Service Trading and Mediation in Distributed Computing Systems

M. Merz, K. Müller, W. Lamersdorf

Hamburg University

Department of Computer Science; Databases and Information Systems
Vogt-Kölln-Straße 30; D-22527 Hamburg, Germany

[merz|kmueller|lamersd] @ dbis1.informatik.uni-hamburg.de

Abstract

The increased availability of global communication infrastructures allows providers and users of various application services to cooperate in nearly unlimited geographic scopes. Problems of heterogeneity and scale have motivated specific standardisation activities for client/server "trading" or service "mediation" components. Motivated by current limitations of the emerging ODP (Open Distributed Processing) trader, this paper argues for a broader concept of general service "mediation" as more appropriate for realistic open distributed environments. The proposed mediation concept addresses some of the heterogeneity and flexibility requirements of open service co-operation by a uniform "Service Interface Description Language" (SIDL). The goal is to support distributed application development for a "Common Open Service Market" (COSM) by facilitating flexible service selection and client/server interaction.

The paper also presents basic components of a generalised trading and mediation architecture as well as the status of a prototype implementation.

1. Introduction

The infrastructure of modern world-wide communication networks provides the basis for the interconnection and co-operation of great numbers and varieties of geographically separated, independent application components. The predominant structuring paradigm for distributed applications in such an environment is the well-known client/server or client/service model which distinguishes clients and service providers (servers) with regard to their different roles in distributed application cooperations [1]. Although this model has proven to be well suited for cooperation in open distributed systems, generalisations from small to large-scale open systems seem to require additional system support.

In general, client/server systems are considered to be *open* as long as they interoperate on the basis of agreed (i.e. standardised) communication mechanisms. From an *application* point of view, however, they may still appear as *closed* systems whenever specific details of the respective cooperation partners have to be known in advance. In order to demonstrate this, a remote "car rental" server which is accessible via standard "Remote Procedure Call" (RPC) mechanisms may serve as a simple example: Besides the use of agreed, standardised communication protocols, most current implementations of such a distributed application would require the client application to have very specific a-priori knowledge of the service addressed as well as about the related protocol. Approaches to client/server cooperation in open distributed environments - as currently discussed in the respective international standardisation committees (here especially: ISO Open Distributed Processing, ODP) - address this problem by making the additional knowledge available through dedicated "trading" functions; they are, however, still limited to fairly *close* (i.e. well-informed about each other, predefined, or 'standardised') client/server cooperations.

In this sense, an ODP trading service (as further described in chapter 2) may perform well for a distributed application that is *closed* from an application point of view, i.e. an application which requires both client and server to be closely linked together (e.g. by a trader function) such that they are well-informed about each other. In such client/server cooperation scenarios as mostly discussed today, both client and server are expected to have a common agreement of, e.g., the service (type) characterisation of the addressed service (e.g. a specific 'CarRentalService' in the example above).

In more general scenarios of *dynamically changing*, unrestricted *global* open communication infrastructures, however, various and heterogeneous kinds of service offers and requests may typically arise at *arbitrary* points in time and space in a distributed network system. Based on the assumptions of the traditional service "trading" scenario as sketched above, this would - for any global trader or specific client application - mean to have dedicated knowledge of and agreement with (potentially) all different service types available in the network. This does not seem feasible beyond the limitations of relatively small networks (resp. service) boundaries and would - at least - prevent new, alternate, competing, or complementary service providers to enter such a distributed service environment - to the potential benefit of the clients - freely and with without utilisation delay.

In order to overcome the obvious mismatch between - on the one hand side - the flexibility requirements of (client) distributed system users and - on the other hand side - the necessity to closely (and correctly!) align suitable networks components to one another based on agreed service characteristics and cooperation protocols, a generic system infrastructure for a "Common Open Service Market" (COSM) is proposed and presented in this paper. In addition to the (ODP) *trading-oriented* cooperation

schema [2], this architecture proposes (and supports) a *user-oriented* cooperation schema which allows clients to stay independent from the details of specific service providers. In this schema, the human user is directly involved in (yet unknown) service selection and application specific interaction. Based on a common (i.e. uniformly defined) *Service Interface Description (SID)*, the proposed open system support infrastructure allows to automatically generate service-specific client components like a so-called "*Generic Client*", a "*Service Browser*" (for more details see section 3.2) etc.. In the resulting open system infrastructure, *any* appropriate service provider in the network can be selected in a flexible manner based on human end-user interaction and supported by appropriate dedicated system support functions.

