclocaltime( ) Converts a binary timestamp that represents time in an arbitrary zone (UTC and an arbitrary TDF) to an ASCII representation of local time. The time component of the output string represents the UTC component of the timestamp adjusted to the local TDF, as supplied by the system. The TDF component of the output string represents the system-derived TDF.

utc_bintime( ) Converts a binary timestamp that represents time in an arbitrary zone (UTC and an arbitrary TDF) to a **timespec** structure and a TDF parameter. The **timespec** structure represents the time component of the timestamp in UTC, and the TDF parameter represents the TDF component of the timestamp.

utc_mkgmtime( ) Converts a **tm** structure into a binary timestamp with its time component determined by the input time and its TDF component set to GMT. That is, *utc_mkgmtime*( ) treats the input time as UTC.

utc_mkanytime( ) Converts a **tm** structure and a TDF parameter into a binary timestamp. To generate the time component of the binary timestamp, the input time is adjusted to GMT using the input TDF. The input TDF determines the TDF component of the binary timestamp.

utc_mklocaltime( ) Converts a **tm** structure into binary timestamp, adjusting the input time as follows. The input time is treated as the local time, and a TDF, supplied by the system, is used to adjust the input time to GMT. The time component of the binary timestamp is set to the adjusted time, and the TDF component is set to the system-supplied TDF.

utc_mkbintime( ) Converts a **timespec** structure and a TDF parameter into a binary timestamp. The input time, which is interpreted as UTC, determines the time component of the output timestamp. The **tdf** parameter determines the TDF component of the resulting binary timestamp.

Chapter 8

Time API Manual Pages

This chapter contains the reference manual pages for the Time API.

NAME

utc_abstime — computes the absolute value of a relative binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_abstime(
        utc_t *result,
        utc_t *utc);
```

ARGUMENTS

Input

utc Relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

result Absolute value of the input relative binary timestamp.

DESCRIPTION

The *utc_abstime*( ) routine computes the absolute value of a relative binary timestamp. The input timestamp represents a relative (delta) time.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

NAME

utc_addtime — computes the sum of two binary timestamps

SYNOPSIS

```
#include <dce/utc.h>

int utc_addtime(
        utc_t *result,
        utc_t *utc1,
        utc_t *utc2);
```

ARGUMENTS

Input

utc1 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

utc2 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

result Resulting absolute binary timestamp or relative binary timestamp, depending upon the operation performed:

```
relative time + relative time = relative time
absolute time + relative time = absolute time
relative time + absolute time = absolute time
absolute time + absolute time
```
is undefined. (See note below.)

DESCRIPTION

The *utc_adtime*( ) routine adds two binary timestamps, producing a third binary timestamp whose inaccuracy is the sum of the two input inaccuracies. One or both of the input timestamps represent a relative (delta) time. The TDF in the first input timestamp is copied to the output. The timestamps can be two relative times or a relative time and an absolute time.

Note: The combination `absolute time + absolute time` must *not* be used. The result is undefined in this case.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_subtime( )

NAME

utc_anytime — converts a binary timestamp to a **tm** structure that expresses time adjusted to the timezone represented by the timestamp's TDF

SYNOPSIS

```
#include <dce/utc.h>

int utc_anytime(
        struct tm *timetm,
        long      *tns,
        struct tm *inacctm,
        long      *ins,
        long      *tdf,
        utc_t     *utc);
```

ARGUMENTS

Input

utc Absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

timetm Time component of the binary timestamp expressed as local time, adjusted to the timezone represented by the TDF component of the binary timestamp.

tns Fractional seconds of the time component of the binary timestamp in nanoseconds.

inacctm Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is specified, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Fractional seconds of the inaccuracy component of the binary timestamp in nanoseconds.

tdf TDF component of the binary timestamp in units of seconds east of GMT.

DESCRIPTION

The *utc_anytime*( ) routine converts an absolute binary timestamp to a **tm** structure. The time returned is adjusted to the timezone represented by the TDF component of the binary timestamp. The inaccuracy and TDF components of the timestamp are also returned. Additional returns are fractional seconds of the time and inaccuracy, expressed in nanoseconds.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_mkanytime( ), *utc_anyzone*( ), *utc_gettime*( ), *utc_getusertime*( ), *utc_gmtime*( ), *utc_localtime*( )

NAME

utc_anyzone — gets the timezone label and offset from GMT

SYNOPSIS

```
#include <dce/utc.h>

int utc_anyzone(
        char         *tzname,
        size_t        tzlen,
        long         *tdf,
        int          *isdst,
        const utc_t  *utc);
```

ARGUMENTS

Input

tzlen Length of the *tzname* buffer.

utc Absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

tzname A character string that must be long enough to hold the timezone label.

tdf Longword with differential in seconds east of GMT.

isdst Integer with a value of −1, indicating that no information is supplied as to whether it is standard time or daylight saving time. A value of −1 is always returned.

DESCRIPTION

The *utc_anyzone*( ) routine gets the timezone label and offset from GMT by using the TDF contained in the *utc* input argument. The label returned is always of the form GMT + *n* or GMT − *n* where *n* is the *tdf* expressed in hours:minutes. (The label associated with an arbitrary timezone is not known; only the offset is known.)

Note: All of the output arguments are optional. No value is returned and no error occurs if the pointer is **NULL**.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or an insufficient buffer.

SEE ALSO

utc_anytime( ), *utc_gmtzone*( ), *utc_localzone*( )

NAME

utc_ascanytime — converts a binary timestamp to an ASCII string that expresses time adjusted to the timezone represented by the timestamp's TDF

SYNOPSIS

