Licensing Service Specification

Version 1.0
New Edition: April 2000

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm.*

Contents

Contents

Preface

About This Document

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

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

Object Management Group

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

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

What is CORBA?

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

X/Open

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

Intended Audience

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

Need for Object Services

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

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

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

What Is an Object Service Specification?

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

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

Associated OMG Documents

The CORBA documentation is organized as follows:

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

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

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

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

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

Service Design Principles

Build on CORBA Concepts

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

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

Other related principles that the designs adhere to include:

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

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

Basic, Flexible Services

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

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

Generic Services

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

Allow Local and Remote Implementations

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

Quality of Service is an Implementation Characteristic

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

Objects Often Conspire in a Service

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

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

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

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

Use of Callback Interfaces

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

Callback interfaces have two major benefits:

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

Assume No Global Identifier Spaces

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

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

Finding a Service is Orthogonal to Using It

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

Interface Style Consistency

Use of Exceptions and Return Codes

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

Explicit Versus Implicit Operations

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

Use of Interface Inheritance

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

Acknowledgments

The following companies submitted and/or supported parts of the *CORBA Services* specifications:

- Digital Equipment Corporation
- Gradient Technologies, Inc.
- International Business Machines Corporation

Service Description 1

Contents

This chapter contains the following topics.

1.1 Background Information

1.1.1 Existing License Management Products

This section, “Background on Existing License Management Products,” is for readers who are unfamiliar with the management of software licenses. It provides an overview of licensing and addresses issues that must be faced in developing and selecting a license management system.

Application suppliers need methods for controlling the access to and use of their products. In most cases, this is necessary to ensure fair compensation for use. The most common control method used by software suppliers is licensing, where the license can be provided through technical (software- or hardware-based) or contractual means. While contractual licensing is a viable option, it does not provide the same level of

control as technical licensing, which uses hardware or software tools to control licensing. Therefore, application suppliers continue to require technical licensing methods to complement legal contracts.

Along with the expanding need for technical licensing, there are specific requirements for licensing that must change to reflect today's computing environments. Traditional licensing methods (nodelocked licensing and site licensing) evolved from computing environments of the past, specifically timesharing systems and stand-alone systems such as PCs and workstations. These older licensing methods are insufficient for current environments.

While today's computing environments provide significant advantages for application suppliers and end users, they also present opportunities. It is apparent that software and hardware resources can be managed on a network-wide basis for maximum efficiency. However, the resulting requirement for network-wide license sharing is less apparent. The traditional licensing methods (expensive site licensing and inflexible nodelocked licensing) do not complement today's fleixible and efficient computing environments.

Given these realities, sophisticated technical licensing tools are required. These licensing tools are important to all constituents in the market: application suppliers; hardware vendors; and application users. Software suppliers need a licensing tool to support their business and pricing models. Hardware vendors embed and offer the technology to support software developers and end users, and act themselves as application suppliers for their internally developed applications. End users interact with licensing technologies when they use, manage, and pay for software applications.

1.1.2 Business Policy

In the development and selection of software licensing systems, the licensing system must not impose its business practices on users. The software license is, in effect, a contract between suppliers and customers that establishes a business relationhip between them. Because a software licensing system plays an important role in regulating this contract, it must provide mechanisms to implement the flexible business practices that suppliers need to deal with a diverse customer base.
One danger in developing a licensing system is that it could reflect the business practices of the developing organization. This is sometimes the case when a licensing system is developed for internal use in a large organization and then offered for general use. A licensing system may work for one company, but will probably not address a wide range of business policies and practices. Often this problem manifests itself in subtle ways.

1.1.3 License Types

If not fully considered beforehand, it is possible to construct a software licensing scheme that forces the software suppliers and end users into a limited model of software licensing. If a licensing system offers only limited license types and/or offers few options for applying them, software suppliers are limited in the way they manage business relationships with their software customers.

Because software licensing touches many aspects of a relationship with a customer, including upgrades, support, enhancements, and follow-on purchases, a licensing system must provide a wide range of license options and many options for applying them. Software suppliers—not licensing system developers—must choose which licensing options they want to use.

The options allowed within various license types are also critical to ensure that application suppliers have all the capabilities necessary to establish the business relationship they desire with their customers. Capabilities such as allowing a grace period to provide unlicensed users access to the software for a limited period may be critical in retaining the goodwill of a large and influential customer. Other licensing features include selective user inclusion or exclusion lists; reserved licensing (to ensure that a license is always available to high-priority users); and multi-use rules that allow multiple use of an application with a single license. In addition, different license types can be used together in a single application. This ensures that the supplier, not the licensing system, determines business policies.

1.1.4 A History of License Types

Providing a wide-ranging portfolio of license types ensures that application suppliers are able to conduct business and arrange business policies as they deem appropriate.

Nodelocked licenses (which evolved from timesharing) allow a software product to be used at the single node for which the license was created. As the stand-alone workstation market grew, new licensing models were required. Major workstations users, such as insurance companies, banks, and industrial corporations, needing a more economical way to purchase software, demanded that application suppliers offer a business model that would provide unlimited use at a given site. That need gave rise to site licensing.

Site licensing often resulted in dissatisfaction of both suppliers and customers. Suppliers were asked to assess a price for usage they did not fully understand. They often felt they were being asked to discount their future revenue too deeply. Customers felt that the site license fees were excessive and made them pay for usage that might not occur in the future.

As networks of computers developed, system vendors began to introduce the notion of a concurrent use license. Concurrent use licenses define the number of users allowed to access an application at a given time. These licenses are allowed to “float” around the network, temporarily appropriated by users as applications are invoked, then returned to the license repository when an application is terminated. Concurrent use licensing allows end users to purchase licenses to match their usage and allows software providers to be compensated for use of their products. Additionally, end users can easily add more concurrent licenses as needed.

1.1.5 Asset Management

Licenses protect expensive corporate assets. Since licenses exist only as data they are harder to secure than a server or workstation, but every bit as important to control and manage. Control helps ensure that licenses are used in a manner which supports

corporate goals, such as improving compliance with paper software license contracts and reducing exposure to legal action. This helps keep the corporation out of court and enhances its relationship with its software suppliers. Large corporate software purchasers want to be treated as equal partners with their suppliers; licensing makes this easier.

Managing both existing and new licenses maximizes their value. Old licenses might need to be redeployed as projects and budgets change. If the license administrator can keep track of software licenses, know which licenses are and are not being used, and can move them to where they are needed, corporate waste will be reduced and productivity improved.

Similarly, if a corporation has software usage metrics, it has a strong basis for understanding future needs. These metrics permit a corporation to purchase licenses in bulk at lower prices with the confidence that they are not over or under buying.

A corporation can also measure whether they have over or under purchased in the past. An important metric is the "shelfware" measure. How much software was purchased (perhaps as unused components of "suites" of software) that never leaves the user's bookcase? Reducing such waste is a major incentive for software customers to use automated software licensing and asset management.

1.1.6 License Usage Practices

Application suppliers can implement one or more of the license types in their software products. An application can be programmed to require multiple license types, to allow the supplier to sell the product in different ways to different customers.

An ideal licensing system should be transparent to end users. For example, a user might invoke an application, which makes calls to a licensing library. Then, the library function locates a server with a valid license. Assuming that a valid license is available and that person is authorized to use the license, a grant is returned to the application, allowing the program to execute, all completely transparent to the end user.

