Supporting transition to open, heterogeneous computing environment

Lea Kutvonen
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, FINLAND
phone: +358 0 708 4168, fax: +358 0 708 4441
email: Lea.Kutvonen@cs.Helsinki.FI

Abstract

Open system architectures and platforms are gaining ground in distributed computing very fast. Several architecture models and even metamodels are created to ensure interoperability between systems that declare themselves to be open. However, the transition from traditional closed systems on traditional platforms to the usage of new paradigms offered by the new models is difficult. In this paper we suggest an open computing architecture that allows the coexistence of the old and new paradigms of cooperation. This architecture offers a new concept, service interface autonomy, for systems within a coalesced network. Therefore, the model is not only to be used in the transition phase but also to support the coexistence and cooperation of several different application area networking environments in the future. We also suggest service invocation scheme comparable to communication paradigms of RM-ODP and describe software supporting the function.

Computing Reviews Categories and Subject Descriptors:
C.2.1. [Computer Communication Networks]: Network Architecture and Design
C.2.4. [Computer Communication Networks]: Distributed Systems

General Terms:
Distributed systems, heterogeneous systems, communication

Additional Key Words and Phrases:
ODP, trading, service invocation, service interface autonomy, interoperability

1. Introduction

A general goal in distributed computing is to support interoperability in heterogeneous environments. There will continuously be heterogeneity in network architectures used in different application areas, because of the differing needs of network exploiters. For example, services supported by telecommunication providers are different from those supported by academic networks. However, there should exist a common meta-architecture that can join the concrete architectures together. Such an architecture is the Open Distributed Processing Reference Model of ISO and ITU (RM-ODP) [ISO10746]. Architecture models like TINA [BaBI93] and OMA [OMG90, Sol92] have adopted ideas comparable to those of ODP.

The utilization of commonly accepted meta-architecture is vital for the future information society. It allows interoperability between systems both horizontally and vertically. Horizontally, systems supporting similar enterprise goals can increase exchange of information. System integration also expands the possible marketplaces of information and information processing products enormously. Vertically, systems with different goals can be integrated seamlessly. By integration of traditional telecommunication services, traditional computer networks and embedded systems, more information can be accessed and more flexible applications can be achieved. For example, even in a small enterprise, integration of computer network, telephone information systems, telephone switch and security control system could give additional value for computer security, reachability of people, and freedom to move around in the buildings and still work efficiently. Ubiquous computing [Wei94] even requires this kind of vertical system integration.

Distributed computing concerns a set of *autonomous* systems, which can, without any external intervention or obligation, decide which services they provide and who are allowed to use them. In order to cooperate, the systems should, however, be consistent on the services and specially their semantics: users should be able, in a controlled way, to use services provided by an other system or to utilize the resources of another owner. The key issue here is the concept of transparency: the users should not be forced to understand the global system configuration and the technical differences in services due to that configuration. Services should be distribution transparent, i. e. the user should not (if not explicitly interested) be aware of the location or migration, failures or recovery, implementation technology or invocation mechanism within the implementing platform of the service provider. Instead, the user should only see an abstract service through a concrete interface selected either by the user itself or introduced to the user by the administrator of the local subsystem.

Benefits from the meta-architectures, such as RM-ODP, can be achieved by following the programming paradigms they implicate. This means that traditional communication paradigms (object models, message passing, client-server configuration) are faciliated with *late binding,* *attribute-based naming, and more precise service invocation mechanism.* Extensions using these features can lead to client-*service* paradigm (in contrast to client-*server* paradigm).

Late binding means that object templates (program files, etc.) do not define exactly which object instances are going to communicate with each other. Instead, the templates describe what type of objects are required to fulfill the required services. At run time, the object instance will request services from its environment following that description, and the supporting infrastructure will select from the available service provider objects the most suitable one at that moment. Attribute based naming is required to achieve late binding. Attribute based naming means here that the target is only described through restrictions upon property values typical to the type of service requested, instead of giving a full address or identity of the target object.

