Trading and Negotiating Stream Bindings

H. O. Rafaelsen[1] and F. Eliassen[2]

[1]University of Tromsø
Dept of Computer Science, 9037 Tromsø, Norway
hansr@cs.uit.no
[2]University of Oslo, Dept of Informatics,
P.O.Box 1080, 0316 Oslo, Norway
frank@ifi.uit.no

Abstract. Distributed multimedia information systems require a range of different interaction styles ranging from simple remote operation interaction to complex patterns of interaction involving both discrete and continuous data. The standardized reference model for Open Distributed Processing (ODP) defines a binding model that encapsulates different interaction styles within explicit binding objects. In this paper we discuss mechanisms for selecting and negotiating appropriate explicit stream bindings as required by the application. We describe the notion of explicit bindings and introduce the idea of using a trading-like facility for selecting potential binding types. We show how an earlier proposed type model for stream interfaces can be used as a basis for binding type selection, and extended to support automatic negotiation of binding properties.

1 Introduction

The notion of stream interface has been proposed as the preferred means to convey media streams in distributed multimedia systems [3]. A stream interface consists of a collection of source and/or sink media flows. The act of stream binding establishes a logical association between compatible stream interfaces for the purpose of exchanging continuous flows as dictated by the type and direction of the flows. During binding interfaces to be bound must be type-checked for compatibility. Informally, type-checking means ensuring that the properties of each source flow are as expected by the corresponding sink flow.

In the reference model for Open Distributed Processing (RM-ODP) stream bindings are explicit. This supports direct client control of the binding during its lifetime. Furthermore, bindings are first class objects and are created, managed and invoked in the same way as other objects [1]. The result of the binding action is a control interface through which the binding object can be controlled. In figure 1, the binding model is illustrated where a binding object connects four interfaces by means of local bindings that associate the interfaces of the objects with the interfaces of the binding object.

J. Sventek and G. Coulson (Eds.): Middleware 2000, LNCS 1795, pp. 289-307, 2000.

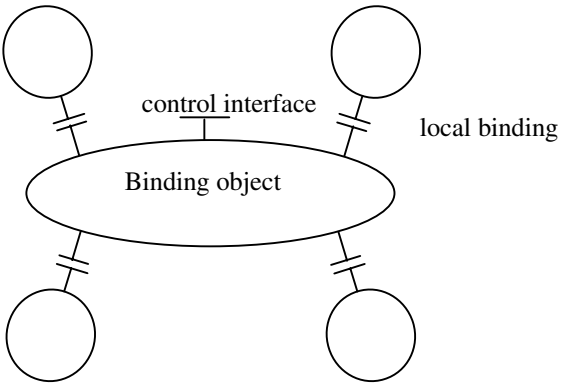

Fig. 1. The explicit binding model

Depending on the type of the binding object, new interfaces can be added to the binding, and existing ones can be removed. After a new (stream) interface has been added to the binding, it can be locally bound to a suitable application (stream) interface. Hence, for example, binding objects of the appropriate type can describe dynamic groups in which membership can change during the lifetime of the binding. The main rationale for explicit bindings is to support QoS management in terms of QoS specification, monitoring and control [1]. For example, the binding control interface of a stream binding can be used to control and monitor the QoS of ongoing streams.

A binding type (or template) defines a particular class of binding objects by identifying the type of interfaces which can participate in the binding, the roles they play, and the way behaviour at the various binding interfaces are linked [9]. For example, a multicast video binding would typically support a producer role and a consumer role. In some cases the binding will support operations for adding interfaces to a binding in a named role (e.g. a new multicast receiver), and for removing interfaces from the binding (e.g. remove a multicast receiver from the binding).

Binding factories are objects that create bindings. In our model, a binding factory is associated to a binding type such that by invoking the factory object's `create` method, a new binding of the associated type is created.

Through the local bindings the binding object receives and delivers information for binding participants according to the causality and type of the bound interfaces. Type checking as explained above, is applied when creating the binding and when adding a new interface to the binding. When creating the binding, type information about the application object interfaces to be initially bound and the corresponding roles they will fulfill in the binding, is provided as parameters of the `create` method call. Corresponding interfaces with the appropriate roles are as a result added to the binding. If an application object interface α is offered to fulfil a role β, then the type of α must be compatible with the type of β.