If no licenses are immediately available, the application developer can program the software to respond in a variety of ways. The application can automatically put the user on a wait queue, query the user as to the course of action to take, recommend that the user try again later, or grant permission to run anyway. (The developer can choose to grant permission to run without a license if, for example, there is a “grace period” instituted to allow for a smooth transition to a network licensing model.) If all licenses are temporarily checked out and users go on a wait queue, the next available license can be granted according to user priority settings defined at the end user site.

These choices and how they are implemented comprise the policy a software provider chooses as a business model. Policy can be further broken into two components: fixed and variable. Fixed components are coded into the application and determine things such as what license types are permitted; whether multiple use rules apply to the application; or if a grace period will be extended when a license is not available. Variable components are defined externally to the client application and include such things as external definition of the hours a product may be used, or an external list of people allowed to use it. Either list may be producer- or end user-created.

1.1.7 Scalability

Some networks are small, consisting of just a few nodes, while others grow to thousands of machines. Typically, large user communities on large networks demand licensed applications from many different vendors. A licensing system and its runtime environment must, therefore, scale well to the network and all its software.

1.1.8 Reliability

Sometimes, an application obtains a license from a license server and never returns it. A licensing system must be designed to prevent licenses from being stranded and to prevent other client-server breakdowns.

Breakdowns occur for several reasons. The application or server could abort, or the network could become partitioned between the application and server. These situations could arise unintentionally or maliciously (for example, in an attempt to gain unauthorized use of an application). Any design must make careful trade-offs between license availability and security enforcement. All designs require a scheme to detect breakdowns.

Generally, there are two detection methods: continuous detection or occasional check-in. Continuous detection methods ensure that while a license is in use by an application, the application and server are both continuously aware of each other's existence and are immediately notified of a breakdown. These schemes are typically implemented by using a connection protocol such as a port. The main advantages of a continuous scheme are its directness, immediacy, and simplicity. The main disadvantage is its negative impact on network performance. If a redundant server high availability model is employed, then continuous connections need to be maintained between the application and each of the servers, thereby multiplying network traffic.

Occasional detection provides a method for the application to check in with the server periodically before some time out has occurred. The breakdown is identified either by the server (if the time out occurs), or by the application (if the check-in is unsuccessful). This method is very scalable and has a negligible impact on performance. The application supplier should be able to adjust the time-out to allow trade-offs between higher security and higher availability.

Additionally, the occasional detection model is very tolerant of momentary interruptions on the network. Continuous detection is not tolerant of such interruptions. Lost connections between the client and the server in a system using continuous detection causes a breakdown or program termination.

Application suppliers will want to determine for themselves which action to take in the case of a client-server breakdown. Some may want more strict enforcement and choose to terminate an application; others might choose to display a polite message and allow the application to continue.

1.1.9 Legacy Applications

Managing a business relationship with a minimum of disruption includes the ability to accommodate existing customer applications within the scope of the licensing system. This must be done without requiring access to or modification of the application's source code, as the apparently simple solution of modifying source code may not always be available. Consider the personal computer, for which there are tens of thousands of small and inexpensive applications. Modifying the sources of all these applications would be an economically unacceptable approach even if the source code were available.

Software suppliers are eagerly awaiting an integrated licensing technology that will take existing “shrink-wrapped” applications and enhance them to function in a licensed environment. It may not be possible to provide a security fence as high as a source-modified application, but the level of license security could be made commensurate with the value of the application and well beyond the economic justification for attempting to defeat the security.

1.1.10 Security

Until recently, licensing systems were required to enforce only simple, single-system application use. Security infractions caused few implications. Today, security requirements must be designed to operate in more complex networks.

The distributed computing networks in use today are designed for easy resource sharing; demand more complex licensing models (presenting new security challenges); and must support mass distribution of software (on compact disk, for example). A supplier's ability to ship trial copies of applications relies heavily on the security of the licensing system to ensure that prospective customers do not transgress the intended use permissions. An application supplier must also rely on the licensing system’s security when it ships a complete set of applications to its entire installed base: the licensing system must ensure that only the purchased applications are used.

Each application supplier has a different security need. Each will want to choose from a spectrum of trade-offs, such as security versus availability, and effect of breach versus development effort. A licensing system should not dictate one particular level of security, but should allow application suppliers to choose the security level appropriate for their business needs.

1.1.11 Client/Server Authentication

A secure licensing system should address the possibility of someone attempting to create an impostor license server (an imposter server always grants licenses). Without security, an impostor could be established by eavesdropping on valid client-server communication and then mimicking the license grant protocol. Impostor clients should also be addressed, since a successful impostor client could disrupt legitimate license activities by artificially returning a license to the license server when it is actually still in use, thereby making the returned license available for other users.

1.1.12 Example: Application Acquiring and Releasing a Concurrent License

This section contains an example of how an application might interact with one of the various license management products that exist today.

In a system that uses concurrenct licensing, end users at their workstations and PCs see no change in their normal working routine. They start applications as they normally would. The application has calls to the license library that transparently go over the network to request a license for the application. Using transport-specific naming and location facilities, a server holding a valid license is located and a “yes” is returned to the requesting application.

The application need not be downloaded over the network to the workstation each time the application is invoked. The application, once loaded, remains at the workstation as it normally would. Only a request for a license and a return grant go over the network, thereby providing a rapid response time that is virtually unnoticed by users.

When end users close an application, the license is “returned” to the server. The server then can make this license available for other requests as they come in.

Administration and reporting tools act as clients to the license server, tapping into server databases and log files to access the stored information. The license servers, though implemented as multiple physical servers, operate as a single repository managing all license activity for the network. This single, “logical” server handles licenses for any number of vendors, for any number of products, with any number of product versions. The server also handles any number of clients making requests for its facilities, thereby automatically scaling to accommodate increases in the number of users, machines, applications, and licenses.

1.2 Licensing Service Overview

Licensing Service terms are defined in Appendix A.

