SOVEREIGN SYSTEMS AND DYNAMIC FEDERATIONS

Lea Kutvonen

Department of Computer Science
PBox 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND
Lea.Kutvonen@cs.Helsinki.FI

Abstract: Modern information services require world-wide cooperation, and involve groups of autonomously administered computing systems, i.e., sovereign systems. Traditionally, system integration has been facilitated by forcing a common interface layer on each of the systems involved. However, autonomous administration causes asynchrony of service evolution and variation in common service behaviour in each system. In such an environment, a single middleware solution cannot be practically required to be the basis for globally integrated software systems. Instead, capabilities are required for dynamical establishment of federations across different middlewares, i.e., capabilities of negotiating on new cooperation relationships amongst independent systems. There is no call for new middleware solutions, but further exploitation of current middleware services could lead to added interoperability with less administrative effort. The essential feature for federation is systems' mutual reflection. This requirement is not too hard to achieve at practical level in systems that are based on the current emerging frameworks, such as OMG/CORBA and TINA. This paper discusses binding processes within federated systems in order to show how various middleware services together build a mutual reflection mechanism.

Keywords: Open federated systems, RM-ODP, computational bindings

1 INTRODUCTION

Modern information services require world-wide cooperation. Areas of cooperation cover not only electronic commerce and EDI applications, but also, for example, computer supported cooperative work across companies, and even independent development of software components that are application-area specific and capable of federated cooperation.

Traditionally, system integration has forced a common interface layer on each of the systems involved. However, autonomous administration causes asynchrony of service evolution and variation in common service behaviour in each system. In such an environment, a single middleware solution cannot be practically required to be the basis for globally integrated software systems. Instead, capabilities are required for dynamical establishment of federations across different middlewares, i.e., capabilities of negotiating on new cooperation relationships amongst independent systems. There is no call for new middleware solutions, but further exploitation of current middleware services could lead to added interoperability with less administrative effort.

Federated systems are able to reflect on their own capabilities, and furthermore, to reflect on the capabilities of their potential cooperation partners. The reflection facilities include infrastructure functions for exchanging meta-information about their services, i.e., the same tools as single middleware architectures suggest. The emerging frameworks, such as OMG/CORBA [15] and TINA [19], answer these requirements to the extent of self-reflection, but not sufficiently at the level of mutual reflection. The open distributed processing reference model (ODP-RM) [3], supports the federation model further.

Cooperation requires that the sovereign nature of the involved organisations and computing systems is acknowledged appropriately. Aspects of autonomy include

- independence of service evolution. The evolution independence covers not only the service implementations, but also service interfaces. Services can be personalised for example for varying user groups. (Personalisation pattern is familiar from the context of micro-kernels [20].)
- autonomy on decisions on cooperation partners. Service based federations (in contrast to node based) require standardised contract schemata. Work on such schemata has already been initiated, for example, within the business object modelling special interest group (BOMSIG) of OMG [17].
- independence of administration. Mostly, the internal operational policies are not relevant for cooperation. Only some behavioural and quantitative aspects of the exploited services need to be agreed on. Such aspects should not be administrative decisions, but subject to negotiation between the service user and the service provider [9].
- sovereignty of choice in languages and specification techniques. Similarly as the organisations are free to choose the methods for their service provision, they freely choose methods for self-reflection. The freedom is limited by the need of mutual reflection, but the common methods should support evolution without forcing the involved systems to synchronise their evolution steps.

Theoretically, a general algorithm that would always resolve all potential for cooperation, is not possible. The aspects of negotiation vary too widely:

- the languages for expressing service interfaces or behaviours are not known,

- the level of detail for service interface or behaviour expressions is not fixed,
- the naming systems of the negotiators are not necessarily comparable,
- there is no single set of underlying services that could be trusted as the platform on which the partners will be executing, and finally,
- the decisions on cooperation have to be done while the system users are waiting, which may mean very hard real-time problems.

