7.3 CORBA Event Service

1. Overview of CORBA Event Service
2. Communication Styles
3. Event Channel
4. Event Service Class Structure
5. An Event Channel Use Example

Appendix: CORBA Time Service

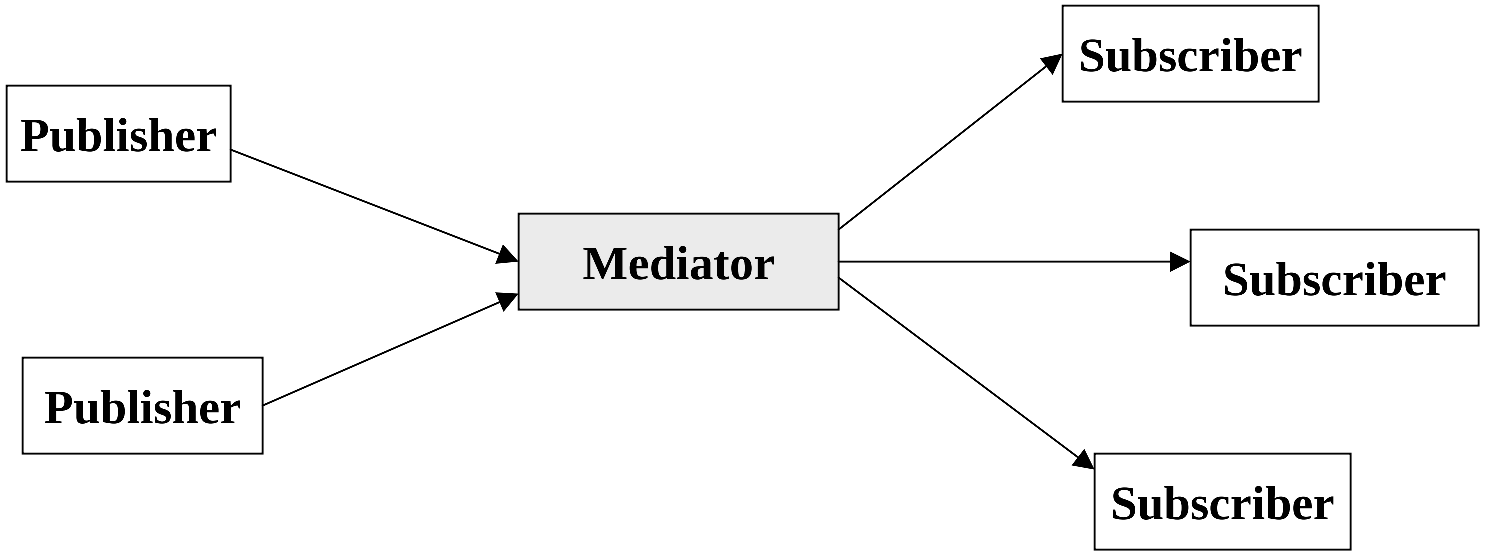
1. Overview of CORBA Event Services

-Standard CORBA method invocations result in synchronous execution of an operation provided by an object

÷For many applications, a more decoupled communication model between objects is required (i.e., asynchronous communication with multiple callers and receivers)

-CORBA standard defines a set of interfaces that enable decoupled, asynchronous communication between objects, called *event service*.

÷CORBA Event model is based on the observer pattern also known as "publish/subscribe" pattern.



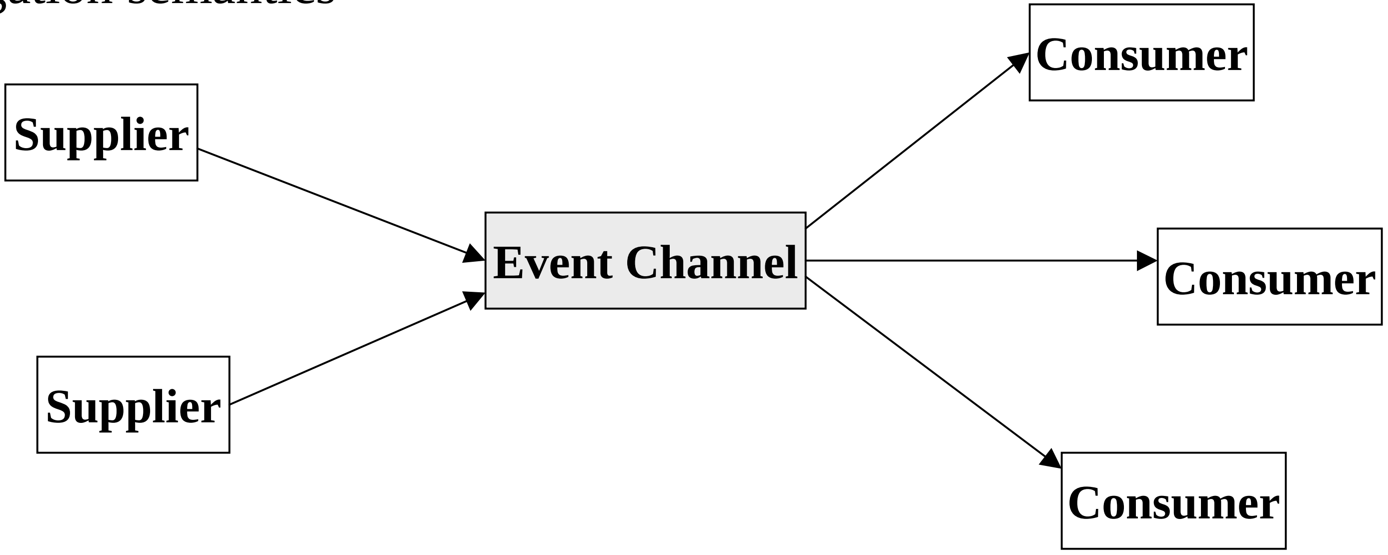
Event Service Participants

-The OMG event service defines three roles:

÷The Supplier role: Suppliers generate event data

÷The Consumer role: Consumers process event data

÷Event Channel: A "mediator" that encapsulates the queuing and propagation semantics

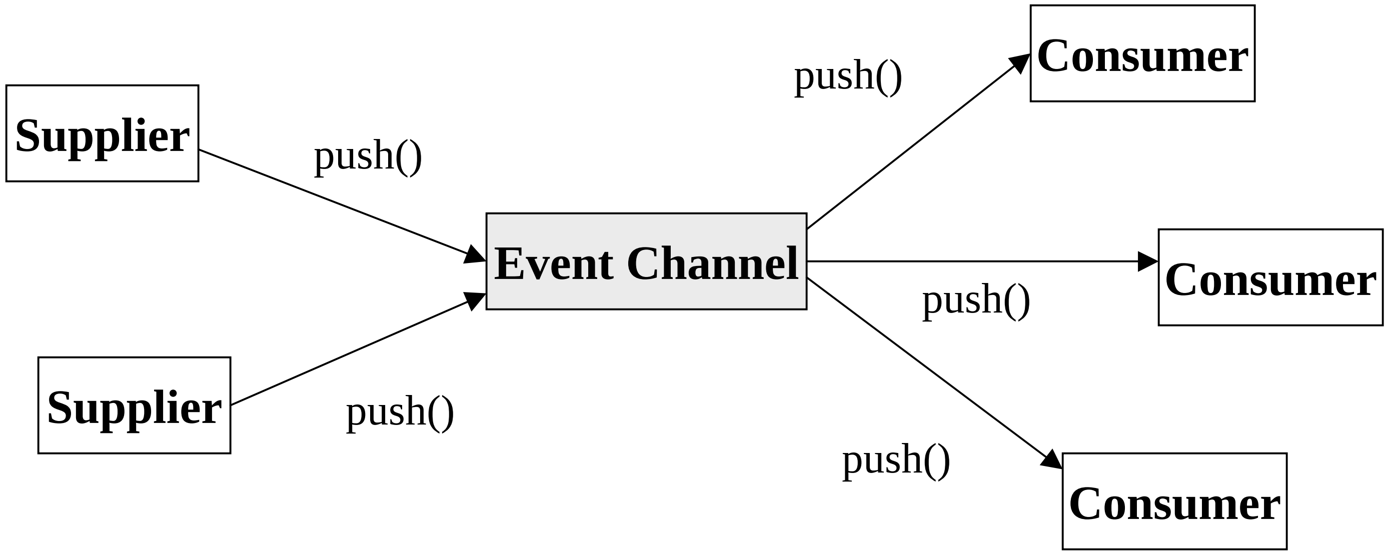


-Event data are communicated between suppliers and consumers by issuing standard CORBA (two-way) requests.

2. Communication Styles

-CORBA Event service supports ***push*** and ***pull*** communication models.