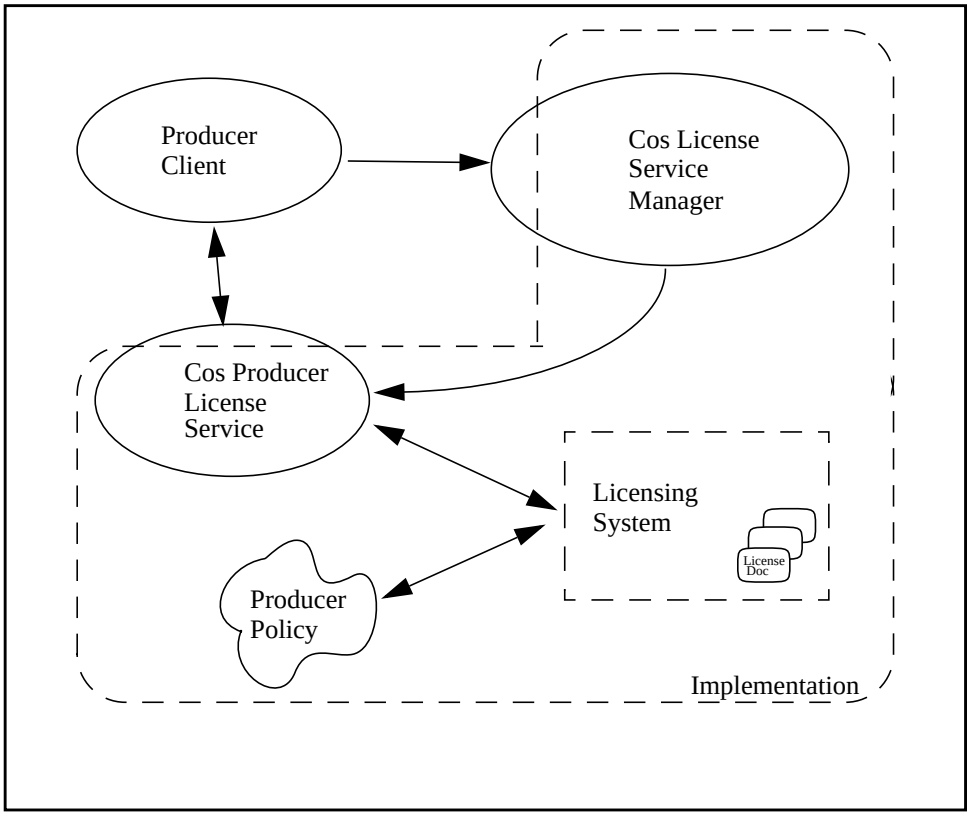

Figure 1-1 Licensing Service Relationships

The Licensing Service provides a mechanism for producers to control the use of their intellectual property in a manner determined by their business and customer needs. In Figure 1-1, the Licensing Service Manager, Producer Licensing Service, and the Licensing System are shown as three distinct objects. Implementations of the Licensing Service may differ. The dotted line indicates components that depend on the implementation design and are addressed in terms of an example solution. Components outside the dotted line are addressed in this specification.

1.3 Key Components of a Licensing System

1.3.1 License Attributes

To implement controls, the Licensing Service needs a set of fundamental attributes. A license can be thought of as having three dimensions of attributes:

Time includes, but is not limited to the attribute of Expiration/Duration. All licenses should be able to have start/duration and expiration dates.

Value Mapping includes, but is not limited to, the following attributes:

- A *unit* is a quantity that can be used by policy mechanisms.

- *Allocative*. Use of an license with an allocative attribute removes it from the pool of available allocative licenses for a given product until it is returned. This is traditionally known as concurrent use licensing.
- *Consumptive*. Use of a license with a consumptive attribute permanently records its use. This can be used to provide metering capability. It can also be used to implement a “grace period” via the use of overflow licenses when the maximum number of allowed concurrent licenses has been met.

Consumer includes, but is not limited to, the following attributes:

- *Assignment or Reservation.* All licenses should be able to be assigned to or reserved for a specific entity or collection of entities. The definition of what an entity may be is implementation-specific. One example is where an entity is defined to be a specific user and a collection of entities is a specific organization comprised of a collection of specific users. Other examples of what an entity might be include a specific machine or collection of machines, a specific system resource or resource collection, such as printers and adapters.

1.3.2 Licensing Policy

The Licensing Service allows the license attributes to be combined and derived from to form any policy deemed necessary. This allows the producer and, where appropriate, the end user administrators to control product use to fit their business environmenst.

The following derived attributes are representative examples of those that can be used for a flexible policy implementation:

- Time windows
- Value
- Use by a collection of related objects
- Postage meter
- Gas meter

Time Windows

It may be necessary for some policies to constrain the time periods within which a particular license unit may be used. A time window attribute can be derived from the expiration/duration attribute.

Value

A Producer can define, as part of their Producer Policy, the mapping between actual use of their intellectual property and the way license units are associated with that use in the Licensing System. A simplistic example might be where a single unit of control represents a single active implementation of a given object with no constraints on the number of instances. A more complex example may be where the number of units of control required may be calculated to satisfy a combination of requirements: a specific machine size where an implementation is active, how many instances, and how many method activations are allowed in parallel.

Use By a Collection of Related Objects

The definition of granularity is very broad. In an OMA-compliant system, the Licensing Service will allow control from the fine grain of a method activation to the coarse grain control of a suite of objects acting together in a relationship to represent an application. The relationship may be defined with the Relationship Service, a future Collection Service or any other Service providing relational capability for objects. The Producer Policy can discover all theseobjects according to the implementation.

Postage Meter

Derived from consumptive, use of a license with a postage meter attribute permanently removes it from the pool of available licenses. The total number of licenses is never less than zero (0) for any product.

Gas Meter

Derived from consumptive, use of a license with a gas meter attribute adds to the pool of consumed licenses. The total number of licenses is initialized to 0.
Examples of how these attributes can be used in license policy are as follows:

- An end user administrator could be empowered by the Licensing Service to combine assignment and time constraints on installed license units to constrain the use of certain products to a set of individuals outside of the normal work week.
- A producer could provide a personal use license by combining an allocative attribute with an assignment attribute to an individual with a unit attribute of 1.
- A producer could enhance the previous example by allowing end user administrators to reassign the license to a particular group.

1.3.3 Interfaces Isolated From Business Policies

The Licensing Service interfaces are isolated from policy issues. The client interface only delivers notification that a producer wants some or all of the producer's intellectual property to be controlled reliably and securely. Once the notification is made, the Licensing Service can identify the appropriate policy.

For example, consider a producer who wants to restrict the activation of a particular method to a certain simultaneous number of users. The producer need only tell the Licensing Service interface to indicate that a method has been activated and who activated it. When the method activation is complete, an indication must be sent that the use is done. The LS can then, in an implementation-specific way, determine if a producer-defined limit has been met. The Licensing Service can notify the object, telling it what to do if a producer policy is activated from overuse or another condition. The Producer can still override a generic policy with an alternate behavior for a particular Producer Client , since policy responses are inside the Client implementation.

A Producer Policy implementation requires the use of other object services such as the Relationship and Property Services. As other services are defined, producer policy implementations will broaden to use them. The producer client might change to

address any new producer policy, but the underlying Licensing Service interface will not require change. These services can be used to find out about objects outside of the objects themselves.

For example, consider the Relationship Service. If producers choose to license a particular set of their objects that are related in a manner defined by the relationship service, the producer policy implementation can obtain relationship information using the relationship service. The objects involved need to have no special knowledge about their relationships to one another other than that required to conspire together in the relation to achieve their desired functionality. Mechanisms provided to support this by an particular implementation will vary. One implementation may choose to support this using a document style of policy delivery, others may support producer policy object implementation. This can not be defined or restricted by the Licensing Service client interface.

A mechanism for license document delivery is not defined in the Licensing Service: it is implementation-specific.

1.4 Licensing in the CORBA Environment

Licensing in the CORBA world faces many issues. The provision of services by objects in the ORB environment must allow for service producers to control use of their intellectual property according to their business models.

Constraint of use must range from strict control to benign monitoring of intellectual property. Strict control might allow only a specific number or combination of services to be used. Benign monitoring mechanisms might allow service use without constraint, but would track usage for later examination.

If producers require strict control, they will also require assurance that the information provided by their licensing mechanism is secure. It would be pointless to choose strict control if it were a trivial matter to replace some component within the ORB which nullified strict control enforcement without the producers' services being aware of it. The level of trust in the Licensing Service must meet the producer's chosen enforcement policy. For example, suppose a producer has selected a policy that allows use of his object service by an end user without constraint, but the policy requires the Licensing Service to log all service usage so a monthly post-facto charge can be made for use of the service. This capability is of limited use if the Licensing Service's logging mechanism allows end users to illicitly modify the logs to show low usage.