In practice, a general algorithm is not needed. In order to achieve additional system support for interoperability, it is sufficient to restrict to some heuristic solutions, and to some areas of negotiation with common interest. The negotiation areas are chosen by market forces and subsequent de-jure and de-facto standardisation. Such areas are, for example, categories of services available at electronic markets. Defining a category would naturally lead to an agreed level of abstraction for describing the related behaviour patterns, and would also restrict the set of applicable languages.

Dynamic federations and mutual reflection is supported by middleware services:

- federated trading services (dynamic repository of service providers),
- federated type repository services (dynamic repository of service types, federation contract schemata, and mappings to local technology solutions), and
- federated binding framework.

Currently, middleware services promise federation. Trading services [2, 16] already have the necessary support for federation. Whether a trader installation actually supports federation or interworking model across homogenising middleware, depends on how the trader is supported by other middleware services [10]. However, the current type repository service [4] supports the federation model adequately only if used together with the open binding framework [10]. Thus, the corresponding meta-object facility (MOF) [14] is not adequate for federation purposes, although it is rigorous with all other interworking models. The current open binding framework [5] allows federation model to be applied, although it does not directly specify the details [10].

In the following, the system requirements induced by federation establishment are discussed. The exploitation of federation model requires that each computing system includes a binding factory capable of federated operation; that each organisation has a private trading service; and that organisations are capable of mapping together their contract semantics. Also, the status of CORBA and TINA architectures is discussed.

2 OVERVIEW OF THE FEDERATED BINDING PROCESS

This chapter studies the special characteristics of federated systems, and reviews the concepts, processes and supporting repositories necessary for federated bindings.

2.1 Federated object model and concept of binding

Only application level service packages, like teleconferencing services or bank interfaces, can reasonably be expected to interoperate in a federated environment. Entities like UNIX processes or Java objects are excluded from the federation discussion, because the mechanism is far too heavy for such detailed integration. Objects within the scope of discussion have the same granularity as those called megamodules elsewhere [21].

Federations between sovereign systems do not form a single static network of nodes. Instead, the application objects are able to join in various communities at run-time. Instead of a server interface, the architecture focuses on the client view to the service: the client object is interested in having a functionality performed and the service can be eventually performed by some group of objects. The scope of the federation network of application services is restricted by the interworking capabilities of the platform services. The relationship between the service liaisons and infrastructure liaisons is separately determined at each sovereign system (Figure 1). A computational object corresponds to an application level object; an engineering object corresponds to an application object together with the supporting platform object.

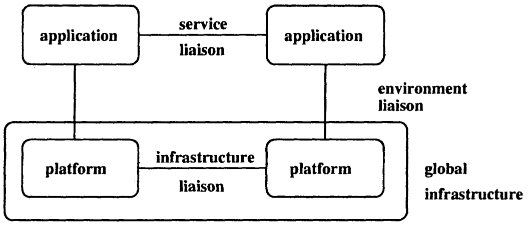

Figure 1. Global system view of federated system.

The appropriate concepts in object-oriented programming environments are the application level service types and the binding types. Service type is defined to capture the expected interface type, and other information about the functionality, including quality of service aspects. The service type defines a structure for those contracts that are established between cooperating objects. A binding type [5, 1, 8] is a computational abstraction that defines the communication partners and the communication rules for a group of cooperating objects. The supporting services of the communication relationship must use the binding type as a specification of the necessary channels between the involved objects. Also, the end-points of the channels must conform to the interfaces of the involved application objects. The concepts of service type and binding type must be supported by infrastructure functions that

- map service types into client-role interface structures and associated QoS contract structures,
- map binding types into appropriate channel configurations, and
- offer facilities for dynamic management of the channel.

A binding facilitates signalling between computational object interfaces. In federated environments, result of the binding process needs to be a binding liaison, that ensures that a communication channel can be created between the communicating objects [5]. The time at which the involved software components are configured and the resources reserved can be optimised based on the actual communication frequencies.

2.2 Binding process

For an application programmer, only the application interface aspects should be visible. The infrastructure layer services enhance these aspects into a full engineering view.

