Events Routing Service for Distributed Applications Computer Telephony - case study

Bartosz Klimek, Dominik Radziszowski, Krzysztof Zieliński
{radzisz, kz}@ics.agh.edu.pl
Dept. of Computer Science, University of Mining & Metallurgy
Al. Mickiewicza 30, 30-051 Cracow, Poland
tel:+48 (12) 617 39 82, fax:+48 (12) 617 39 66

August 5, 2002

Abstract

This paper presents Event Routing Service (ERS) that represents a pattern for lightweight session between a source and a destination of events establishment and processing. It exploits CORBA Notification Service at its basics. To illustrate the features of the proposed solution its application in Computer Telephony has been discussed in details. Most of the ERS positive features are inherited from Notification Service but its implementation requires a novel protocol of lightweight session establishment mechanism. The implementation of ERS is described and its application for routing events in CT system is presented. The paper is concluded with performance measurement study.

Keywords: distributed applications, notification service, events routing, computer telephony.

Introduction

Most existing systems of automatic redirection of telephone calls employ Call Redirection Module (CRM) [1] implemented around a standard simple client-server model [2]. With CRM a client is a program which switches telephone calls, and a server takes care of providing a proper reaction to these calls. Although these systems work well, they are very limited in terms of performance and adaptation to new services. The creation of a new service in the CT (Computer Telephony) system very often requires a modification of the existing code of CRM, especially when more complicated interaction between the CT hardware and the service is to be realized.

The goal of this paper is to present Event Routing Service (ERS) that represents a pattern for lightweight session between a source and a destination of events establishment and processing. It exploits CORBA Notification Service [3, 4] at its basics. The proposed pattern is general and could be used for many applications.

To make the presented investigation more concrete and to illustrate the features of the proposed solution its application in CT has been discussed in details. Most of the positive features of ERS are inherited from Notification Service but its application in context of CT events routing requires a novel mechanism of session establishment and fault-tolerance [5].

In the presented system ERS is well isolated from the underlying hardware by an object oriented library, CTLIB [6], that provides uniform abstraction of CT telephony cards provided by such vendors as Dialogic or Picka Technology. It is also separated from application servers providing them with lightweight session establishment mechanisms over event notification service. The system allows coexistence of a variety of different services, which will react to incoming calls. Call Center can serve as an example. We can imagine services interested in other kinds of events, e.g. Billing Service will handle the start and termination of calls only to record the duration of the call. ERS provides coexistence of many copies (*instances*) of the given service. It allows load balancing, simplifies concurrent processing and increases reliability.

The structure of the paper is as follows. In next section the requirements of event routing service in CT applications are considered in details. Three different viewpoints: functional, management and fault tolerance are analyzed. In the following section the ERS architecture is specified and the proposed pattern of lightweight session establishment and processing is explained. This pattern has been compared to a well known Session and Abstract Session Patterns [7]. Next the ERS implementation environments is presented and evaluated. The application of ERS is presented in the following section. The problems of ERS scalability and flexibility is discussed in the last section. The paper ends with conclusions.

Requirements for events routing in CT systems

Functionality of ERS has been defined in context of requirements for CT applications, which may be divided into three groups:

- information about the events generated by CT hardware such as: the appearance of a new call, the and of playing an audio file, etc.
- the supply of an interface allowing the control of CT hardware, e.g. switching the call to the given destination, starting to record a file, etc.
- the delivery of the information about the given call (state, duration, etc.).

The use case diagram of a typical CT application, called Telephone Exchange Interface (TEI), is shown in Fig. 1. It specifies the interaction between two categories of actors i.e.: CT Devices and CT Services. CT Device generates events and processes commands issued by CT Service in response to received events in context of external information retrieved from database.

It is easy to notice that events dispatching plays a very important role for this application. It is necessary to route events to the proper services, however, it cannot be assumed that events should only be obtained by one service. The mechanism of events distribution to many services supporting events filtration should be