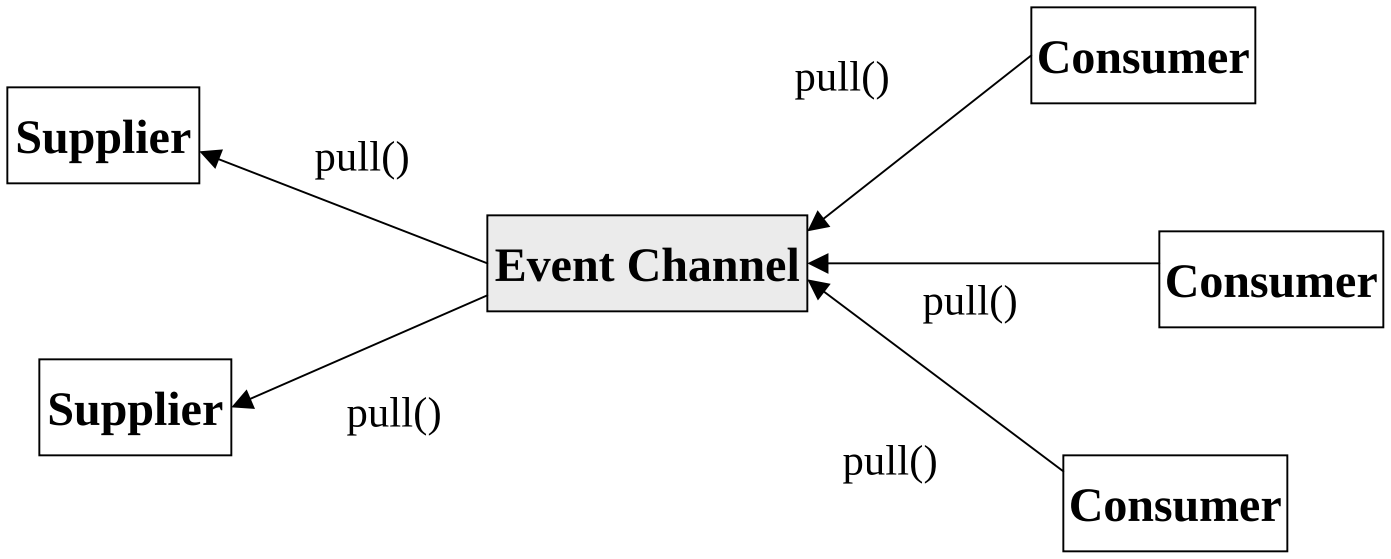
Push Communication Model



-The supplier pushes event data to an event channel

-The event channel, in turn, pushes event data to all consumers: note that an event channel need not make a complex routing decision, e.g., it can simply deliver the data to all consumers.

Pull Communication Model



-The consumer pulls event data from the event channel
-The event channel, in turn, pulls event data from the suppliers: this can be optimized by adding a queuing mechanism in the Event Channel

3. Event Channel

Overview

-The CORBA Event channel:

÷is an object that allows multiple suppliers to communicate with multiple consumers in a highly decoupled, asynchronous manner.

÷plays the roles of both a consumer and supplier of event data that it receives; it acts in its simplest form as “broadcast repeater”

÷is a standard CORBA object, and communicates with an event using standard CORBA requests.

÷However, an event channel need not supply the incoming event data to its consumer(s) at the same time it consumes data from its supplier(s) (i.e., it may buffer data).

Multiple Consumers and Multiple Suppliers

-An event channel may provide many-to-many communication
-The channel consumes events from one or more suppliers, and supplies events to one or more consumers
-Subject to the quality of service of a particular implementation, an event channel provides an event to all consumers

Mixed-style Communication with an Event Channel

-An event channel can communicate with a supplier using one style of communication, and communicate with a consumer using a different style of communication
-How long an event channel must buffer events is defined as a “quality of implementation" issue.

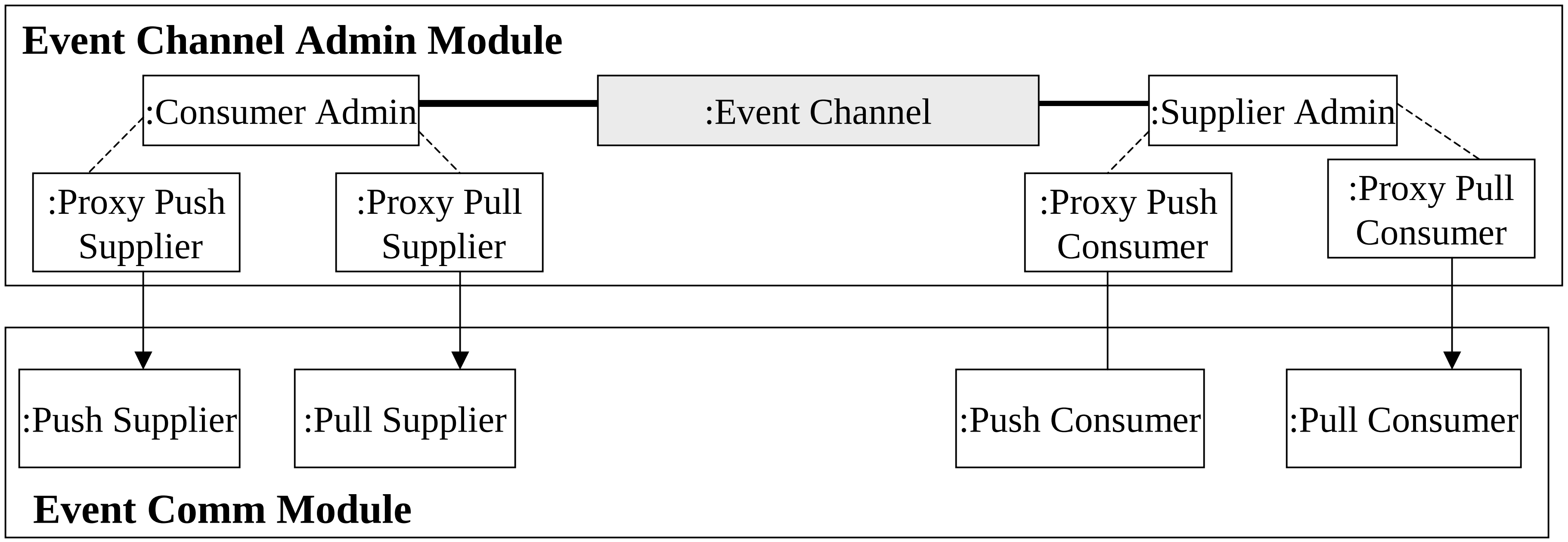
4. Event Service Class Structure

-There are two orthogonal approaches that CORBA event-based communication may take:

÷*Generic*: all communication is by means of generic push or pull operations; these operations involve single parameters or return values that package all the events into a generic CORBA ***any*** data structure.

÷*Typed*: communication is via operations defined in OMG IDL; Event data is passed by means of typed parameters, which can be defined in any desired manner.