For the application programmer the objects and the binding process can be seen as follows (see Figure 2). First, objects are composed of a set of interface objects and private state information. An object can be requested to modify the encapsulated information by sending it a signal that matches an interface signature. Second, for each server object, the signature and behaviour associated with an interface are described by object property values. The property values can be stored into a trader, as service offers (potential contracts). In the trader, each service offer is structured according to an abstract service type definition, that specifies a contract schema. At run-time, when a client object presents a service request (potential contract), the infrastructure requests the trading service for matching service offers. When a suitable set of offers is found, a binding liaison is formed. As a result of a successful binding process, each application object can be expected to have a communication channel to its peers.

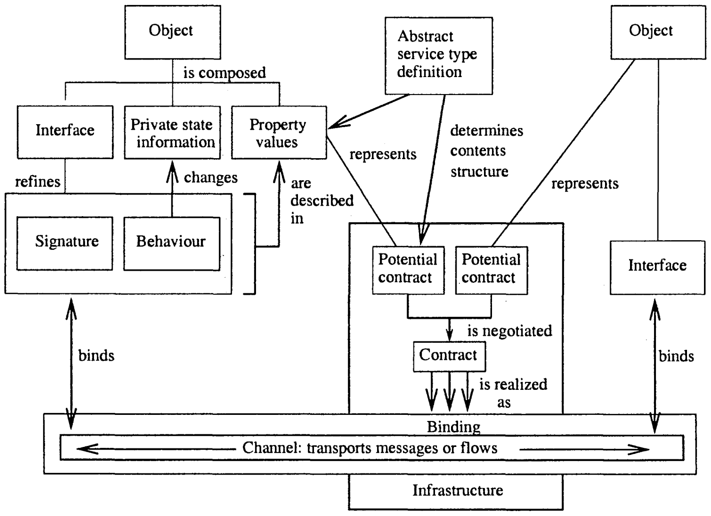

Figure 2. Object structure and application level view to binding.

Viewed from the infrastructure, the federated binding process is more complicated, as shown in Figure 3. The negotiation task is split into two parts, service liaisons and infrastructure liaisons (Figure 1). A service liaison captures the application interface involved, while an infrastructure liaison captures the facilities of the platforms. Consequently, a binding liaison between application objects consists of the application specific part and the infrastructure specific part. The first guarantees application interoperability, the latter interworking properties of the platform.

The infrastructure liaison naturally restricts the possibilities of application level liaisons. Therefore, the phases of the federated binding process are as follows (the concepts used in this summary are discussed later in more detail):

Step 1. For the application level potential contacts, find the platform level potential contracts. The application level potential contracts are created by application programmers and eventually stored into traders. For each application, also requirements for platform services are stored (e.g., as references to platform offers), and therefore the platform level potential contracts can also be retrieved from a trader. For each possible platform offer, a joint potential contract is formed.

Step 2. Use the trading function to match together the joint potential contracts of each involved system. The client system directs a trading request to the other involved systems, in order to find the required application service, with the limitation that also the supporting platform service is similar to that of the client's. Quality of service values, for example, are not necessarily considered at this stage.

Step 3. Choose one of the matching pairs. The selection heuristics is dependent on the administrative choices of the initiating system: it may be random, cost dependent, analyse quality of service guarantees, etc. As the decision may be time-critical and the process essentially automatic, there is no need for an elaborated negotiation protocol here, but the decision can be localised. Although the selection of QoS attribute values is an interesting aspect, there is not much that can be said in general: each potential contract captures some QoS offers and QoS requirements, from which a commonly accepted subset of values can be collected into a QoS agreement [6]. The details of the matching process are specific for the service type in question.

Step 4. Create a binding contract and fill it in with attribute values acceptable to all binding liaison members (both application and platform layer aspects).

Step 5. Deliver the binding contract to all involved systems and create a channel controller to manage it. Transform the contract information into the locally understood formats using the type repository services for the mappings (see Section 2.4.2).

Step 6. Instantiate the channel using the binding contract information.

