18

Trader Supported Distributed Office Applications

Ch. Mittasch[a], W. König[b] and R. Funke[b] chris@ibc.inf.tu-dresden.de
[a]Institut für Informatik, TU Bergakademie Freiberg
D-09596 Freiberg, Germany
[b]Fakultät Informatik,IBDR, TU Dresden D-01062 Dresden, Germany

Abstract

Comfortable and effective working applications are required in large open distributed environments. Therefore traders are becoming more and more interesting - as they deliver a more precise overview of available services as well as allowing the integration of QoS-attributes of service offers. Distributed office applications - as described here - promise to use the potentials of modern heterogeneous network systems for a more flexible and reliable control of office workflows.

Thus, an appropriate object-oriented trader (X*-Trader) is presented. It uses the DC++ extension of OSF/DCE. Its architecture, interfaces and functionality are discussed. Second, our system for design and decentralized control of distributed office applications (CodAlf) is introduced (also a DC++ based system). Furthermore an extension for the management of standardized application server interfaces (a type management system) is introduced. Finally, the integration of the X*-Trader in such a representative distributed system is shown.

Keywords

Computer Communication Networks; Distributed Systems; Distributed Applications, Trading, Workflow Management Systems

1 INTRODUCTION

Distributed applications are of increasing importance in network environments. Dif-ferent approaches for the collaborative use of connected resources are being developed. On the one hand different distributed platforms are available, while on the other hand a number of workflow management systems appeared - up to now mainly as database extensions.

The necessity and the functionality of trading have been investigated in that context (Mittasch,94; Mittasch,94b). Based on that, we designed and implemented a dedicated trader-instance (X*-trader) in our workgroup. In this paper we present essential characteristics of this trader and its application in our workflow system for distributed office applications.

A. Schill et al. (eds.), *Distributed Platforms*

Demonstrating this application of trading may help to establish trading as a standardized middleware service, too.
Currently known concepts of traders and of request brokers are going to fill up the gap between current naming services and the actual demands upon such distributed environments. In particular, attribute-based naming as well as explicit management of service types impose new requirements. Thus, the general tasks of a trader are:

- definition of interfaces for service requesting clients (importers) at various levels of abstraction (with regard to logical expressions of service attributes, terms with expressions of lowest allowed QoS-rates (Quality of Service), random choice,.. ;
- selection between various equally matching servers;
- search for appropriate servers, also if no suitable service offer exists in the home - trader domain (via trader interworking);
- possibility to restrict the search by so called constraints;
- establishment of new kinds of service offers (despiction of a broad variety of services and (maybe) to force competition (Mittasch,94);
- service type browser as a structured information base for application development.

An object based approach to a trading realization is presented in chapter two the X*-Trader. X* works on top of DC++ (Schill,95) and DCE. For the design of our trader we took into consideration the current developments in trading standardization (ODP- Trading - Open Distributed Processing (ISO,94)) and the framework for object oriented distributed environments based on the Common Object Request Broker Architecture (CORBA-OMG,91) and their extensions: Common Object Services Specifications (COSS1, COSS2,..), eg. for Naming, Object's Live Cycle, Persistence, Transactions and Security. CORBA based system implementations, as eg. DOE (Distributed Objects Everywhere, Sunsoft,93) were also taken into account. Further trader implementations were considered.
Three main ideas influenced the concept of this trader.
(1)to offer a structured database of service types (potentially available in the trader domain and in other trader domains, too);
(2)to manage the currently existing entries of server instances in a decentralized way. (That means to realise a two-level-trading).
(3)to support distributed application development by the knowledge based analysis of earlier service allocations.
The third chapter introduces a system supporting development and use of distributed, object oriented office applications (CodAlf) also based on DC++. Moreover the advantages of using that trader for the workflow system are outlined. For example, tasks of office procedures can be mapped dynamically to currently available employees and server processes while explicitly considering dynamic attributes as availability of employees, costs of the transmision quality.

2 THE X-*TRADER - A DC++ BASED TRADER

2.1 DCE and Trading

The Distributed Computing Environment (DCE) by the Open Software Foundation (OSF) realizes a platform for development and use of decentralized and heterogeneous applications. It covers a set of system components, tools and runtime libraries. Mainly used components are

shown - in combinition with DC++ - in figure 1, for more details see for instance (Lockhart,94).

