

## Simple Service Developed with Ericsson OSA/Parlay Simulator

Michał Rój, Jarosław Domaszewicz  
*Institute of Telecommunications*  
*Warsaw University of Technology, Warsaw, Poland*  
*E-mail mroj@elka.pw.edu.pl, domaszew@tele.pw.edu.pl*

*Keywords:* telecommunications services, open APIs, OSA/Parlay

**SUMMARY:** This paper gives an example of applying a novel approach to developing telecommunications services. The new approach introduces mainstream IT technologies into service creation, making the process accessible to a wide community of software developers. Services can be developed for all kinds of networks, including wireless ones. In this paper, a simple call center service implemented using the OSA/Parlay API and tested with the Ericsson OSA/Parlay Simulator is presented.

### 1. INTRODUCTION

The OSA/Parlay API (application programming interface) aims to capture different areas of functionality of a telecommunications network and make them available to the programmer by means of standardized, object-oriented interfaces ([YAT,2000], [MOE,draft]). The programmer implementing a service logic application uses the interfaces to program interactions between the application and the telecommunications network. Using the API, the application can request that it be notified by the network about relevant, network-related events. It can also instruct the network to perform specific tasks required by the service logic.

The service logic application can be developed using mainstream IT technologies. The application can (but does not have to) be hosted by a service provider different from a network operator. Such an approach enables wide deployment of specialized services. OSA/Parlay is now a part of ETSI and 3GPP standards.

This paper is of technical nature and is organized as follows. In the second section, essential features of the OSA/Parlay service model are reviewed. The third section covers, very briefly, capabilities of the Ericsson OSA/Parlay Simulator. The simulator is a program that, for debugging and testing purposes, replaces an OSA/Parlay gateway (a server that interfaces a service provider to a telecommunications network). The fourth section explains what the implemented call center service does. The fifth section covers briefly fundamentals of OSA/Parlay programming. The next two sections present how to use the OSA/Parlay API to implement the call center. The goal is to give the reader a sense of what it takes to develop with the OSA/Parlay API. The last section concludes the paper.

During the paper presentation, the service program and the Ericsson OSA/Parlay Simulator will be started on a notebook computer, and service operation will be demonstrated.

### 2. OSA/PARLAY SERVICE MODEL

The OSA/Parlay API implies a specific service model. The key point is that service logic is separated from the telecommunications network. Moreover, the service logic may be easily placed outside the network operator's domain. The services are then provided by independent *value added service providers* (VASPs), as shown in Fig. 1 [ROJa,2002].

The network's functionality is made available through a so-called *gateway*. VASP's processes that implement the service logic are in OSA/Parlay referred to as *applications*. Applications run on

*application servers* and communicate with the gateway using the OSA/Parlay API. The communications between an application server and a gateway is based on CORBA.

The current version of the API covers most of functionality offered by modern telecommunications networks. It allows, among other things, call creation, interaction with users, user localization, and charging/accounting [ROJb,2002].

### 3. ERICSSON OSA/PARLAY SIMULATOR

The application described here has been wholly tested with the Ericsson OSA/Parlay Simulator. The simulator includes both a network simulation part and a gateway part. The network is controlled by means of a graphical user interface (GUI). This GUI allows to add terminals to the network, to start and release calls, and to type on a terminal's keypad. Text messages can be displayed on a terminal's screen. Announcements can be replayed by a terminal. The GUI makes it simple to inspect the state of the network. The architecture of the simulator is presented in Fig. 2.

The simulator supports two OSA/Parlay-defined so-called *Service Capability Features* (network functionality building blocks offered by the gateway and used to create services). The supported service capability features are Call Control and User Interaction. The former is used to create and manage calls, while the latter allows exchanging textual and audio information with terminals. The exact version of supported API is 3GPP Release's 4 TS 29.198 [3GPP,2001].

### 4. SIMPLE CALL CENTER SERVICE

A simple call center service developed with the Ericsson OSA/Parlay Simulator works as follows. The call center consists of a number agents helping end users (customers). The agents can be located anywhere, for example at their homes. A list of available agents is maintained all the time. The list may keep changing, depending on the day of the week, the current time, etc. Updating the available agents list does not involve OSA/Parlay and is not covered here. It is based on information specific to the company running the call center.