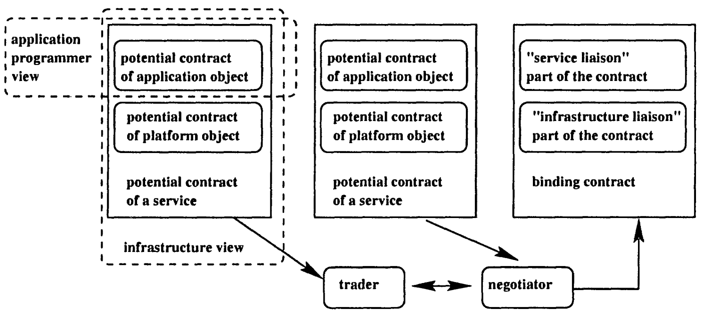
application programmer view
potential contract of application object
potential contract of platform object
potential contract of a service
infrastructure view
potential contract of application object
potential contract of platform object
potential contract of a service
"service liaison" part of the contract
"infrastructure liaison" part of the contract
binding contract
trader
negotiator

Figure 3. Infrastructure view to the federated binding process.

2.3 Concepts for binding representation and management

The essential concept, the result of the federated binding process, is a binding liaison. It can technically have two states: (i) a binding state, that is a state where no communication resources are reserved but a binding contract has been established, and (ii) a channel state, that is a state where resources are in use and managed by a channel controller, which in turn is governed by the binding contract. The concept of interface reference is used for addressing interface instances; the concept partially overlaps with binding contract of an established channel.

2.3.1 Channel. A channel is a configuration of intermediate objects (traditional components, like stubs for marshaling, binder objects for controlling the channel connectivity, and protocol objects for data transfer) that are able to route signals (operation invocations, terminations, flow signals) from one application object to another. The channel components can be selected at run-time, instead of compilation time as in many RPC implementations. Also, several stubs can be active concurrently, using the same protocol link underneath. Separate concurrent protocols for channel components are, for instance, group management [18], and QoS management [22]. In a general case, the stubs are not self-sufficient, but require services from management functions like authentication services [7].

A channel does not necessarily form a static circuit through the network; a channel can be based on connectionless protocols.

A channel reaches from one sovereign system to another, becoming thus logically partitioned into independent channel sections. Each channel section is administered by a single sovereign system.

2.3.2 Binding liaison. The binding liaison is realised by a set of independent channel sections and a channel controller. The channel controller carries the binding liaison information even when the channel sections are not present. The number of channel sections involved and the interface types supported at each channel end-point is dependent on the binding type.

The binding liaison information is captured to a binding contract object that is replicated for each of the sovereign systems involved. The binding contract object is encapsulated into the channel controller and can therefore be managed through the channel controller interfaces.

The interfaces to be interconnected can be either identified by the binding initiator or searched during the binding process based on their properties. Thus, only the computational interfaces are bound together, instead of creating a channel between the initial locations of the interfaces.

2.3.3 Channel controller. The role of the channel controller [10] is dependent on the binding type selected by the object requesting to be bound to other objects. For instance, the channel controller may monitor the membership of the binding liaison and act as a leader in failure detection protocols.

Similarly, the actual channel structure varies depending on which distribution transparencies are selected by the user, and which communication protocols are in use. The actual channel structure varies also depending on the platform architecture and administrative rules within the systems that administer channel sections.

A channel controller is a direct client to all of the channel component's management interfaces, and therefore it offers a combined control interface to all of them. The channel controller has a specific object at each domain, and those objects may cooperate in order to offer a joint binding liaison management service.

The use of a channel controller even allows the channel configuration and parameters to be modified during the service liaison duration [10]. Changes can involve, for example, multi-cast group members or timeout values when a fixed network line is switched to a mobile network connection. Changes can also involve QoS aspects: any of the bound objects may instruct the channel controller to change a QoS attribute value to something valid within the QoS agreement between them.

In traditional systems, the functionality of channel controllers is often embedded to applications, which again makes the model unsuitable for federation, as all management actions would violate the immunity of sovereign systems.