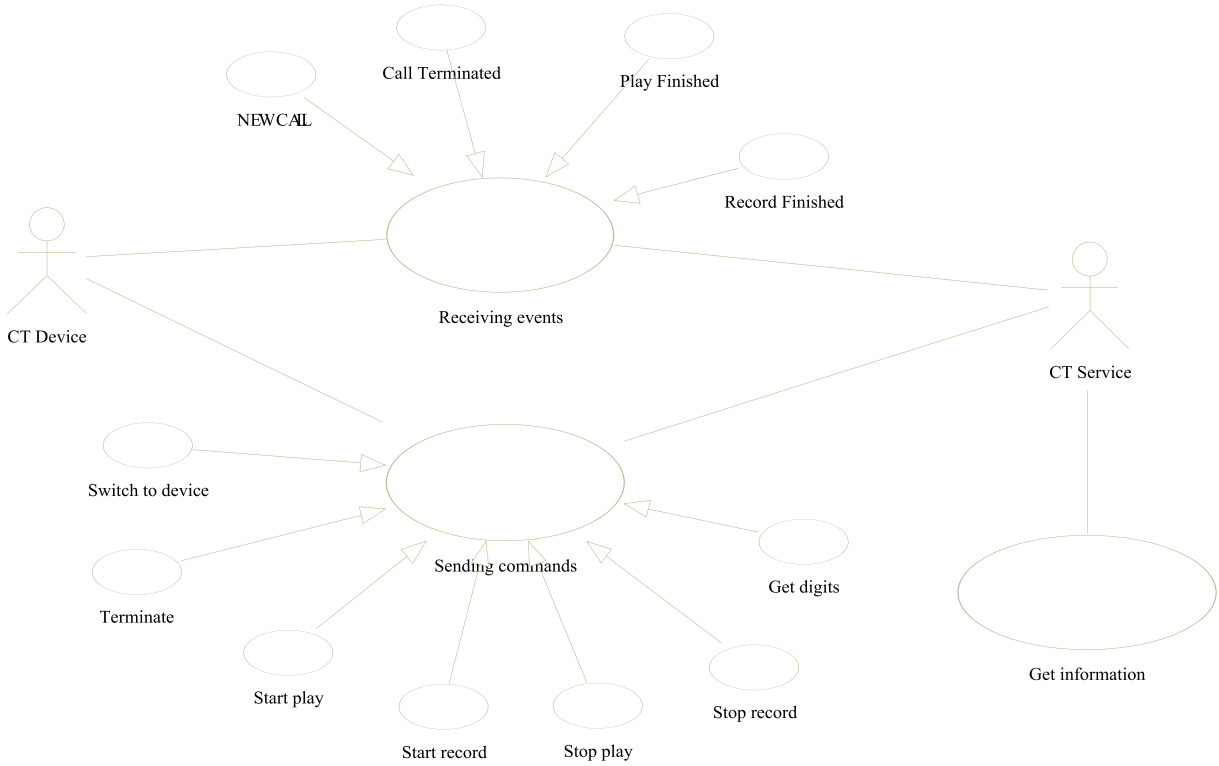

Figure 1: The main use case diagram for TEI subsystem

provided and the events should forward the suitable routing information. The following events dispatching scenarios should be implemented:

- Sending events to the given service instance (e.g. to the service that handles the call),
- Dispatching events to all instances of the given service type (e.g. when a new call appears),
- Delivering events to all services (e.g. administrative messages).

The system should be equipped with a management interface that makes the dynamic setting of system services properties possible as well as allows the defined events routing scenarios to be realized.

CT applications are typically built of various different independent services. In this case:

- The given call processing should be allocated only to the relevant service or more precisely to the instance of the service. It is necessary to avoid the situation when several (more than one) services (instances) try to control the call.
- The authorization of the call processing should be restricted only to the services which handle the pre-allocated range of number extensions. Such allocation should prevent from conflicts and unlimited access to information.

An activity in a distributed environment also requires the implementation of mechanisms which supply information about the system components accessibility and their state to ensure the proper system faults handling.

ERS service definition

Event Routing Service introduced in this section has been defined in the context of CT application requirements already defined, yet it provides far more general functionality. This is why this service is defined in general terms not related to this particular application. The CT oriented usage will be described by the mapping ERS components to applications domains objects.

ERS has been designed as an extension of CORBA Notification Service with mechanisms which support lightweight session establishment and control the events between the source object (producer) and the service objects instance (consumer). The point is that many applications require a sequence of events generated from a given source, which has to be processed in the same context i.e. by the same service instance. The session implementation concept over Notification Service is central to the proposed ERS service.

The proposed solution is similar to the Session pattern [7] which imposes a three-phase protocol upon the interactions between the source of event, playing the client role, and the server object instance.

- Client sends an initial "request" event to the event routing engine specifying the requested service parameters. The event is forwarded to all service instance from the pool of active instances. Any free server instance passes its unique services instance identifier to the source of the event and in return it obtains a session identifier. The service instance with the allocated session identifier is in a connected state and ready to accept the events marked with this identifier.

- During the session period the service instance and session identifiers are used in the following way:

  - both identifiers are put to the subsequent events generated by the source of events,
  - service instance identifier is checked by the events routing mechanism in order to forward an event to the right destination service instance.
  - session instance identifier is checked by the destination service instance and the event is processed only if its instance identifier is in perfect agreement with the session identifier earlier received.

- When the client has finished using the service instance, it sends a final "release" event, passing in the server instance and session identifiers. On Receiving this event the service instance enters into a disconnected state and the session identifier is removed from the instance.

The proposed solution is to some extent similar to the Session pattern [7] in that the event source obtains a form of the session object pointer by receiving a server object instance identifier. This object processes subsequent messages. It differs from this pattern by application of a semantic event routing engine to forward a message to the destination server object. Semantic routing of event messages which implies forwarding the

messages based on checking event parameters and then passing them to the right pool of service instances. It makes the proposed pattern even more general.

The session is called lightweight because the only required common information between the source and the destination of event is the session identifier and the event forwarding based communication is asynchronous. It is enough to ensure that the sequence of messages from the same source is processed by the same service instance, which could preserve the state between subsequent events processing. The session identifier could be passed to another source of events that allows many sources to participate in such defined session. This schema could be further enhanced by the Abstract Session pattern application in which many sessions have to be served by the same server object.

ERS architecture components

Event Routing Service architecture, which implements the proposed pattern, consists of four basic types of components:

- Event Source Adapter (ESA),
- Event Routing Service Manager (ERSM),
- Notification Service,
- Event Destination Service (EDS),

depicted in Fig.2.

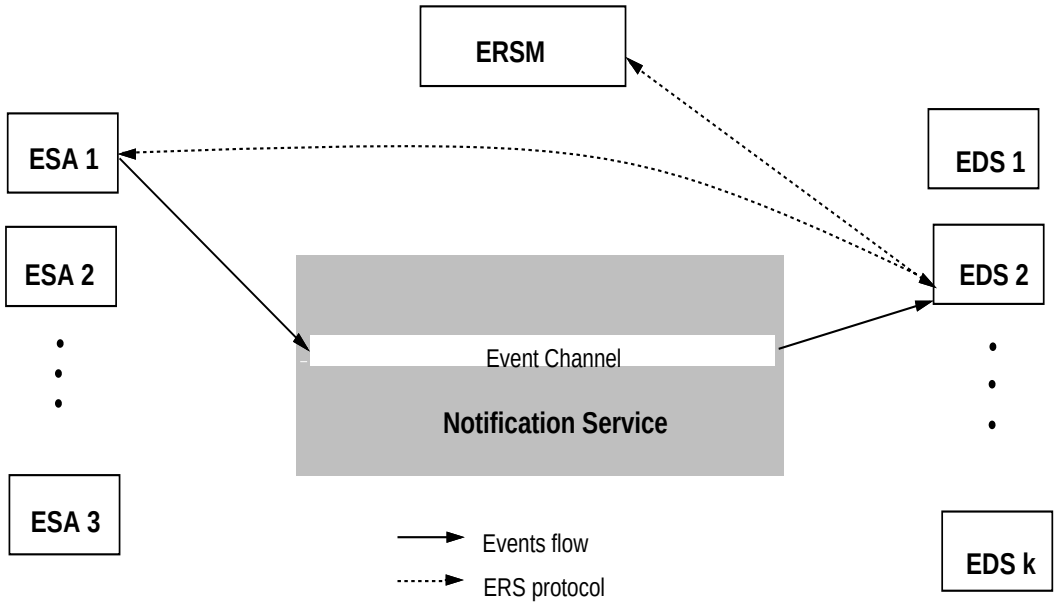

Figure 2: ESR architecture

The ESA object represents a single device that generates events that have to be served by the selected EDS object. Notification Service performs events routing between ESA objects and EDS objects and processes the specific ERS events parameters.