In this paper we present an approach for applications to select binding types and associated binding factories according to their needs based on a trading model. In this

scheme a set of binding factories are located based on a specification of the required properties of the binding. The located set of binding factories are all capable of instantiating binding objects with properties conforming to those specified by the trading client. Furthermore, in those cases where binding objects specify alternative stream behaviour (e.g. alternative encodings, resolutions, or frame rates) at its supported interfaces, we also show how the notion *of policy specification* supports automatic negotiation to choose the actual stream interface behaviour to be used.

In general terms, a trading facility as indicated above, supports the reuse of binding factories. More specifically, it will allow evolution of supported binding types in a distributed computing environment without sacrificing support of existing applications. This can be ensured by requiring that a new version of a binding type must *conform* to the older version. Run time trading of binding types is also required for multimedia databases. In this case the required properties of the binding to be used to present a query result are generally not statically known, but rather depends on the specifics of each query [18].

The remainder of the paper is organized as follows. In section 2, we offer an overview of the model of bindings and streams that we base our work on. Following that, in section 3 we present a trading model for selecting binding types from binding type requirement specifications and introduces a simple middleware architecture illustrating the application of a binding type trader. Section 4 presents our approach to automatic negotiation of binding properties for those cases where binding objects specify alternative behaviour at its supported interfaces. In section 5 we discuss some related work while section 6 concludes with an outlook to further work.

2 Model of Bindings and Streams (MBS)

For the explicit binding and stream abstractions, we have developed a generic model called MBS (Model of Bindings and Streams). MBS will constitute a part of the foundation for the programming model of an adaptive multimedia ORB called MULTE-ORB [15]. The engineering of MULTE-ORB is based on the flexible protocol system Da CaPo [17]. MBS is based on a proposed generic type model for stream flows and associated type checking rules earlier proposed in [6]. This model is open-ended and can in principle support any set of flow parameters. It also includes compatibility rules ensuring the correctness of binding attempts of flow endpoints, and conformance rules expressing conditions for substitutability [7]. An implementation of the flow type model and examples of its application are described in [8].

2.1 Flow Type Model

In the flow type model a flow type is specified by indicating the media type of the flow such as audio or video, its causality (source or sink), and a set of quality attributes such as rate and resolution. Furthermore, an attribute value is specified as a set of "atomic" values. This will in general enhance the chances of successful binding.

For example when a sink flow type specifies a set of different names on the video encoding attribute, it actually declares that it can accept flows where the video can have any of the indicated formats.

The following example of an H.261 video flow type features optional playback rates.

```
flow videoPhone {
  video V {encoding:H.261, rate:2..24)};
  audio A {encoding:PCM, rate:{8000,16000}};
  constraint (V & A) | A };
```

This specification states that a `VideoPhone` flow consists of two different element types labeled `V` and `A` respectively. Each element type includes a declaration of the generic media type such as `Video` or `Audio`, and a specific set of attributes, referred to as a media descriptor, specifying quality properties of the element type. The expression `(V & A) | A` is referred to as the structural constraints of the flow [8]. It specifies legal configurations of flow elements that may occur in an instance of the flow. The above structural constraint indicates that an instance of a flow can consist of video and audio elements, or audio elements only. Hence, we may think of the specification as modeling an adaptable flow endpoint.

A flow type specification is interpreted as a set of potential flow qualities (`QI`) and flow configurations (`SI`) that can be produced by a source flow endpoint or is acceptable to a sink flow endpoint. This interpretation of a flow type allows us to define a variety of flow type relationships based on set theory. The quality interpretation of a flow type is defined as the combination of the interpretation `I` of each of its element types (for further details see e.g. [6]).

Flow quality subtype relationship. A flow type `M` is a *subtype* of the flow type `N` if both the quality and structural interpretation of `M` is a subset of the corresponding interpretations of `N`. Suppose `A`, `B` and `C`, `D` are element types of flows `M` and `N`, respectively. We derive that `M = Flow[A;B]` is a *strict quality subtype* of `N = Flow[C;D]`, denoted $M <_q N$, if $I(A) \subseteq I(C)$ and $I(B) \subseteq I(D)$. On the other hand, a *relaxed quality subtype* may support fewer element types than the super type such that, for example, `M = Flow[A]` is a relaxed quality subtype of `N = Flow[B;C]`, denoted $M <\sim_q N$, when $I(A) \subseteq I(B)$.