2.3.4 Binding contract. The binding contract information is replicated to each of the object interfaces involved in the liaison. Because the interfaces can reside at separate systems, the data representing the contract information may have different local formats and coding. Each object can use the local copy of the binding contract as policy information or parameters to its internal activities. This mechanism can be used to provisioning of the binding contract as part of the object behaviour (see [13] for an example).

The binding contract collects together information required by all binding related functionalities (the sufficiency of the description techniques discussed in [9, 10]):

- a service type identifier (for each type system involved),
- a binding type identifier (for each type system involved),
- service type specific QoS agreements as name-value pairs,
- names for binding type specific failure detection and recovery protocols, failure defined as not being able to meet the QoS agreement,
- name for a remuneration protocol,
- technical descriptions (IDL or other language, also names or identifiers can be used) for interface signature expected by the client (can be differently selected at each sovereign system),
- name of a communication protocol,
- channel type identifier (for each type system involved), and
- interface reference for the channel controller at each sovereign system.

Figure 4 illustrates how a binding contract can be realized. The binding contract structure includes both the service and infrastructure liaison related aspects, and agreements related to the maintenance of the binding contract itself.

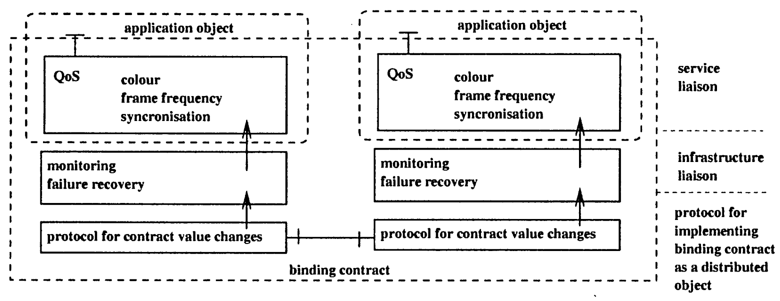

Figure 4. Example of federating a binding contract.

2.3.5 Interface references. According to the ODP reference model an interface reference is a structured identifier for an interface. An interface reference is created together with the interface and used during channel section instantiation for accessing the interfaces [5].

This definition resembles the ones for potential contracts and binding contracts. Indeed, the concepts overlap. However, in systems like CORBA, the interface references capture only the platform level aspects of the binding contract, leaving out the aspects related only to the application layer. In addition, the contract negotiation is not a run-time activity, but the contract is implicit and static. Therefore, problems arise, for example, when quality of service aspects between application interfaces need to be captured. In TINA model, the binding contract information appears partially separated from interface references, but only as QoS contracts.

2.4 Concepts for binding factories

This section studies the concepts required for binding establishment by federated binding factories. The federated process of independent factories create a set of cooperating channel sections, based on the type and template descriptions stored into type repositories.

2.4.1 Channel instantiation. Each sovereign system needs an independent binding factory for instantiating sections of communication channels defined to be located at their domain [9, 5]. The channel sections are instantiated based on channel templates that specify the required configuration of stubs, binders, protocol objects, and interceptors. The required configuration may be different at each platform or at each channel administration domain. For local bindings within a (single administration) system, optimised channel templates can be used.

Although the channel sections are instantiated independently from each other, they must work together. Therefore, binding contracts must use shared concepts for expressing the required interfaces and functionality of the channel end-points.

In the binding contract, the concept of channel type is used. Channel type denotes what is the expected channel functionality (distribution transparency and QoS requirements, security support) and required behaviour in case of channel failures.

In some cases, the application federation can be retained while the channel is totally reconstructed. During channel creation, each factory receives a copy of the binding contract and tailors it according to the local platform requirements.

2.4.2 Type repository contents. The concepts required by binding factories are mainly supported by the type repository function. The concepts of binding type, channel type, channel template and channel controller template need to be represented as target concepts in the type repository system [4]. These concepts are illustrated in Figure 5 which also denotes the relationships between the concepts. The illustration shows two type domains, e. g. sovereign systems.

In each system, a set of binding types is known and offered for programmers. The binding types that are expected to support federations must be known at other systems as well (not necessarily with the same name).