```
#include <dce/utc.h>

int utc_ascanytime(
        char    *cp,
        size_t  stringlen,
        utc_t   *utc);
```

ARGUMENTS

Input

stringlen The length of the *cp* buffer.

utc Absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

cp An ASCII string that expresses time adjusted to the timezone represented by the TDF component of the binary timestamp.

DESCRIPTION

The *utc_ascanytime*( ) routine converts an absolute binary timestamp to an ASCII string. The time component of the string expresses the time component of the binary timestamp, adjusted to the timezone of the TDF component of the binary timestamp. The TDF component in *cp* represents of the TDF component of the binary timestamp.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_ascgmtime( ), *utc_asclocaltime*( )

NAME

utc_ascgmtime — converts a binary timestamp to an ASCII string that expresses GMT time

SYNOPSIS

```
#include <dce/utc.h>
int utc_ascgmtime(
        char    *cp,
        size_t  stringlen,
        utc_t   *utc);
```

ARGUMENTS

Input

stringlen" Length of the *cp* buffer.

utc An absolute binary timestamp.

Output

cp An ASCII string that represents the time value of the binary timestamp and the GMT TDF.

DESCRIPTION

The *utc_ascgmtime*( ) routine converts an absolute binary timestamp to an ASCII string. The time component of the output string represents the UTC component of the timestamp, and the TDF component of the output string represents the GMT zone.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_ascanytime( ), *utc_asclocaltime*( )

NAME

utc_asclocaltime — converts a binary timestamp to an ASCII string that represents time adjusted to the local timezone

SYNOPSIS

```
#include <dce/utc.h>

int utc_asclocaltime(
        char    *cp,
        size_t  stringlen,
        utc_t   *utc);
```

ARGUMENTS

Input

stringlen Length of the *cp* buffer.

utc An absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

cp An ASCII string that expresses the time adjusted to the local timezone.

DESCRIPTION

The *utc_asclocaltime*( ) routine converts an absolute binary timestamp to an ASCII string. The time component of the string expresses the time component of the binary timestamp, adjusted to the local timezone.

When the environment variable **TZ** is defined, it determines the TDF used in the conversion. When **TZ** is not defined, the TDF is determined in a system-dependent manner.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_ascanytime( ), *utc_ascgmtime*( )

NAME

utc_ascreltime — converts a relative binary timestamp to an ASCII string that represents the relative time

SYNOPSIS

```
#include <dce/utc.h>

int utc_ascreltime(
        char          *cp,
        const size_t  stringlen,
        utc_t         *utc);
```

ARGUMENTS

Input

utc A relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

stringlen Length of the *cp* buffer.

Output

cp An ASCII string that represents the time.

DESCRIPTION

The *utc_ascreltime*( ) routine creates an ASCII string representation of a relative binary timestamp.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_mkascreltime( )

NAME

utc_binreltime — converts a relative binary timestamp to two **timespec** structures that express relative time and inaccuracy

SYNOPSIS

```
#include <dce/utc.h>

int utc_binreltime(
        reltimespec_t *timesp,
        timespec_t    *inaccsp,
        utc_t         *utc);
```

ARGUMENTS

Input

utc Relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

timesp Time component of the relative binary timestamp, in the form of seconds and nanoseconds since the base time (1970-01-01:00:00:00.0+00:00I0).

inaccsp Inaccuracy component of the relative binary timestamp, in the form of seconds and nanoseconds.

DESCRIPTION

The *utc_binreltime*( ) routine converts a relative binary timestamp to two **timespec** structures that express relative time and inaccuracy. These **timespec** structures describe a time interval.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_mkbinreltime( )

NAME

utc_bintime — converts a binary timestamp to a **timespec** structure

SYNOPSIS

```
#include <dce/utc.h>

int utc_bintime(
        timespec_t *timesp,
        timespec_t *inaccsp,
        long       *tdf,
        utc_t      *utc);
```

ARGUMENTS

Input

utc An absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

timesp Time component of the binary timestamp, in the form of seconds and nanoseconds since the base time.

inaccsp Inaccuracy component of the binary timestamp, in the form of seconds and nanoseconds.

tdf TDF component of the binary timestamp in the form of signed number of seconds east of GMT.

DESCRIPTION

The *utc_bintime*() routine converts a binary timestamp to a **timespec** structure and a TDF argument. The **timespec** structure represents the time component of the timestamp in UTC, and the TDF argument represents the TDF component of the timestamp.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_binreltime( ), *utc_mkbintime*( )

NAME

utc_boundtime — given two UTC times, returns a single UTC time whose inaccuracy bounds the input times

SYNOPSIS

```
#include <dce/utc.h>

int utc_boundtime(
        utc_t *result,
        utc_t *utc1,
        utc_t *utc2);
```

ARGUMENTS

Input

utc1 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

utc2 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

result Spanning timestamp.

DESCRIPTION

Given two UTC times, the *utc_boundtime*( ) routine returns a single UTC time whose inaccuracy bounds the two input times and their inaccuracies.

Note: The TDF in the output UTC value is copied from the *utc2* input argument. If one or both input values have unspecified inaccuracies, the returned time value also has an unspecified inaccuracy, and the returned time component is the average of the time components of the two input values.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid argument order.

SEE ALSO

utc_gettime( ), *utc_pointtime*( ), *utc_spantime*( )

NAME

utc_cmpintervaltime — compares two absolute binary timestamps or two relative binary timestamps

SYNOPSIS

