

FLEXIBLE SUPPORT OF ORB INTEROPERABILITY

N. Fischbeck, E. Holz, O. Kath
Humboldt University Berlin
Axel-Springer-Str. 54a, 10099 Berlin, Germany
{fischbec|holz|kath}@informatik.hu-berlin.de

V. Vogel
Deutsche Telekom AG
Am Kavalleriesand 3, 64295 Darmstadt, Germany
vogel@fz-telekom.de

ABSTRACT

The Camouflage Project is a national german research initiative led by the Deutsche Telekom. It aims at the provision of a TINA conform distributed processing environment based on existing legacy telecommunication systems.

To achieve this goal new concepts for the interoperability between CORBA islands have been developed and implemented. This comprises pluggable Message Transfer Protocols and Remote Operations Protocols. Both features enable ORB's to interwork highly efficient on top of existing telecommunication networks, although the ap-

proach is not restricted to telecommunication networks.

Selected results of the project have been brought into the OMG standardization process.

KEYWORDS

CORBA, TINA, INTEROPERABILITY, TELECOMMUNICATION, REAL TIME, COMMUNICATION PROTOCOLS,

1. INTRODUCTION

To deploy CORBA technology as a central building block of a Distributed Processing Environment in a specific domain often the sole availability of the General Inter-ORB Protocol and the TCP/IP based IIOP is not sufficient. In the telecommunications domain for example, the use of GIOP/IIOP instead or on top of existing telecommunication protocols would result in limitations on the efficiency and quality of ORB interconnection services. Moreover, it would pay no attention to proven networking technologies as the reliable and efficient SS.7. However, a direct inclusion of alternative mappings into the ORB does currently require for each protocol changes to the ORB core.

Within CAMOUFLAGE a well defined but small interface between the ORB and the transport protocols for GIOP messages has been developed, which decouples the ORB from the underlying networking technology. This does not only make the ORB open for the introduction of new interoperability protocols but enables also the use of different message transfer protocols by the same ORB.

Furthermore, within the telecommunication as well as in other domains special remote operation protocols and mechanisms suitable to support object interaction do already exist. An example for such protocols is the Transaction Capabilities Application Protocol (TCAP) of the Signaling System Number 7 (SS.7) in the telecom domain. The application of these protocols for object interaction in a CORBA environment requires a more abstract plug-in service. This approach will make it more easy to define, implement and plug-in an Environment Specific Inter-ORB Protocol (ESIOP) for a given domain.

2. THE OPEN COMMUNICATION INTERFACE

To achieve the objectives described above, the Open Communications Interface (OCI) has been defined as a common mechanism for pluggable protocols. The OCI is structured into two components, the *Message Transfer Service* and the *Remote Operation Service*.

TCP/IP as the currently only transport protocol for inter-ORB interactions is obviously a candidate for a Message Transfer Service plug-in. More interesting for the telecommunications domain however

are other protocols as SCCP (Signaling Connection Control Part, part of SS.7) or SAAL (Signaling ATM Adaptation Layer). If the implementation of the plug-in takes care of issues as connection management and reliability, even non-reliable or non-connection-oriented protocols as UDP/IP can be used

For the implementation of the second OCI component, the *Remote Operation Service* also different and domain specific solutions are possible. Certainly the GIOP falls is one of them, however, going beyond the current CORBA scope also DCE-RPC and TCAP (defined by SS.7) provide the appropriate mechanisms and can therefore be used as *Remote Operation Service* plug-ins.

This two-level approach to add interoperability mechanisms to an ORB allows not only to deploy a wide range of different protocols but also to select dynamically an appropriate service according to application requirements during runtime.

Depending on the protocol such plug-ins can be very light-weight having the ORB handle most of the protocol logic. Additionally, it is also possible to make use of existing remote procedure call mechanisms in domains where this is desirable. This enables the ORB itself to be lightweight. The OCI does also allow to combine plug-ins for the *Remote Operation* and the *Message Transfer Service*, provided both plug-in support each other. The combination of a GIOP Remote Operation Service plug-in with a TCP/IP Message Transfer Service plug-in is one example. These two plug-ins together then implement the IIOP protocol.