The Notification Service [8] provides a mechanism for a more loosely coupled method of communication between objects, rather than by invoking operations directly. Event suppliers and event consumers are de-coupled by an event channel which handles supplier registration and broadcast of events to consumers.

The service is an extension to the Event Service [9]. It is the engine of an event-driven system. Its architecture allows to construct a kind of a virtual bus, with advanced addressing of events sent through it, including unicast, broadcast and multicast, based on the information contained in the events. This is particularly interesting for developers of telephony and telecommunication systems, in which events generated by the hardware may trigger various actions, possibly in many places at once (e.g. logging, billing, gathering information for statistics purposes).

The events sent through the channels of the Notification Service can be of any IDL type, in particular they can have a special structure defined by OMG called structured event, in which case the spectrum of possibilities grows.

A very powerful feature of the Notification Service is event filtering, specified with a constraint language. It can be easily applied to user-defined fields of various types. It is used to organize the flow of events in our system, which is quite complex.

ERS exploits the structured events with a few predefined attributes proposed by service definition presented later in this section. The values of these attributes are processed by the ESR protocol in collaboration with ERSM, EDS, and ESA.

ERS components interfaces

The ERS components are equipped with interfaces which enhance the standard Notification Service interfaces and provide support for the ERS protocol used for session establishment implementation. The session is represented by `session_id` which is a unique number generated by the ESA object implementation. The `session_id` is used as a tag parameter for each event generated by the ESA object during the session and pushed to Notification Service. This tag provides temporal association between the source of events and the destination object. ESA interface, presented below, provides only two additional operations related to `session_id` handling.

EDS object is a standard push consumer object. The only difference is that it has a built-in logic of the ERS protocol and implements sophisticated event queuing mechanism. The communication with EDS is performed via events using standard Notification Service interface [3]. ERSM plays the role of a central registry of service instances provided by the system. Each service is described by a name and an `instance_id`. The `instance_id` makes it possible to distinguish diverse instances of the same service type. It is generated during the service instance registration process and passed back to the EDS object in `Register_info` structure. The registration procedure should be performed by the EDS object during the start-up procedure. Other operations of the ERSM interface, shown in Fig. 4, are used for service activity

```
interface ESA {

   long get_session_id(in long service_id, in long instance_id);

   Service_Configuration_Seg get_configuration ();

   void terminate(in long session_id);
};
```

Figure 3: ESA interface

testing and the module management.

```
interface ERSM {

   Register_info register_instance(in ServiceName service_name);

   void ping(in InstanceId instance_id);

   Service_Configuration_Seg get_configuration ();

   oneway restart ();

   void shutdown ();
};
```

Figure 4: ERSM interface

ERS Protocol

The ERS Protocol implements the lightweight session establishment mechanism. Instead of formal definition of the protocol its sphere of activity will be described. The example of the structure events used by the proposed ERS protocol are shown in Fig.5, which presents the collaboration diagram of the protocol activity during the session establishment procedure.

The ESA object opens a new session by sending the event with the type field set to `New_Session` value and `service_name` set to the required service. The field `call` of this event is equivalent with the reference of the ESA Corba object. This event will be called an opening event. The Notification Service dispatches this event to all active EDS instances representing this service. The standard filtering mechanisms are provided by Notification Service which is its main purpose. Each idle EDS instance 1 ESA object using the obtained reference by invoking `get_session_id` operation with set `service_id` and `instance_id` parameters. Only one invocation is successful and returns with proper value of `session_id`. The identifier of the successful