```
#include <dce/utc.h>

int utc_cmpintervaltime(
        enum utc_cmptype *relation,
        utc_t            *utc1,
        utc_t            *utc2);
```

ARGUMENTS

Input

utc1 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

utc2 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

relation Receives the result of the comparison of *utc1*:*utc2* where the result is an enumerated type with one of the following values:

utc_equalTo	*utc1* is equal to *utc2*.
utc_lessThan	*utc1* is less than *utc2*.
utc_greaterThan	*utc1* is greater than *utc2*.
utc_indeterminate	*utc1* overlaps *utc2*.

DESCRIPTION

The *utc_cmpintervaltime*( ) routine compares two binary timestamps and returns a flag indicating that the first time is greater than, less than, equal to or overlapping with the second time. Two timestamps are equal if and only if their UTC time components are equal and both have 0 (zero) inaccuracy. Two times overlap if the intervals (UTC time – inaccuracy, UTC time + inaccuracy) of the two times intersect.

The two input binary timestamps either must both be absolute binary timestamps or must both be relative binary timestamps. The results are undefined when one timestamp is a relative binary timestamp and the other is an absolute binary timestamp.

RETURN VALUE

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument.

SEE ALSO

utc_cmpmidtime( )

NAME

utc_cmpmidtime — compares two absolute binary timestamps or two relative binary timestamps, ignoring inaccuracies

SYNOPSIS

```
#include <dce/utc.h>

int utc_cmpmidtime(
        enum utc_cmptype *relation,
        utc_t            *utc1,
        utc_t            *utc2);
```

ARGUMENTS

Input

utc1 An absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

utc2 An absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

relation Result of the comparison of *utc1*:*utc2* where the result is an enumerated type with one of the following values:

utc_equalTo	*utc1* is equal to *utc2*.
utc_lessThan	*utc1* is less than *utc2*.
utc_greaterThan	*utc1* is greater than *utc2*.

DESCRIPTION

The *utc_cmpmidtime*( ) routine compares two binary timestamps and returns a flag indicating that the UTC time component of the first timestamp is greater than, less than or equal to the UTC time component of the second timestamp. Inaccuracy information is ignored for this comparison; the input values are therefore equivalent to the midpoints of the time intervals described by the input binary timestamps.

The two input binary timestamps either must both be absolute binary timestamps or must both be relative binary timestamps. The results are undefined when one timestamp is a relative binary timestamp and the other is an absolute binary timestamp.

RETURN VALUE

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument.

SEE ALSO

utc_cmpintervaltime( )

NAME

utc_gettime — returns the current system time and inaccuracy as an absolute binary timestamp

SYNOPSIS

```
#include <dce/utc.h>
int utc_gettime(
        utc_t *utc);
```

ARGUMENTS

Input

None.

Output

utc	System time as an absolute binary timestamp.

DESCRIPTION

The *utc_gettime*() routine returns the current system time and inaccuracy in an absolute binary timestamp. The routine gets the TDF in a system-dependent manner.

RETURN VALUE

0	Indicates that the routine executed successfully.
−1	Generic error that indicates the time service cannot be accessed.

NAME

utc_getusertime — returns the current system time as an absolute binary timestamp, using an environment-specific TDF

SYNOPSIS

```
#include <dce/utc.h>

int utc_getusertime(
        utc_t *utc);
```

ARGUMENTS

Input

None.

Output

utc System time as an absolute binary timestamp.

DESCRIPTION

The *utc_getusertime*() routine returns the system time and inaccuracy in an absolute binary timestamp.

When the environment variable **TZ** is defined, the routine uses it to determine the TDF. If **TZ** is not defined, the system TDF, which is determined in a system-dependent manner, is used. In this case, this routine is equivalent to **utc_gettime**.

RETURN VALUE

0	Indicates that the routine executed successfully.
−1	Generic error that indicates the time service cannot be accessed.

SEE ALSO

utc_gettime()

NAME

utc_gmtime — converts an absolute binary timestamp to a **tm** structure that expresses GMT or the equivalent UTC

SYNOPSIS

```
#include <dce/utc.h>

int utc_gmtime(
        struct tm *timetm,
        long      *tns,
        struct tm *inacctm,
        long      *ins,
        utc_t     *utc);
```

ARGUMENTS

Input

utc An absolute binary timestamp to be converted to **tm** structure components. When **NULL**, the routine shall use the current time in place of this argument.

Output

timetm Time component of the binary timestamp.

tns Fractional seconds of the time component of the binary timestamp, expressed in nanoseconds.

inacctm Seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is specified, then **tm_mday** returns a value of –1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of –1.

ins Fractional seconds of the inaccuracy component of the binary timestamp, expressed in nanoseconds. If the inaccuracy is unspecified, *ins* returns a value of –1.

DESCRIPTION

The *utc_gmtime*( ) routine converts an absolute binary timestamp to a **tm** structure that expresses GMT (or the equivalent UTC). Additional returns include fractional seconds of time and inaccuracy from the binary timestamp, expressed in nanoseconds.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_anytime( ), *utc_gmtzone*( ), *utc_localtime*( ), *utc_mkgmtime*( )

NAME

utc_gmtzone — gets the timezone label for GMT

SYNOPSIS

```
#include <dce/utc.h>

int utc_gmtzone(
        char   *tzname,
        size_t  tzlen,
        long   *tdf,
        int    *isdst,
        utc_t  *utc);
```

ARGUMENTS

Input

tzlen Length of buffer *tzname*.

utc A binary timestamp. This argument is ignored.

Output

tzname A character string long enough to hold the timezone label.

tdf Differential in seconds east of GMT. A value of 0 (zero) is always returned.

isdst Daylight saving time indicator. A value of 0 (zero) is always returned, indicating that daylight saving time is not in effect.

DESCRIPTION

The *utc_gmtzone*( ) routine gets the timezone label and zero offset from GMT. Outputs are always *tdf* = 0 and *tzname* = GMT. This routine exists for symmetry with the *utc_anyzone*( ) and the *utc_localzone*( ) routines.

Note: All of the output arguments are optional. No value is returned and no error occurs if the *tzname* pointer is **NULL**.

RETURN VALUE

0 Indicates that the routine executed successfully (always returned).

SEE ALSO

utc_anyzone( ), *utc_gmtime*( ), *utc_localzone*( )

NAME

utc_localtime — converts an absolute binary timestamp to a **tm** structure that represents time adjusted to the local timezone

SYNOPSIS