When a customer dials the call center number, an OSA/Parlay application running on a call center server is contacted. The application picks an available agent or decides that no agents are available. An agent selection algorithm applied does not involve OSA/Parlay and is not covered. Agent selection may be based on a number of criteria specific to the company.

If an agent is available, the application connects the customer to the agent. Otherwise, a polite message asking the customer to call later is displayed on his terminal.

### 5. OSA/PARLAY PROGRAMMING FUNDAMENTALS

From the OSA/Parlay application programmer's point of view, any running OSA/Parlay application can be considered a part of a single, distributed, multi-threaded process. (In what follows, the term "programmer" is used to denote the OSA/Parlay application programmer.) Some threads of that process run mostly (but not exclusively) on the application server (these threads constitute what is called an OSA/Parlay application), while others mostly on the gateway computer. The programmer provides the code for the former ones, which use the network's functionality to implement service logic. The latter ones are beyond the programmer's control; they are implemented by the gateway vendor. Their task is to make the network's functionality available in a convenient way (by hiding complexity of telecommunications signaling protocols). The OSA/Parlay API specifies how the application server-based threads communicate with the gateway-based ones.

The OSA/Parlay API separates the two components of the process, and makes it unnecessary for the programmer to know any specifics about the gateway-based threads. For example, it is perfectly valid for her to assume that there is only one such thread. Moreover, due to services provided by CORBA, the programmer need not even remember that the threads of these two types

run on different computers (the gateway and application server). The number and internal organization of the application server-based threads are up to the programmer (as long as they stick to the OSA/Parlay API). In a simple case, there may be only one such thread.

To summarize, the task of the programmer of a simple OSA/Parlay application can be stated as follows: implement a *service logic thread* that will interact with a vendor supplied peer *gateway thread* by means of the OSA/Parlay API.

In the OSA/Parlay model, the two threads communicate by creating (instantiating) a number of objects of OSA/Parlay-standardized types and then using the objects to exchange information. Some of these objects reside at the application server, and some at the gateway. Information flows between the two threads in both directions. If thread *A* wants to pass a piece of information (e.g., a notification or command) to thread *B*, thread *A* calls a method on an OSA/Parlay object residing at the location of thread *B*. Obviously, prior to that, the OSA/Parlay object must be created, and a reference to it must be made available to thread *A*. Passing the reference may be done using the same mechanism, but applied in the opposite direction. To start the communications, at initialization, one of the threads is given an initial reference to a single OSA/Parlay object. This initial reference is passed by a completely different means. (The initialization phase is not covered in this paper.)

An example of the OSA/Parlay communications is presented in Fig. 3. The figure is a UML collaboration diagram. In the figure, objects residing at the application server are located on the left hand side, and those at the gateway on the right hand side. Only OSA/Parlay method invocations and some OSA/Parlay object creation are shown. The same rules apply to all the UML collaboration diagrams in this paper.

In Fig. 3, the gateway thread has already created an object of an OSA/Parlay specified type called `IpCallControlManager`. The object resides at the gateway and represents call control functionality. A possible use of the `IpCallControlManager` type is to make it possible for the service logic thread to request notifications of events occurring in the network. For example, assume the service logic thread wants to be notified by the gateway thread when an end user dials a specific number. Then, as shown in Fig. 3, it calls the `enableCallNotification()` method on the `IpCallControlManager` object.

*Note:* assume that the service logic thread calls a method on an object created by (and residing at) the gateway thread. It is then that the service logic thread runs on the gateway computer, and not the application server. That is why we said that service logic threads execute mostly (but *not* exclusively) on the application server. A similar argument applies to gateway threads. On the other hand, CORBA makes the exact location of thread execution irrelevant.

As in any project involving communicating threads, the OSA/Parlay programmer has to take care of making the communications correct by, for example, making some methods thread-safe, using mutexes, etc.