Therefore, the basic question addressed in this paper is how the two alternate approaches to matching client requests with the 'right' service offers in large-scale open system environments can be best combined: the basic idea is to *integrate* the approaches of - on the one hand side - (ODP) *trader*-oriented and - on the other - more flexible *human*-oriented, interactive service selection mechanisms into a common open system support infrastructure.

Accordingly, the rest of the paper is organised as follows: First, an introduction to client/server "trading" functions and their limitations is given in section two. Then, ODP trading is generalised into open service "mediation" in chapter three. This chapter also reviews and extends some aspects of a "Service Interface Description Language" (SIDL) as an adequate uniform service specification technique for open service trading and mediation in distributed computing systems. Finally, the last chapter presents an architecture and the current status of a distributed prototype system integrating ODP trading and open service mediation concepts in system support for an example heterogeneous network scenario.

2. Service *trading* in distributed systems

Trading is the process of matching client service requests with corresponding service offers accessible somewhere within a distributed open systems network. The specification of a trader function is currently carried out within the ISO ODP standardisation activities [3]. Therefore, in the remainder of the paper the abbreviation *ODP trader* is used for ODP-conform trading services [2]

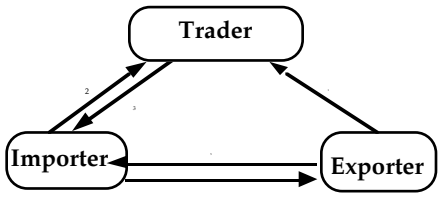

Figure 1: AN ODP Trader and Its Users

2.1. Functionality of the ODP trading service

A closer review of the overall ODP trader functionality shows that a compound trader service is comprised of different interfaces offered to its different kinds of potential clients (Fig. 1) [4]. *Exporters* are service providers which register their interface in a client role at the trader (step 1) while *importers* issue a request for providers of a particular service (step 2). After an appropriate "best possible" (according to given criteria) service is selected and one or more *service identifiers* are returned to the importer (step 3), a direct *binding* can be established eventually between the importing client application and the selected service provider (steps 4 and 5).

In this triangular relationship, the basic role of the trader is to provide and maintain means of *classifying* services by given *service type*s [5]. Such service types identify distinct operational interface signatures, characterised by an *interface type*, and a predefined set of characterising attributes (service properties). So, the notion of the *service type* plays a central role in an ODP trading context. By defining an interface type and a set of attributes for a given service, it provides the basis for a common understanding of the function and semantic of a special service class. Therefore, a server exporting a service to the trader, always has to refer to a distinct, predefined service type.

The above mentioned car rental service may also serve as an example for an ODP trader function. A specific new car rental service may register its service in an open network environment by first referring to a (predefined) service type *CarRentalService*, which can be defined as follows:

```
ServiceType CarRentalService {
    // Service Attribute Types
    ServiceAttributeTypes {
        CarModel :          Enum { AUDI, FIAT-
                            Uno, VW-Golf }
        AverageMilage :     Integer
        ChargePerDay :      Float
        ChargeCurrency :    Enum{ USD, DEM, FF,
                            SFR, GBP }
    };
    // Service Signature
    ServiceSignature {
        SelectCarReturn_t   SelectCar ( SelectCar_t);
        BookCarResult_t     BookCar( BookCar_t);
    }
};
```

Then, the actual service offer as exported by a particular car rental service has to specify the values for all attributes of the service type according to the template given by the service type *CarRentalService*, for example "CarModel = FIAT-Uno".

In general, the *computational interface* of an ODP trader provides operations for the export, withdrawal, and replacing of exported services as well as operations to retrieve a list of services which conforms to any given client request, including the possibility to obtain a *best-fitting* service according to some given criteria. Furthermore, a *management interface* allows to modify the domain of service offers by inserting or deleting specific service type entries.

An important advantage of involving a trader function for service acquisition in open distributed environments lies in a gain of additional *distribution transparency*: For example, a service user in such a scenario is not required to

know about the exporters network address, host type, or other implementation details.

2.2. Limitations of the current ODP trader

In principle, the concepts underlying the ODP trading function are well suited in case of services which are well-established, i.e. those for which their respective functionality, service type and other characteristics are well-known (i.e. standardised) to all requesting clients. Referring to the example given above, that means that an exporter of a "car rental" service has an explicit knowledge of the existence of a service type 'CarRentalService' managed by the trader with prescribed service semantics. Similarly the importer 'knows' which service type to specify if a car rental service is required, and further, the application programmer of such an importing application knows how to interact with the remote server after a service identifier is retrieved by the import call.