```
#include <dce/utc.h>

int utc_localtime(
        struct tm *timetm,
        long      *tns,
        struct tm *inacctm,
        long      *ins,
        utc_t     *utc);
```

ARGUMENTS

Input

utc An absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

timetm A **tm** structure that holds the time component of the binary timestamp, adjusted to the local timezone.

tns Fractional seconds of the time component of the binary timestamp, expressed in nanoseconds.

inacctm A **tm** structure that holds seconds of the inaccuracy component of the binary timestamp. If the inaccuracy is specified, then **tm_mday** returns a value of –1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of –1.

ins Fractional seconds of the inaccuracy component of the binary timestamp, expressed in nanoseconds. If the inaccuracy is unspecified, *ins* returns a value of –1.

DESCRIPTION

The *utc_localtime*( ) routine converts an absolute binary timestamp to a **tm** structure that expresses the time component of the timestamp, adjusted to the local timezone.

When the environment variable **TZ** is defined, it determines the TDF used in the conversion. When **TZ** is not defined, the TDF is determined in a system-dependent manner.

Additional returns include fractional seconds of the time and inaccuracy components of the binary timestamp, expressed in nanoseconds.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_anytime( ), *utc_gmtime*( ), *utc_localzone*( ), *utc_mklocaltime*( )

NAME

utc_localzone — gets the local timezone label and offset from GMT, given an absolute binary timestamp.

SYNOPSIS

```
#include <dce/utc.h>

int utc_localzone(
        char   *tzname,
        size_t  tzlen,
        long   *tdf,
        int    *isdst,
        utc_t  *utc);
```

ARGUMENTS

Input

tzlen Length of the *tzname* buffer.

utc An absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

tzname Character string long enough to hold the timezone label.

tdf Differential in seconds east of GMT.

isdst Integer with a value of 0 (zero) if standard time is in effect or a value of 1 if daylight saving time is in effect.

DESCRIPTION

The *utc_localzone*( ) routine gets the local timezone label and offset from GMT, given an absolute binary timestamp.

When the environment variable **TZ** is defined, it determines the TDF. When **TZ** is not defined, the TDF is determined in a system-dependent manner.

Note: All of the output arguments are optional. No value is returned and no error occurs if the pointer is **NULL**.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or an insufficient buffer.

SEE ALSO

utc_anyzone( ), *utc_gmtzone*( ), *utc_localtime*( )

NAME

utc_mkanytime — converts a **tm** structure and TDF to a binary timestamp

SYNOPSIS

```
#include <dce/utc.h>
int utc_mkanytime(
        utc_t     *utc,
        struct tm *timetm,
        long       tns,
        struct tm *inacctm,
        long       ins,
        long       tdf);
```

ARGUMENTS

Input

timetm — A **tm** structure that represents time in the timezone of the *tdf* argument; **tm_wday** and **tm_yday** are ignored on input; the value of **tm_isdt** must be −1.

tns — Fractional seconds to add to the time component, expressed in nanoseconds.

inacctm — A **tm** structure that expresses days, hours, minutes and seconds of inaccuracy. If a **NULL** pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified; **tm_mday**, **tm_mon**, **tm_wday** and **tm_isdst** are ignored on input.

ins — Fractional seconds to add to the inaccuracy component, expressed in nanoseconds.

tdf — Time differential factor to use in converting the local time representation to an absolute binary timestamp.

Output

utc — Resulting absolute binary timestamp.

DESCRIPTION

The *utc_mkanytime*( ) routine converts a **tm** structure and a TDF to an absolute binary timestamp. To generate the time component of the binary timestamp, the input time is adjusted to GMT using the input TDF. The input TDF determines the TDF component of the binary timestamp. Other inputs include fractional seconds to add to the time and the inaccuracy, expressed in nanoseconds.

RETURN VALUE

0 — Indicates that the routine executed successfully.

−1 — Indicates an invalid time argument or invalid results.

SEE ALSO

utc_anytime( ), *utc_anyzone*( )

NAME

utc_mkascreltime — converts a **NULL**-terminated character string that represents a relative timestamp to a binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_mkascreltime(
        utc_t *utc,
        char  *string);
```

ARGUMENTS

Input

string A **NULL**-terminated string that represents a relative timestamp. (See Section 7.2.1 on page 63 for string representation of time.)

Output

utc Resulting relative binary timestamp.

DESCRIPTION

The *utc_mkascreltime*( ) routine converts a **NULL**-terminated string, which represents a relative timestamp, to a relative binary timestamp.

Note: The ASCII string must be **NULL**-terminated.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_ascreltime( )

NAME

utc_mkasctime — converts a **NULL**-terminated character string that represents an absolute time to an absolute binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_mkasctime(
        utc_t *utc,
        char  *string);
```

ARGUMENTS

Input

string A **NULL**-terminated string that represents an absolute time.

Output

utc Resulting absolute binary timestamp.

DESCRIPTION

The *utc_mkasctime*( ) routine converts a **NULL**-terminated string that represents an absolute time to an absolute binary timestamp.

Note: The ASCII string must be **NULL**-terminated.

RETURN VALUE

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

SEE ALSO

utc_ascanytime( ), *utc_ascgmtime*( ), *utc_asclocaltime*( )

NAME

utc_mkbinreltime — converts a **timespec** structure that represents a relative time to a relative binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_mkbinreltime(
        utc_t          *utc,
        reltimespec_t  *timesp,
        timespec_t     *inaccsp);
```

ARGUMENTS

Input

timesp A **timespec** structure that represents a relative time.

inaccsp A **timespec** structure that expresses inaccuracy. If a null pointer is passed, or if **tv_sec** is −1, the inaccuracy is considered to be unspecified.

Output

utc Resulting relative binary timestamp.