The DC++-implementation (Distributed C++ - Schill,93; Heuser,95) is an exten-sion of DCE towards object-oriented and distributed applications. It uses DCE (RPC, Threads, CDS) inside and offers the use of C++ in such an environment. Program-mers work with substitute classes instead of application classes. So, objects achieve location independence. The DC++ generator creates automatically their source code using a declaration file. Every substitute class owns additional elements and methods concerning to it's associated application class.

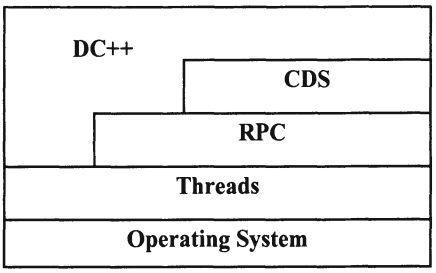

Figure 1 Systemplatform: DCE and DC++

Every process owns hash tables for the administration of process locations and the addresses of local and remote objects of substitute classes. The system class 'Object_Reference' implements forwarding chains, for locating mobile objects together with these dynamic vectors. Furthermore, the function static *A* get_ref_by_name (char*)* realises referencing to a remote object of class A by it's name. This function is generated automatically for every class by DC++. The CDS-database is able to manage DC++-components (processes and objects), too.

Today different DCE based traders and related trading approaches exist (examples are described in Mittasch,94; Müller,95; Kuttvonen,93). A common feature is the ability to map service requests to servers based on quality attributes. The TRADE-trader (Müller,95) uses the XDS/XOM interface of CDS for the management of additional attributes. It was extended for handling new and changing service types. So it better meets the needs of exporters (service providers) in a service market scenario. The DRYAD trading system (Kuttvonen,93) contains a lot of features allowing very flexible use, eg. the use of the trader is possible at different levels of abstraction. The ExTra-trader (Mittasch,94b) works on top of CDS and RPC. It manages all attributes at it's own, including dynamic attribute acquisition.

2.2 The Concept of the X*-Trader

The requirements of distributed and object oriented programs caused new features in comparison to the above mentioned DCE-traders. They led to the concept of the X*-Trader, an object oriented trading service using DC++. So, the main differences to procedure-oriented traders and the basic mechanisms of using the DC++ middleware services were investigated.
The components of a trading community are basically trader(s), servers and clients. They do not correspond to processes of operating systems, instead they are units (objects) in such

processes. With regard to the object oriented programming paradigm only objects interact with each other. Therefore the required granularity is of objects and not of processes. The need for such a fine granularity is determined by the need for applications with large amounts of data or a high frequency of repetition, as for instance in workflow management systems.