Class Structure



The EventComm Module

-The event communication module ***EventComm*** illustrated below defines a set of CORBA interfaces for event-style communication

```
module CosEventComm {
        exception Disconnected {};

        interface PushConsumer {
           void push (in any data) raises (Disconnected);
           void disconnect_push_consumer ();
         };

        interface PushSupplier {
           void disconnect_push_supplier ();
         };

       interface PullSupplier {
           any pull() raises (Disconnected);
           any try_pull() (out boolean has_event) raises (Disconnected);
           void disconnect_pull_supplier ();
         };

        interface PullConsumer {
            void disconnect_pull_consumer ();
        };
```

Event Channel Administration Module

Overview

-An event channel is built up incrementally (i.e., when a channel is created no suppliers or consumers are connected).

-An ***EventChannelFactory*** object is used to return an object reference that supports the ***EventChannel*** interface.

-The ***EventChannel*** interface defines three administrative operations:

÷***ConsumerAdmin***: a factory for adding consumers

÷***SupplierAdmin***: a factory for adding suppliers

÷An operation for destroying the channel

-The ***ConsumerAdmin*** factory operation returns a proxy supplier

÷A proxy supplier is similar to a normal supplier; it inherits the supplier interface; however, it includes a method for connecting a consumer to the proxy supplier.

-The ***SupplierAdmin*** factory operation returns a proxy consumer

÷A proxy consumer is similar to a normal consumer; it inherits the interface of a consumer; however, it includes an additional method for connecting a supplier to the proxy consumer.

Supplier and Consumer Registration

-Registering a supplier with an event channel is a two-step process:

1. An event-generating application first obtains a proxy consumer from a channel
2. It then “connects” to the proxy consumer by providing it with a supplier object reference

-Likewise, registering a consumer with an event channel is also a two-step process

1. An event-receiving application first obtains a proxy supplier from a channel
2. It then “connects” to the proxy supplier by providing it with a consumer object reference

The EventChannelAdmin Module

-The ***EventChannelAdmin*** module defines the interfaces for making connections between suppliers and consumers

```
#include "EventComm.idl"
module CosEventChannelAdmin {
  exception AlreadyConnected {};
  exception TypeError {};

  interface ProxyPushConsumer: CosEventComm::PushConsumer {
            void connect_push_supplier(in CosEventComm::PushSupplier push_supplier)
                        raises (AlreadyConnected);
  };

  interface ProxyPullSupplier: CosEventComm::PullSupplier {
            void connect_pull_consumer(in CosEventComm::PullConsumer pull_consumer)
                        raises (AlreadyConnected);
  };

  interface ProxyPullConsumer: CosEventComm::PullConsumer {
            void connect_pull_consumer(in CosEventComm::PullSupplier pull_supplier)
                        raises (AlreadyConnected, TypeError);
  };
// module CosEventChannelAdmin ctd.

 interface ProxyPushSupplier: CosEventComm::PushSupplier {
        void connect_push_consumer(in CosEventComm::PushConsumer push_consumer)
                raises (AlreadyConnected, TypeError);
 };

 interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier ();
        ProxyPullSupplier obtain_pull_supplier ();
 };

 interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer ();
        ProxyPullConsumer obtain_pull_consumer ();
 };

 interface EventChannel {
        ConsumerAdmin for_consumers ();
        SupplierAdmin for_suppliers ();
        void destroy ();
 };
};
```

The EventChannel Interface

-The EventChannel interface defines three administrative operations

1. Adding consumers
2. Adding suppliers
3. Destroying the channel

```
interface EventChannel {
ConsumerAdmin for_consumers ();
SupplierAdmin for_suppliers ();
void destroy ();
};
```