In reality however, an open distributed computing system consists of a multitude of individual and unrelated service offerings from different companies and organisations which all provide distinct services with individual goals of gaining profit from their respective usage by external client applications. The establishment of markets for software modules and services via "CompuServe" may be considered as an example for the pace of module and service proliferation between human clients and "servers". Generally speaking, from the viewpoint of *competitors* in such market scenarios, it is usually rewarding to offer innovative services *before* others have already accessed the market with similar service offerings ("being the first pays most").

Considering the concepts of ODP trading mechanism in the light of a realistic open market scenario, trader utilisation can be even counter-productive for an *innovative* and generally yet *unknown* service provider, for example, if the new service offer would require establishment (and registration at the trader function) of its new, dedicated service type and, therefore an increased time-to-market.

In cases where the visibility of such a new service is not restricted to a single trader but, e.g., to a trader *federation* (as -optionally - envisaged by ODP [2]] for geographic scopes), the delay and costs of establishing of a new service type become even less acceptable and more counter-productive for the service providers and thus the open market itself.

In summary, in an ODP trader environment as defined today, obstacles to innovative service establishment can be characterised by the additional overhead of service registration and establishment which includes the following phases:

- service type *standardisation* (by global agreement),
- service type *registration* at a trader's type manager,
- *availability* of registered services to potential importers, and
- development of client *applications* to achieve the ability to cooperate with remote servers.

Apart from such delays - and additional costs - of a new server's (de-facto) availability in open network environments, the prescription of service functionality - which is a precondition for service type standardisation - may, for example, facilitate follow-up competitors to imitate the innovators service. Taking these phenomena into account, the ODP trading mechanism as introduced in section 2.1 has some limitations when applied to open service environments which have to satisfy, for example, the requirements of realistic, profit-oriented competitors planning, e.g., to introduce innovative services easily, efficiently, with great flexibility, and without compromising their respective competitive advantages.

Nevertheless, after a time of "maturation" (i.e. after several other market participants have provided comparable services) a general *standardisation* of new service types may still be desirable and achievable. Therefore, an *additional* generalisation from the concept of *trading* to a broader concept of *service mediation* seems appropriate and is therefore proposed in this paper. It *includes* ODP trading as a distinct mechanism for linking importers to exporters of well-known services. In addition to that, however, service mediation supports flexible and efficient access to (yet generally unknown, i.e. not yet standardised) services in a "pre-tradable" stage of their development. Common descriptional - and most important - basis for both service trading and mediation in open systems environments is a powerful "*Service Interface Description Language*" which characterises all necessary aspects of arbitrary heterogeneous remote services in a uniform way. In this way, appropriate SIDL concepts also help to protect open distributed client applications from as many of the potential server differences (and, in result, access complexities) as possible.

2.3. Requirements for dynamic access to remote services

A predominant goal of a system infrastructure to support a "Common Open Service Market" (COSM) – in the sense explained above – is to flexibly support innovative services in order to make them available to requesting clients and to support client/server interaction at lowest-possible effort. In this sense, any change of the COSM configuration is considered a *transition* effort with related specific, additional *transition costs* [6]; for example :

- Making an innovative service available on the market requires cost of administration and adapter stub development.
- The utilisation of a new service, or switching between the utilisation of very similar services may involve costs of adaptation and configuration. Usually a client application developer has to re-write adaptation code to be able to connect to different services since these may differ in both syntactical and semantical aspects of their programmatic interface.
- Acting as a value adding service to pre-existing ones causes transition costs: For example, if there is a demand for a graphics image server in format X, but a suitable image server only supplies format Y, it may be profitable to provide a value-adding service by

converting Y to X. Establishing this service would first cause adaptation costs at the image server and, secondly, adaptation cost for the service itself to be available to potential clients.

- Finally, the set of service attributes and, generally, the interface description shall be *extensible* by individual services without involving a respective adaptation of remote components. E.g., a group of services may extend their interface descriptions by supplying an additional finite state machine specification of allowed state transitions. If - ideally - there were no adaptation cost for server-unspecific clients in this scenario, such an extension would cause no transition costs.

If such transition costs are significantly higher than the amount that an innovative service provider charges, the overall service utilisation cost for potential clients becomes prohibitively high due to a too inflexible system infrastructure design. In contrast, an appropriate open distributed system infrastructure also makes newly created and innovative services fast and easily accessible for many clients at negligible adaptation costs, thus creating enough incentive for new, profit-driven suppliers of such services to the benefits of potential distributed client applications.