## 6. CALL CENTER IMPLEMENTATION: EVENT REGISTRATION

The network's functionality made available by the gateway is split into modules called *Service Capability Features* (SCF). The call center application uses two SCFs: Generic Call Control Service (GCCS) and User Interaction (UI). GCCS ([PARa,2000], [PARb,2000]) allows the service logic thread to manage calls, while UI ([PARc,2000], [PARd,2000]) allows it to play announcements or display messages on an end user's terminal. In order to use an SCF, the service logic thread has to be given a reference to an OSA/Parlay *manager object* for that SCF. References to manager objects (one per SCF) are obtained by the service logic thread during its initialization (not covered here).

We assume that during initialization the service logic thread has obtained two manager object references. One, of type `IpCallControlManager`, makes it possible to access GCCS, and the other one, of type `IpUIManager`, does the same for UI.

*Note:* all the types mentioned in this paper are standardized by OSA/Parlay. Most OSA/Parlay types are so-called interfaces. Each type is specified by a list of its methods and a description of what each method is supposed to do. The purpose of different OSA/Parlay objects (and their types) should become clear, as we cover methods invoked on them. Also, all the OSA/Parlay types and methods mentioned in this paper are collected in Tab. 1.

The call center service logic thread needs to be notified each time an end user dials the call center number. This functionality belongs to GCCS. According to the general scheme outlined above, these notifications amount to the gateway thread's call of a method on an OSA/Parlay object residing at the application server. Hence, the first thing for the service logic thread to do is to create an object that can receive the notifications. The type of the object is `IpAppCallControlManager`. The gateway thread will use the `callEventNotify()` method invoked on that object to make a notification. Before this can happen, however, the gateway thread needs to be given (a) a description of what kind of a network event is to be monitored (an end user dialing the call center number), and (b) a reference to the object of type `IpAppCallControlManager` (so that `callEventNotify()` can be called). Such a registration, shown in Fig. 3, is performed by the service logic thread by invoking the `enableCallNotification()` method on the GCCS manager object (of type `IpCallControlManager`). Once the registration is done, the service logic thread awaits notifications of the call center end users' activity.

## 7. CALL CENTER IMPLEMENTATION: CALL PROCESSING

Now assume an end user dials the call center number. The action initiates a new call. The gateway thread finds out about the event by means of a network signaling protocol. Since that type of event has been registered for monitoring and notification by the service logic thread, the gateway thread creates an object of type `IpCall`. The object represents this specific call at the gateway. The `IpCall` type contains call management methods to be invoked by the service logic thread.

Now is the time to notify the service logic thread. (A respective collaboration diagram is shown in Fig. 4). The gateway thread calls the `callEventNotify()` method on the service logic thread's `IpAppCallControlManager` object. A reference to the `IpCall` object is passed as an argument of `callEventNotify()` thus enabling the service logic thread to manage the call. An integer *call ID* is passed the same way.

One of the responsibilities of the `callEventNotify()` method is to create a representation of this specific call at the application server. This is done by creating an object of type `IpAppCall`. The `IpAppCall` type contains call-related notification methods to be invoked by the gateway thread. A reference to the `IpAppCall` object is acquired by the gateway thread as a return value from `callEventNotify()`.

At this point the service logic thread knows that the call center number has been dialed. What follows is the service logic proper: the thread picks an available agent (and his routable number) or decides that there is no agent available. The determination is based on call center status data maintained by the service logic thread. This activity does not use the OSA/Parlay API in any way.

The first possible outcome is that an available agent has been picked. The agent is marked as busy, and the end user can now be connected to the agent. As making the connection is a call management function, the service logic thread uses the gateway thread's `IpCall` object. Specifically,

the `routeReq()` method is called on that object. The routable number of the call agent is passed as an argument to the method.

The invocation of the `routeReq()` method triggers making a connection between the end user and the agent. To that end, the gateway issues corresponding signaling messages to the network. When `routeReq()` returns, the service logic thread suspends processing the call.

Some time later, the end user, having been serviced by the agent, hangs up, and the call is terminated. The gateway thread finds out about the event by means of a network signaling protocol and notifies the service logic thread. As this is a call-related notification function, the gateway thread uses the service logic thread's `IpAppCall` object. Specifically, the `callEnded()` method is called on that object.