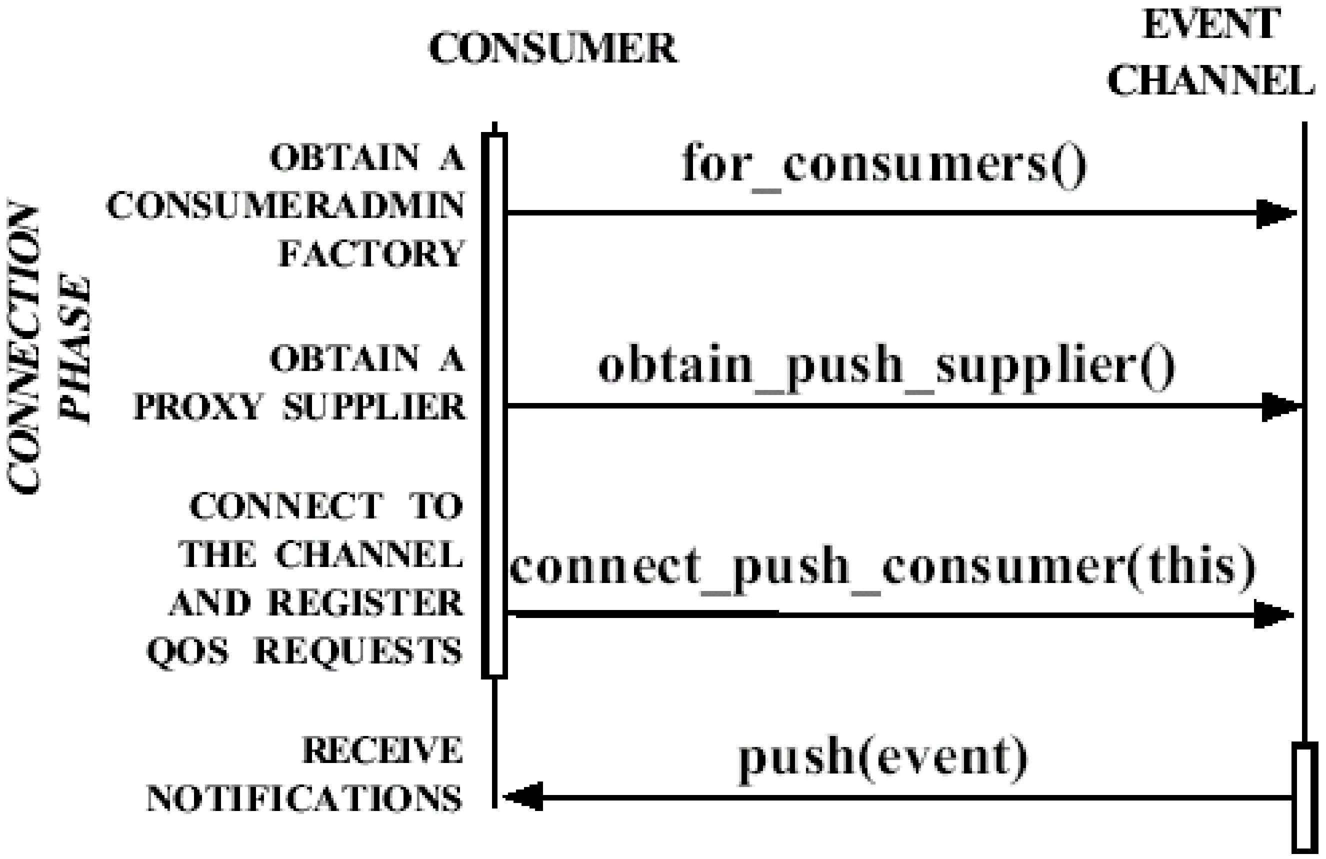
-Consumer administration and supplier administration are defined as separate objects so that the creator of the channel can control the addition of suppliers and consumers:

÷An event channel creator might wish to be the sole supplier of event data, but might allow many consumers to be connected to the channel

÷In this case, the creator would simply export the ConsumerAdmin object. Example:

```
interface Document {
        ConsumerAdmin title_changed ();
};
```

Connecting a Consumer to an Event Channel



Connecting a Consumer To an Event Channel (ctd)

```
PushConsumer psconsumer;
EventChannel channel;

//Obtain a ConsumerAdmin Factory
ConsumerAdmin cadmin = channel.for_consumers();

//Obtain a proxy supplier
ProxyPushSupplier proxyps = cadmin.obtain_push_supplier();

//Connect to the channel and register QOS requests
proxyps.connect_push_consumer(psconsumer);

//Receive notifications
psconsumer.push(event);
```

5. An Event Channel Use Example

-This section illustrates an example use of the event channel, including the following:

- –Creating an event channel
- –Consumers and/or suppliers finding the channel
- –Suppliers using the event channel
- –In this example, the document object creates event channels and defines operations in its interface to allow consumers to be added.

- The **Document** interface defines two operations to return event channels:

```
interface Document {
   ConsumerAdmin title_changed();
   ConsumerAdmin new_section();
                       :
};
```

÷The **title_changed** operation causes the document to generate an event when its title is changed;

÷The **new_section** operation causes the document to generate an event when a new section is added.

÷Both operations return **ConsumerAdmin** object references. This allows consumers to be added to the event channel.

-The **title_changed** implementation contains instance variables for using and administering the event channels.

```
/* Factory for creating event channels. */
EventChannelFactoryRef ecf;


/* For title changed event channel */
EventChannelRef event_channel;
ConsumerAdminRef consum_admin;
SupplierAdminRef supplier_admin;
ProxyPushConsumerRef proxy_push_consumer;
PushSupplierRef doc_side_connection;
```

-At some point, the document implementation creates the event channel, gets supplier and consumer administrative references, and adds itself as a supplier.

```
event_channel = ecf->create_eventchannel(env);
supplier_admin = event_channel->for_suppliers(env);
consumer_admin = event_channel->for_consumers(env);
proxy_push_consumer =
      supplier_admin->obtain_push_consumer(env);
proxy_push_consumer->connect_push_supplier(env,
                        doc_side_connection)
```

-The **title_changed** operation returns the **ConsumerAdmin** object reference.

```
        return consumer_admin;
```

Clients of this operation can add consumers.

-When the title changes, the document implementation pushes the event to the channel.

```
        proxy_push_consumer->push(env,data);
```

Appendix: CORBA Time Service

-The OMG has chosen to use only the Universal Time Coordinated (UTC) representation from the *X/Open DCE Time Service*.

÷Global clock synchronization time sources, such as the UTC signals broadcast by the WWV radio station of the National Bureau of Standards, deliver time, which is relatively easy to handle in this representation.

-UTC time is defined as follows.