Accordingly, the requirements of an open systems infrastructure that supports service trading *and* mediation of innovative services is based on the following key aspects which are discussed in detail in the next section:

- A *uniform* service description technique and extensive use of extensible representations of service interface descriptions, and
- Provision of specific software components of, e.g., a "*Generic Client* " for dynamic service access which enables and supports distributed application users to access innovative services in a *common* way - i.e. independent of specific knowledge about details of the remote services.

3. Service *mediation* in open systems

In a broader sense than the notion of trading, service *mediation* is shall denote in this context the general task of dynamic cooperation support for clients and servers in open systems. In particular, service mediation does not depend on a pre-defined service type but rather on human user service browsing and selection. Accordingly, this principle will be referred to as *browser mediation.*

3.1. Service interface description technique

Traditionally, service descriptions are used as an input for stub code generation as known from several existing open RPC implementations. In the COSM environment, however, interface descriptions are regarded as objects which can be *communicated* between distributed application components. This idea allows not only a *dynamic marshalling* of transferred parameters, it also provides a prerequisite for a *generic client* component (as presented below) which hides as many differences of remote services to client applications as possible and, thus, reduces their respective effort for adaptation to innovative services substantially.

In summary, a "*Service Interface Description*" (SID) [7] can be considered a container for various descriptional elements for services in open systems, including not only the respective operation signatures but also additional information on, e.g.:

- restrictions to legal invocation sequences of service operations, based on a finite state machine (FSM) model,
- how to generate an individual user interface at the client site,
- user-understandable annotations to the SID elements, or at a later point of time,
- service type information that is required for traders.

For the above mentioned simple "car-rental" example, the FSM model may, e.g., comprise two different communication *states*, INIT and SELECTED, and two transition *operations*, SelectCar(...) and Commit(...). In addition, the service may, e.g., restrict the *allowed* transitions to the following list (consisting of tuples of: 'current□state', 'allowed transition', and 'resulting state') of:

(INIT, SelectCar, SELECTED),
(SELECTED, SelectCar, SELECTED)
and (SELECTED, Commit, INIT)

So, the FSM specification is a first example for necessity of optional extensions to a signature-based ('object-oriented'?!) SID. Consequently, existing SIDL techniques have to be augmented by, at least, specification mechanisms for FSM service protocol restrictions (see also [7]). Additional SID extensions - and corresponding SIDL extensions - are briefly explained below.

Extended service interface descriptions

In an extended SIDL, a SID should be considered a communicable *first class* object. In an example, a SID can, accordingly, be a data value of, e.g., the assumed type SIDBASE as defined below: Its type definition denotes the descriptional elements embedded into the SID; additional elements of an extended SID represent extensions to the corresponding type which is then considered a *subtype* of the base type SIDBASE:

```
Let SIDBase   = record     typespec : TypeSpec_t
                           opspec : OpSpec_t  end;
Let SIDSub    = record     typespec : TypeSpec_t
                           opspec : OpSpec_t
                           fsmspec : FSMSpec_t end;
let browse    = fun ...
```

As an example, the base type for a service description may be viewed as a record type, as known, e.g., from *polymorphic programming languages* like Quest [8]] or TL (Tycoon Language) [9]. Such record types comprise distinct elements like a list of type definitions, a list of operation signatures, etc.. So, in the example above, a specific (sub-) type SIDSUB conforms to the base type SIDBASE in a *type-safe* way if it contains *at least* the elements of SIDBASE, but possibly additional ones as well. Instances of the extended subtype SIDSUB still

conform to operations that expect instances of the base type. As far as service descriptions are concerned, the type extension may comprise an additional element "FSMDefinition" which contains a finite state machine specification of allowed server state transitions (see Fig. 2). If a service extends its SID type as shown above and transfers corresponding SID instances to other components within the communication environment, these remain capable of interpreting the SID since it remains conform to the base type. Only those components, however, which are capable of processing the more specific SID subtype can be aware of the respective SID extensions.

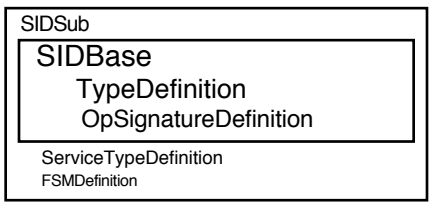

Figure 2: Extending a base interface description by additional elements

So, the introduction of *subtype polymorphism* into a SIDL and its application to concrete service specifications enables individual services (or parts of the service community) to communicate and autonomously extend a given base SID to a more specific one. Therefore, extended SIDL techniques provide a descriptional basis for generalising 'tradable' services from (only) well-know, predefined to (also) innovative, yet unknown ones - an important prerequisite for flexible open service 'mediation' in realistic large-scale distributed systems.