DESCRIPTION

The *utc_mkbinreltime*( ) routine converts a **timespec** structure that expresses relative time to a binary timestamp.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_binreltime( ), *utc_mkbintime*( )

NAME

utc_mkbintime — converts a **timespec** structure to an absolute binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_mkbintime(
        utc_t      *utc,
        timespec_t *timesp,
        timespec_t *inaccsp,
        long        tdf);
```

ARGUMENTS

Input

timesp A **timespec** structure that expresses time since 1970-01-01:00:00:00.0+0:00I0.

inaccsp A **timespec** structure that expresses inaccuracy. When a null pointer is passed, or **tv_sec** is −1, the inaccuracy is considered to be unspecified.

tdf TDF component of the binary timestamp.

Output

utc Resulting absolute binary timestamp.

DESCRIPTION

The *utc_mkbintime*( ) routine converts a **timespec** structure to an absolute binary timestamp. The input time, which is interpreted as GMT, determines the time component of the output timestamp. The *tdf* argument determines the TDF component of the resulting binary timestamp.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_bintime( ), *utc_mkbinreltime*( )

NAME

utc_mkgmtime — converts a **tm** structure that expresses GMT or UTC to an absolute binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_mkgmtime(
        utc_t     *utc,
        struct tm *timetm,
        long       tns,
        struct tm *inacctm,
        long       ins);
```

ARGUMENTS

Input

timetm A **tm** structure that expresses GMT or UTC. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdt** must be –1.

tns Fractional seconds to add to the time component, expressed in nanoseconds.

inacctm A **tm** structure that expresses days, hours, minutes and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_wday** and **tm_isdst** are ignored.

ins Fractional seconds to add to the inaccuracy component, expressed in nanoseconds.

Output

utc Resulting absolute binary timestamp.

DESCRIPTION

The *utc_mkgmtime*( ) routine converts a **tm** structure that expresses GMT or UTC to an absolute binary timestamp. Additional inputs include fractional seconds to add to the time and inaccuracy components, expressed in nanoseconds.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_gmtime( )

NAME

utc_mklocaltime — converts a **tm** structure that expresses local time to an absolute binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_mklocaltime(
        utc_t     *utc,
        struct tm *timetm,
        long        tns,
        struct tm *inacctm,
        long       ins);
```

ARGUMENTS

Input

timetm A **tm** structure that expresses the local time. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdst** should be –1.

tns Fractional seconds to add to the time component, expressed in nanoseconds.

inacctm A **tm** structure that expresses days, hours, minutes and seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_wday** and **tm_isdst** are ignored.

ins Fractional seconds to add to the inaccuracy component, expressed in nanoseconds.

Output

utc Resulting absolute binary timestamp.

DESCRIPTION

The *utc_mklocaltime*( ) routine converts a **tm** structure that expresses local time to an absolute binary timestamp.

When the environment variable **TZ** is defined, it determines the TDF used in the conversion. When **TZ** is not defined, the TDF is determined in a system-dependent manner. The input time is treated as the local time, and the TDF is used to adjust the input time to GMT. The time component of the binary timestamp is set to the adjusted time, and the TDF component is set to the environment-derived TDF.

Additional inputs include fractional seconds to add to the time and inaccuracy components, expressed in nanoseconds.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_localtime( )

NAME

utc_mkreltime — converts a **tm** structure that expresses relative time to a relative binary timestamp

SYNOPSIS

```
#include <dce/utc.h>

int utc_mkreltime(
        utc_t     *utc,
        struct tm *timetm,
        long       tns,
        struct tm *inacctm,
        long       ins);
```

ARGUMENTS

Input

timetm A **tm** structure that expresses a relative time. On input, **tm_wday** and **tm_yday** are ignored; the value of **tm_isdst** must be −1.

tns Fractional seconds to add to the time component, expressed in nanoseconds.

inacctm A **tm** structure that expresses seconds of inaccuracy. If a null pointer is passed, or if **tm_yday** is negative, the inaccuracy is considered to be unspecified. On input, **tm_mday**, **tm_mon**, **tm_year**, **tm_wday**, **tm_isdst** and **tm_zone** are ignored.

ins Fractional seconds to add to the the inaccuracy component, expressed in nanoseconds.

Output

utc Resulting relative binary timestamp.

DESCRIPTION

The *utc_mkreltime*() routine converts a **tm** structure that expresses relative time to a relative binary timestamp. Additional inputs include fractional seconds to add to the time and inaccuracy components, expressed in nanoseconds.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

NAME

utc_mulftime — multiplies a relative binary timestamp by a floating-point value.

SYNOPSIS

```
#include <dce/utc.h>
int utc_mulftime(
        utc_t  *result,
        utc_t  *utc1,
        double  factor);
```

ARGUMENTS

Input

utc1 Relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

factor Scale factor.

Output

result Resulting relative binary timestamp.

DESCRIPTION

The *utc_mulftime*( ) routine multiplies a relative binary timestamp by a floating-point value. Either or both may be negative; the sign of the resulting relative binary timestamp is negative if and only if the sign of the input binary timestamp is the opposite of the sign of the floating point multiplier. The unsigned inaccuracy in the relative binary timestamp is also multiplied by the absolute value of the floating-point value.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_multime( )

NAME

utc_multime — multiplies a relative binary timestamp by an integer factor

SYNOPSIS

```
#include <dce/utc.h>

int utc_multime(
        utc_t *result,
        utc_t *utc1,
        long   factor);
```

ARGUMENTS

Input

utc1	Relative binary timestamp.
factor	Integer scale factor.

Output

result	Resulting relative binary timestamp.

DESCRIPTION

The *utc_multime*( ) routine multiplies a relative binary timestamp by an integer. Either or both may be negative; the sign of the resulting relative binary timestamp is negative if and only if the sign of the input binary timestamp is the opposite of the sign of the integer multiplier. The unsigned inaccuracy in the binary timestamp is also multiplied by the absolute value of the integer.

RETURN VALUE

0	Indicates that the routine executed successfully.
–1	Indicates an invalid time argument or invalid results.

SEE ALSO

utc_mulftime( )

NAME

utc_pointtime — converts a binary timestamp to three binary timestamps that represent the earliest, most likely and latest time

SYNOPSIS