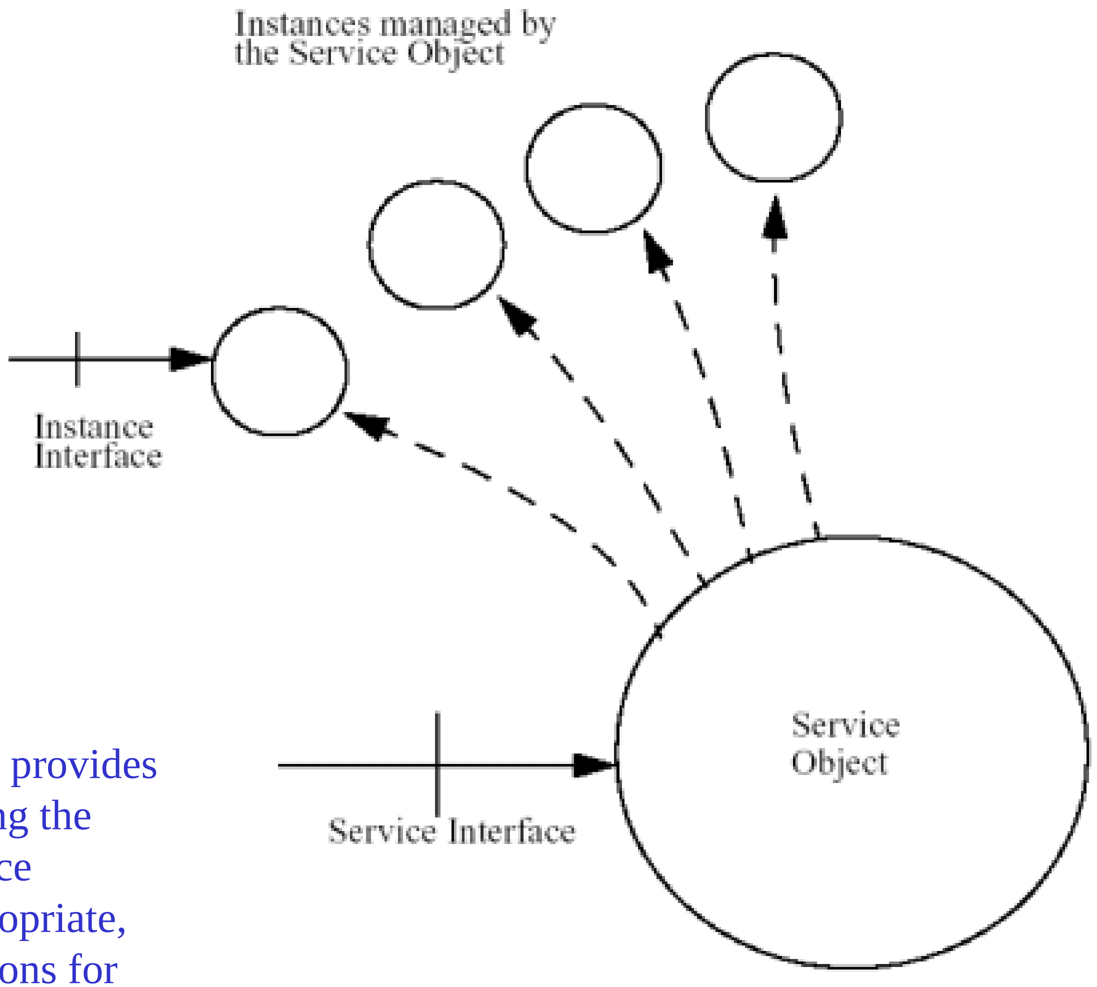
```
Time units 100 nanoseconds (10-7 seconds)
Base time 15 October 1582 00:00:00.
Approximate range AD 30,000
```

÷UTC time in this service specification always refers to time in Greenwich Time Zone.

-The corresponding binary representations of relative time is the same one as for absolute time, and hence with similar characteristics:

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

General Object Model for Service



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

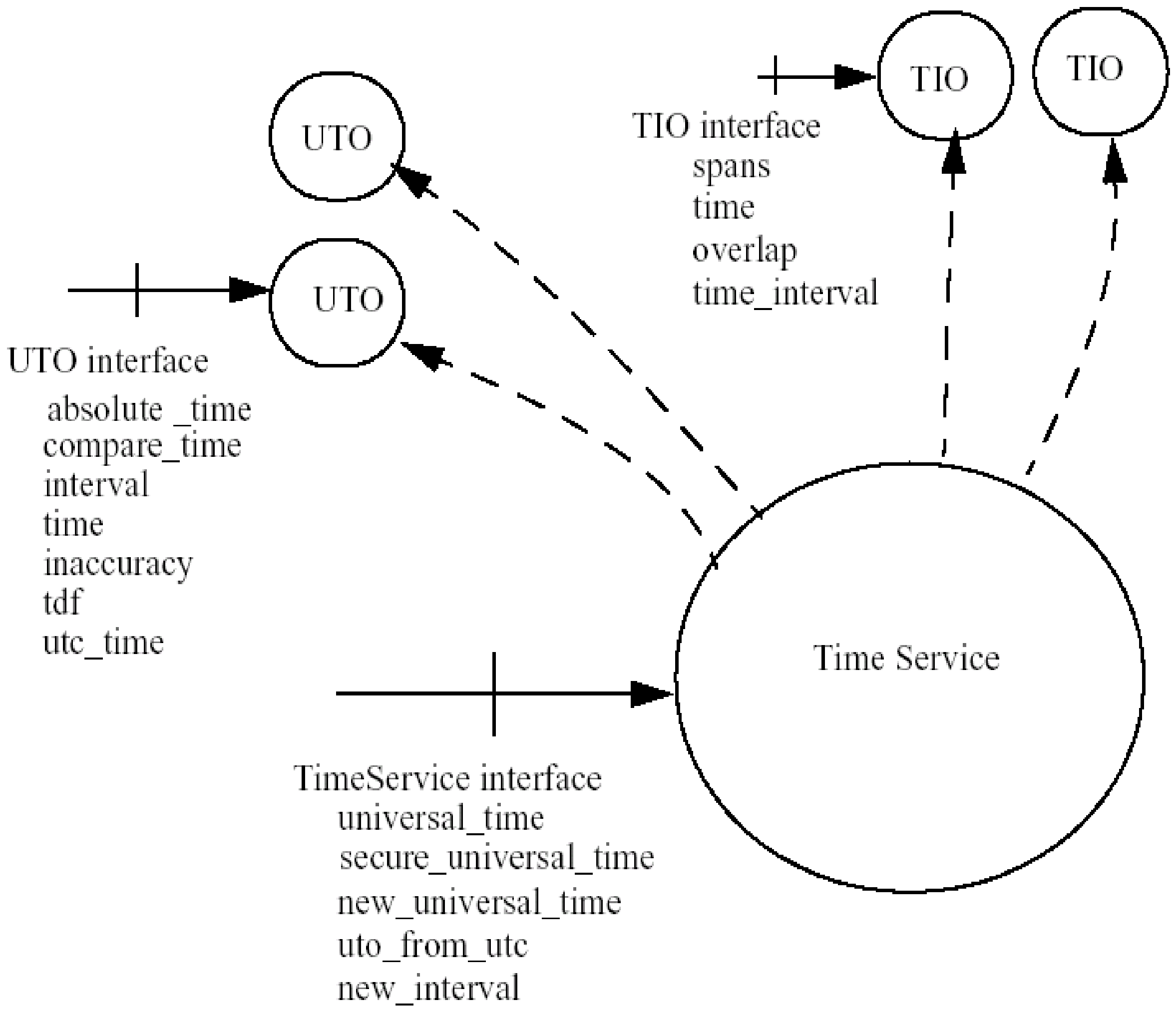
Time Service

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

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

    - *Timer Event Service* manages *Timer Event Handler* objects, and is represented by the **TimerEventService** interface.

Object Model for Time Service



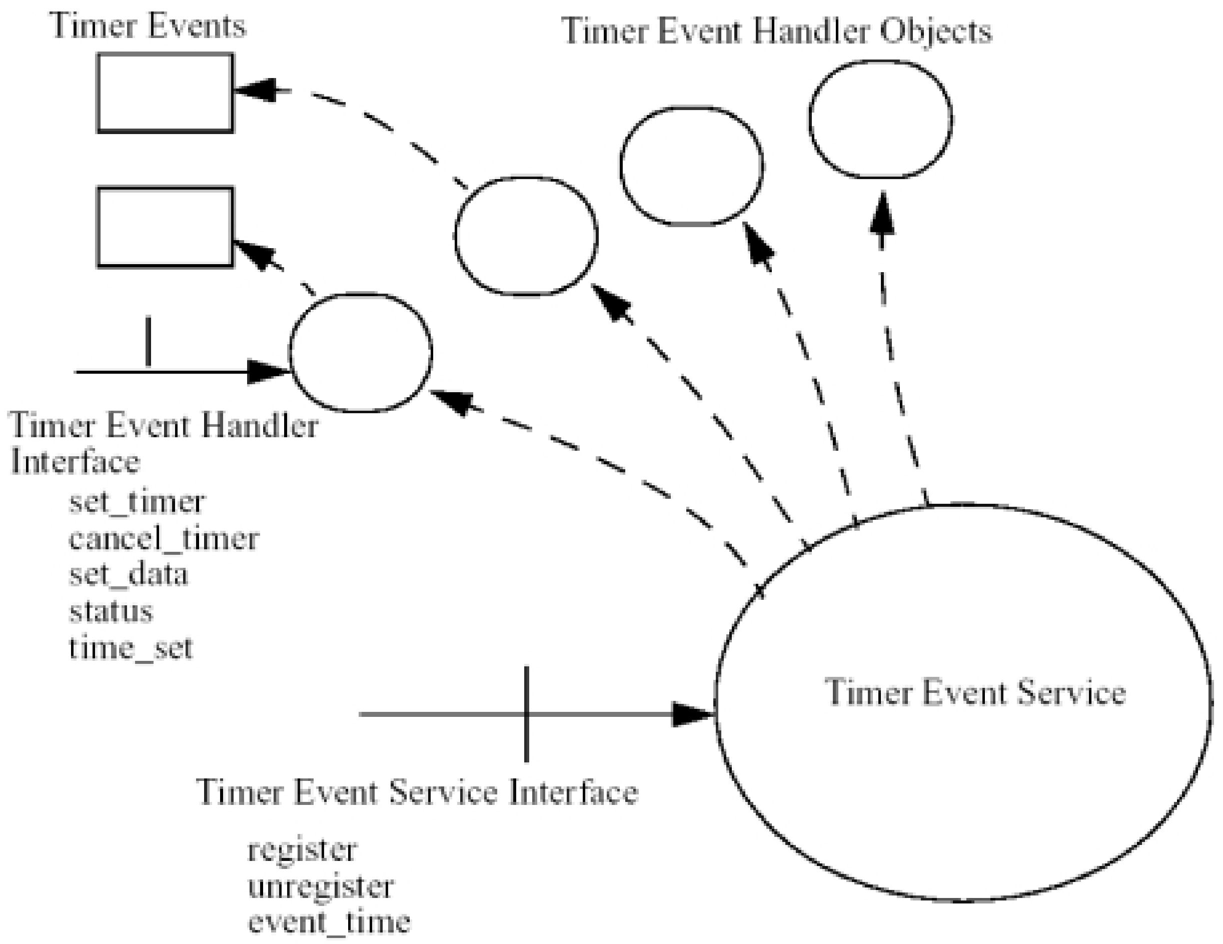
-The Time Service object manages Universal Time Objects (UTOs) and Time Interval Objects (TIOs).