2.1. The Message Transfer Interface

2.1.1. Design Patterns

The Message Transfer Interface is based on the commonly used design patterns Connector and Acceptor [7]. These patterns already provide the basis for most ORB implementations, so in many cases this allows ORB vendors to switch to the Message Transfer Interface by simply “wrapping” existing code.

2.1.2. Exceptions

The Message Transfer Interface does not define any new exceptions. This would result in a severe performance penalty since such exceptions would

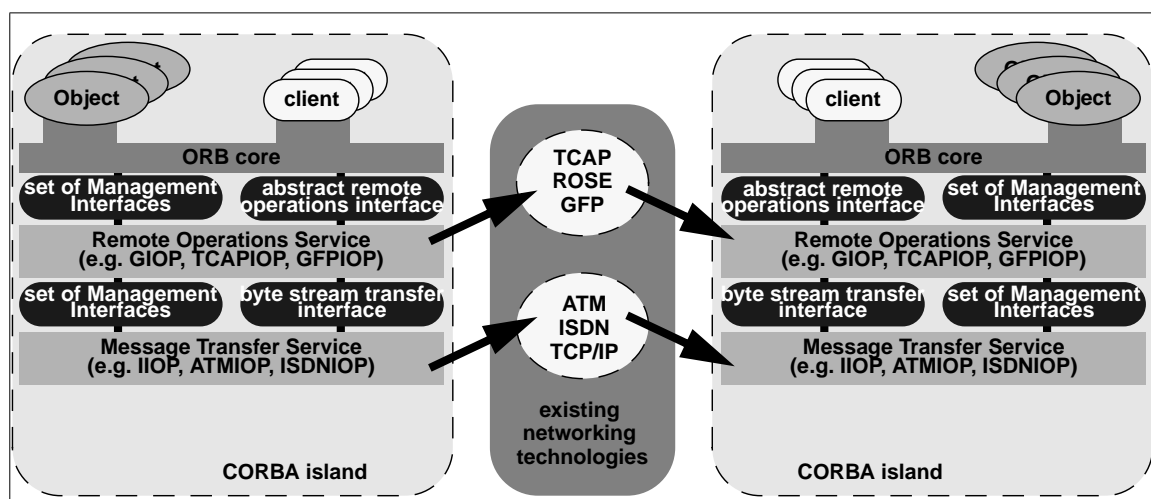


Figure 1 Open Communication Interface - Schematic Structure

have to be caught by the ORB and translated into standard CORBA system exceptions.

Instead, system exceptions (i.e. exceptions derived from `SystemException`) are used. That is, all operations are allowed to throw system exceptions but no other exceptions. A separate set of minor exceptions codes has to be defined for each protocol plug-in.

2.1.3. Thread Safety

Plug-ins for the Message Transfer Interface do not have to be thread safe. It's the ORB's responsibility to ensure a serialized access to the interfaces. This allows using plug-ins for single-threaded and multi-threaded ORBs without performance loss.

2.1.4. Object References

For the representation of Object References the Message Transfer Interface uses Interoperable Object References (IORs) as described in "An Information Model for Object References" in the CORBA Core specification.

2.1.5. Locality Constraints

All objects used by the Message Transfer Interface must be local to the process.

2.1.6. Packet Size and Fragmentation

According to the General Inter-ORB Protocol (GIOP) Version 1.2, fragmentation of GIOP messages is supported. For optimization of inter-ORB com-

munication, the preferred packet size for a particular message transfer protocol can be obtained from an instance of `Transport`. In case of a SCCP plug in this preferred packet size should be set to an appropriate value for SCCP SDUs, i.e. 4096 octets.

2.1.7. Quality of Service Issues

Parts of the Open Communication Interface relate to proposals for Quality of Service framework definitions from [2]. In particular, the definitions of appropriate policies given to the operation `compare_with_policies` of interface `Connector` will be aligned with those defined in the Messaging Service.

The proposal of the Open Communication Interface does not include any definition of new policies.