For each binding type, a set of channel types can be used. The channel types can differ by the support they offer, for example, in terms of selective distribution transparencies (migration transparency, transaction transparency, etc). Also the channel types used in federated environments must be recognisable at multiple systems. However, differences for example in data representation techniques are allowed and transformation information is captured by references to suitable interceptors.

Finally, for each channel type a set of channel templates and channel controller templates are supported. The channel templates selected for a federated binding should fulfil two requirements:

- The supported application object should be offered the service view that was denoted in the binding type. The service view covers logically the service behaviour and QoS aspects, and technically the binding object interface signature.
- The protocol objects that support the data transfer between the federating systems should be similar. Also the channel controllers have a predefined channel with the same requirement.

2.5 Summary

In federated environments, interoperation capabilities are often explored and enabled as part of the interface binding process. A client may request a named service from its environment, i.e. a behaviour pattern, instead of addressing a server object to perform an operation. For binding purposes, the origin of the service providing object is not interesting, only the object type. Therefore, the object can either be selected based on a specification of required behaviour and QoS aspects, or instantiated from a system specific template even as late as at service request time. The concepts available from the type repository support both the matching of offered and requested service types, and the subsequent mapping into templates for instantiation.

In a federated environment, it is essential that the result of the binding process has two abstraction levels; a specification of the required communication aspects and an implementation of that specification. The control of the binding can thus be managed through the specification level, which can be understood by all federating parties. The

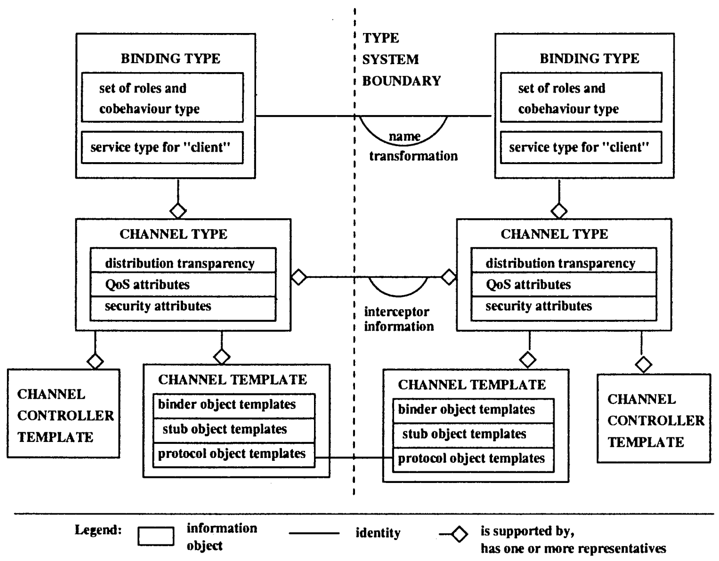

Figure 5. View of the federated type system.

indirect control of the implementation level is then delegated to each system involved. The concepts of a distributed channel controller and a distributed binding contract support this pattern.

Separation of types and templates – and correspondingly, binding specification and implementation – originate from the sovereign nature of the federating systems. As each system is free to choose technological solutions, the object templates and channel implementations differ. However, these implementations have a common conceptual background, which is captured by the types and binding specifications.

The fundamentals of the federated system model can be found by studying how the two abstraction levels interlock: The type repository carries information on abstract types and concrete templates, and furthermore, on the relationships between types and templates. The binding contracts capture information both in terms of types and templates. This information is then used and manipulated by distributed channel controllers.

In current systems, the binding state information is often held by the application objects themselves. This effectively excludes federative control of bindings, because no direct management operations are allowed across the sovereign system boundary. However, the solution optimises performance in a single protocol environment.

3 DISCUSSION

The federation model presents new requirement for the middleware architectures. Moreover, it presents requirements for the world-wide organisation of the cooperation between traders, type repositories and names servers.