The service logic thread marks the agent as available again and releases all the resources associated with the call. The `deassignCall()` method is called on the gateway thread's `IpCall` object. That indicates that the gateway thread can dispose of the `IpCall` object. The `IpAppCall` object is then disposed of by the service logic thread.

The second possible outcome of the service logic proper is that there is no available agent. (This case is shown in Fig. 5.) The end user is supposed to be notified about that by a message displayed on his terminal. To display the message, the service logic thread uses the UI (User Interaction) SCF. Recall that the thread has access (reference) to the gateway thread's UI manager object (of type `IpUIManager`).

First, a *user interaction session* associated with the end user call has to be created. To do that, the `createUICall()` method is called on the `IpUIManager` object. In order to associate the user interaction session with the call being processed, the call ID is passed as an argument to `createUICall()`. The method (executing at the gateway) creates an object of type `IpUICall`. The object represents the user interaction session at the gateway. Methods of the `IpUICall` type allow the service logic thread to interact with the end user. The `createUICall()` method returns a reference to the `IpUICall` object so that it can be accessed by the service logic thread.

Now, the service logic thread can send a message to the end user's terminal. This is accomplished by invoking the `sendInfoReq()` method on the gateway thread's `IpUICall` object. A text to be displayed by the terminal is specified by an argument passed to the method. An additional argument instructs the gateway thread to destroy the user interaction session, including the `IpUICall` object (once the message has been displayed).

Finally, the service logic thread has to release all the resources associated with the original call. This is done, as described above, by calling the `deassignCall()` method on the gateway thread's `IpCall` object and disposing of the service logic thread's `IpAppCall` object.

## 8. CONCLUSION

A simple call center was implemented using the OSA/Parlay API and tested with the Ericsson OSA/Parlay Simulator. Basics of programming with the API were explained. Specifically, two important service capability features, Call Control and User Interaction, were presented.

The OSA/Parlay approach makes it much simpler to develop telecommunications services. The development process is accessible to people familiar with object-oriented software technology, but not necessarily with telecommunications protocols. Easier development and the fact that the new service model includes value added service providers enable rapid deployment of highly specialized services.

REFERENCES

- [YAT,2000] Yates M. J., Boyd I.: The Parlay Network API Specification, *BT Technology Journal*, no. 2, vol.18, 2000
- [MOE,draft] Moerdijk A.-J., Klostermann L.: Opening the Networks with Parlay/ OSA APIs: Standards and Aspects behind the APIs. Draft version to be resubmitted to IEEE Communications Magazine.
- [ROJa,2002] Rój, M.: An Introduction to Parlay/OSA APIs, *M.S. thesis, Warsaw University of Technology*, 2002.
- [ROJb,2002] Rój, M., Domaszewicz J.: Service Creation with Parlay/OSA APIs, *Proceedings KST 2002*, 2002 (in Polish).
- [3GPP,2001] Third Generation Partnership Project, OSA APIs Specification, 3GPP Standard TS 29.198 (R4), 2001
- [PARa,2000] The Parlay Technical Team, Parlay APIs 2.1 – Generic Call Control Service Interfaces, version 2.1, 2000
- [PARb,2000] The Parlay Technical Team, Parlay APIs 2.1 – Generic Call Control Service Data Definitions, version 2.1, 2000
- [PARc,2000] The Parlay Technical Team, Parlay APIs 2.1 – Generic User Interaction Service Interfaces, version 2.1, 2000
- [PARd,2000] The Parlay Technical Team, Parlay APIs 2.1 – Generic User Interaction Service Data Definitions, version 2.1, 2000