-It does so by providing methods for creating UTOs and TIOs.

-Each UTO represents a time, and each TIO represents a time interval, and reference to each can be freely passed around.

Object Model of Timer Event Service

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



Timer Event Service Usage

•In a typical usage scenario of this service, the user must first create an event channel of the “push” type (see the *Event Service Specification*).

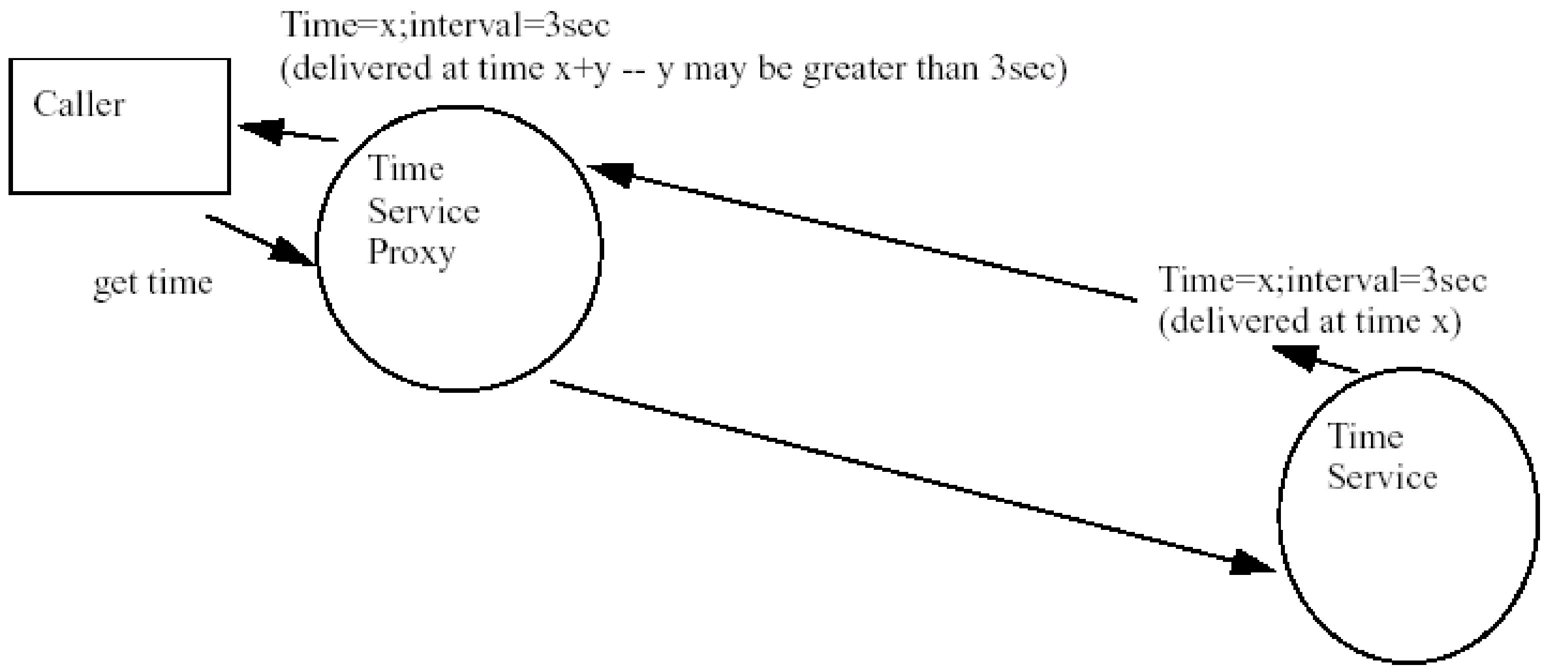
•The user must then register this event channel as the sink for events generated by the timer event handler that is returned by the registration operation. The user can then use the timer event handler object to setup timer events as desired.

•The service will cause events to be pushed through the event channel within a reasonable interval around the requested event time.

•The implementor of the service will document what the expected interval is for their implementation.

•The data associated with the event includes a timestamp of the actual event time with the error envelope including the requested event time.

Proxies and Time Uncertainty



Two possible ways of preventing proxy latency are:

• Prohibit proxies of the time server object (i.e., require a Time Service implementation in every address space that will need to make Time Service calls).

• Create a special time server proxy, which measures latency between the Time Service object and the proxy, recalculates the time interval's uncertainty, and adjusts the interval value before returning the timestamp to the caller.