The client-service paradigm concentrates around client objects and their need of services from the underlying, possibly very large, coalesced computing environment. In our architecture the clients communicate their needs of service as task descriptions (in ODP terminology: interrogations, or even announcements) to the infrastructure. The task descriptions specify what type of service is required and under which conditions a service provider is accepted to perform the task. The infrastructure in turn has contracts with a group of service providers. Therefore, the infrastructure is able to select a performer for the task and also give some promises of the service quality to the client.

With the assistance of the ODP concepts we have created an architecture called DRYAD and partially implemented its services at the University of Helsinki. The DRYAD architecture has adopted especially the ODP communication model, but the architecture also allows both the client-server and the client-service paradigms to coexist, and thus it provides a smooth transition path from one paradigm to another.

In this paper we first describe the developed architecture. It resembles the ODP infrastructure as we think it will look like in the future. Then we show an overview of the software developed for this infrastructure and our future implementation plans.

2. Architecture

The basic requirement in RM-ODP and other related frameworks is distribution transparent access of services. Distribution transparency means that, from the client point of view,

(i) services are always available at the same location (i.e. existence of location transparency),

(ii) services are highly available (i.e. existence of failure transparency), and

(iii) services have always similar interface (i. e. existence of access transparency).

In DRYAD architecture, these properties are based on *service access domains* and usage of *service invocation function*. We have tried to construct these services so that the task delegation paradigm becomes flexible enough for requirements of heterogeneous environment, and would probably also suit the needs of mobile communication. Important goal has also been to allow variety in service interfaces in different user organizations.

We consider first service access domains and then describe task delegation semantics (interrogations, announcements) of the service invocation function.

2.1. Service organization

In the DRYAD architecture one of the basic concepts is a domain [ISO10746]. Domain is an area in which a shared rule is applied: A technology domain may consist of a shared computer, a communication domain has one communication protocol in use, and a security domain restricts a set of objects allowed to communicate with each other. Different domains may affect objects in the system simultaneously, but there is no need for the domains to have similar granularity or equal borderlines. In each case, where communicating objects are in different domains, interceptors are used. For example, two objects in the same security domain may need a protocol interceptor in order to exchange data.

In the DRYAD architecture, a special domain of service access interfaces is introduced. Support of these domains is based on the ideas of abstract service types and the concrete interfaces as instances of these types. In order to interoperate, there must be some consensus about the semantics of services generally used. We call this consensus *abstract service type*. The service access interface domain consists of interfaces administered together in order to provide a consistent platform for a group of users or clients. The offered platform would normally follow some standard about platforms (POSIX, etc.), but it could also support locally required special services. In a domain, the administrator can autonomously decide what services the users of that domain should be able to access and what concrete form of access interface they are offered. The offered interfaces are not necessarily identical to those interfaces directly supported by the service providers, which possibly are situated into remote parts of the network. In a technology domain, there can coexist several service access domains, e.g., in one computer there can simultaneously be (i) clients using the native operating system services and (ii) clients using modified access interfaces to the same services, backed up with similar services from another computer in case of partial failure of the original computer.

We utilize service access domains in two ways. Firstly, we allow, in general, different views to abstract services to coexist in the global system. These views are presented as different interface types (sets of operation signatures). The access paradigm, as viewed by clients, is based on the dynamic selection of a representative instance of the abstract service class. Because of the common abstract class, heterogeneity of interfaces can be handled and the service may appear access transparent. Secondly, we allow, in particular, the existence of domains, where clients' view is restricted to statically preselected instances. This can be done, because the client-server paradigm is a special case of the client-service paradigm, and references to individual instances can be replaced by references to classes. To support traditional software, we offer *virtual server* interfaces that intercept the original communication paradigm of the client to the task delegation paradigm. Additional value is obtained by extended service availability, because the fixed configuration can now be augmented by automatically and dynamically selected fallback servers.

Because of the required interoperability in large networks, there must be some standards about the services. There must be a framework like RM-ODP to divide the tasks in distributed environment into logical components, but there also has to be standards about the abstract service types. However, there can be several different standards about the concrete service interfaces, as long as they can be mapped to the abstract services standardized. Technically, this mapping is done with interceptors [ISO10746], which are small software components working as "light-weight protocol transformers" where needed.