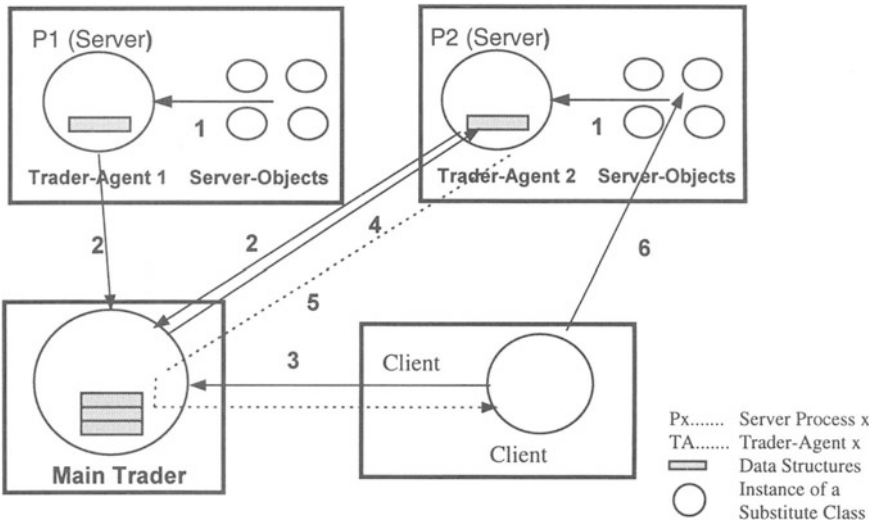

Figure 2 Typical course of events in the trading community using the X*-trader

1. The instance of a trader-agent registers an exported service offer by the function *register_Service()* at it's node.
2. The local-trader tests, if the service type of this offer is already available by another object of that exporting process - if not, this service type is exported to the main-trader by *export_ServiceType()*.
3. A client object sends a service request with *import_Service()* or with *lookup_Service ()* to the main-trader.
4. The main-trader forwards the request to matching trader-agents (amount and sequence of requested trader agents depend also on the client's inquiry expression or constraints).
5. The local-trader(s) selects a matching server object depending on the inquiry expression and responds back to the main-trader. This enables the main-trader to select the best matching server object (if necessary). In the lookup-case all matching service-types are collected and delivered as a list to the client object.
6. The client receives an object name and gets the reference with the method *get_ref_by_name* (of DC++). Now it is able to establish its application.

The definition of a class describes a service type in DC++. A service is specified by methods and attributes. The member functions of the defined datatype (class) are interfaces, the instance variables are properties of the service. By its nature, the represented object oriented office application system does not require strongly limited processing times. Thus, trading is

realizable in a two step approach - a main trader with the service type management and local traders on every participating process. So the run time system allows parallel processing of client requests. The trading service is realized by objects of two different classes in X*:

- The central component is the so called 'main-trader', with knowledge about the existing service types in every participating program. The main-trader exists once only in every trading community (cell).
- Decentralised components, the so called 'trader-agents', exist in every process containing server objects. Such agents manage service offers of their processes, including static and dynamic properties and carry out the local server selection.

Client requests are sent to the main-trader. Afterwards the central trader asks the trader agents with matching service types. That decreases the number of required RPC's for the acquisition of dynamic attributes. The trader agents deliver the status of „their" server objects, that means only one connection of the main-trader to every participating process is essential.

Based on that architecture, the export of a service offer is realised by a two-stage protocol. First the exporter registers its service offers at the local trader-agent. 'ServiceType' represents the service type specified by a class definition. 'ExporterName' is the name of the service exporting instance. The third argument organises the service property entries, the dynamic attributes contain defaults. After that, the trader-agent tests the new service type. It sends a message to the maintrader only if this service type is offered the first time.

void register_Service (ServiceType, ExporterName, PropertyList)

void export_ServiceType (ServiceType, LocalTraderName)

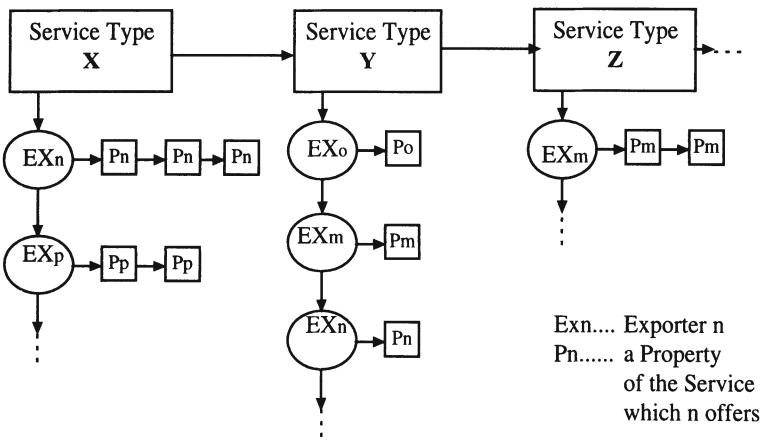

Figure 3 Data structure of a local trader-agent

The import protocol works in a similar way, eg. also as a two-stage protocol. First, a client sends an import request to the main-trader ('ExporterName' represents an output parameter here). This string allows the client to detect the desired remote server instance with the help of the automatically generated method of the DC++ substitute class 'get_ref_by_name'. The client can use the whole functionality of a trader by specifying further characteristics:

- 'strategy' can restrict the server selection by choosing first-choice strategy. Random and best choice cause a search in the whole trader community.
- 'inquiry' allows one to express details of characteristics of the required service. Additionally in the next version of our prototype implementation the inquiry expression will allow the specification of fuzzy requests.
- 'constraints' optionally restrict the search domain.