Flow structure subtype relationship. A flow type `M` is a *structural subtype* of `N` denoted $M <_s N$, if $SI(M) \subseteq SI(N)$. This means that the subtype supports a sub-set of the configurations supported by the supertype. Suppose `b & c` is the structural constraint of `M` and `(a | b) & c` is the structural constraint of `N`. The $SI(M) = \{\{b,c\}\}$ and $SI(N) = \{\{a,c\},\{b,c\}\}$. Clearly we have $SI(M) \subseteq SI(N)$.

Flow quality compatibility relationship. Compatibility is determined by computing the set of common flow qualities and configurations supported by the two endpoints. Compatibility requires that this set is not empty. If the two endpoints can support more than one common flow quality and flow configuration, a flow property negotiation protocol may be employed to choose the actual flow quality to be used. Our approach for type checking binding attempts is to require that the source and the sink can at least support one common flow quality. This relationship we refer to as

quality compatibility. Informally, two flow types are strict quality compatible (denoted $<>_q$) if there exist a bijection between their respective sets of element types such that each pair in the bijection have non-empty set intersection of their respective interpretations. We may for example conclude that `Flow[A;B]` $<>_q$ `Flow[C;D]` if `I(A)` $\cap$ `I(C)` $\neq \emptyset$ and `I(B)` $\cap$ `I(D)` $\neq \emptyset$. Two flow types `M` and `N` are relaxed flow quality compatible (denoted $<\sim>_q$) if there exist a bijection between subsets of their respective element types such that each pair in the bijection is compatible. Thus if `I(A)` $\cap$ `I(C)` $\neq \emptyset$, `Flow[A;B]` $<\sim>_q$ `Flow[C;D]` even if `B` and `D` are incompatible, i.e. `I(B)` $\cap$ `I(D)` $= \emptyset$.

Flow structure compatibility relationship. Two flows of type `M` and `N` are structural compatible, denoted `M` $<>_s$ `N`, if their structural interpretations have non-empty intersection. This means that they support a least one common flow configuration. Suppose `(a | d) & c` is the structural constraint of `M` and `(a | b) & c` is the structural constraint of `N`. Then `SI(M) = {{a,c},{d,c}}` and `SI(N) = {{a,c},{b,c}}`. Clearly we have `SI(M)` $\cap$ `SI(N)` $\neq \emptyset$.

Different variants of the compatibility relationship where some variants are weaker than others, are the following:

i) fully strict compatible, if `M` $<>_q$ `N` and `M` $<_s$ `N`.
ii) partially strict compatible, if `M` $<>_q$ `N` and `M` $<>_s$ `N`.
iii) fully relaxed compatible, if `M` $<\sim>_q$ `N` and `M` $<_s$ `N`.
iv) partially relaxed compatible, if `M` $<\sim>_q$ `N` and `M` $<>_s$ `N`

For example, fully strict compatible is a stronger relationship than partially strict compatible in the sense that the former logically implies the latter. These different kinds of compatibility can be used by applications to state their requirement to the degree of *matching* that must be fulfilled when an application object interface is locally bound to a corresponding interface provided by the binding object.

2.2 Stream Type Model

In [6] a stream interface is simply specified as a collection of flows. In MBS we extend this specification by adding the notion of configuration constraint which is a specification of alternative combinations of flows that may be configured in a stream binding. For example, a stream interface modeling an access point to a video conference provider, may exploit this feature to express the alternative audio and video flow configurations and qualities that can be supported.

A stream configuration constraint is written as a structural constraint over flow labels. The set of alternative configurations of flows of a stream interface is referred to as its structural interpretation `SI`.

Stream compatibility relationship. Two stream interfaces **`S`** and **`T`** are *compatible*, denoted **`S`** `<>` **`T`**, if **`S`** and **`T`** have a common configuration of flows and there exist a bijection between the set of flows in these configurations of **`S`** and **`T`** such that for each pair of flows in the bijection, the pair is compatible. The kind of compatibility required we assume is specified by the application. Consider, for example, the following stream interface type of a video conference binding type

```
stream videoConfProducer {
  sink flow a {
     audio a1 {encoding:PCMA,
            rate:{8000,16000}};};
  sink flow v {
       v1: Video[encoding:H.261,
            rate:(2..24)};};
  constraint a|(a&v) }; //end stream
```

and the interface type `audioTalk` offered by a potential participant of the binding

```
stream audioTalk {
  source flow a {
     audio a1 {encoding:{PCMA,GSM},
         rate: 8000 }; }; //end flow
  constraint a }; //end stream
```

The absence of a flow configuration constraint means that all element types of the flow are required. The above interface types are compatible because they have a common configuration {{a}} where the label a refers the audio flow in both stream interfaces, and the two audio flows are compatible. By closer inspection it can be seen that the audio flow of `audioTalk` is fully strict compatible to the audio flow of `videoConfProducer`.