Of course, there must also be support for creating and using new services, before they get standardized. There must be place for new services locally, but the interworking mechanism should also allow sharing these unstandardized services, as well as the standardized ones. We have tried to address this when planning type management.

2.2. Service invocation function

Service invocation function unifies all service requests to interrogations and announcements. The function is supported by an open infrastructure [Kut94b]. We consider service providers as members of the infrastructure. They may join and leave this membership as wanted.

When the *servers* join the infrastructure, they establish *a liaison with the global infrastructure*. The contract characterizising this liaison states what services they are willing to perform, what interface style (operation signatures) they are able to understand, what limits there are for servers environment for correct performance, and the conditions on acceptability of service requests. On the other side, clients initiate contract establishments by invoking announcements and interrogations. Both of these task delegation types involve a contract with the infrastructure. This contract states quality of service (QoS) for two stages: firstly, there is a *contract* about communication *between the client and the infrastructure*, and secondly, there is *a contract about the service* that the infrastructure takes responsibility of mediating *to the client*.

We define QoS attributes on interrogations and announcements in two groups. The first group expresses the contract between peer (application) objects. This group includes

- clients reservation to cancel the operation at the peer object, and
- clients implication in whose environment the operation is virtually executed, i.e. whether the objects interacting are loosely or very tightly coupled.

Technically, the first attribute gives a hint of need to use transactional procedures within the transportation layer and execution on tasks at the server provider side. The client may desire to cancel the operation for any reason dictated from the nature of the application. This attribute does not deal with cancellations due to technical communication failures. The second attribute can hint for a need of distributed, shared memory. A client may allow the server to directly change its own internal state. (This feature would allow behaviour reminiscent of messengers [Tsch93] or ships [ChPh94].)

The other group of attributes expresses the contracts between the requesting object and the supporting and mediating infrastructure object. This group includes

- decision on whether the communication is synchronous or asynchronous in respect to the peer object,
- resource consumption decisions, stating for example for how long the operation at the peer object may last, and
- implication, how eagerly the client expects the infrastructure to retry and watch that the service is actually in progress.

Synchrony in regard with the peer object means that the client aims to wait until the server has finished its task. Synchronous delegation of a task does not necessarily require any results to be returned. However, when the client is released from a synchronous interrogation or announcement, it knows that the delegated task has been performed. In the asynchronous case, the client only waits until the server can safely be guaranteed to receive the task invocation.

Resource consumption limits may reflect costs, if accounting is important for the application area. From a communication technology point of view, resource consumption deals with time limits. Now, the client has means to individually announce how long it is going to wait for an answer and when it considers reasonable to break the contract and assume that no processing or return of results is going to take place.

The third attribute, "eagerness", allows clients to tell if the operation is idempotent from a semantic point of view. This would allow as many retries as needed to complete the task. Or, if the client really does not care whether the task is completed in every case, unnecessary computing can be avoided. For example, when sending broadcast messages to a group of workstations just to pass information to current users, there is no need to retry on a failed workstation. A normal case would naturally be something like exactly once semantics with remote procedure calls.

Failure conditions arising during announcements are not part of the peer object communication scheme. If the infrastructure notices that a failure of the server or the transportation system or execution system has occurred, it uses the binding between client and itself to inform the client about the exceptional situation. Some part of the infrastructure is always in the same failure group as the client itself, so the infrastructure is always able to give some information to the client.

The liaisons between infrastructure and other objects are characterized by contracts private to each liaison. Therefore, attribute values can be negotiated individually for each liaison. Moreover, individual contracts can be dynamically renegotiated and changed, if the environment of involved objects change during the existence of the liaison.

The service invocation function utilizes *trading function*, *type management* and *policy management*. Both service invocation function and trading function are distributed over several interworking domains, whereas type management and policy management are strictly local and support autonomy of each domain.