The main-trader engages all trader-agents with matching service types to select the required service offer using threads. The result is exactly one server (specified in ExporterName) and it's current service properties, or 'null'. Finally, the main-trader evaluates the messages of the trader agents and selects one server depending on the selection strategy. There exist some cases that allow to find the required server in a more efficient manner (such as first choice strategy).

returncode Import_Service (ServiceType, ExporterName, Inquiry, Strategy, Constraint)

returncode deliver_Exporter (ServiceType, ExporterName, Inquiry, Strategy, PropList)

Now the local trader-agents are called and acquire the current values of the dynamic properties of exporter instances with compatible service type. Then, an incorporated parser tests the syntax of the inquiry-expression. At the same time the expression is analysed step by step. In the course of this the characteristics of all suitable servers are tested and remaining candidates are written to a list. Finally, the trader agent selects one entry and returns objectname and property-list to the main-trader.

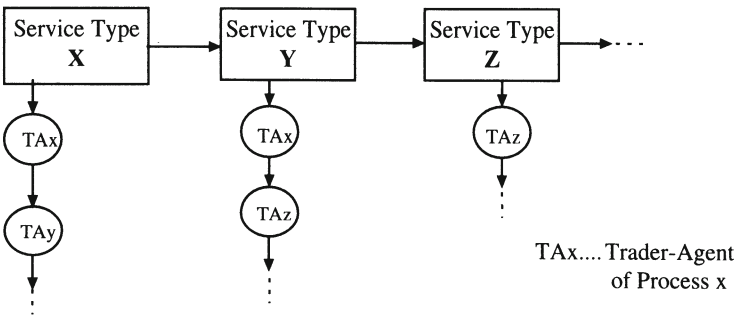

Figure 4 Data structure of the main-trader

A simple subtyping approach was designed by using inheritance relations of classes. We integrate this extension with the aim to establish a more flexible handling of service types and to support a more efficient search for matching servers in the application's design phase.
The inclusion of service offers in other trader domains is under development, corresponding to the Interworking Trader Project. That represents the possible support or the integration of employees, the use of software, processor times, etc. of other departments of an enterprise. At the moment the X*-Trader is capable of interacting with other domains containing an instance of the X*-Trader.

3 A SYSTEM SUPPORTING DEVELOPMENT AND USE OF DISTRIBUTED OFFICE APPLICATIONS

3.1 The system CodAlf: Enabling distributed office applications

Complex applications require a structured description in event lists, graphs, etc. representing partial tasks and parallelism. Naturally their application field is not restricted to the office automation environment. Such descriptions are a valuable basis for the transformation of often repeated, office applications to distributed systems, for instance based on DCE. Our own Petri-Net-based description language has been developed specifically for the mapping of office procedure execution onto DC++ and DCE. Their translation to other platforms is under consideration, too.

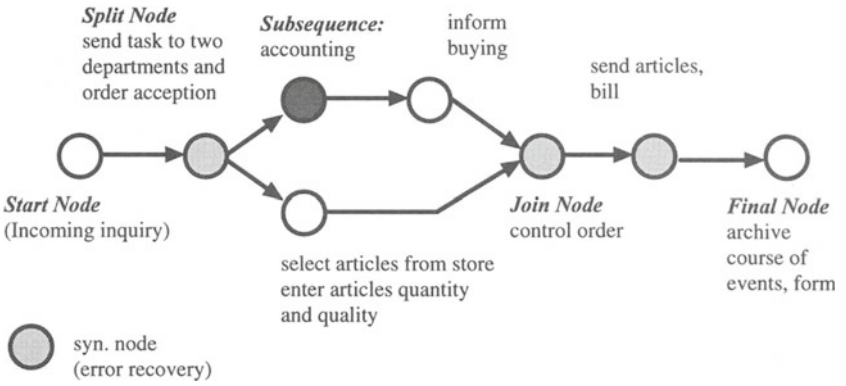

Figure 5 An example of a processing sequence of a distributed application

The so called procedure description or processing sequence of the application contains the graph of the distributed procedure (see fig. 5) including a specification of various quality requirements at nodes and edges (transmission paths). The following structure elements describe the variety of execution components of office applications:

- sequences;
- parallel edges;
- alternative edges;
- loops;
- subsequences.