To enable usage control, there must be a mechanism that provides the end user with appropriate authorization. This authorization is usually conveyed as a text string that can be thought of as a License Document. The size of this document may vary from a few tens of characters to a few thousand characters depending on the functionality provided by the underlying Licensing Service. The content of the document must be protected by an implementation-specific mechanism.

To support a wide variety of business models, producers require usage constraint policies (producer policies) that can vary for end user conditions. For example, a producer might deliver a demonstration of a client service that allows unlimited use of the service during the demonstration period, but upon purchase requires a strict usage

enforcement policy. The enforcement policy may need to be varied depending on customer needs. A large customer may negotiate a post-sale period where analysis of use is supported by benign monitoring and later moved to strict enforcement. Interfaces to the Licensing Service allow this and many other varieties of usage controls without requiring changes to the producer's fundamental product.

The ability for an end user to apply constraints beyond those specified by a producer is a well-recognized benefit to the end user. The capability in this area will vary across implementations of the Licensing Service.

Because we live in a dynamic economic environment, a producer's policies must be easily changed. The best approach for a Licensing Service specification is to separate the "I want to be controlled!" requirements of the application or service from the "how am I to be controlled?" requirements of the policy that have to deal with all of the exceptions and producer business practices. This separation enables a producer to choose a Licensing Service implementation based on considerations of how well a specific Licensing Service supports the producer's business practices, as instantiated by the producer policies.

The interface to the Licensing Service accomplishes this by allowing the controlled applications or services to notify the Licensing Service of its wish to be controlled specifying how the enforcement is to be performed.

Administration and policy issues are not addressed in detail by the Licensing Service interface; instead, they are left to implementors. End users need to control their own interface and reporting capabilities. The ability of the underlying Licensing Service to generate management reports, both of historical and snapshot-of-time usage, will vary widely depending on the implementation. The administrative interfaces for the Licensing Services include command line only, GUI only, and combined GUI and command line. An administrative interface would affect the ability of end users to manage their environments as they choose, so it is not defined by the Licensing Service.

1.5 Design Principles

The design of the Licensing Service interface satisfies the following principles:

Neutrality. The Licensing Service should not introduce any constraint on the way a Producer can use the interfaces because of some underlying dependency on the LS implementation. Producers need to be able to choose Licensing Service implementations that allow them to deliver their products in a manner best suited to the individual Producer's business needs without requirements on the way the interface is used. It is expected that LS implementations will allow many Producer Client objects to reference a single instance of the associated **ProducerSpecificLicenseService** interface to reduce the overhead of object creation.

Extensibility. The Licensing Service allows for extensions to support styles of Producer Policies that are not currently obvious. The Licensing Service provides extensibility in its object reference in the returned Action structure in the check-use

operation. This allows implementation-specific extensions to the notification mechanism. The interface can also be extended by adding additional arguments and/or operations; for example, in support of the Security Service.

Security. The Licensing Service provides a mechanism such that a degree of trust can be established between the users of the interface (the Producers) and the underlying license management system. This is different from a typical secure environment since, the Producer does not usually trust the end user or the end user security environment. A mechanism is provided to allow the Producer to authenticate, in real time, that the underlying license management system is a legitimate provider of the Licensing Service. End user administration can not circumvent this authenticating mechanism.

Performance. Implementations of the Licensing Service may choose to optimize performance by the manner in which Producer Specific Licensing Service objects are managed. For example, an implementation could choose to allow multiple copies of a Producer Specific Licensing Service to distribute client operations.

The Licensing Service mechanisms must allow both synchronous and asynchronous messages so a producer can decide what is best for its application. For example, a very short duration method activation may well be best suited, for performance reasons, to using asynchronous meechanisms. On the other hand, if producers want to be extremely strict, they might choose synchronous messages to prevent misuse and accept the resulting loss of performance.

The Licensing Service provides mechanisms so that an application using the Licensing Service cannot accidentally orphan a license by acquiring an allocative style of license and never releasing when an application fails. Current mechanisms include connection-oriented, client-server communications; client-server heartbeat mechanisms; and server-based, client status query mechanisms. Keep in mind that the mechanism chosen may place a performance burden on the producer client.

Licensing Service Interfaces 2

Contents

This chapter contains the following topics.

2.1 Licensing Service Interfaces

The Licensing Service defines the interface between the Producer Client and the Licensing Service Manager (**LicenseServiceManager** interface) and the interface between the Producer Client and the Producer Licensing Service (**ProducerSpecificLicenseService** interface). The interfaces enable Producers to control use of their intellectual property in any manner they deem appropriate for their business model. The isolation of policy from the Licensing Service interfaces enhances Producer flexibility. The interfaces for administration, policy creation, and license document creation are not addressed, because they are implementation-specific.

The **LicenseServiceManager** interface provides a mechanism for the Producer to locate an object supporting the second interface, **ProducerSpecificLicenseService**. All of the operations required to constrain use of producers’ intellectual property are supported by the second interface. This design allows the implementors of Licensing Services to make trade-offs such as those between client performance, licensing system performance, and ease of administration.

Once a Producer Client implementation has obtained a **ProducerSpecificLicenseService** object reference, the three operations (**start_use**, **end_use**, **check_use)** can be performed on this interface within the

Client where the Producer deems it correct. The information returned from these operations provides the basis for the Producer to enforce its chosen usage constraint policy.

2.1.1 Interfaces are Mandatory

All the interfaces are mandatory for all implementations. Optional arguments exist in the **LicenseServiceManager** interface. For the **check_use** operation the **ProducerNotification** component of the returned Action can be a nil object reference indicating that the implementation does not support this kind of notification mechanism. In the **start_use** operation the **call_back** argument can be a nil object reference indicating that the Producer client implementation is not using event services and is designed to operate in a poll only mode. The properties argument to **start_use**, **check_use**, and **end_use** can be nil.

2.1.2 Constraints on Object Behavior

The Licensing Service interface assumes the provision of an Event Service . If an Event Service implementation supports true asynchronous events—where delivery of an event can interrupt an object's task to invoke the push operation—then the Producer Client implementation must manage its internal state in a re-entrant world.