3.2. Dynamic open service access

Dynamic support for innovative services requires distinguished access and interaction functions in the COSM system support infrastructure. Therefore, the two following aspects of client genericity and dynamic client/server bindings require particular attention:

Uniform remote service access via generic clients

In order to make as much of the heterogeneity of the multitude and variety of accessible remote services in open systems transparent to potential clients, a "*Generic Client* " function can now be defined, based on the above mentioned (SIDL) techniques for uniform extended service interface descriptions. Generic clients allow human users to access arbitrary innovative services but still to remain *independent* of any given service as far as interface signatures, interaction modes or human user interfaces to these services are concerned. So, generic clients allow applications to generate service-specific components (as, e.g., user-interfaces, parameter marshalling stubs etc.) *automatically* based on their uniform respective SID. Therefore, an important property of generic clients in a COSM is a well-defined relationship of linguistic service description elements to corresponding (graphical) user interface (Gui) management system (UIMS) components at the client site (Fig. 3).

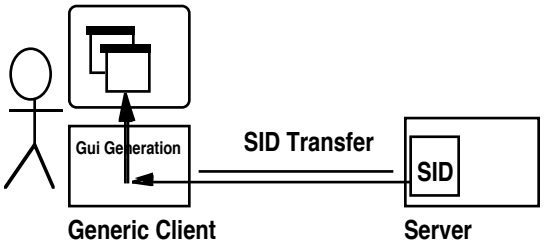

Figure 3: Dynamic Binding to Innovative Services in Open Systems

Components of SIDL service descriptions like type definitions, operation signatures and textual annotations result in respective UIMS components. Therefore, operation-specific value editor forms can be generated automatically that allow to present or enter data values (for an example see also Fig. 7). Other controller elements (e.g. buttons, list items), that can be activated by mouse events are related to respective remote operation invocations as defined in SIDL UIMS descriptions.

Binding support through generic clients and browsers

Since there is no predefined service type for innovative service classifications, application services register their SID together with their globally identifying service reference (and, thus, make it available to clients) at a well-known generic "*Browser*" component. The service reference belongs to a SIDL base type, SERVICEREFERENCE. In turn, the browser may also act as an application service as well and register its own SID at yet another browser etc..

When involving a generic client, the supply of a service reference is required to identify the server to be bound to. As a SIDL base type, values of SERVICEREFERENCE are first class objects which can be transferred forth and back as parameters or return values. Accordingly, the generic client component provides a distinct UIMS controller representation to enable human users effecting binding establishment via user interface interactions. In fact, the actual binding establishment is made transparent by a seamless transition between UIMS dialogue interactions from the users point of view. This basic COSM mechanism enables a powerful principle of *service mediation* (Fig. 4):

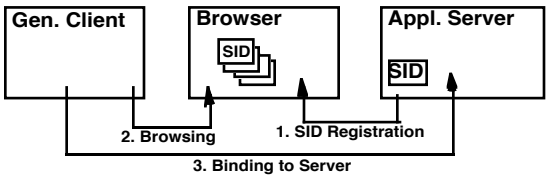

Figure 4: Bindings between a Generic Client, a Browser, and an Application Server

As shown in Figure 4, binding via browsers is effected by obtaining a distinct service reference from the currently selected entry. In turn, a further binding can be effected out of the user interface based on this service reference. Each binding corresponds to an individually generated user interface. Accordingly, a cascade of bindings and

corresponding user-interfaces can evolve from several consecutive binding establishments.

3.3. Benefits and limitations of service mediation

Involving a generic client component based on uniform and powerful (extended) service descriptions leads to a flexible client/server support infrastructure. Here, the uniform SID enables *any* type of service to "remote control" generic clients automatically. The only components to be (application) specifically developed for that are the server components themselves, and their respective uniform (SIDL) service descriptions. Therefore, transition costs for individual participants can be reduced substantially since there is *no* adaptation effort required for generic clients; and service mediation is facilitated by automatic binding support via service references. The only programming effort that is left for the distributed application programmer is to adapt their respective services to the COSM system support infrastructure.