2.1.8. Relationship to POA

The proposal of the Open Communication Interface does not rely on a particular Object Adapter, except

- the definition of policies for the POA, which have an impact on the underlying transport protocol and
- the access to `add_acceptor` operation of `AcceptorRegistry` interface as a read-only attribute of a POA.

With the POA, it is possible to register acceptors of different transport protocols to a particular POA or a set of POA's.

2.2. Interface Summary

2.2.1. Buffer

An interface for a buffer. A buffer can be viewed as an object holding an array of octets and a position counter, which determines how many octets have already been sent or received.

2.2.2. Transport

The Transport interface allows the sending and reception of octet streams in the form of Buffer objects. There are blocking and non-blocking send/receive operations available, as well as operations that handle time-outs and detection of connection loss.

2.2.3. Acceptor and Connector

Acceptors and Connectors are Factories for Transport objects. A Connector is used to connect clients to servers. An Acceptor is used by a server to accept client connection requests.

Acceptors and Connectors also provide operations to manage protocol-specific IOR profiles. This includes operations for comparing profiles, adding profiles to IORs or extracting object keys from profiles.

2.2.4. Connector Factory

A Connector Factory is used by clients to create Connectors. No special Acceptor Factory is necessary, since an Acceptor is created just once on server start-up and then accepts incoming connection requests until it is destroyed on server shutdown. Connectors, however, need to be created by clients whenever a new connection to a server has to be established.

2.2.5. The Registries

The ORB provides a Connector Factory Registry and the Object Adapter provides an Acceptor Registry. These registries allow the plugging-in of new protocols. Transport, Connector, Connector Factory and Acceptor must be written by the plug-in implementors. The Connector Factory must then be registered with the ORB's Connector Factory Registry and the Acceptor must be registered with the POA's Acceptor Registry.

2.2.6. Class Diagram

Figure 2 shows the classes and interfaces of the Message Transfer Interface. The ORB must provide abstract base classes for the interfaces Connector Factory, Connector, Transport and Acceptor. The protocol plug-in must inherit from these