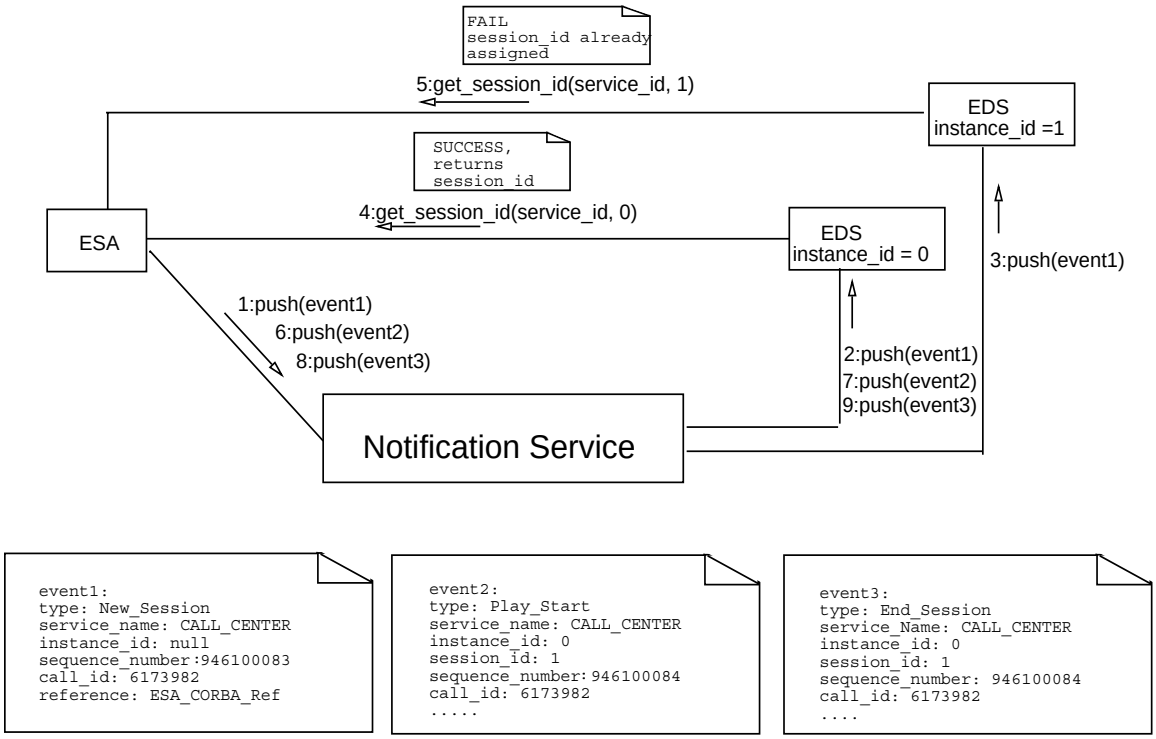

Figure 5: Example of ESR protocol activity

EDS instance object is used as `instance_id` in subsequent events generated by the investigated ESA object. This field directly points the service instance object during the session. The granted `session_id` plays a role of a password which authorizes events processing.

If no idle instance of the required service is available, one of the two following scenarios may be chosen by the application programmer:

- All opening events are destroyed by the busy EDS object instances and there will be no `get_session` operation invocation. This situation is recognized using the time-out set at the moment when opening event is generated. If there is no ESA object invocation during the time-out, it means that no service object is available and that the opening event should be re-sent later.

- The opening event is queued in busy EDS objects and processed in due time. In this case, the late invocation of `get_session` operation by EDS objects results with only one successful operation. The other invocations are unsuccessful and create addition overhead.

The choice between these two options are discussed in detail later together with scalability issues.

The proposed protocol may be further refined to reduce some scalability problems when many EDS objects instances are ready to join a service or all objects are busy.

The successful EDS instance either processes receiving events locally or performs operations on EAS object or any object which references are handed off by EAS events during the session. It makes EDS

possible to play the role of events processing and dispatching engine for the given session.

The session closes with sending the final event with type field set to `End_Session` value. The EDS object processes final event, sets its status to idle and then it is ready to response for a new session establishment invitation.

The described EDS Protocol may be seen as a dynamic service allocation procedure which automatically finds an available service.

ERS fault-tolerant extension

The ERS fault-tolerant mechanisms exploit the leasing concept based on the following rule - resources are hired for a specified time. When the user of the resource still wants to use them, they have to ask the manager of the resource to prolong the hire for another period of time.

Looking at the ERS system, it may be said that service objects hire proxy suppliers from Notification Service. Periodically services have to inform ERSM that they are still interested in receiving events by calling `ping` method on it's interface. ERSM has a time-out set for each instance of a service, if it expires the appropriate proxy suppliers are destroyed so the instance of a service will not get any more events. The time-out may happen when CT service is stopped, destroyed or there have been serious communication problems. The periodical call of `ping` method is realized by EDA objects automatically - the programmer does not need to take care of it.

ESA also needs to notify the services whether it is working. This process is performed through an auxiliary event channel. All service objects listen to this channel, and if so called ping event from ESA appears, they know that EAS is running and wants them to wait for events and call handling. If ESA pings do not come on time, all service objects have a time-out set, the communication session is released. The activity described above is presented in Fig.6

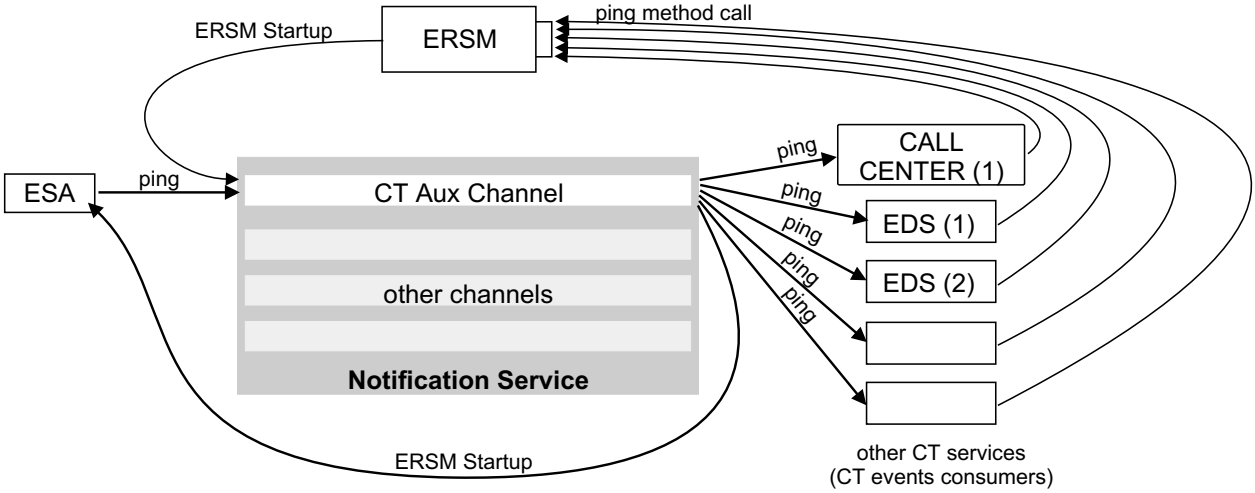

Figure 6: Ping 'flow' diagram

This activity is particularly important during the session establishment. The problem is that ESA does not know how many, if any instances of a given service object are ready for processing. That is why (but

not only) ESA has a time-out for each `New_Session` event, and if no instance of a desired service handles the event before timeout expires, the call is terminated.

The auxiliary channel is also used by ERSM. At ERSM startup, an event informing about this fact is sent. This event informs each ESA to refresh its configuration. The service object gets this event too but it ignores it.