In summary, depending on the stage of service standardisation either service *mediation* supported by generic clients or service *trading* may be the more suitable approach: In a pre-standardised stage - due to the generic client approach - only *browser mediation* is possible at all, which represents a flexibility advantage per se - yet with restrictions of the descriptive power of the SID and the components that can be generated on this basis. On the other hand, a *trader* may be more appropriate for linking client and server components together which already have enough knowledge about each other and, therefore, agree on a common service type. Here, the compatibility among services of the same type allows to select a distinct service based on well-known quality attributes and selection policies. However, the larger the visibility of a service in geographic terms is, the more trading service suppliers and, therefore, standardisation authorities, are involved in defining a particular service type. In those cases, traditional service trading in the sense of the current ISO ODP standardisation seem hardly able to bridge the gap between fast (innovative) service availability and - eventual - standardised accessibility.

Consequently, the following chapter presents an *integrated* distributed system support architecture and prototype implementation that *combines* the benefits of *both* approaches.

4. COSM system architecture and prototype implementation

As stated above, the implementation of an integrated system support for both service trading and mediation in open systems requires, first of all, extended *SID concepts* for highly expressive, but still uniform service type descriptions. Secondly, its implementation requires specific system *software components* (as, for example, generic client, browsers, graphical user interface generator components etc. as realised in the current prototype system) which are based on SIDL technology and provide the abstract and uniform client application interface for service selection both in direct (human) interaction with arbitrary remote service providers (i.e. by service *mediation*) as well as - indirectly - with an ODP *trader* function, whenever possible.

4.1. Integrating innovative and tradable services

The most interesting challenge for the COSM approach to system support for open distributed applications lies in an elegant and cost efficient transition support for innovative services to become - eventually - *tradable*, i.e. accessible and usable via standardised and publicly known instances of corresponding well-known service types. If, in the above example, eventually (if at all!) the suppliers of "car rental" services have agreed on a common functionality of their respective services offered, the respective service signature and semantics can be standardised in a respective service type definition. For the "car rental" service example *service property types* as presented in section 2.1 can, for example, be used as selection criteria for registered services at an ODP trader function which requires this additional service type information also to register a service offering in an appropriate way. At the same time, however, such a service shall also remain accessible for generic clients in the more general service mediation environment. Therefore, an additional SID component is implemented which contains the required service type description.

The previous chapters introduced SID and SIDL techniques and concepts in ways which seem most adequate for supporting very general distributed open system scenarios. As this approach, however, explicitly aims at *realistic* open system environments as increasingly used and - in part - provided by distributed system software vendors, the concrete prototype of the COSM infrastructure as currently implemented and presented here, tries to integrate - as much as possible - new ('de-facto' standardised) open system support platforms. In the context of service trading and mediation in open system, especially the "*Common Object Request Broker*" (CORBA) "Interface Description Language" (IDL) as recently proposed by the multi-vendor "Object Management Group" (OMG) [10] seems to have a potential for playing an important role for future open distributed application implementations.

Accordingly, the *concrete syntax* selected for service interface descriptions in the current prototype system conforms to (and extends) the OMG CORBA IDL. In this environment, the SID elements as reviewed and extended above are embedded as distinguished CORBA IDL *modules* with a COSM/SIDL-specific structure. The example below shows how the embedding of a particular service signature description can be integrated in a surrounding framework based on CORBA IDL terms. The great benefits of formulating service interface descriptions in terms of IDL embeddings lies in the fact that - despite of the presented COSM-specific SID extensions - the original IDL syntax may remain unchanged. In its framework, optional descriptional elements, like, e.g., the FSM specification, can rather be embedded into the given CORBA IDL module structure. In order to incorporate, e.g., the subtype polymorphism concepts as presented in section 3.1, IDL interpreters can be extended to recognise only known module names and skip those that do not bear any

meaning to them. Thus, COSM SIDs remain processable also by (de-facto) standard CORBA compliant components, like parsers or interface repositories, independently of the number an kind of additional extensions.

The following example shows the structure of a base SID type that is extended by service attribute descriptions for traders in open systems:

```
module CarRentalService {
    // the base part:
    typedef CarModel_t enum { AUDI, FIAT-Uno, VW-Golf };
    typedef SelectCar_t struct {
                                enum CarModel;
                                string BookingDate;
                                ... };
    interface COSM_Operations {
        SelectCarReturn_t SelectCar ( [in] SelectCar_t
                                                    selection );
        BookCarReturn_t BookCar ( );
        };
    // the extension:
    module COSM_TraderExport {
        const    ID          ServiceID = 4711;
        const    String      TOD = "CarRentalService";
        const    CarModel_t Model = FIAT-Uno;
        const    float       ChargePerDay = 80;
        const    ChargeCurrency_t  ChargeCurrency = USD;
        };
    ... };
```

Here, the COSM architecture provides a SID extension COSM_TRADEREXPORT to describe the characteristics of specific tradable services. The standardisation of a particular "car rental" service type may therefore require a set of attributes as, e.g., shown in the example above.