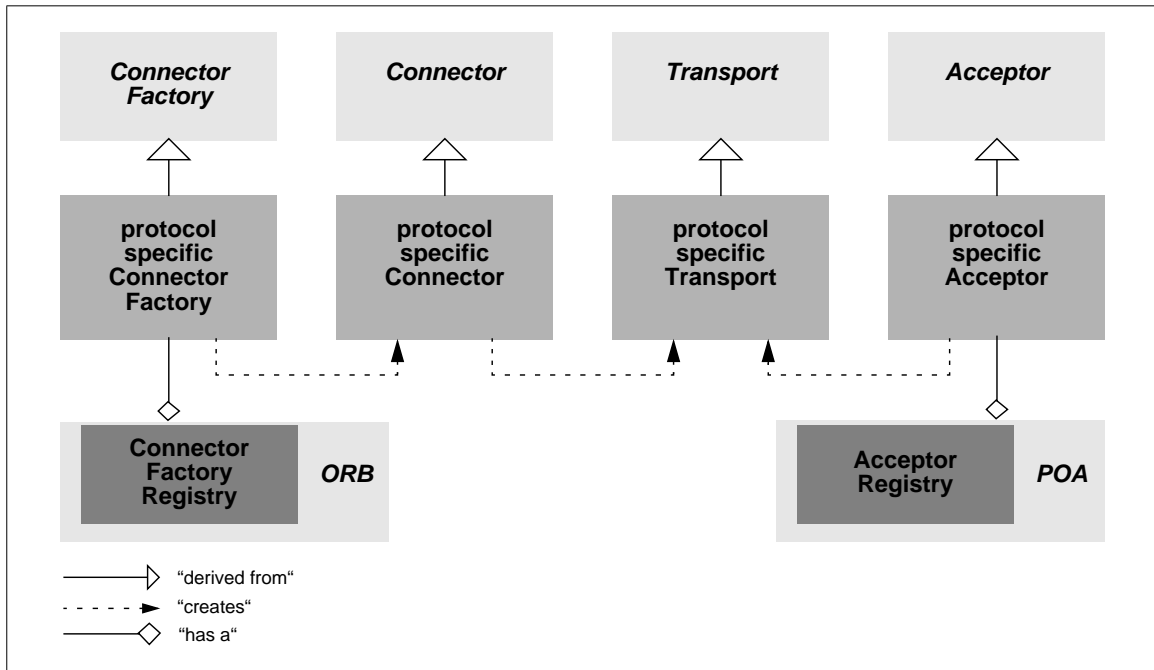


Figure 2 Interface Diagram and Basic Relationships

classes in order to provide concrete implementations for a specific protocol. The ORB must also provide concrete classes for the interfaces Buffer, Connector Factory Registry and Acceptor Registry. An instance of the Connector Factory Registry and the Acceptor Registry is provided by the ORB and Object Adapter, respectively. Concrete implementations of the Connector Factory must be registered with the ORB's Connector Factory Registry and concrete implementations of the Acceptor must be registered with the Acceptor Registry of a POA.

2.3. The Remote Operations Interface

In a same manner, like the Message Transfer Interface supports Plug-in of transport protocols, like TCP/IP, UDP or SCCP, the Remote Operations Interface will support the Plug-In of so called RPC-style protocol Plug-ins, like GIOP, TCAP, Real-time Interaction Protocols or DCE-RPC.

3. APPLICATION

3.1. Usage of Message Transfer Part of OCI

This section describes,

- which parts of OCI has to be implemented by the protocol plug-in provider
- how a protocol plug-in will be added to an ORB and an OA,
- how a connection will be established,
- how messages are exchanged.

The protocol example used in this paragraph will be IIOP.

3.1.1. Protocol Plug-in Implementation Example

The protocol specific parts of a particular transport protocol has to be implemented by the plug-in provider. Those specific parts regard management of IOR profile bodies, accept incoming connections, connection establishment, send and receive of octet sequences.

3.1.2. Management of IOR profile bodies

Since an acceptor for a particular transport protocol is registered with a POA, the operation **add_profile** of **acceptor** will be called for example, if a servant is registered with a POA. This operation adds the protocol specific address information to the given IOR. Assuming, that the class

implementing an IIOP acceptor is named **IIOPAcceptor**, the implementation of **add_profile** could look like:

```
void IIOPAcceptor::add_profile (const
OCI_ObjectKey& key, OCI_IOR& ior)
{
CORBA_IIOP_ProfileBody body;
body.iiop_version.major = 1;
body.iiop_version.minor = 0;
body.host = // the host;
body.port = // the port;
body.object_key = key;
// add an IIOP profile to ior.profiles
ior.profiles.length(
ior.profiles.length()+1);
ior.profiles[ior.profiles.length()-1].tag
= CORBA_IOP_TAG_INTERNET_IOP;
// ... marshal CORBA_IIOP_ProfileBody
// ... add it to IIOP profile
}
```

3.1.3. Accept incoming Connections

If an object is activated, for instance using

```
my_poa -> the_POAManager() -> activate ();
orb -> run ();
```

the operation **accept** of **Acceptor** will be called.

An implementation of **accept** returns a **transport** object, which is afterwards responsible for the established connection. an implementation of **accept** for an IIOP Plug-in can look like

```
OCI_Transport_ptr
IIOPAcceptor::accept()
{
// ... find free port or use given port
// ... listen

// accept
if((_fd = ::accept(fd_, (struct
sockaddr*)&newAddr, &len)) == -1)
throw
CORBA_COMM_FAILURE(getError(),MinorAccept,
CORBA_COMPLETED_NO);

// set socket options
if(setsockopt(_fd, IPPROTO_TCP,
TCP_NODELAY, (Char*)&no, sizeof(int))
== -1)
throw
CORBA_COMM_FAILURE(getError(),MinorSocketopt,
CORBA_COMPLETED_NO);
// create a Transport
OCI_Transport_ptr _transport = new
IIOPTransport(_fd);
return _transport;
}
```