Note that the proper work of ERSM is not required for the system to work if it is already running, so there is no need to check ERSM activity.

It is also necessary to point out that ERS is using TCP protocol for events delivery (as Notification Service does), so if the suitable objects are active the message event should not be lost.

ERS Implementation Issues

The requirements for the ERS system motivate to choose the technologies used in building their components very carefully. In particular, the attention should be drawn to: performance, portability (concerning both hardware platforms and operating systems), reliability, and flexibility.

In the following subsections major implementation decisions and the reasons that have convinced authors to make them have been described.

Choice of Implementation Environments

The choice of implementation environment has to be considered in the following issues context: (i) Programming language, (ii) Distributed programming environment, (iii) Event transport mechanism, and (iv) Operating systems and hardware dependencies. The general assumption has been that the ERS system has to be implemented using distributed object oriented technology that is a common standard approach nowadays.

Selection of programming languages

Because authors have attempted to use well known, proved and easily accessible tools, they decided to implement the system in C++ and Java. There was an option to write all code in Java, however they were not certain how well the servers would perform, for they must be robust and stable and work continuously for a long time. The asynchronous nature of garbage collection in Java results in that the usage of CPU and memory resources change with time. Therefore it was perceived as an disadvantage in a heavily loaded server processes, because the performance of such servers may periodically go down. This in turn, may make the whole system unstable. That conclusion led to the decision to implement all core elements in C++ for effectiveness and stability.

On the other hand, C++ programming is more difficult and bug-prone than Java, especially when using high-level, complex environments, such as Common Object Request Broker Architecture (CORBA) [10]. Since the system was intended to be open for extensions and flexible, Java has been chosen to implement the service interface. A very similar interface has been also developed in C++, so that service developers are not bound to one language.

An important advantage of Java is its portability. Nevertheless, the source code of our servers is independent of the operating system or hardware and uses only standard ANSI compliant libraries and CORBA. This means they can be compiled on any platform on which an ANSI C++ compiler and CORBA are available. Authors are certainly aware of possible slight incompatibilities between various C++ compilers and libraries.

Choice of distributed programming environment

One of the basic requirement of ERS is the distribution of its components. This is primarily due to its heterogeneous nature, and the need for load balancing and reliability. At the early phase of the design it has been decided to use CORBA. Authors found the following features of CORBA to be essential for ERS system:

- Heterogeneity — there exist mappings from IDL to C++ and Java, while ORB software is available for all popular platforms [11].
- Existence of basic services, as well as advanced ones, which makes design and implementation easier and much more effective [12, 13, 3, 14].
- High level of abstraction and transparency — most of low-level issues are hidden from the programmer (e.g. the format of data sent through the network, the location of the server process).
- Support for multi-threaded environment [15, 16].

OmniORB [17] was chosen for ERS system, because it is reliable, robust and available to a variety of platforms and compilers. The important arguments were also that omniORB is available with documentation and source code, it is free, and there is a developer's forum dedicated to this product.

Event transport mechanism

Instead of developing a new, specific for the application, event delivery service, one of the existing CORBA Services [8, 18] was proposed. Event Service [12] has been rejected because it doesn't support event filtering, which is important for overall ERS system performance. Notification Service [3] has been chosen. Its functionality allows to use, advanced delivery schemes of sent events, including unicast, broadcast and multicast, based on the information contained in the events. This is of great import to developers of telephony and telecommunication software when events generated by the hardware may trigger various actions, possibly in many places at once (e.g. logging, billing, gathering information for statistics purposes).

Operating systems and hardware platforms

As it has already been mentioned, the ERS code is highly portable, so it can be compiled and run on a variety of platforms. However, for the development the following platforms have been selected: Windows NT/x86, Unix, and Linux/x86. The choice of Windows NT/x86 is obvious for many applications. For instance, CT

software is often available for x86 platform while interface libraries are accessible only for Windows. It requires ESA objects implementation over Windows/NT. Authors prefer to use Unix for other parts of the system because it is generally more flexible and stable than Windows NT. The servers run on Solaris/SPARC or on Linux/x86.

ERS application in CT system

The proposed ERS service has been used for implementation of the CT application. The general requirements of this system have already been analyzed. The Telephone Exchange Interface (TEI) is a part of a computer aided telephone call handling system, CT Service System (CTSS). The TEI system is implemented with object-oriented API wrapper of CT hardware [6]. For each telephone line `Abstract_Terminal` is created. When a new call arrives `Abstract_Terminal` answers it and calls `Call_Factory` object to create new `Call_Impl` object. The interface of `Call_Impl` object contains such typical operations as for instance: `start_play()`, `stop_play()`, `start_record()`, `stop_record()`, `get_digits()`, etc.

The `Abstract_Terminal` object acts as EAS and sends call related events using ESR protocol and Notification Service. There has also been built CT System Manager (CTSM) server that implements functionality of ERSM.

CT-Service class implements logic of the telephone call processing. It receives all events from the associated `Abstract_Terminal` object via ESR protocol and performs relevant operations on `Call_Impl` object. The CT-Service object is registered with CTSM server and identified by the name of a service type. Many instances of the same service type may be activated and awaiting for a new call.

ERS component	CT application component
ESA	Call object of TEI
EDS	CT-Service
ERSM	CTSM

Table 1: Mapping of ERS to CT components

The structure of CT-Service module is presented in Fig.7. The application independent part consists of StructuredPushConsumer and an implementation of internal Event_Queue mechanisms. The application dependent part is CT-Service class which also performs the ERS protocol (`register_instance()` and `get_session_id()` calls).

The straightforward mapping of the general ERS terms to the investigated CT-application components is shown in Table 1.

The internal structure of the CTSM server is almost completely independent of the application. It is a part of the system which may be reused in other applications.