The basic concepts for the federated binding process discussed in Section 2 follows the lines of DIS14753 | X.930 Interface references and binding. However, even this ISO standard does not specify federated binding factories, channels and binding liaisons in detail, it only outlines that such special circumstances exists and points out the locations, at which related information can be stored. The standard makes a mapping to CORBA IIOP and shows how the general binding framework is implementable. Naturally, the interworking facilities are restricted to those understandable for IIOP protocol that does not support group communication, streams, QoS monitoring or negotiation, etc. Moreover, the IIOP protocol does not allow any inter-ORB reflection to take place, but instead, forces a single unit capable for limited self-reflection.

In comparison of the CORBA and TINA models with federated binding mechanism, major differences can be found [10, 11]. First, in the CORBA or TINA architectures client-role interfaces are not explicitly presented. Both in the CORBA and TINA models, binding contract information is present in some extent, but only associated with the server side, thus implicating that only a shared controller for the communicating partners exists. Therefore, there are no facilities for noting discrepancy between client and server views of the shared interface. The client and server role interfaces should be separated from each other and the client requirements emphasised.

Second, by definition, the CORBA platforms can interoperate only with other CORBA systems, i.e., only the ORB interfaces and IIOP protocols are accepted, although these may be achieved by adding a bridge over, for example, a DCE system. As the distributed computing platform of TINA is directly an ORB, this applies to TINA model as well. The federative model relaxes this requirement.

Third, CORBA specification acknowledge interface substitutability (suitability for being bound together) to be based only on type inheritance, not on comparison of types. This is basically due to not having separation between the concepts of type (externally visible properties) and templates (necessary for private instantiation process). Separation of those concepts would allow sovereignty of template systems that are dependent on the selected platform architecture. Meanwhile, the types could be used as a mapping tool between the various template systems. The concepts of template and type should be separated and domain based type repositories adopted. In addition, sub-typing concepts should be based also on other mechanisms besides inheritance.

In the federation of, for example, type repositories and binding contracts, the major problem is how to compare expressions of behaviour, quality of service attribute values, protocol specifications, etc. Section 2 suggested the use of identifiers, names and expressions in freely selected languages. This is based on the joint offerings of type system federation techniques and federated naming systems, as described below.

For the federated system model, the relationships between various type expressions or type names should be stored into the type repositories. That provides connectivity with reasonable time scale. Heuristics and systematic analysis on type correspondencies can be run as separate processes, thus not delaying the federated binding process at application run-time. It also allows entering human decisions without algorithmic reasoning. The current CORBA model does not support interceptor information nor name transformations between type repositories; the MOF follows an interworking model instead of a federation model [10].

The importance of integrating name systems of sovereign computing systems varies depending on the kind of name system used. In addressing systems, a set of worldwide systems is necessary (e.g., IP addresses). The systems must be joined together through a gateway system that also routes the communication traffic to the network via different protocols. In identification systems (e.g., interface identifiers), the need is similar. However, the scoping rule is not just technical but also organisational, administrative. Still, integration requires that the name spaces are joint. There is no theoretical reason for this, instead a practical reason: current distributed platforms already include such naming services. In systems, where plainly ideal concepts are named (e.g., type systems including behaviour names), other mechanisms can be used. The naming domains are rather large and extra overhead is created only when a domain boundary is crossed. There has not yet been a practical integration step for these names, so the naming systems are very different. Integrating existing systems to a single name space would be impractical and would lead to non-evolving system design.

4 CONCLUSION

This paper presents a current view to interoperability and federation based on work in ISO and other consortia. The approach differs from earlier attempts to global consensus. Instead of forcing a shared control structure, the model trusts on mappings between similar concepts in separate systems. Each mapping can be created either because of a theoretical equality or because of a practically sufficient resemblance. The benefit of the approach is that interoperability can be achieved while the systems take their time to evolve and congregate.

References

[1] BERRY, A., AND RAYMOND, K. The $A1\surd$ Architecture Model. *The Third International Conference on Open Distributed Processing – Experiences with Distributed Environments.* Brisbane, Australia, 1995. Chapman & Hall, pp. 55 – 66.