3.1.4. Connection Establishment

If a client application wishes to request operations on a servant, the clients ORB will try to establish a connection to the host, where the servant resides. It gets this information through the given IOR, but the evaluation, whether the host can be reached through a particular transport protocol can only be

done by the appropriate protocol plug-in itself. So the ORB performs the operation

```
boolean consider reference (in IOR ior, in
CORBA::PolicyList policies)
```

on each registered **ConFactory** object. This operation retrieves its known profile from the given IOR and evaluates the profile body part with respect to the given policies. If this operation returns **CORBA_TRUE** and the ORB decides to select an IIOP connection, the operation **create** or **create_with_policies** on that **ConFactory** object will be called. This operation will return a new **Connector** object. Assuming, that the constructor of an **IIOPConnector** takes a host and a port as parameter, an implementation of **create(in IOR ior)** will be something like

```
OCI_Connector_ptr
IIOPConFactory::create(const OCI_IOR& ior)
{
    for(CORBA_ULong i = 0 ;
        i < ior.profiles.length() ; i++)
    {
        // get the IIOP profile
        if(ior.profiles[i].tag ==
            CORBA_IOP_TAG_INTERNET_IOP)
        {
            // Get the IIOP profile body
            const CORBA_Octet* oct =
                ior.profiles[i].profile_data.data();
            CORBA_IIOP_ProfileBody body;
            Unmarshal(body, oct, endian
                /* the appropriate endian */);
            return new IIOPConnector(body.host,
                body.port);
        }
    }
    return OCI_Connector::_nil(); // oops!
}
```

The actual connection establishment can then be done by calling the operation **Transport connect()** on the instantiated **Connector** object. An example implementation of **connect** for IIOP is something like

```
OCI_Transport_ptr IIOPConnector::connect()
{
    // Get address
    GetInAddr(host, addr);
    // Create socket
    if ((fd_ = CreateSocket()) == -1)
    {
        throw
        CORBA_COMM_FAILURE(getError(), MinorSocket,
            CORBA_COMPLETED_NO);
    }
    // ...
    int no = 1;
    // set socket options
    if(setsockopt(fd_, IPPROTO_TCP,
        TCP_NODELAY, (char*)&no, sizeof(int))
        == -1)
    throw
    CORBA_COMM_FAILURE(getError(), MinorSocket,
        CORBA_COMPLETED_NO);
    // Connect
```

```
if (::connect(fd_,
    (struct sockaddr*)&addr_,
    sizeof(addr_)) == -1)
    throw
    CORBA_COMM_FAILURE(getError(), MinorConnect,
        CORBA_COMPLETED_NO);

// create transport
OCI_Transport_var _transport = new
IIOPTransport(fd_);
return _transport;
}
```

Note, that transports for clients and servants side are the same. After successful execution of **connect**, the object interaction in terms of sending GIOP messages through the established connection can take place.

3.1.5. Send and Receive of Octet Sequences

Since a Transport object is being instantiated, sending and receiving of messages in terms of octet sequences is possible through the instantiated connection.

An implementation example for receive could look like

```
void
IIOPTransport::receive(OCI_Buffer_ptr
buf, CORBA_Boolean block)
{
    // The Block Parameter Will Be
    // Ignored In This Example
    while(!buf -> is_full())
    // a buffer is full, if the number of
    // received octets
    // equals to a number of required octets
    // provided by
    // the caller, for instance 12 to receive
    // the message header
    {
        int result = ::recv(fd, (char*)buf ->
            rest(), buf -> rest_length(), 0);
        // buf -> rest() is the start of
        // allocated memory,
        // where the received bytes have to be
        // written
        // buf -> rest_length() is
        // (number of required bytes) - (number of
        // already received bytes)
        if(result == 0)
            throw
            CORBA_COMM_FAILURE("", MinorRecvZero,
                CORBA_COMPLETED_NO);
        else if(result == -1)
            throw
            CORBA_COMM_FAILURE(getError(), MinorRecv,
                CORBA_COMPLETED_NO);
        buf -> advance(result);
        // buf -> advance ( int ) sets the number
        // of already received
        // bytes to the new value after this round
    }
}
```