The procedure description consists of two parts: First, there exist a procedure description with general attributes. Second follow the description of all task nodes. Furthermore, different kinds of service allocation are possible: Static service binding, dynamic service binding via a trader at the generation time of the distributed application or stepwise dynamic binding depending on the progress in the processing of the application. The definition of an individual node contains the following information:

- the specification of the service with requested quality parameters

- the kind of server allocation
- the description of the following edges (communication services) with requested QoS-parameters of communication links (eg. transmission rate)
- control mechanisms (eg. parallel paths or the condition-dependent splitting of the procedure with subsequent synchronisation points in both cases)
- reservation requests
- determination of the behaviour in the case of errors (eg. fault recovery mechanisms, see also fig. 5 - synchronization mode).

```
PROCEDURE order_form;
      ATTRIBUTES
            COMPLETION_TIME=5;
            PRIORITY=3;
      DATA
            calalogue_type articles;
            form_structure_type delivery_note;
            form2_structure_type bill;
      INITIATOR
            NOTIFY inquiry_server WITH STATUS procedure started;
      TERMINATION
            COPY form TO data_server;
START_NODE accounting;
      SERVICE do_delivery_note;
            PARAMETERS   articles [IN];
                         delivery_note[OUT];
                         bill [OUT]
      NODE_ATTRIBUTES
            time_out ==1;// one day
      BINDINGS
            TO_ATTRIBUTES articles.available == in_stock;
            SELECT WITH PRICES LOW;
      NODE_ACTIONS
            RESERVATION AT_NODE control_order
                  RESOURCE employee.name ="Lehmann";
                  NOTIFY INITIATOR WITH COMPLETION;
            EXCEPTION
                  CAUSE employee.available=false REACTION employee.name="Meier";
      NEXT_NODES
            LINK_TO control_order;
NODE_END

FINAL_NODE control_order
      INPUT accounting;
      SERVICE control;
            PARAMETERS   delivery_note [IO];
                         bill [IO];
                         control_form [OUT];
      NODE_ACTIONS
            EXCEPTION
                  CAUSE bill==false REACTION CANCAL;
NODE_END;
```

Figure 6 Example of a procedure description (translated into text-mode)

A simple example shows a procedure description with the two nodes „accounting“ and „control_order“ (see fig. 6). The design of distributed office procedures is supported by an editor with graphical user interface. It delivers a design editor for the Petri-Net-based description of the distributed procedure (see fig. 5), including syntactic and semantic tests. The graphical description can be translated into the textual description (and viceversa) after finishing the design.

A procedure generator translates the procedure description (graphical- or textmode) into a procedure plan - the input to the run time system. When a procedure is started, a procedure object is generated and migrated to the first application server object of the procedure. A procedure object contains its procedure plan and the actual state of the procedure. So it moves from application server to application server without the control of a centralized manager. Its data or data references and information about its creation (name, owner,...) belong to an application object, too. All components of the system for decentralized distributed applications and their interaction are shown in figure 7.