4.2. Prototype implementation

Finally, an overview of the current status of a prototype architecture is presented as well as the implementation for a distributed open systems COSM support infrastructure providing both SIDL-based service trading and service mediation as well as automatic local (graphical) user interface generation based on common SIDs and corresponding generic client and browsing system software components.

Overall prototype architecture

The overall prototype architecture (see Fig. 6) can, conceptually, be subdivided into, first, a "*User Level*" which contains all application-specific - including interactive - end user interface components for which distributed software development, based on (potential) involvement of all accessible services in open network environments, shall be supported as much as possible. Below that level, the "*Client/server Level*" comprises - on the one hand side - all (still application-specific) client application, application server, as well as interactive client applications. On the other hand, it contains the server independent generic client resp. service browsing components required for flexible service mediation in open systems as described above. Finally, the rest of the prototype architecture components are those which altogether comprise the functions of the "*COSM Support Interface*": For the "*Controlling Level*", the ODP trader function is the most important one addressed so far. Also additional important system support functions for, e.g., distributed "Transaction" or "Activity Management" would belong to this layer but are - currently - outside the scope of the ongoing prototype implementation. The "*Service Support Level*" comprises "Type" and "Interface Management", a "Name Server" as well as "Group Management" and the client/server "Binder" Function. Finally, the "Communication Level" contains the basic open communication subsystem. functions as, most importantly, provided by - standardised - "RPC" and extended multicast and broadcast functions.

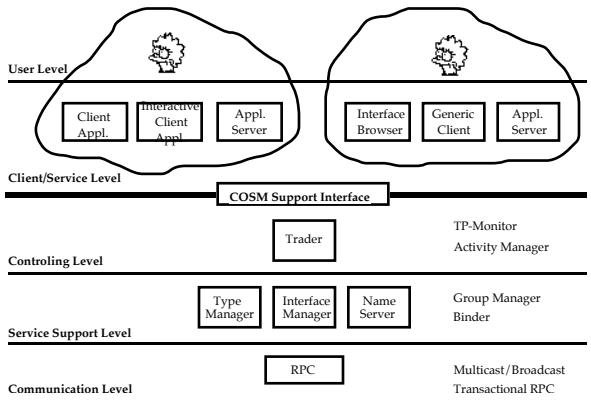

Figure 6: Overall architecture of the prototype COSM support system infrastructure

Prototype implementation

The current version of the prototype was developed on a heterogeneous workstation cluster, consisting of both Sun SPARC□stations as well as IBM RS/6000 AIX workstations. Currently, the Sun RPC interface serves as a common communication basis. Additional standardised 'middleware' services - as provided on various hardware platforms by, e.g., the OSF Distributed Computing Environment (DCE) - are currently evaluated and will be used as an extended distributed operating and communication system basis.

Currently, one part of the COSM support system prototype implementation concentrates on completing the implementation of the basic functions of an ODP *trader* function - closely integrated into already existing flexible COSM service *mediation* functions as presented above. At the end user interface, a first version of the prototype implementation required client service invocations to explicitly supply RPC calls with actual parameter values. In order to support access to such innovative services more appropriately, the current prototype's generic user interface *automatically* generates a typed form for local parameter entry and analysis (See Fig. 7) [7]. For such a SIDL-based user interface generation, all necessary type descriptions can be automatically retrieved from the servers SID. Accordingly, return values can be presented in the same way by the user interface. Service invocations which do not conform to the current communication state (as specified by the FSM), can also be automatically

intercepted by the generic client and, therefore, already be rejected locally.