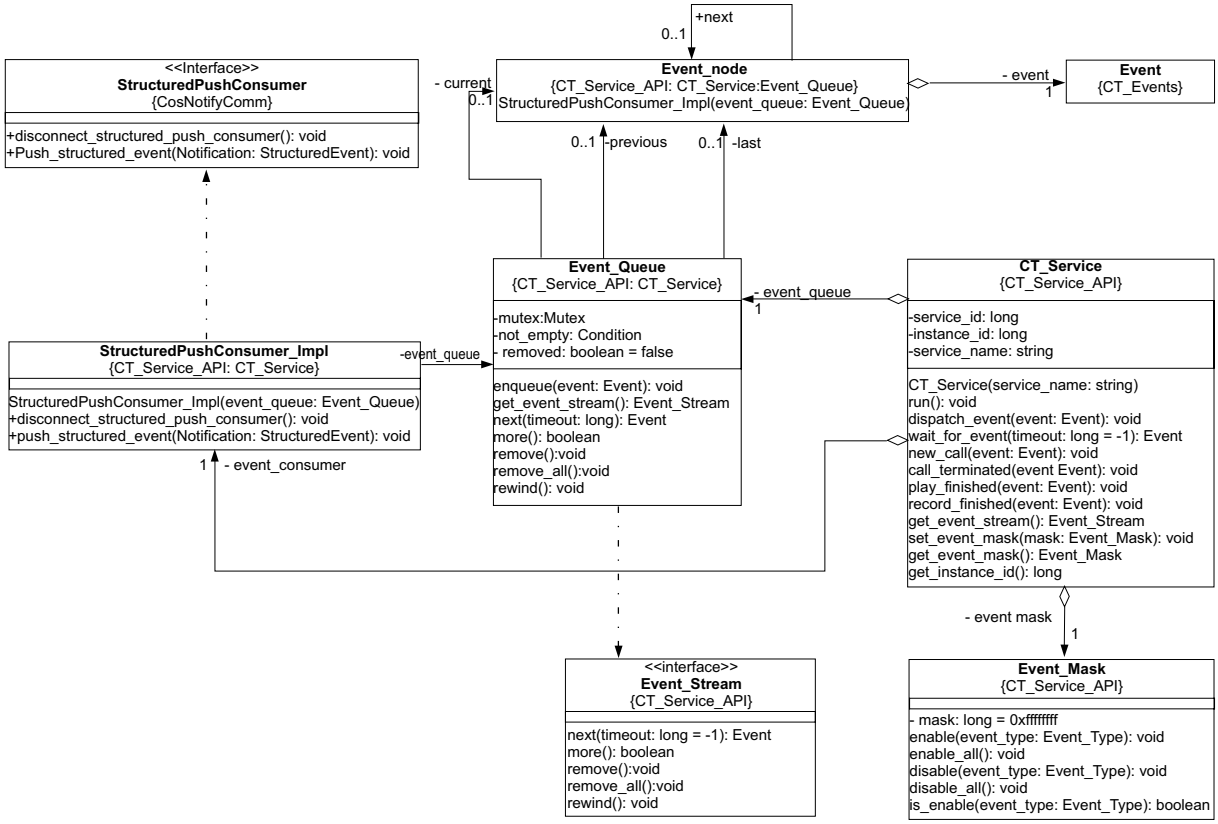

Figure 7: CT-Service API - structure

ERS performance and scalability analysis

The ERS service should be evaluated from the performance point of view, where the most important index is the time of handling typical operations such as: time to join a service by an EDS object, time of an event delivery, etc. Another important aspect is an overhead caused by the ERS protocol activity under different load of the application. The performance issue of the proposed service in the state when the session between ESA and EDS is already established is determined by the implementation of Notification Services. The performance analysis of this service could be found elsewhere [19, 4].

The ERS protocol activity should be analyzed in two situations: (i) the system is not very busy and many EDS instances are ready to join the session, (ii) the system is overloaded and no EDS objects is willing to open a new session. In the first situation many EDSes simultaneously perform the `get_session_id` operation, from which only one is successful. It may create an increase of network load in a short period of time and unnecessary processing overhead on the network node where the ESA process is running. It is not risky, thought, the system as a rule by principle, is not very busy and it is absolutely ready to cope with this extensive traffic.

Otherwise, when EDS objects are very busy they will respond to the invitation for joining a new session in due time when current processing is finished or suspended. In this case, the danger of system overload

is rather limited. Despite this facts, the number of unnecessary `get_session_id` operation calls should be reduced significantly.

The behavior of the system could be easily improved by grouping EDS objects of the same and delegating the task of joining sessions to the group representative (GR). The structure of such system is presented in Fig.8. GR performs `get_session_id` operation only when one or more of its EDS objects are idle.

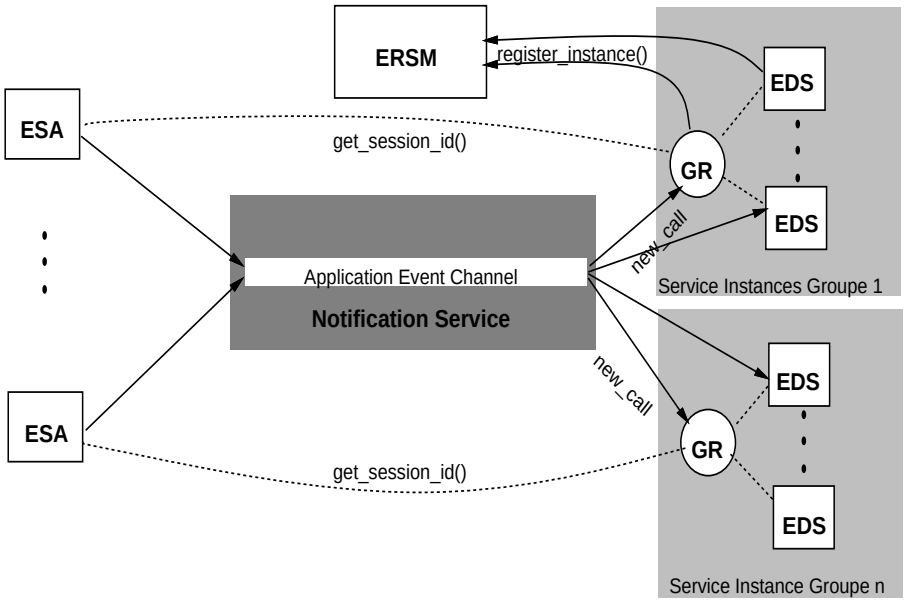

Figure 8: ERS with Group Representatives

This almost eliminates the number of unnecessary operations invocations. This effect increases the complexity of the system and prolongs the time of joining a session when the EDS instances which create a group are spread around many network nodes. In such a case, the hybrid solution, where objects collocated on the same node have common GR, could be a better solution.