To receive a Request message, the operation **receive** can be called two times

```
// receive the GIOP Message Header,
// which always has a length of 12 byte
```

```

_my_buf = allocateBuf ();
_my_buf -> length ( 12 );
_my_transport -> receive ( _my_buf,
                          CORBA_TRUE );
// demarshal message header, find out
// message type, find out message length
// set new length of buffer
_my_buf -> length ( 12 + <message length>);
// receive GIOP request header and
// request body
_my_transport -> receive ( _my_buf,
                          CORBA_TRUE );
// demarshal all
// dispatch request

```

Note, that the transport could of course fill the buffer in the first invocation of receive with a complete message, like it would be the case in the GIOP over SCCP approach, so the second call of receive is unnecessary.

The `send/send_detect/send_timeout` operations of a `Transport` object are responsible for sending encoded messages through the connection it is responsible for.

```

void
OBIIOPTransport::send(OCI_Buffer_ptr buf,
                     CORBA_Boolean block)
{
// The Block Parameter Will Be Ignored
// In This Example
while(!buf -> is_full())
{
// same approach for is_full like receive,
// but for sent bytes
int result=:send(fd,
                (const char*)buf ->rest(),
                buf -> rest_length(),0);
if(result == 0)
    throw CORBA_COMM_FAILURE("",
        MinorSendZero,_CORBA_COMPLETED_NO);
else if(result == -1)
    throw
CORBA_COMM_FAILURE(getError(),MinorSend,
        CORBA_COMPLETED_NO);
buf -> advance(result);
}
}

```

An application of `send` would be the transfer of an

operation request:

```

// allocate a buffer
_my_buf = allocateBuf ();
// set buffers length
_my_buf -> length (
<marshaled message header length> +
<marshaled request header length> +
<marshaled request body length> );
// put marshaled GIOP message header,
// GIOP Request Header
// and Request body to the buffer
_my_transport -> send ( _my_buf,
                      CORBA_TRUE );

```

3.2. Plug-Ins for Telecommunication Protocols

Within the CAMOUFLAGE project different OCI plug-ins have been developed resp. are under investigation. The protocols used here come from the telecommunications domain. Especially protocols from B-ISDN (Generic Functional Protocol, Q.2932) and narrowband ISDN (based on X.31) have been in the centre of interest. However, current studies do also cover the area of Intelligent Networks (IN) and ATM protocols and remote-procedure-call technologies.

Pluggable protocols in IN networks should allow dynamic binding of the ORB core to GIOP-based protocols and ESIOPs of the TCP/IP and SS7 protocol families. Figure 3 shows at the left side the GIOP and the TCP/IP plug-in forming an IIOp stack and on the right the DCE-RPC plug-in and the TCP/IP plug-in as an ESIOP.

Transforming this to the SS7-world, we get the configuration which is given in Figure 4 with GIOP over SCCP on the left side or TCAP and SCCP plug-ins at the right establishing an ESIOP on top of the MTP transport network .