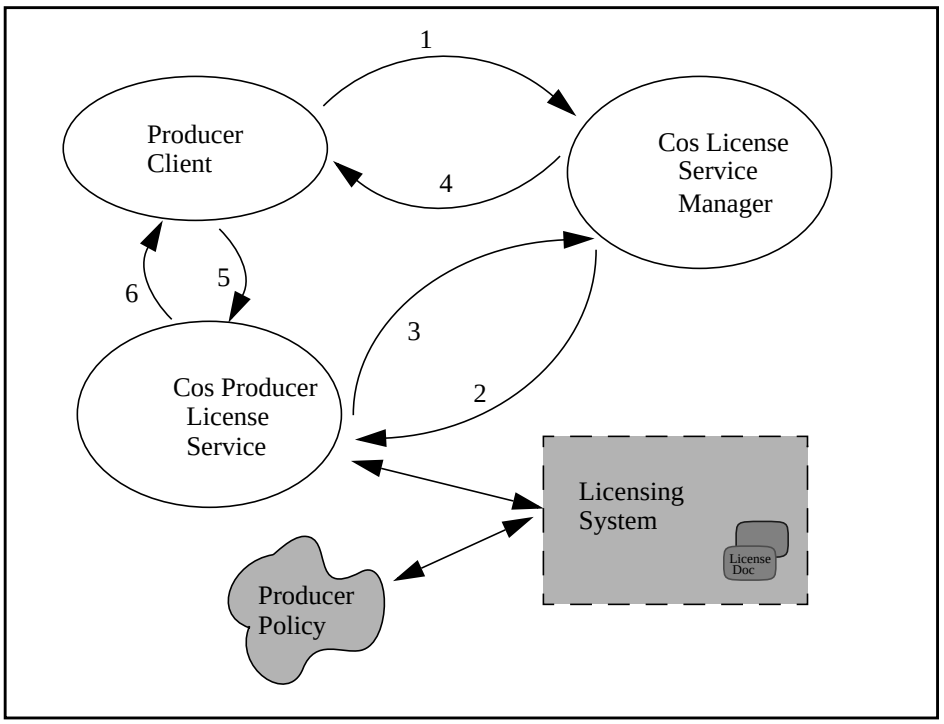

Figure 2-1 Licensing Service Instance Diagram

In Figure 2-1, the Producer Client performs the operation **obtain_producer_specific_license_service** on the **LicenseServiceManager** interface (Step 1). The Licensing Service Manager implementation creates an object (Steps 2 and 3) or locates an object reference to an object who has an interface **ProducerSpecificLicenseService** and who is capable of responding to the particular producer challenge. It then returns the reference to the Producer Client (Step 4). The producer client now uses the reference to perform the operations **start_use**, **check_use**, and **end_use** (Steps 5 & 6). In implementations that support true asynchronous events, the **ProducerSpecificLicenseService** object can asynchronously perform the push operation using the reference to the interface in the Producer Client provided as one of the arguments to the **start_use** operation in a previous step (in Step 5).

2.1.3 Licensing Event Trace Diagram

Figure 2-2 on page 2-4 represents the flow of events through Producer Client objects and a Licensing Service implementation. The steps below are illustrated in the diagram.

1. Producer Client gets an object reference to the Producer Specific Licensing Service.
2. Producer Client determines that usage control is required and performs the **start_use** operation.
3. Producer Client does an intial **check_use** call to retrieve the initial **recommended_check_interval**.
4. Producer Specific Licensing Service instance interprets policy and interacts with the Licensing System as necessary.
5. If asynchronous events are supported, the Producer Specific Licensing Service asks for event notification to the particular Producer Client at an interval determined by Policy.
6. Event Service delivers the event to the Producer Client.
7. Producer Client responds to the event by performing the **check_use** operation.

Steps 4,5,6,7 are repeated until the Producer Client instance indicates that usage control is no longer necessary.

Producer Client performs the **end_use** operation when usage control is to be terminated.

If asynchronous events are not supported, the Client implementation will need to “poll” the Producer Specific Licensing Service with the **check_use** operation at an interval defined by the **check_interval** argument to the **check_use** operation. To initially retrieve this **check_interval** value, the Client will need to invoke a **check_use** immediately after the **start_use** call.

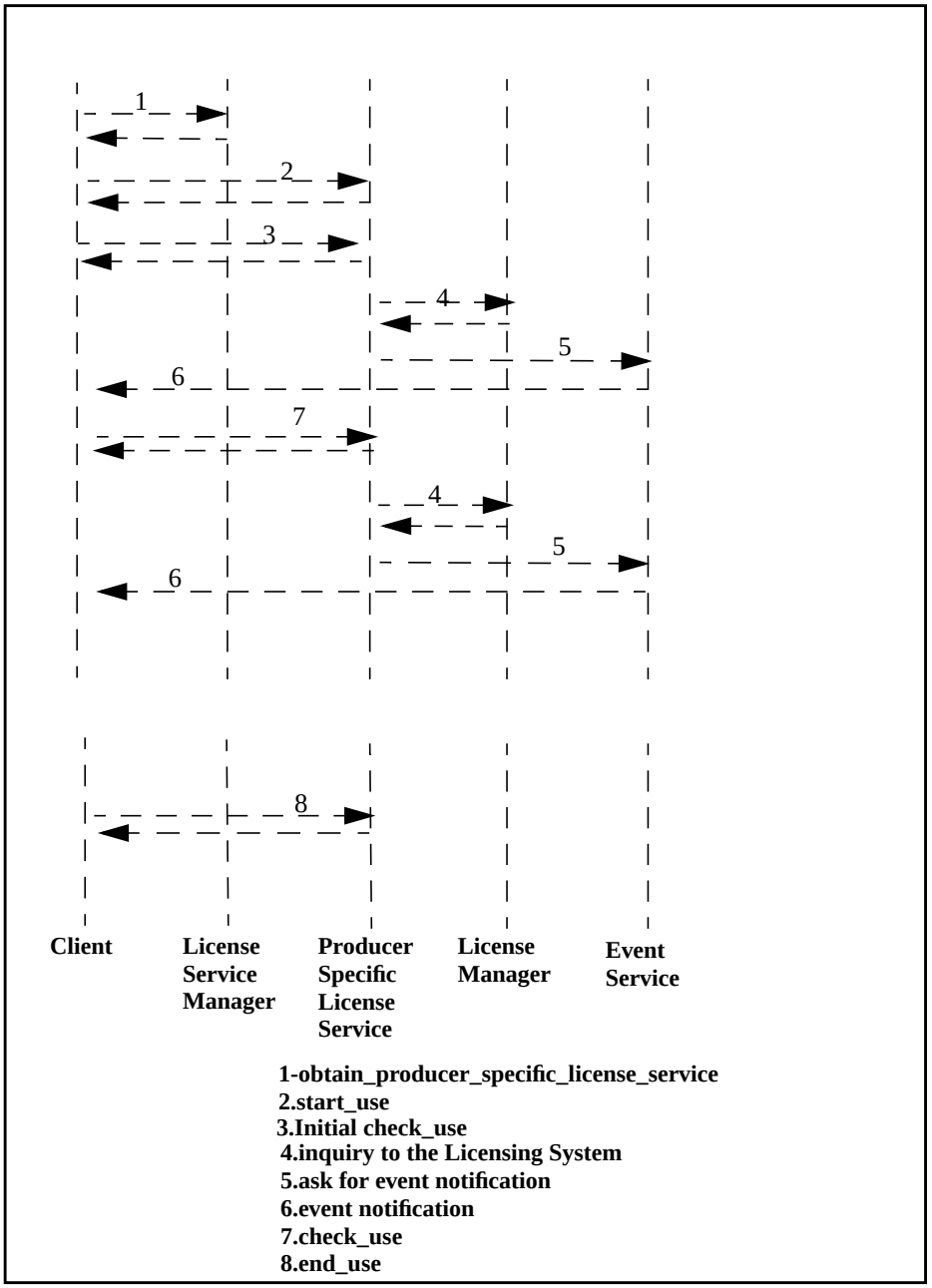

Figure 2-2 Licensing Event Trace Diagram

2.2 The CosLicensing Module

The **CosLicensing** module is a collection of interfaces that together define the Licensing Service. The module contains two interfaces:

The **LicenseServiceManager** interface consisting of the following operation:

- **obtain_producer_specific_license_service**

The **ProducerSpecificLicenseService** interface consisting of the following operations:

- **start_use**
- **check_use**
- **end_use**

This section describes the **LicenseServiceManager** and **ProducerSpecificLicenseService** interfaces and their operations.