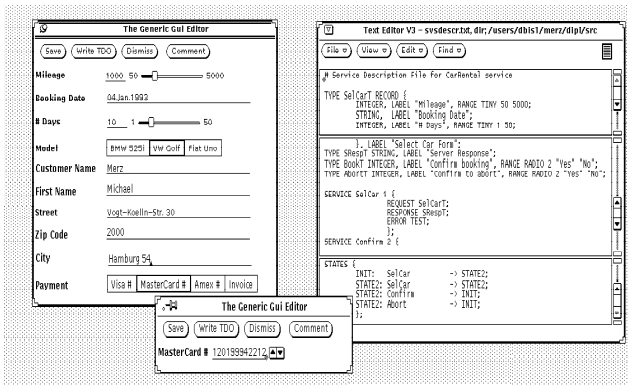

Figure 7: Service description and the resulting user interface at the generic client site

By involving such a *generic* style of automatic user interface generation for remote services, type *conformance* between co-operating client and server interfaces is always given implicitly. Therefore, the prototype also supports the integration of user interface and service description aspects. So, developing new server applications just requires to implement service operations and to describe the respective procedures by means of the extended service interface description language: the formal parts as type, procedure, state, and export description, and, optionally, the informal part of the user interface description as natural language annotations.

5. Concluding remarks

The COSM support infrastructure aims at improved system support for flexible client/server cooperation in modern distributed and heterogeneous open systems. Specifically, it addresses problems of matching distributed application program client requests with arbitrary generic remote server interface functions as provided at dedicated server nodes anywhere in an open network environment. The goal here is not just to support specific client/server cooperations but rather to design an architecture for service management in large-scale open systems. The two - principally conflicting - goals addressed in balance are - on the one hand side - to keep a maximum of local server *autonomy* while - on the other hand side - to *reduce service transition costs* as much as possible and thus make flexible decisions to "make or buy" specific components of distributed applications feasible and efficiently implementable at all.

The extendibility of the SIDL language as outlined in this paper leads to scalable and - most important - *uniform* formal specifications of *any* open server's functionality. In result, an open distributed application support system implementation based on a COSM infrastructure and SIDL service descriptions supports client/server cooperation in ODP by reducing both the *complexity* of accessing heterogeneous services in open systems, as well as the *implementation effort* required for developing open distributed applications, substantially.

In addition, as demonstrated in the COSM prototype, SIDL specifications could also be used for automatic creation of local human user (e.g. window graphics) or computer program *interfaces* to all respective remote services in open systems and, thus, for supporting automatic user interface generation in environments of great multitudes and varieties of potential services accessed.

The specific emphasis of this paper and an important goal of the COSM design and prototype implementation is to *integrate* both well defined and already established techniques for service *trading* in open systems (as, e.g., proposed by ODP) with new mechanisms for flexible and efficient client server *mediation* which are also applicable for realistic, large-scale open network scenarios where service providers and user may hardly have any knowledge about each other. Decisive for the success for such an integration seems - first of all - the definition of an adequate SIDL as a common basis for, at least, service characterisation and request trading in open systems, and - secondly - the realisation of an integrated architecture for both service trading and mediation in a common distributed system infrastructure. The ongoing work as presented in this papers aims at such an integration both at a conceptual and at a systems implementation level.

6. References

[1] L. Svobodova: "Client/Server Model of Distributed Processing", Proc. GI/ITG-Conf. 'Kommunikation in verteilten Systemen', Informatik-Fachberichte, Springer-Verlag, Heidelberg, 1993, pp.485-498

[2] ISO/IEC JTC1 SC21 WG7: Trader, WD N7047, ´92

[3] ISO/IEC JTC1 SC21 WG7: Basic Reference Model of Open Distributed Processing, Working Document N7053, 1992

[4] M. Y. Bearman: ODP - Trader, in [11], pp 37-51

[5] J. Indulska, M. Bearman, K. Raymond: A Type Management System for an ODP Trader, in [11], pp.□169–180

[6] M. Merz, W. Lamersdorf: Cooperation Support for an Open Service Market,, in [11], pp 329-340

[7] M. Merz, W. Lamersdorf: Generic Interfaces to Remote Applications in Open Systems, in: Proc. Intern. IFIP Workshop on Interfaces in Industrial Production and Engineering Systems, North-Holland, 1993, pp.267-281

[8] L. Cardelli: Typeful Programming, DEC SRC Research Report #45, Palo Alto, CA, 1989

[9] J.W: Schmidt, F. Matthes: Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems, Proc. Int. IEEE/RIDE Workshop on Inter-operability, IEEE Computer Soc. Press, Los Alamitos, 1993

[10] The Common Object Request Broker: Architecture and Specification, OMG Document No. 91.12.1, 1991

[11] J. de Meer/ B. Mahr/ S. Storp (Eds.): Proc. International Conference on Open Distributed Processing (ICODP '93) Elsevier Science Publishers B.V. (North-Holland), Amsterdam London New York Tokyo, 1993