```
#include <dce/utc.h>

int utc_pointtime(
        utc_t *utclp,
        utc_t *utcmp,
        utc_t *utchp,
        utc_t *utc);
```

ARGUMENTS

Input

utc Binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

utclp Lowest (earliest) possible absolute time or shortest possible relative time that the input timestamp can represent.

utcmp Midpoint of the input timestamp.

utchp Highest (latest) possible absolute time or longest possible relative time that the input timestamp can represent.

DESCRIPTION

The *utc_pointtime*( ) routine converts a binary timestamp to three binary timestamps that represent the earliest, latest and most likely (midpoint) times. If the input is an absolute binary time, the outputs represent absolute binary times. If the input is a relative binary time, the outputs represent relative binary times.

All outputs have zero inaccuracy. An error is returned if the input binary timestamp has an unspecified inaccuracy.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument.

SEE ALSO

utc_boundtime( ), *utc_spantime*( )

NAME

utc_reltime — converts a relative binary timestamp to a **tm** structure

SYNOPSIS

```
#include <dce/utc.h>

int utc_reltime(
        struct tm *timetm,
        long      *tns,
        struct tm *inacctm,
        long      *ins,
        utc_t     *utc);
```

ARGUMENTS

Input

utc Relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

timetm Relative time component of the relative binary timestamp. The field **tm_mday** returns a value of −1 and the fields **tm_year** and **tm_mon** return values of 0 (zero). The field **tm_yday** contains the number of days of relative time.

tns Fractional seconds of the time component of the relative binary timestamp, expressed in nanoseconds.

inacctm Seconds of the inaccuracy component of the relative binary timestamp. If the inaccuracy is specified, then **tm_mday** returns a value of −1 and **tm_mon** and **tm_year** return values of 0 (zero). The field **tm_yday** contains the inaccuracy in days. If the inaccuracy is unspecified, all **tm** structure fields return values of −1.

ins Fractional seconds of the the inaccuracy component of the relative binary timestamp, expressed in nanoseconds.

DESCRIPTION

The *utc_reltime*( ) routine converts a relative binary timestamp to a **tm** structure. Additional returns include fractional seconds of the time and inaccuracy components of the binary timestamp, expressed in nanoseconds.

RETURN VALUE

0 Indicates that the routine executed successfully.

−1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_mkreltime( )

NAME

utc_spantime — given two absolute binary timestamps, returns a single UTC time interval whose inaccuracy spans the two

SYNOPSIS

```
#include <dce/utc.h>

int utc_spantime(
        utc_t *result,
        utc_t *utc1,
        utc_t *utc2);
```

ARGUMENTS

Input

utc1 Absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

utc2 Absolute binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

result Spanning timestamp.

DESCRIPTION

Given two absolute binary timestamps, the *utc_spantime*( ) routine returns a single UTC time interval whose inaccuracy spans the two input timestamps; from the earliest possible time of the two timestamps to the latest possible time of the two timestamps.

The *tdf* argument in the output binary timestamp is copied from the *utc2* input. If either input binary timestamp has an unspecified inaccuracy, an error is returned.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument.

SEE ALSO

utc_boundtime( ), *utc_gettime*( ), *utc_pointtime*( )

NAME

utc_subtime — computes the difference between two binary timestamps

SYNOPSIS

```
#include <dce/utc.h>

int utc_subtime(
        utc_t *result,
        utc_t *utc1,
        utc_t *utc2);
```

ARGUMENTS

Input

utc1 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

utc2 Absolute binary timestamp or relative binary timestamp. When **NULL**, the routine shall use the current time in place of this argument.

Output

result Resulting absolute binary timestamp or relative binary timestamp, depending upon the operation performed:

```
absolute time − absolute time = relative time
relative time − relative time = relative time
absolute time − relative time = absolute time
relative time − absolute time
```
is undefined.

DESCRIPTION

The *utc_subtime*() routine subtracts *utc2* from *utc1* (*utc1* – *utc2*). The two binary timestamps may express either an absolute time and a relative time, two relative times or two absolute times. However, if *utc1* is a relative time, then *utc2* must also be a relative time. The result of the operation:

```
relative time − absolute time
```

is undefined.

The inaccuracies of the two input timestamps are combined and included in the output timestamp. The TDF in the first timestamp is copied to the output.

RETURN VALUE

0 Indicates that the routine executed successfully.

–1 Indicates an invalid time argument or invalid results.

SEE ALSO

utc_addtime( )

Appendix A

Time Representation

The Time Service interfaces and messages use three data types for time. These are binary absolute time, binary relative time and simple binary relative time.

Note: The formats described here are not visible to applications. Instead, time is presented to applications as opaque binary timestamps. The application programmer's interface includes functions to set, interpret and manipulate fields in binary timestamps.

Absolute time is represented in binary form as the integral number of 100ns time units that have elapsed since a base time which is 1582-10-15-00:00:00+00:00 (the beginning of October 15, 1582 at the Greenwich meridian). The inaccuracy is also expressed as an integral number of 100ns time units. The TDF is expressed as an integral number of minutes east of the Greenwich meridian.

The time, inaccuracy, TDF and a version control number are represented by a data structure consisting of 16 8-bit bytes partitioned into 4 fields. The first field consists of the first 8 bytes and represents a date and time value in 100ns units since the base date encoded using two's complement arithmetic. With this encoding, all dates from 0001-01-01-00:00:00 (January 1, 1) to 9999-12-31-23:59:59 (December 31, 9999) can be represented in binary absolute time. The second field consists of the next 6 bytes and represents the inaccuracy in units of 100ns encoded as an unsigned integer. With this encoding, inaccuracies over 300 days can be represented. The third field is 12 bits long and consists of the next (second from last) byte and the 4 least-significant bits of the last byte. This field represents the TDF in units of minutes and is encoded using two's complement arithmetic. The last field is used for version control and consists of two subfields — a 3-bit number, and a 1-bit endian flag. The endian flag is 0 if the timestamp is stored in little endian format, that is, the natural format for little endian processors. It is 1 if the timestamp is stored in big-endian format.

The arrangement and interpretation of the fields depends on the endian flag. Machines generate timestamps in their native format, but are responsible for interpreting timestamps in either format. This permits interoperation, without favouring either addressing scheme. Further, this ''user makes right'' strategy permits[12] timestamps to be stored in files that are shared between disparate architectures.

The little-endian representation consists of the following 16 octets:

12. This is especially useful for event logs whose records typically consist of a timestamp and a Latin-1 string (refer to ISO 8859: 1987).