The **CosLicensing** module is shown below. Note that this module definition uses some definitions from the **CosEventComm** module (in the Event Service) and the **CosPropertyService** module (in the Property Service).

```
#include “CosEventComm.idl”
#include “CosPropertyService.idl”

Module CosLicensingManager {
    exception InvalidProducer{};
    exception InvalidParameter{};
    exception ComponentNotRegistered{};

    typedef Object ProducerSpecificNotification;

    enum ActionRequired { continue, terminate};

    enum Answer { yes, no };

    struct Action {
        ActionRequired action ;
        Answer notification_required ;
        Answer wait_for_user_confirmation_after_notification ;
        unsigned long notification_duration;
        ProducerSpecificNotification producer_notification;
        string notification_text;
};

    struct ChallengeData {
        unsigned long challenge_index;
        unsigned long random_number;
        string digest;
    };

    struct Challenge {
        enum challenge_protocol { default, producer_defined };
        unsigned long challenge_data_size;
        any challenge_data;
    };

    typedef any LicenseHandle;
```

```
        interface ProducerSpecificLicenseService {

readonly attribute   string producer_contact_info
readonly attribute   string producer_specific_license_service_info

    LicenseHandle start_use (
        in Principle principle,
        in string component_name,
        in string component_version,
                inProperty::PropertySet license_use_context,
            CosEventComm::PushConsumer call_back,
            inout Challenge challenge
                )

     raises ( InvalidParameter, ComponentNotRegistered);

        void check_use (
            in LicenseHandle handle,
                    in Property::PropertySet license_use_context,
                out unsigned long recommended_check_interval,
                out Action action_to_be_taken,
                inout Challenge challenge
                    )

             raises ( InvalidParameter );

        void end_use (
                in LicenseHandle handle,
           Property::PropertySet license_use_context,
                inout Challenge challenge
            )

          raises ( InvalidParameter );

    };

    interface LicenseServiceManager {
    ProducerSpecificLicenseService
            obtain_producer_specific_license_service (
                in string producer_name,
                inout Challenge challenge
              )

              raises ( InvalidProducer, InvalidParameter };
    };
};
```

Table 2-1 Exceptions Raised by Licensing Service Operations

Exception Raised	Description
InvalidProducer	Indicates that the producer argument is not correct or that an appropriate producer cannot be found.
InvalidParameter	Indicates that one of the parameters is invalid. No additional detail is provided in this document since this will include a failed challenge. Additional information could assist if someone wanted to make a deliberate attempt to work out the challenge of a producer.
ComponentNotRegistered	Indicates that the specific component has not been registered with the Licensing System.

2.2.1 LicenseServiceManager Interface

The **LicenseServiceManager** interface defines a single operation: obtaining the producer specific Licensing Service object.

The **LicenseServiceManager** interface allows a producer to control the use of their intellectual property. The **obtain_producer_specific_license_service** operation returns an object reference that supports the **ProducerSpecificLicenseService** interface. This operation is protected by the use of a producer challenge.

It is likely that implementations of the **License ServiceManager** will make use of other Object Services, such as Life Cycle, to create a producer-specific instance of the Licensing Service. The Life Cycle Service is not used directly in order to allow the service implementation to cache object references for performance reasons. Requiring instance creation on every use of the **obtain_producer_specific_license_service** operation is not desirable, but can be allowed in a particular implementation.

The operation **obtain_producer_specific_license_service** raises the InvalidProducer and InvalidParameter exceptions.

2.2.2 ProducerSpecificLicenseService Interface

The **ProducerSpecificLicenseService** interface defines three operations:

1. notification that a product has started to be used,
2. notification that a product is still in use, and
3. notification that a product has finished being used.

Any object that possesses an object reference that supports the **ProducerSpecificLicenseService** interface and is capable of satisfying the challenge for that particular instance of the **ProducerSpecificLicenseService** interface can perform the following operations:

- The **start_use** operation which allows producers to notify the License Service that some aspect of their product has started to be used and is to be controlled by the service.
- The **check_use** operation which allows the producers to notify the Licensing Service that some aspect of their product that previously notified the service using a **start_use** operation is still in use.
- The **end_use** operation which allows the producers to notify the Licensing Service that an aspect of their product, previously notified to the service in the **start_use** operation, has completed its use.

All of the previously listed operations are protected by a challenge mechanism to allow a producer to be satisfied that the instance of the Licensing ServiceManager is a legitimate one to control the producer's intellectual property.

The attribute **producer_contact_info** may be used to provide information that can be displayed to an end user. The attribute **producer_specific_license_service_info** can be used, if necessary, for a Producer Client to alter the way it interacts with different **ProducerSpecificLicenseService** objects. These attributes are defined at creation of the **ProducerSpecificLicenseService** instance and do not change during the instance's life.

The **start_use**, **check_use**, and **end_use** capture and propagate information about the user's runtime context to the Licensing Service via the **license_use_context** parameter. This information will typically include the user's name, their node's name, network address, local time, and so on. This information can then be used by the License System for a variety of purposes:

- In an access control mechanism to determine whether or not to allow the user to continue.
- In a private, possibly secure, usage logging mechanism.
- To provide data for peripheral management functions, such as triggering an e-mail message to the network administrator when resources run out.

The operations **start_use**, **check_use**, and **end_use** raise the InvalidParameter exception.

The **action_to_be_taken** output parameter in the **check_use** operation is used to give the **ProducerClient** information on actions to be taken as a result of its request to be active or running. The following describes the Action structure in more detail. Note that only the **action** field must be specified. All other fields can return a value of NULL in which case behavior is determined by the coded policy defined within the **ProducerClient** implementation.

- **action** : This field indicates if the **ProducerClient** should continue or terminate its processing depending on whether the requested license is available from the Licensing System.

- **notification_required**: Indicates whether or not the **ProducerClient** needs to prompt the local user with a message indicating the results of the licensing request.
- **wait_for_user_confirmation_after_notification**: Indicates whether the **ProducerClient** needs to wait for a confirmation before continuing its processing. This is applicable only if a notification has been requested.
- **notification_duration**: If the user notification is required without confirmation, this indicates how long the **ProducerClient** needs to wait before continuing with its processing.
- **producer_notification**: This provides a reference to an object used by a Licensing System to return implementation specific results and control information to the **ProducerClient**. For example, producer policy instructions can be part of this object interface. It could also communicate the expiration date and time.
- **notification_text**: This provides the text to be communicated to the local user if required.

The **check_use** operation thus collects into a single client **action** the ability to address the following requirements:

- Give the capability to the producer client to get both the results from and the actions to be performed following a request for permission to be active and/or running.
- Give the capability to the producer client to periodically verify the right to be active and/or running in the case of 'time dependent' licensing policy (for example, time based consumable licenses, expiration times, and so forth). The **recommended_check_interval** is the parameter strictly tied to this verification.
- Give the capability to both the producer client and the Licensing Service implementation to detect the following *unexpected* conditions and then either release the related active license and/or stop the usage accounting:
  - Abnormal termination of either the producer client or the Licensing Service.
  - Unrecoverable breakdown in communication between the Producer Client and the Licensing Service.
  - The indirect detection of these conditions is performed by forcing the producer client to issue a check request within the check interval.