[2] ISO/IEC IS13235. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Trading function*, 1997.

[3] ISO/IEC IS10746. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing*, 1996.

[4] ISO/IEC CD14746. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. ODP Type repository function*, Jan. 1998.

[5] ISO/IEC DIS14753. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – ODP Interface References and Binding*, Sept. 1998.

[6] ISO/IEC FCD14769. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – Quality of service in ODP*, Jan. 1998.

[7] KITSON, B. Intercessory Objects within Channels. *The Third International Conference on Open Distributed Processing – Experiences with Distributed Environments.* Brisbane, Australia, 1995. Chapman & Hall, pp. 233 – 244.

[8] KONG, Q., AND BERRY, A. A General Resource Discovery System for Open Distributed Processing. *The Third International Conference on Open Distributed Processing – Experiences with Distributed Environments.* Brisbane, Australia, 1995. Chapman & Hall, pp. 79 – 90.

[9] KUTVONEN, L. Management of Application Federations. *International IFIP Working Conference on Distributed Applications and Interoperable Systems (DAIS'97).* Cottbus, Germany, 1997. Chapman & Hall, pp. 33 – 46.

[10] KUTVONEN, L. *Trading services in Open Distributed Environments.* Department of Computer Science, University of Helsinki, 1998. PhD thesis. A-1998-2.

[11] KUTVONEN, L. Why CORBA systems cannot federate? To appear in Distributed Systems Engineering Journal special issue on *OMA/ODP Workshop* (Cambridge, UK, Nov. 1997).

[12] KUTVONEN, L. Supporting Global Electronic Commerce with ODP Tools. *International IFIP Working Conference on Trends in Electronic Commerce (TREC'98).* Hamburg, Germany, 1998. Dpunkt Verlag, pp. 43 – 56.

[13] MEYER, B., AND POPIEN, C. Flexible management of ANSAware applications. *The Third International Conference on Open Distributed Processing – Experiences with Distributed Environments.* Brisbane, Australia, 1995. Chapman & Hall, pp. 271 – 282.

[14] OBJECT MANAGEMENT GROUP. *Common Facilities RFP-5: Meta-Object Facility,* 1997. OMG TC Document cf/96-05-02.

[15] OBJECT MANAGEMENT GROUP AND X/OPEN. *The Common Object Request Broker: Architecture and Specification,* May 1996. OMG Document No. 91.12.1. (Revision 2.1.).

[16] OBJECT MANAGEMENT GROUP AND X/OPEN. *The OMG Trader Object Service,* May 1996. OMG Document orbos/96-05-06.

[17] OMG *OMG Business Application Architecture,* March 1995.

[18] OSKIEWICZ, E., AND EDWARDS, N. A Model for Interface Groups. Tech. Rep. APM.1002.01, APM, May 1994.

[19] TELECOMMUNICATIONS INFORMATION NETWORKING ARCHITECTURE CONSORTIUM (TINA-C). *Requirements upon TINA-C architecture,* Feb. 1995.

[20] LEPREAU, J., HIBLER, M., FORD, B., AND LAW, J. In-kernel servers on Mach 3.0: Implementation and performance. *Third USENIX Mach Symposium,* USA, 1993, pp. 39 – 55.

[21] WIEDERHOLD, G., WEGNER, P., AND CERI, S. Towards Megaprogramming. *Communications of the ACM 33,* 11 (Nov. 1992), pp. 89 – 99.

[22] VOGEL, A., KERHERVÉ, B., VON BOCHMANN, G., AND GECSEI, J. Distributed Multimedia Application and Quality of Service – A Survey. *IEEE Multimedia 2,* 2 (Summer 1995), pp. 10 – 19.

Biography

Lea Kutvonen received her Ph.D. degree in Computer Science from University of Helsinki. She joined the permanent personnel of the Department of Computer Science in 1990; currently carries resposibilities in administration, teaching and research. Her interests include middleware architectures, ODP-RM, CORBA, and object systems. She is the current editor of DIS14753 | X.930. Member of Finnish Society of Computer Science, IEEE, and ACM.