```
Byte 0                     Least significant 8 bits of the time
  .
  .
Byte 7                     Most significant 8 bits of the time
Byte 8                     Least significant 8 bits of the inaccuracy
  .
  .
Byte 13                    Most significant 8 bits of the inaccuracy
Byte 14                    Least significant 8 bits of the TDF
Byte 15
 Least sig 4 bits          Most significant 4 bits of the TDF
 Next least sig bit        1
 Next least sig bit        0
 Next least sig bit        0
 Most significant bit      0 (little-endian)
```

The big-endian representation consists of the following 16 octets:

```
Byte 0                     Most significant 8 bits of the time
  .
  .
Byte 7                     Least significant 8 bits of the time
Byte 8                     Most significant 8 bits of the inaccuracy
  .
  .
Byte 13                    Least significant 8 bits of the inaccuracy
Byte 14                    Least significant 8 bits of the TDF
Byte 15
 Least sig 4 bits          Most significant 4 bits of the TDF
 Next least sig bit        1
 Next least sig bit        0
 Next least sig bit        0
 Most significant bit      1 (big-endian)
```

Note that the TDF field is always stored in little-endian format. This is because the two subfields of the TDF field are not logically contiguous when viewed on a big-endian processor. Also note that the endian flag is always the most-significant bit of the last byte.

In either addressing scheme, the time field is considered as a 64-bit signed two's complement quantity. The inaccuracy field is considered a 48-bit unsigned quantity. Further, the inaccuracy value consisting of all 1s is special and represents an infinite inaccuracy. The TDF field is considered a 12-bit signed two's complement number that represents the integral number of minutes ahead of UTC. TDF values must be in the range −780 to +780 (−13 hours to +13 hours), inclusive. Values outside this range are reserved for future extensions.

There is no representation for the times corresponding to a leap second. That is, there is no representation of the form:

```
yyyy-mm-00-HH:MM:60
```

When a leap second occurs, the inaccuracy must be increased to reflect this.

Relative time is represented in binary form with a syntax identical to absolute time. However, the notion of a TDF is meaningless and so the TDF field is reserved and must be 0.

Finally, simple relative time is represented in binary form with a syntax identical to absolute time. However, the notion of a TDF or an inaccuracy are meaningless and so both of these fields must be 0.

Appendix B

Parameters, Constants and Names

This appendix describes a set of parameters, architectural constants and names for which implementations must define default values. These parameters and constants appear throughout the specification text. All time values are given as *hours*:*minutes*:*seconds*.

B.1 Parameters

Parameter default values are described in Table B-1.

Time Service Parameter Default Values	
Parameter	**Default**
checkInt	01:30:00.00 (90 min.)
epochNumber	0
errorTolerance	00:10:00.00 (10 min.)
TDF	00:00:00
MaxInacc	00:00:00.100 (100 msec.)
minServers	server: 3 clerk: 1
repetitions	3
syncHold	server: 00:02:00.00 (2 min.) clerk: 00:10:00.00 (10 min.)
courierRole	**BackupCourier** = 2
LStimeOut	00:00:10.00 (10 sec.)
GStimeOut	00:00:15.00 (15 sec.)
serverEntryName	"dts-entity"
groupName	"dts-servers"

Table B-1 Default Parameter Values

B.2 Constants

Architectural constant values are described in Table B-2.

Time Service Architectural Constant Values	
Constant	**Type**
cacheRefresh	2:00:00.00
minLocalServers	3

Table B-2 Architectural Constant Values

B.3 Names

The following well-known default names are recommended:

- The default server entry name is constructed by appending the parameter *serverEntryName* to the host's cell relative name. Using the *serverEntryName* value given in Table B-1 on page 107, the name is therefore **/.:/***hostname***/dts-entity**.
- The default cell profile is **/.:/cell-profile**.
- The default LAN profile is **/.:/lan-profile**.
- The default time server group name is constructed by appending the value of *groupName* to the name **/.:/subsys/dce/**. Using the *groupName* value given in Table B-1 on page 107, the name is therefore **/.:/subsys/dce/dts-servers**.

Appendix C

LAN Services Interface Definition

The LAN services interface definition is given below:

```
[uuid (6f264242-b9f8-11c9-ad31-08002b0dc035),
    version(1)
]

interface lan_profile
{
 import "dce/nbase.idl";

        /*
         * Do not export any procedures. We just want the interface
         * specification for the LAN  services profile structure,
         * generated by client stub.
         *
         * The structure generated is lan_services_v1_0_c_ifspec
         * of type rpc_if_handle_t
         */

void dummy_lan ([in] handle_t    dummy_handle );