The check request concept is left to the specific Licensing System implementations. However, that does not prevent the Licensing Service from using the check operation as the heartbeat mechanism. The heartbeat mechanism is a general purpose mechanism required inside a client/server based application to determine if the other end is still active. Some applications dedicate a specific process or task to this purpose and rely on event detection, others use a polling mechanism, others use system notification exits, and so on. Furthermore, because of the different concepts, the polling and exits could not be fully satisfied by a single checking rate.

Glossary

License Document: Represents the fundamental element of control. It provides a secure delivery vehicle describing such things as how many copies of the intellectual property are allowed, how long each copy may be used, and other elements of how producers wish to constrain usage of their intellectual property.

Licensing Service: The general term for the complete service, it consists of three components: Producer Client; Producer Licensing Service; and Licensing Service Manager.

Licensing Service Manager: The Common Object Service Licensing Service Manager is responsible for managing and creating the Producer Licensing Service objects.

License Unit: License documents may contain the concept of license units that are interpreted in a producer-specific manner by the producer policy. A typical example of a license unit could be one where a single unit is to represent a single concurrent use of a producer's intellectual property by an individual user. The term license can be used to refer to the smallest indivisible quantity of license units that a given Licensing System implementation supports.

Licensing System: The implementation-specific component that provides fundamental usage control that, in conjunction with the Producer Licensing Service, provides sophisticated producer policies. The Licensing System is responsible for securely managing the fundamental units of control - the License Documents for all Producers.

Producer: The company or individual who owns the intellectual property that requires usage control.

Producer Client: Any object, or component of an object, that wants to have its usage controlled or metered via a Licensing Service.

Producer Policy: A Producer Policy is a collection of data that describes the detailed terms and conditions, or business policies, which govern control and monitoring of a producer's intellectual property wherever the property can be used. The implementation of producers' policies is very specific to the Producer's selection of a

Licensing System. There are two components to business policy implementation in a licensing system. One component is contained in the License Document and includes fundamental things like expiration date and quantities. The other component, the Producer Policy, includes the broader aspects of business policy and may be derived from the License Document. As an example of the broader issues that require Policy, the Producer Policy deals with all possible licensing exceptions such as when no license is found.

Producer-Specific Licensing Service: A producer-specific implementation that interacts with and selects the particular Licensing System and Policy used by a specific Producer to control the Producer's intellectual property. In this chapter, the Producer-Specific Licensing Service is is also referred to as the Producer Licensing Service.

References *A*

Object Management Group. *Object Services RFP 4*, OMG Document Number 94.4.18, May, 1994.

A

Use of Other Services *B*

This appendix describes the relationship between the Licensing Service and these Object Services: Property; Relationship; and Security.

B.1 Property Service

The properties argument to the `start_use`, `check_use` and `end_use` operations enables implementations to choose between using the Property Service or providing name value pairs directly to the Licensing Service. This decision can be based on performance considerations or other practical concerns. For example, the inability to differentiate ownership where a single property is used in a single operation (method) but has differing values (as far as the Licensing Service is concerned) because more than one principal is using the particular instance's method at one time.

Examples of properties that are useful:

- UNITS_TO_RESERVE provides a hint to the producer policy implementation indicating that the currently controlled aspect of the producers intellectual property has some idea about what it is going to 'use' over the next amount of time.
- VALUE_TO_CONSUMER provides a hint to the producer policy implementation indicating that the currently controlled aspect has some idea of the value of what it is currently doing.
- NODE_NAME provides a hint to the producer policy implementation about where the currently being controlled object is executing.

These are currently always producer-specific. The Licensing Service places no semantic or syntactic interpretation on these properties but makes them available, in an implementation-specific way, to the producers policy.

B.1.1 License_Use_Context

There will need to be a set of information about each producer client made available to the *ProducerSpecificLicenseService* as a "PropertySet" as specified by the Property Service. The PropertySet is a dynamic equivalent of CORBA attributes. This set of information is made available to the *start_use*, *check_use* and *end_use* operations for the Licensing System to use in determining various aspects of policy. As one example, this data structure could contain:

- All data from the *principal*, as retrieved through the new context information provided by the *CORBA 2.0* specification and as used, for example, by the Transaction Service.
- Any data the producer client may need, either in the present or the future. Being all inclusive early on reduces the need to re-deploy the licensed software if subsequent licenses need additional data.
- Fields from the example list of licensing attributes (provided below.)

The example list is useful to allow people other than the original producer to create license documents for an object implementation. This happens in the case of either acquisitions or distribution agreements. The example list makes it easier for one object implementation to be licensed by multiple license systems depending on the environment in which it finds itself.

The list items are suggestions. Currently, no central registry of names exists; also, many items are not clearly defined. The list is a starting point and can serve as a check list for Producers.

Canonical List of user_context Properties:

- DATE_TODAY
- Today's date and time.
- GROUP_ID
- Integer group
- ID GROUP_NAME
- Name of group of users
- HARDWARE_FAMILY
- String of compatible hardware family
- HARDWARE_MODEL
- Hardware model
- HARDWARE_PRODUCER
- Manufacturer name
- NETWORK_ID
- Integer network identifier
- NETWORK_NAME
- String network identifier
- NETWORK_PROTOCOL
- String protocol name, for example, "TCP/IP" or "DECnet"
- NETWORK_STYLE

- 1 is local, 2 is LAN, 3 is WAN.
- NODE_ID
- Integer node identifier
- NODE_NAME
- Name of computer
- OPERATING_SYSTEM
- String identifying the OS
- OS_VERSION
- String identifying the OS version
- PROCESS_FAMILY
- String identifying a group of related processes
- PROCESS_ID
- Integer identifying a process number
- PROCESS_NAME
- String identifying the name of the process
- PROCESS_TYPE
- 1 is batch, 2 is interactive, 3 is other
- PRODUCT_NAME
- Name of intellectual property being protected
- PRODUCT_PUBLISHER
- Owner of intellectual property being protected
- PRODUCT_VERSION
- Version string of intellectual property
- PUBLIC_KEY
- String containing public key to test against Product
- RELEASE_DATE
- Integer indicating the date the software was released
- USER_ID
- Integer indicating user
- USER_NAME
- String containing user name

B.1.2 Dependent Licenses

The Licensing Service can examine not only the most recent set of user runtime environment data but it can also examine data from previous runtime contexts collected along a particular thread of control. For example, a user may log in as "Fred" and begin some action under that name. This action may include an operation being dispatched to an object implementation logged in as "root". If this second process needs to obtain a license which was reserved for "Fred" then it ought to be able to do so. The user should be known by all the names associated with that thread of control.