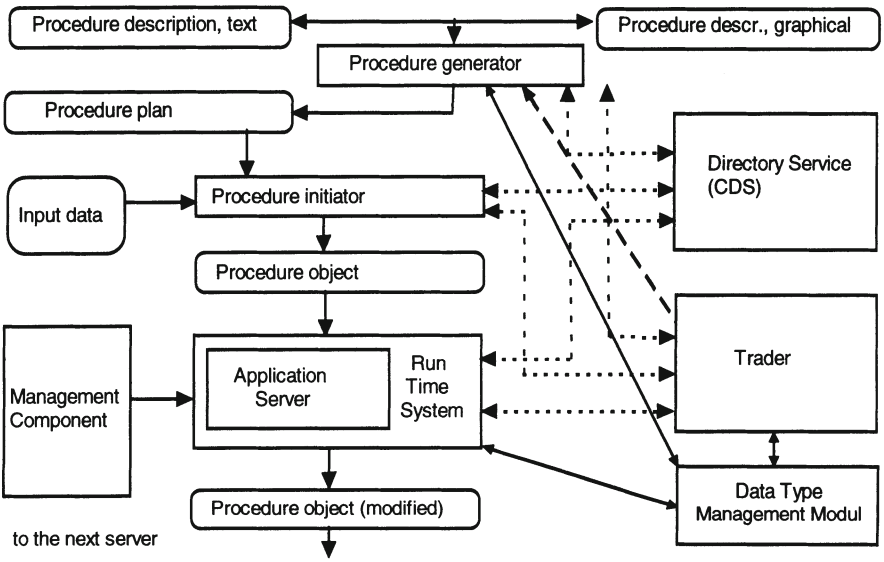

Figure 7 Architecture of the system for decentralized distributed applications

Their work is described in the faultless case here. Moreover, different error-semantics are supported by the CodAlf-system, for instance for the restart of the application by the initiator or the restart at defined synchronisation points (see fig.5).

- General application servers exist at fixed locations. They present the steps of the application (as shown in fig. 5 and 7, respectively). The application executes while the procedure object(s) moves from application server to server. The necessary application management tool manages these servers independently, i.e. also in a decentralized way. Every procedure object possesses a copy of the procedure plan. The present state of processing is represented by a state pointer of the plan. The next picture (fig. 8) illustrates the collaboration between the runtime system (managing the procedure object(s)) and the application server object at every node.

The run time system at each node contains three parts:

- *The entrance module.* This module receives the procedure object, separates the data part and passes it to the application server. The remaining part of the procedure object is passed to the so-called control module.

- *The control module.* This module prepares the continuation of the procedure with the help of the enclosed procedure plan and the knowledge of the actual state during data processing by the application server.
- *The exit module.* This module receives the data produced or manipulated by the application server, adds them to the modified procedure object and causes the continuation of the procedure in accordance with the procedure plan (migrating the procedure object to the next selected server(s). Synchronisation is realised here in the case of parallel paths that meet at this node of the procedure plan.

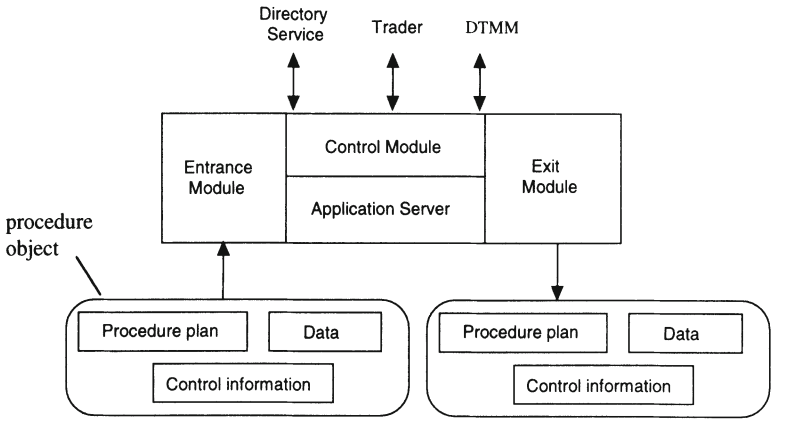

Figure 8 Run time system, procedure object and application server

The runtime system communicates with the application servers via the server interface. This instance consists of three parts: an interface for data exchange, an interface for reservation requests and an interface for management purposes.

As mentioned above, an additional management system is necessary. It consists of agents on each participating node and allows administrative intervention in working procedures. The runtime system uses the ability of the management component to start and terminate servers. All application server processes are started initially via the management component.

The embedding of the application server into the runtime components causes problems with included data. The runtime system should be applicable for all application servers without changing the source code. On the other hand any data type should be passed through the runtime modules to the application server module. We resolve this problem by using a common unique data type which can represent any other data type. It consists of a byte array with variable length, the byte counter belonging to it and an identifier (UUID) which specifies the real data type. The conversion from a special data type into its common representation and vice versa is carried out by a separate module called Data Type Management Module (DTMM).

The base for this conversion is a so-called type description. Such a description is generated by the DTMM from a type description file with an ASN.1 like syntax and is exported to the Type Repository. The realization of a Type Repository is not an objec-tive of our work. Any Directory Service can be used for storing the type descriptions.