The objective of the performance study has been to evaluate on the lightweight session establishment mechanism which is essential for the proposed service. Authors have intentionally eliminated from the experiment all issues related to Notification Services implementation performance not to ERS protocol characteristics.

The performance study has been performed for the two CT system configurations depicted in Fig.9. The configuration 2 in contrast to configuration 1 had Group Representative object, which was processing each `new_call` event.

The measurement of the number of calls handled per minute has been performed for different number of: active telephone lines, service instances, and various call duration time. To facilitate the interpretation of the obtained results all services were of the same type. It has been assumed that the call processing complexity is represented by the call duration. This was the time after which the service instance had finished the call processing. Each line become active immediately after the previous call from this line had been served, that is `terminate ()` operation had been called on the line adapter ESA interface. This kept the system load on

the maximum, so the results obtained represent the maximum performance.

It has also been assumed that each line `proxy_push_consumer`, and each service `proxy_push_supplier` had been created with Notification Service before the start of the measurement. Each line adapter object before the start of the experiment performed also `get_configuration` operation on ERSM interface to obtain `service_id` of the requested service.

The reported results of the experiments concern a distributed configuration where Notification Service and the service instances where running on SUN Enterprise 3000 (3 x 450 MHz UltraSPARC CPU). The other part of the system, that is the line adapter objects stimulated by TEI, and ERSM has been processed by an Intel-PC laptop. Both computers were connected by 100Mbps Ethernet.

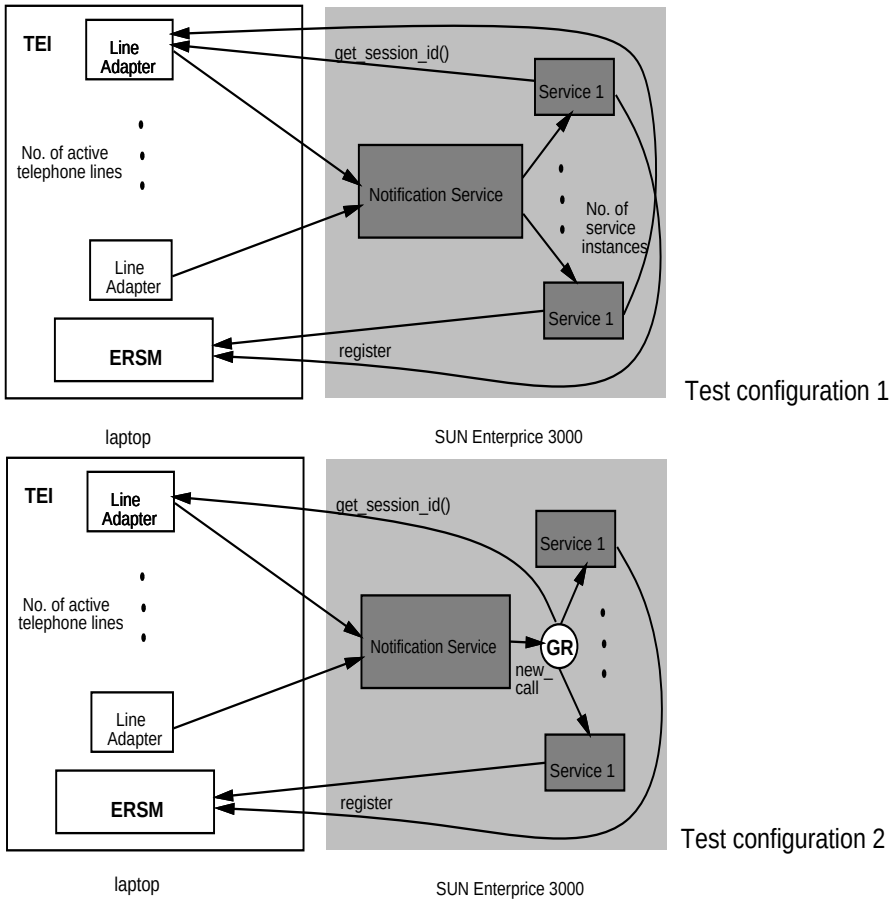

Figure 9: Test configurations

The results of the experiments have been represented in Fig.10 and Fig.11 for the call duration of 1[ms] and 1000[ms] to illustrate the most interesting observations.

It is evident that configuration 1 performance decreases when the number of service instances is greater than a certain value depending on call duration. For a very short duration time 1[ms], the existence of other instances which all are not busy creates collisions at the moment of getting `instance_id`. For 1000[ms] study this effect is observed for 5 instances.

The system with GR, that is configuration 2, performs substantially better than configuration 1. This system is able to process more than five times greater number of calls and shows only rather not dramatic reduction of performance when the number of telephone lines is substantially lower than the number of service instances. For longer call duration time the best system behavior was observed when the number of active lines was close to the number of service instances (see Fig.9). This result can be easily deduced applying intuition.