The trading function allows selecting suitable service providers for clients at service request time, instead of compilation or configuration time. Therefore, trading supports late binding, when it is used together with service invocation. At the trader interface, a client can describe through property constraints and abstract service names what kind of service is needed and what kind of service providers could perform the task. Thus, additionally to the attribute based naming facility, the trading function can also support user tailorable services. For example, there could be a general editing interface to produce pieces of text. Each user would have a profile introduced to the system, and each time an editing command is entered, the system would request the local trader to select the most suitable editing program for the requesting user. The result would probably need to respect also the terminal equipment characteristics.

Type management has the responsibility of checking whether a requested service interface could be mapped to available interfaces through an abstract service type, and also providing information about required interceptor components.

Policy management is shared between the trading function and the service invocation function. Therefore, the acceptability of invocation can be checked during the trading phase and thus no retries for security reasons are needed.

3. Software

Within the DRYAD project, software has also been implemented to support the described architecture. There are five interesting components:

(i) service invocation function supporting announcements and interrogations,
(ii) trading function to select service providers to perform a task; selection is based on dynamic state of client, servers and infrastructure,
(iii) type management facilities,
(iv) virtual server support to facilitate service access interface domains and
(v) a graphical browser to view available service types and service provider in the network.

When implementing these components, we have aspired services that can be added on top of any traditional operating system. Thereby we can gradually introduce new services required by open environments.

The service invocation function is under implementation at the time of writing. The implementation is based on the function specification written using ODP modelling technique [Kut94a]. The service invocation function requires the help of trading function, policy management function and type management function. It also requires that all objects have the properties shown by the ODP infrastructure model, in order to make binding between objects possible.

The DRYAD trader functionality is rather close to the current ODP-Trading function [ISO13235, Kut94d]. The DRYAD trader supports, like the current ODP-trader, import operations with matching criteria and preferences, and it allows usage of dynamic property values in describing servers. It is also able to take part in dynamic configurations of traders for interworking between different domains or communities [Ven94]. The new policy framework in [ISO13235] is supported by this trader [Kut94c]. The DRYAD trader uses Debbie database [KuKu93a] as storage service.

The main difference between the DRYAD trader and the current ODP-trader is that the ODP-trader expects all information about available service providers to be exported to it by exporter objects. In DRYAD trading community, a type manager knows most of the needed information all the time, and the "exporter object" proclaims the availability and the current location of the service provider only to the trader. Also, type manager can be dynamically updated by new service types. A third component, a policy manager, which is shared by the invocator and the trader, keeps record of the service providers allowed to proclaim themselves within each service type.

Access to the trading services is supported by a library with functions like call-trader, proclamation and connect-to-trader. Call-trader includes all import operations, proclamation supports the exportation of services, and connect-to-trader supports binding of objects to the trading function. We plan to increase the functionality of this library to contain communication with the invocator, that is, to also support interrogations and announcements.

The current type management system is very simple -- only a flat database structure. However, the trading system in DRYAD is purely a mechanism, and all the knowledge of service types comes through the type management system. Therefore, the dynamic introduction of new service types is straightforward. To support this, and also the policy management, we have implemented a graphical management interface, Nereid [HHKL93]. It allows configuration of trader cooperation, addition of new service types and manipulation of old ones, and also allows creation of access control lists for both clients and servers in the system.

As we support the transition from client-server environment to the client-service environment, we have a support function ("a secretary", Daphne) for virtual servers [KuKu93b]. A virtual server looks like a traditional server (for example a program file or a stub for remote procedure call), but at each reference to it, it actually interrogates a concrete service provider through the invocator mechanism. By this construction, we can utilize the new trading services and task delegation semantics (interrogations and announcements) from old applications, without modifying the software.

For testing purposes and for the users of distributed systems, we have implemented a service browser called Artemis [KKPS93]. It allows users to view the available service types, to read descriptions about them, and also to browse the service providers available at the moment.