If an application server receives such a common data type it passes it to the DTMM. This module extracts the UUID, obtains the matching type description and calls its decode routine which reconstructs the memory structure of the data. An obtained type description remains available locally until it is deleted explicitly. Therefore an application server can look for all required type descriptions only at start time. Thus, the communication with a Type Repository is no longer necessary. Figure 9 shows the structure of a Data Type Management Module.
The functionality of the DTMM can be used by almost all components of CodAlf. The procedure generator can carry out additional semantic tests. The runtime system can realize alternative paths. It can extract any component by element name from a common data type without knowledge of the real data structure. Additionally, the procedure plan can use it for several required comparisons. Furthermore, a developer of application servers for CodAlf can generate the necessary C type definitions by using a management interface to the DTMM.

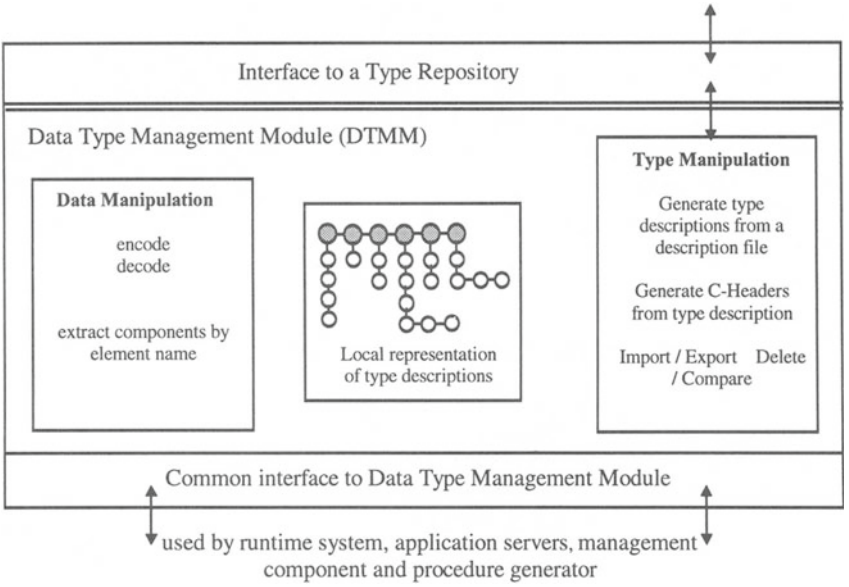

Figure 9 Data Type Management Module

3.2 The integration of the X*-Trader into CodAlf

The architecture of the CodAlf-System (see fig. 7) shows two ways in which the runtime-system and the procedure-generator are able to use: The naming service (here: CDS) or the trading service (here: X*-Trader). The application gets information about matching server objects and their location from there. Moreover the trader offers a comprehensive server selection method - it includes a variety of server attributes and also dynamically changing attribute values. So the use of a trader realises the following tasks of a workflow management system:

- Precise server selection according to user demands (based on requirements and constraints, respectively);
- server selection depending on matching attribute values and Boolean expressions of attributes;
- the consideration of system state changes (such as the availability of employees or service providers or similarly, queue lengths or net loads;
- reservations for subsequent steps can be carried out according to predefined resource requirements;
- application development support by service type and implementation repositories and by statistics of successful and unsuccessful service negotiations (feedback for the procedure generator (fig. 7) and the application developer).

We decided to integrate the X*-Trader into our office procedure environment based on these considerations. The trader functionality is being used in detail as follows:

- Static service mapping: At the beginning, an office procedure description is compiled in order to generate an executable form. The trader is used in order to check whether the service of the employee is available and whether the static quality of service requirements can be met at this stage. This is environment-dependent and time-dependend, of course.
- Dynamic service mapping: The control module of each server is responsible for dynamic server mapping. At runtime: It contacts the trader via an interface and sends the procedure object in accordance with the internal description graph. The trader performs static attribute mapping and selects potential server candidates as described above. In addition, it is able to perform an optimised selection of a specific server based on dynamic information.
- The knowledge of the trader about successful and failed service binding will be used for a better translation of the service description, optionally. This feature is not implemented yet.