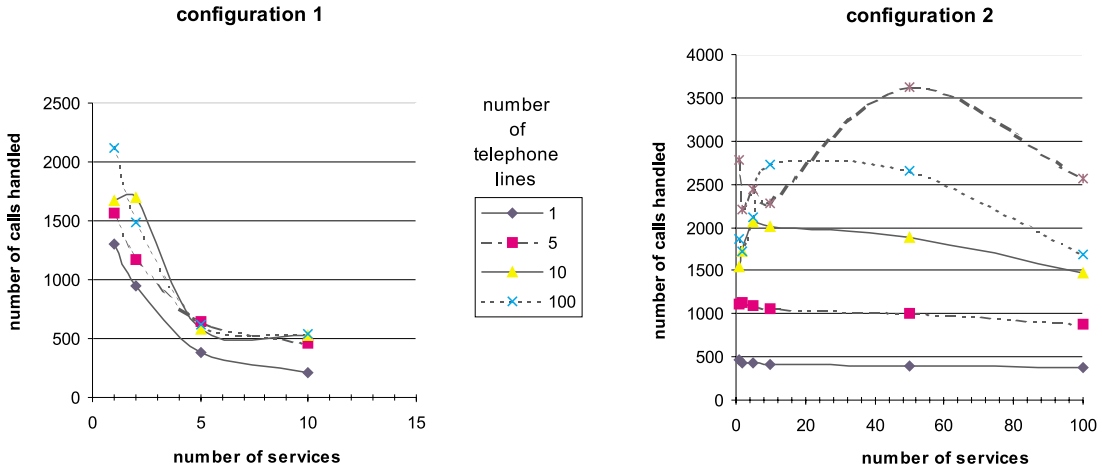

Figure 10: Number of calls handled for service delay 1 ms

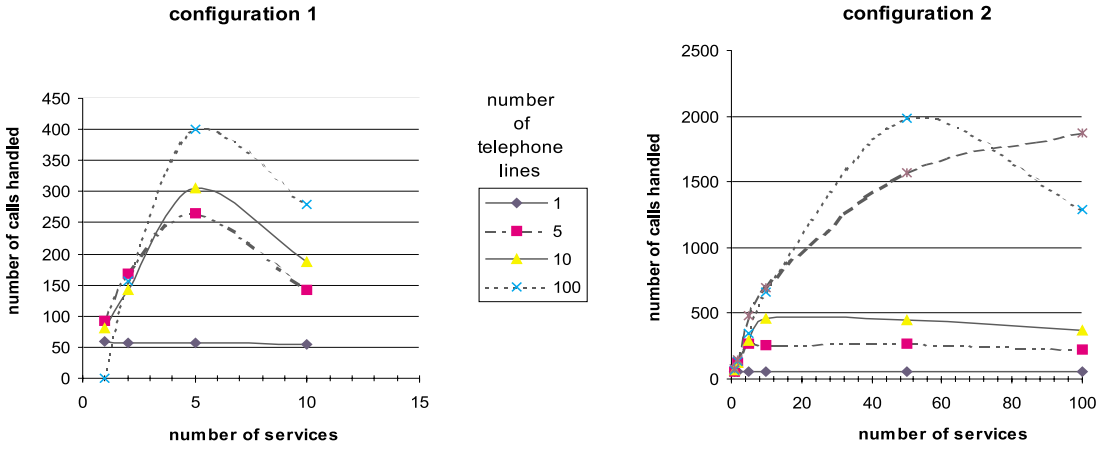

Figure 11: Number of calls handled for service delay 1000 ms

Conclusions

The presented ERS service has provided a valuable framework for structuralization of a very sophisticated system of events routing and processing in the CT system. This service enhances Notification Service with the paradigm of the lightweight session establishment. This provides mechanism for events processing in a well defined context of a given session. This places the proposed service between the fully connection oriented client server model and the completely detached Notification Service.

It has been proved, as far as scalability is concerned, it is crucial to structure the system in groups of service instances represented by single objects which participate in ERS protocol. The performance study has shown that the system is able to process a few thousand calls per minute, which satisfies the requirements of most real applications.

The experience gained from the CT-system development, which uses the proposed ERS service shows that it is possible to build an efficient system. The system which outperforms other systems implementations using standard client-server paradigm as far as flexibility and configurability are concerned.

References

[1] Ericsson. Route manager. Technical report, http://www.ericsson.cz/for_enterprise/ncc03rou.pdf, 2000.

[2] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall International Inc., 1995.

[3] Object Management Group. *CORBA Notification Service*, 1998.

[4] Timothy H. Harrison, Carlos O'Ryan, David L. Levine, and Douglas C. Schmidt. Design and performance of a real–time corba event service. *IEEE Journal on Selected Areas in Communications. Special issue on Service Enabling Platforms for Networked Multimedia Systems*, June 1997.

[5] Michael R. Lyu. *Software Fault Tolerance.* John Wiley & Sons Inc., 1995.

[6] Dominik Radziszowski Bartosz Klimek. *Object-Oriented, Event-Driven Platform for Distributed Computer Telephony Service.* UMM M.Sc. Thesis, 2000.

[7] Nat Pryce. Abstract session. In *Pattern Languages of Program Design.* Addison-Wesley, 1999.

[8] PrismTech Limited. *OpenFusion CORBA Services, Version 1.1. Developer's Guide*, 1999.

[9] Douglas C. Schmidt Carlos O'Ryan and J. Russel Noseworthy. *Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations.* University of California, Object Sciences Corp., 2001.

[10] Object Management Group. *CORBA/IIOP Specification, Revision 2.2*, 1998. OMG formal/98-07-03.

[11] Michi Henning and Steve Vinoski. *Advanced CORBA programming with C++.* Addison Wesley Longman, 1999.

[12] Object Management Group. *Common Object Services – Event Service*, 1998.

[13] Object Management Group. *Common Object Services – Naming Service*, 1998.

[14] Object Management Group. *CORBA Messaging Service specification*, May 1998. OMG orbos/98-05-05.

[15] Tristan Richardson. *The OMNI Thread Abstraction.* Olivetti & Oracle Research Laboratory Cambridge, 1997.

[16] Iona Technologies Ltd. *Orbix 2.3 programming guide*, 1998.

[17] David Riddoch Sai-Lai Lo. *The omniORB2 version2.8. User's Guide.* AT&T Laboratories Cambridge, 1999.

[18] Iona Technologies Ltd. *OrbixTalk 1.2 programmers' guide*, 1998.

[19] Douglas Schmidt and Steve Vinoski. Overcoming drawbacks in the omg event service. *C++ Report*, 10, June 1997.