}
```

Time Interval Arithmetic

The time service returns an absolute binary timestamp in response to a request to read the current time. Encoded in an absolute binary timestamp is an absolute time interval along with other information. The absolute time interval expressed by some absolute binary timestamp, A_j, can be represented as:

$$[T_j - I_j, T_j + I_j]$$

where T_j is the midpoint of the interval and corresponds to an absolute time, and I_j is the distance from the midpoint to either endpoint.

The time service also provides a notion called relative time. A relative time represents the difference or duration between a pair of absolute times. The time service provides a relative binary timestamp to express information about a relative time. Encoded in a relative binary timestamp is a relative time interval corresponding to the difference between some pair of absolute time intervals along with other information. The relative time interval expressed by some relative binary timestamp, R_j, can be represented as:

$$[D_j - I_j, D_j + I_j]$$

where D_j is the midpoint of the interval and corresponds to the difference between some pair of absolute times, and I_j is the distance from the midpoint to either endpoint.

The following arithmetic operations are defined for time intervals and result in an interval of the type indicated:

```
absolute time + relative time = absolute time
relative time + absolute time = absolute time
relative time + relative time = relative time
absolute time − absolute time = relative time
absolute time − relative time = absolute time
relative time − relative time = relative time
p * relative time = relative time
```

where:

\+ represents the addition operation

– represents the subtraction operation

* represents the multiplication operation

= indicates the result of the operation.

Let A_i and A_j denote two arbitrary absolute time intervals, and R_k and R_l denote two arbitrary relative time intervals, and p denote a real number, then the following holds:

$A_i + R_k$ results in the absolute time interval defined by:

$$[(T_i + D_k) - (I_i + I_k), (T_i + D_k) + (I_i + I_k)]$$

$R_k + A_i$ results in the absolute time interval defined by:

$$[(T_i + D_k) - (I_i + I_k), (T_i + D_k) + (I_i + I_k)]$$

$R_l + R_k$ results in the relative time interval defined by:

$$[(D_l + D_k) - (I_l + I_k), (D_l + D_k) + (I_l + I_k)]$$

$A_i - R_k$ results in the absolute time interval defined by:

$[(T_i - D_k) - (I_i + I_k), (T_i - D_k) + (I_i + I_k)]$

$A_i - A_j$ results in the relative time interval defined by:

$[(T_i - T_j) - (I_i + I_j), (T_i - T_j) + (I_i + I_j)]$

$R_l - R_k$ results in the relative time interval defined by:

$[(D_l - D_k) - (I_l + I_k), (D_l - D_k) + (I_l + I_k)]$

$p * R_k$ results in the relative time interval defined by:

$[(p*D_k) - (|p|*I_k), (p*D_k) + (|p|*I_k)]$

where $|p|$ denotes the absolute value of p.

Index

AdjustClkEnd 30
AdjustClock 28
architecture 7
CalcInaccuracy 32
ClerkRequestGlobalTime 54
ClerkRequestTime 52
clock
  adjustment 16
clocks 4
ComputedTimeMinimum 26
computing correct time 13
configuration 34
constants 107
ContactProvider 58
conversion rules 66
correct time 5
couriers 37
data types 51, 57
distributed systems 3
epochs 46
  initialisation 46
EstimateServerTime 25
faulty servers 49
global set
  export 36
  import 36
inaccuracy
  determination 17
initialisation 45
input representations 64
LAN profile
  location 35
LAN services
  interface definitions 109
leap seconds 19
  inaccuracy 19
  obtaining time 22
local faults 23
local set
  export 35
  import 35
names 107-108
next synchronisation
  determination 42
output representations 64
parameter ranges 51
parameters 107
primitive procedures 24
procedures
  AdjustclkEnd 30
  AdjustClock 28
  CalcInaccuracy 32
  ComputedTimeMinimum 26
  EstimateServerTime 25
  SetClock 31
ranges 51
reltimespec 65
security 36
server list
  maintenance 43, 50
ServerRequestGlobalTime 55
ServerRequestProviderTime 58
ServerRequestTime 53
SetClock 31
structures
  reltimespec 65
  timespec 65
  tm 65
synchronisation 47
  determination of next 49
  other servers 47
  TP 47
system clock
  initialisation 39
time
  clocks 4
  distributed systems 3
time API 61
  manual pages 71
time conversion 68
time interval arithmetic 111
time provider interface 56
time representation 105
  character 63
  non-opaque 63
time service
  clerk specification 39
  configuration 33
  functional overview 9
  IDL declarations 51
  rationale 6
  server specification 45

time service API
  taxonomy 67
time service interface
  global set 54
  local set 52
time value
  server 9
  time provider 12
time zones 23
TimeResponseType 57
timespec 65
timestamps 62
tm 65
TPctlMsg 57
TPtimeMsg 57
type
  character absolute time 63
  character-relative time 64
  utc_t 62
utc structure 51
utc_abstime( ) **72**
utc_addtime( ) **73**
utc_anytime( ) **74**
utc_anyzone( ) **75**
utc_ascanytime( ) **76**
utc_ascgmtime( ) **77**
utc_asclocaltime( ) **78**
utc_ascreltime( ) **79**
utc_binreltime( ) **80**
utc_bintime( ) **81**
utc_boundtime( ) **82**
utc_cmpintervaltime( ) **83**
utc_cmpmidtime( ) **84**
utc_gettime( ) **85**
utc_getusertime( ) **86**
utc_gmtime( ) **87**
utc_gmtzone( ) **88**
utc_localtime( ) **89**
utc_localzone( ) **90**
utc_mkanytime( ) **91**
utc_mkascreltime( ) **92**
utc_mkasctime( ) **93**
utc_mkbinreltime( ) **94**
utc_mkbintime( ) **95**
utc_mkgmtime( ) **96**
utc_mklocaltime( ) **97**
utc_mkreltime( ) **98**
utc_mulftime( ) **99**
utc_multime( ) **100**
utc_pointtime( ) **101**
utc_reltime( ) **102**
utc_spantime( ) **103**
utc_subtime( ) **104**