4 Conclusions and Future Work

The functionality of the trader is suitable for the presented project of distributed office applications, especially after enhancing it with the described extensions. The implementation already meets most of the recommendations of the ODP-Trading standardization (ISO,94 at present Committee Draft). The integration of X* into a workflow management system has shown that both components have their own advantages other the other. So, the need for a stable middleware service „Trading“ is outlined. On the other hand the design of the CodAlf-systems was essentially influenced by the X*-trader. Runtime system as well as procedure generator use the knowledge of the trader's database

The next step in the development of the X*-Trader - federative collaborations with other DCE-based traders - is under development now. Trader federations with other platforms (CORBA-based systems) will also be possible in near future. These approaches will allow the integration of commercially offered servers in different businesses. It also will offer their combined functionality, eg. via VPN (Virtual Private Networks, Sasse,95).

Applications may require reservations: Server capacity and transmission capacity have to be requested in advance. That will be added into the runtime-system, also based on our office procedure specification. These requests will also be handled by the trader. It shall contact the selected server(s) and pass the reservation requests (scaled amounts of resources). Each trader agent has to handle local tables for managing reservations and registering the requested

reservation. The trader will be able to select alternative servers in the negative case (reservation impossible). Alternatively, it is able to offer servers with restricted quality characteristics. This feature is also valuable for the use of restricted transmission bandwidth, if only one limited transmission path is usable (similar service offers).

Studies of the global behaviour of the system will be carried out based on examples, in the office automation. Other application fields are under consideration, eg. the integration of environmental systems of different municipal and government authorities.

5 REFERENCES

Beitz,A.; Bearman,M.: An ODP Trading Service for DCE. - 1st Int. Workshop on Services in Distr. and Networked Environments, Prague, 1994, p.42-49

Gütter, D.; König, W.; Mittasch, Ch.; Schill, A.: Systemunterstützung für dezentrale verteilte Abläufe. - 39. Int. Wiss. Kolloquium Ilmenau 1994, Konf-Bd. 1, S.200-205

Heuser, L.; Schill, A.: DC++. - Bonn: Int. Thomson Publ., 1995 (TAT 15)

ISO/IEC JTC1/SC21/WG7 N963: ODP Trader. Report on the Joint Meeting of ISO/IEC JTC1/SC21/WG7 Southampton, 1994.

Kuttvonen, L.; et al.: Overview of the DRYAD Trading System. Helsinki, 1993.

Lockhart, H.W.: OSF DCE: Guide to developing distributed applications. Mc Graw Hill, Inc. 1994.

Mittasch, Ch., Irmscher, K.: On the Way to Competitive Market of Services in Heterogeneous Networks. - IFIP World Computer Congress, Hamburg 1994 - Conf. Vol. 2 (ed. Raubold) p.

Mittasch, Ch. ,Böttcher, R.: Trader in DCE - Funktionsumfang und Realisierung. - 39. Int. Wiss. Kolloquium Ilmenau 1994, Konf-Bd. 1, S.206-211

Müller-Jones; Merz; Lamersdorf: Integrating Trading into DCE.- ICODP-Conf. Brisbane: 1995, In: Conference Proceeedings, pp. 459-470

Object Management Group (OMG): The Common Object Request Broker Architecture and Spec. Revision 1.1, OMG 1991.

Sasse, O.; Mittasch, Ch.: Offener Dienstemarkt und ODP-Trading. - Workshop Anwendungsunterstützung für heterogene Rechnernetze. - Freiberg 1995 - In: Tagungsband zum Workshop (Hrsg. Mittasch, Ch.) S.33 - 40

Schill,A.; Mock, M.: DC++: Distributed Object-Oriented System Support on Top of OSF DCE. - Distr. Sys. Engg. 1 (1993) - p. 112 - 125

SunSoft: Project DOE: Future Solaris Technologies - Techn. White Paper, 1993.

Vogel, A.; Bearman, M.; Beitz, A.: Enabling Interworking of Traders. - Tech. Report, - DSTC Brisbane, 1994.

Wolisz, A.; Tschammer, V.: Performance Aspects of Trading in Open Distributed Systems. In: Computer Communications - vol. 16 (1993) p. 277 - 28