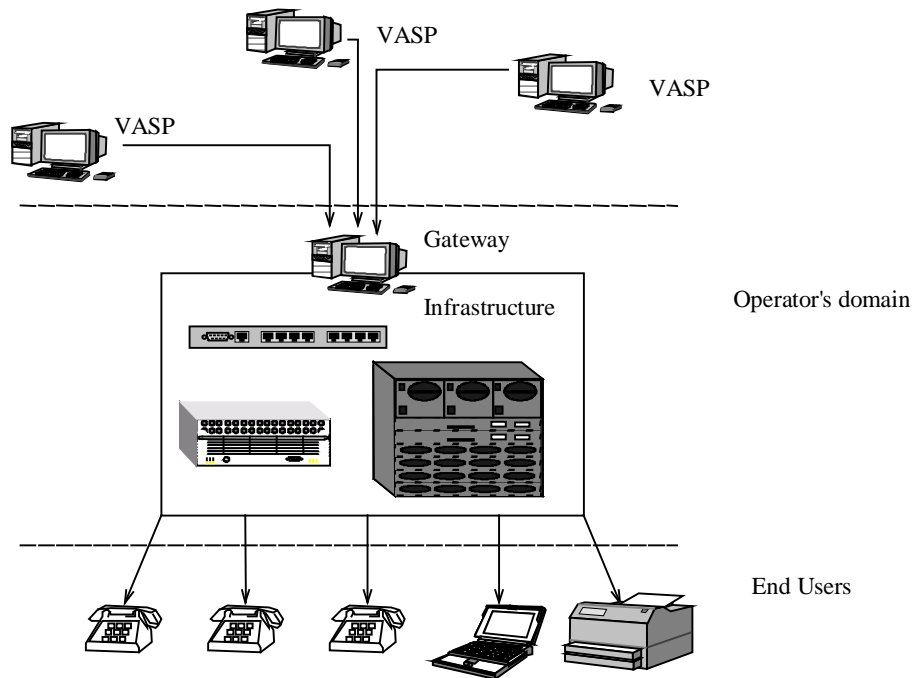


Fig.1. The OSA/Parlay service model

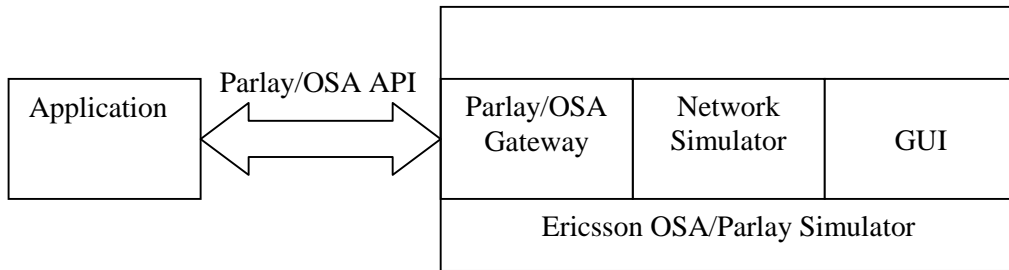


Fig. 2. The application and the Ericsson OSA/Parlay Simulator

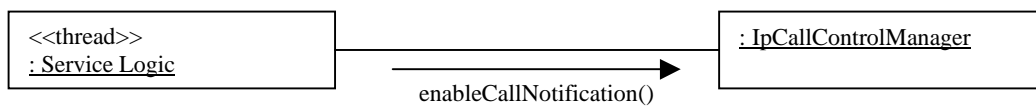


Fig. 3. An example of the OSA/Parlay communications between the service logic thread and gateway thread