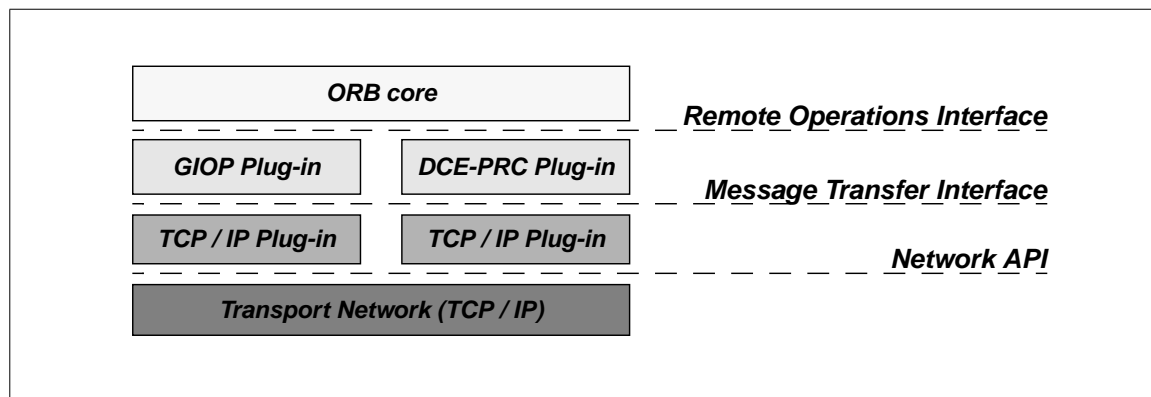


Figure 3 Different Remote Operation Protocol Plug-ins using TCP/IP

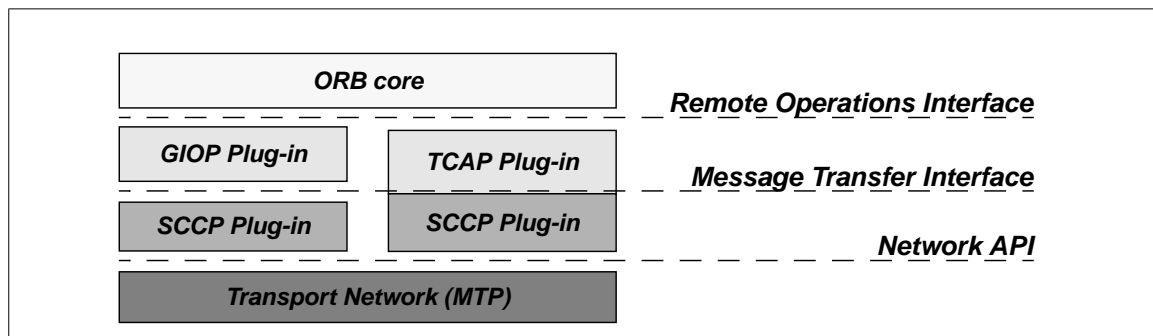


Figure 4 Different Remote Operation Protocol Plug-ins using SCCP

4. CONCLUSION

The Open Communication Interface concepts, which are explained in this paper, were implemented and tested with the following protocols and scenarios

- For the *Message Transfer Interface* part of the *Open Communication Interface*,
- an IIOP protocol Plug-in was implemented in JAVA and C++ and tested with OmniBroker from Object Oriented Concepts Inc. These Plug-ins are now part of the OmniBroker product.
- a SSL supporting Plug-in was implemented and tested with the same ORB product. It is in a Beta-state, which means feature complete.

- a Plug-in based on Broadband-ISDN/ATM was implemented and tested with the same ORB product. It is in a Beta-state, which means feature complete.
- a Plug-in based on Narrowband-ISDN was implemented and tested with the same ORB product. It is in a Beta-state, which means feature complete.

The definition of the Open Communication Interface has been submitted to the OMG standardisation process (Real-Time, CORBA-IN-Interworking). Parts of this work have been contributed also to the TINA.

ACKNOWLEDGEMENT

This work was supported by the Deutsche Telekom AG within the Telekom-project SIGTINAL. We would like all colleagues at Humboldt-University, Deutsche Telekom AG and Omnibroker Inc. who are involved in this work for the helpfull discussions.

REFERENCES

- [1] Object Management Group, CORBA 2.1, OMG, 1997
- [2] Messaging Service Revised Submission , OMG document number orbos/98-03-11
- [3] ITU-T Rec.Q.2932.1: B-ISDN, Digital Subscriber Signalling System No.2, Generic Functional Protocol. Core Functions.; Geneva, 1996
- [4] ITU-T Rec. Q.931: Digital Subscriber Signalling System No. 1 (DSS 1) - ISDN User-Network Interface Layer 3 Specification for Basic Call Control, ITU-T 1993
- [5] ITU-T Rec. Q.771: Signalling System No. 7 - Functional Description of Transaction Capabilities
- [6] O.Kath, E. Holz, M. Geipl, G.Lin, V. Vogel: The Camouflage Project - Introduction of TINA into Telecommunication Legacy System, Proc. of TINA'97
- [7] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.