Another example of a recursive license right is the "embedded" license. Such a license is not valid unless another object implementation was used earlier on the thread of control. A database software vendor might issue License Documents for use within, say, an accounting package. Other uses which might be worth more must be licensed separately. An example of an interface which would support a stack of License Use Context is as follows:

```
interface UserContext {
   Property::PropertySet  License_Use_Context  create ( );
   void push( in Property::PropertySet License_Use_Context);
   void pop ();
   unsigned long getDepth ();
   Property::PropertySet License_Use_Context   top ();
   Property::PropertySet License_Use_Context get (in unsigned long
which_frame );

   void clear ();
void remove ();
}
```

B.2 Relationship Service

Support for collections and relationships will be determined by the mechanisms made available to producers by the particular implementations of the Licensing Service. It is expected that the preferred mechanisms will be to allow the Producer Policy to make use of Object Services such as the Relationship and Property Services, but this is not a requirement of the Licensing Service.

Each implementation of the Licensing Service can address the problem of how to manage the relationships among licenses. The types of relations one can assume exists among licenses can be generically classified as follows:

- Prerequisite licenses, for example. the previous example of a database vendor.
- Corequisite licenses, that is, a set of licenses which must all coexist to give the producer client the right to be running.
- Exrequisite licenses, that is, a set of licenses that can run only if others are not active.
- Generic dependent licenses, that is, a set of licenses whose dependencies are described through a specific constraint expression.

B.3 Security Service

The Security Service will probably replace the logic in each Licensing System that deals with producer client authentication and access control.

Implementation Issues C

C.1 Producer Client Implementation Issues

C.1.1 Client Implementation

In this example, a Producer decides to control method activation. In the Producer's object activation, the implementation performs the obtain_producer_specific_license_service operation on the *LicenseServiceManager* interface and stores the resultant object reference. In the implementation of each method that is to be controlled, the start_use operation is performed on the stored object reference.

Depending on whether asynchronous events are supported, the Producer implementation will vary as follows:

- If true asynchronous events are supported, the Producer implementation needs to provide an interface inherited from CosEventComm, the *PushConsumer* interface.
- If asynchronous events are not supported, or the Producer chooses to not use events, then each implementation that uses the start_use operation needs to use the check_use operation no less frequently than the period specified in the recommended_check_interval argument until the implementation performs an end_use operation. If, within the recommended check interval, the Producer Client does not perform the check_use operation, the Producer Policy may choose to release the associated licenses assuming that the Producer Client has ceased functioning.

Producers must decide how they want to use the Property Service to provide properties to the start_use, check_use and end_use operations. In the Producer implementation, the returned argument action_to_be_taken from the check_use operation needs to influence how the object continues after each check_use operation.

The Producer needs to determine the name for each component and the version for each component. The Producer will then need to produce the Licensing System implementation dependant policy and license document for the Producer's chosen policy.

When a particular use of the Producer object is completed the end_use operation is used to let the Licensing Service know that control is no longer required for that component.

C.2 Asynchronous Events

In CORBA implementations where true asynchronous events are supported, provision is made in the start_use operation to provide the Licensing Service with the object reference that corresponds to a client *PushConsumer* interface. This will allow the license service to asynchronously send a push event to the specified interface with the arguments defined in the following pseudocode:

C.3 Pseudocode

```
struct AsyncLicenseData{
    ProducersSpecificLicenseService service;
    LicenseHandle handle;
    Challenge challenge;
};

/* Producer client implements an interface for the 'push' operation:
*/

void xxxx_push(Object o, Environment *e, any data)
{

    struct AsyncLicenseData *check;


    /* get the actual information that is needed to proceed */

    check = (struct AsyncLicenseData *)(data->_value);

    /*
    perform producer specific testing and lookup on:
```

```
        check->handle
        check->challenge

    need to make sure that the component of this instance
    that handle refers to is still active and that the
    challenge is valid.
     */
     /*
       providing all is well, cause a check_use operation for
       the handle. Have to assemble the challenge, decide which
       properties are important for this handle and so forth.
     */

    check->service->check_use(ev,
                        check->handle,
                        properties,
                        interval,
                        action,
                        challenge);

    /* test the challenge returned and so forth */

}
```

When the Producer Client has the push operation invoked, activating the routine xxxx_push in the pseudocode example, the producer implementation should determine which aspect of the implementation is referenced by the handle argument and then invoke the check_use operation on the handle provided as one of the arguments to the push operation. At thar point, the implementation should determine if the object related to the handle is still active; determine if the challenge is valid; and then perform the check_use operation on the provided object reference. The results from this operation will indicate whether any action is to be taken and, if so, the implementation should proceed according to the Producer Policy.

Challenge Mechanism D

D.1 Default

For a producer to verify that a particular instance of the *LicenseServiceManager* is legitimate, a challenge mechanism is required. This requirement may either disappear or be reduced if the Security Service delivers a similar mechanism that can then be inherited by the *LicenseServiceManager*.

The mechanism proposed, by default, assumes the use of *shared secrets* in the producer implementations of their objects and the specific instance of the Licensing Service that is involved to control the producer's intellectual property.

The challenge mechanism is straightforward. When any operation is requested by a producer's instance a challenge structure is provided along with the normal parameters. This challenge structure consists of the MD5 of all the arguments to the operation, a random number, and a *forward secret value* known only to the producer. The Licensing Service instance for this producer can confirm that the client instance is legitimate by verifying that the challenge is correct. In return the instance of the license system sends back the MD5 of the same random number and a *reverse secret value* again known only to the producer. The instance invoking the operation on the Licensing Service can verify that the Licensing Service is legitimate by validating the generated MD5.[1] The challenge mechanism defined in the proposed interfaces supports more than one set of shared forward/reverse secrets. As part of the ChallengeData structure

1. MD5 is a message digest algorithm defined by R. Rivest in the Internet RFC 1321. It is in the public domain and provides a mechanism to generate a 128-bit "fingerprint" of messages of arbitrary length. It is conjectured that the difficulty of coming up with two messages that have the same digest is 2^64 operations and that generating a specific digest for a message is 2^128 operations, making it suitable for the basis of the challenge protocol described in this specification.

an index is provided, challenge_index, that allows the client to choose which shared secret set is to be used in the challenge. A conforming implementation of the LS needs to support at least four sets of shared secrets whose indices are 0 through 3.

This mechanism is not intended to be completely secure. Instead, it provides trust between the producer and the producer-specific instance of the Licensing Service. Eventually, the Security Service will probably replace the need for the challenge mechanism.

D.2 Alternative

As an alternative to the default challenge, a Producer can choose to define its own challenge protocol. By setting the challenge_protocol enumerated element of a challenge to 'producer_defined' the definition of what the challenge element represents becomes the responsibility of the producer and not the Licensing Service directly. This will depend on the implementation of the Licensing Service, since the mechanisms available to the producer to support this are defined by the way a Licensing Service is implemented.

Note

If the object producer so chooses, the same program can be licensed by more than one Licensing System. It is simply a matter of who satisfies the challenge. In fact, the challenge mechanism supports as many Licensing Service providers as an object producer chooses to pick up. They can choose sets of challenge data to deal with particular providers and use a standard set of challenge data to get the first available service provider.

It is not guaranteed to be true that all object producers will use the same challenge mechanism. However, as long as the object producer chooses to use the default challenge, this will be the case. As soon as an object producer decides to use an alternate challenge, that will be defined by the license system provider. At that point, only that implementation of the Licensing Service can satisfy the challenge and remove the multiple service provider capability. Default challenge mechanisms **must** be supported; however, if licensing system providers offer an alternative, a producer need not use the default.