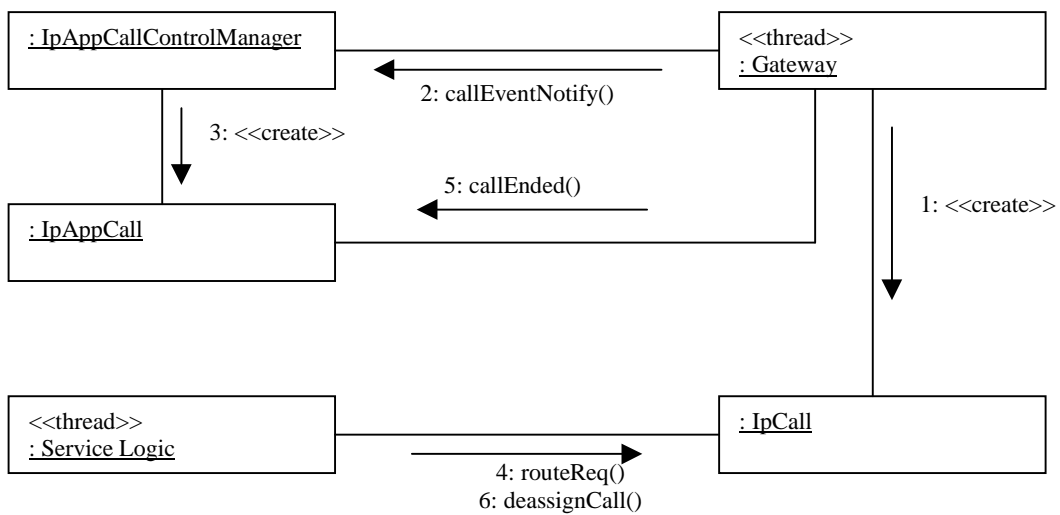


Fig. 4. Call processing (an available agent has been found)

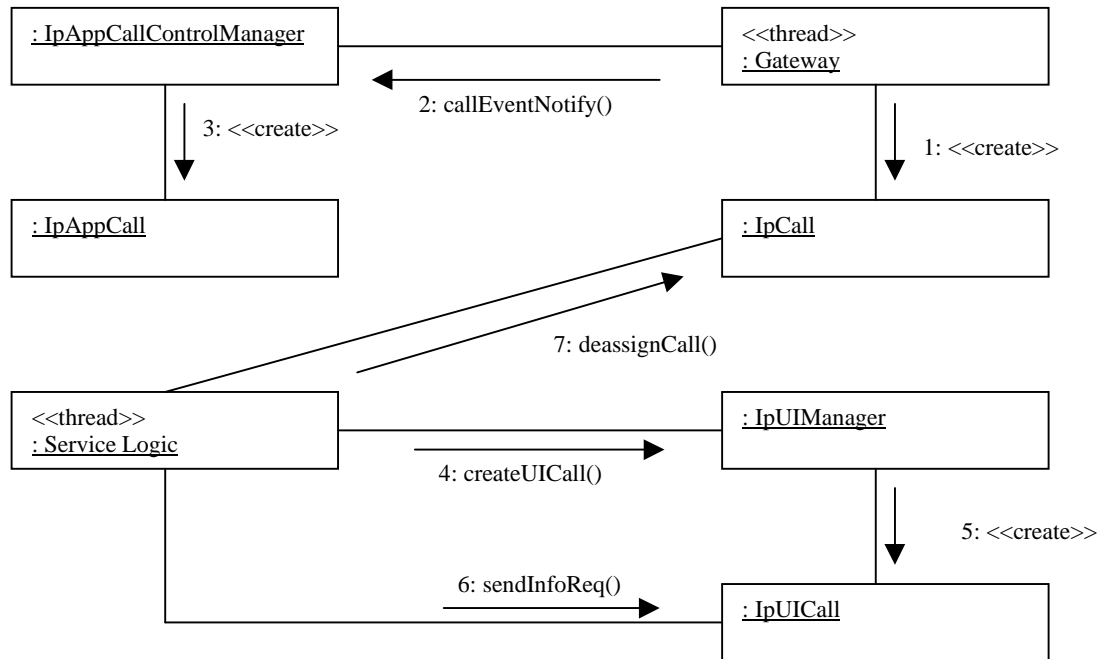


Fig. 5. Call processing and user interaction (an available agent has not been found)

Tab. 1. OSA/Parlay types and methods mentioned in this paper

<i>OSA/Parlay data type</i>	<i>Purpose of the type</i>	<i>Methods of the type</i>
IpCallControlManager	Access to GCCS functionality (by the service logic thread)	enableCallNotification()
IpAppCallControlManager	Notification of registered events (by the gateway thread)	callEventNotify()
IpCall	Single call management (by the service logic thread)	routeReq()
IpAppCall	Notification of call-related events (by the gateway thread)	callEnded()
IpUIManager	Access to UI functionality (by the service logic thread)	createUICall()
IpUICall	Sending and receiving messages to/from end users (by the service logic thread)	sendInfoReq()