4. Conclusion

In the DRYAD architecture and its supporting software, we have aspired three main goals. Firstly, it is required to offer task delegation paradigms that are flexible enough to answer the requirements of heterogeneous environments. Therefore, we adopted the concept of contracts from the ODP framework and utilized it between client and servers, client and infrastructure, and servers and infrastructure. Secondly, we think it is important to offer a task delegation paradigm that allows variety in services and service interfaces to be constantly maintained. Therefore, we based our architecture on the idea of abstract services that take different concrete forms as service interface types at different service access domains. Thirdly, we think that a smooth propagation route to new task delegation paradigms is needed. Therefore, we created software to be added on top of traditional operating system services in such a way that services required by open environments could be gradually introduced and taken in use. We also introduced the virtual server concepts to support the coexistence of several communication solutions. With this experiment, we wanted to show that current systems can smoothly evolve into modern, open and distributed environments.

Acknowledgments

This article presents work done in the DRYAD project at the University of Helsinki. The project has received financial support from the Finnish Technology Development Center TEKES and four Finnish industrial partners during a two year study period in 1992-1994.

References

BaBI93 Barr, W., Boyd, T., Inoue, Y., The TINA Initiative, IEEE Communications Magazine, March 1993.

ChPh94 Chanson, S. T., Phillips, G. W., The Ship Model for Communication and Mobility in Distributed Systems. First International IEEE Workshop on Services in Distributed and Networked Environments. Prague, Czech Republic, June 1994.

HHKL93 Hatakka, J., Hyvätti, J., Koskinen, I., Liljeberg, M., Moilanen, M., Ylitalo, J., Nereid - A graphical management interface to DRYAD. Project exercise. November 1993. Department of Computer Science, University of Helsinki. C-1993-69. In Finnish.

ISO10746 Information Technology - Open Systems Interconnection, Data Management and Open Distributed Processing - Basic Reference Model of Open Distributed Processing. Part 1, Part 4, Committee Drafts, July 1994. Part 2, Part 3, Draft International Standards, November 1993.

ISO13235 Information Technology - Open Systems Interconnection, Data Management and Open Distributed Processing - Trading function. Suggested Committee Draft (output of Southampton SC21 meeting in July 1994), September 1994.

KKPS93 Kekkonen, S., Korkiakoski, J., Partanen, R., Siivola, V., Vepsäläinen, H., Artemis - A graphical user interface to DRYAD Trader. Project exercise. June 1993. Department of Computer Science, University of Helsinki. C-1993-43. In Finnish.

KuKu93a Kutvonen, P., Kutvonen, L., Debbie - A database management system. Report #22 of DRYAD project. University of Helsinki, Department of Computer Science. August 1993.

KuKu93b Kutvonen, L., Kutvonen, P., Broadening the User Environment with Implicit Trading, International Conference on Open Distributed Processing. Berlin 1993. Elsevier North Holland Publishers B. V.

Kut94a Kutvonen, L., Coordinating the access of services within DRYAD architecture. Report #25 of DRYAD project. Department of Computer Science, University of Helsinki. March 1994.

Kut94b Kutvonen, L., An infrastructure for distributed service invocations in a heterogeneous network. Manuscript submitted to ICODP'95. June 1994.

Kut94c Kutvonen, L., Achieving interoperability through ODP Trading function. To appear in the Second International Symposium on Autonomous Decentralized Systems, ISADS'95.

Kut94d Kutvonen, L., Comparison of the Dryad trading system to ODP Trading function draft. Internal technical report. Department of Computer Science, University of Helsinki. C-1994-51. October, 1994.

OMG90 Object Management Group, Inc., Object Management Architecture Guide 1.0. OMG TC Document 90.9.1. November 1990.

Sol92 Soley, R. M., An Object-Oriented Architecture for Distributed Systems. Information Networks and Data Communication '92, Espoo.

Tsch93 Tschudin, C. F., On the Structuring of Computer Communications. Universite de Geneve, Departement d'Informatique. PhD Thesis. Geneve, Switzerland, 1993.

Wei94 Weiser, M., Ubiquous computing, Keynote address in the 14th IEEE Conference on Distributed Computing Systems. Poznan, Poland, June, 1994.

Ven94 Venetjoki, T, Cooperation of traders in DRYAD. Department of Computer Science, University of Helsinki. MSc Thesis. April, 1994. In Finnish.