Stream conformance relationship. The MBS *conformance rules* express conditions for substitutability of stream interfaces. If the stream interface **T** *conforms* to the stream interface **S**, then **S** may be replaced transparently by **T**. A stream interface **T** conforms to a stream interface **S** if and only if SI(**S**) ⊆ SI(**T**) and for each stream configuration of **T** that is also a stream configuration of **S** there exists a bijection between the set of flows in the two configurations such that for each pair in the bijection the flow of **S** is a subtype of the flow of **T**.

The kind of flow subtype relationship required is subject to application policy. For example, when trading for binding types, the client of the trader must specify the required relationship as a parameter to the `look_up` method of the trader.

The following example illustrates a case where a video source `mpgSource` is upgraded to support additional playback rates, video encodings and audio, all encapsulated in the stream interface `mpg_mjpgSource` such that `mpg_mjpgSource` conforms to `mpgSource`.

```
stream mpgSource {
  source flow mpgFlow {
    video h {encoding:mpeg,rate:{20,25};}; };
  constraint mpgFlow  }; //end stream

stream mpg_mjpgSource {
  source flow  mpgFlow {
    video h {encoding:mpeg,rate:{20,25,30};};};
  source flow  mjpgFlow {
    video hj {encoding:mjpeg,rate:{20,25,30};};
    audio au {encoding:{PCMA,GSM},rate:8000};};
  constraint mpgFlow | mjpgFlow }; //end stream
```

The stream interface `mpg_mjpgSource` conforms to `mpgSource` because the structural interpretation of `mpgSource` (which is {{mpgFlow}}) is a subset of the

structural interpretation of mpg_mjpgSource (which is {{mpgFlow},{mjpgFlow}}), and the video flow labeled mpgFlow in the stream interface mpgSource is a (strict) flow subtype of the video flow labeled mpgFlow in the stream interface mpg_mjpgSource.

2.3 Binding Types

Our approach for specifying binding types is similar to the RIVUS template language [9], the main difference being that binding requirements are specified using the stream flow type model referred to above.

A binding type is defined as a 5-tuple $\langle T,P,M,\Delta,E\rangle$ where T denotes a set of role types, P a set of roles, M a set role matching requirements (one for each role), Δ a set of role causalities, and E a set of role cardinality requirements.

A *role type* τ is defined as a set of stream interface types, i.e. $\tau=\{T_1,\ldots,T_n\}$. A *role* defines binding object roles and is specified as a role name and a role type, r : τ. For example, a video conference binding type could define the roles talk and listen where the role talk could be of the role type {videoConfProducer}.

Role matching requirements apply to local bindings and specify for each role of the binding type the kind of type matching required when an interface is offered to fulfil the role. The kind of matching that can be specified is either a subtype or a compatibility relationship. Thus we model role matching requirements as a set of pairs $\{<r_1,m_1>,\ldots,<r_n,m_n>\}$ where r_i is a role name an m_i is a match kind.

A binding type may support several instances of each role. Binding behaviour defines *causalities* between instances of roles. We model role causality as a tuple <C,r_1,r_2,m> where C specifies a causality option, r_1 and r_2 are roles, and m specifies whether conversion between r_1 and r_2 is supported by the binding. Conversion is supported if m=conv, otherwise m=no_conv. Conversion is required if the roles r_1 and r_2 have incompatible roles types. Conversion may be required in those cases where alternative behaviour is specified at the corresponding interfaces (e.g. alternative encodings) and the behaviour of r_1 is allowed to be incompatible with the behaviour of r_2 as a result of local binding negotiation. For example, suppose the type of r_1 and r_2 is both video, and the type video specifies a flow with the two alternative encodings mpeg and mjpeg. If during local binding of r_1 to an application object interface a configuration with mpeg is negotiated, and during local binding of r_2 to another application object interface a configuration with mjpeg is negotiated, then a conversion between mpeg and mjpeg is needed. This might, for example, be realized as a suitable transcoder running within an active network.

As in [9] we define three options for how roles can be mapped together. Specifying <ONE-ONE,r_1,r_2,m> means that the binding object creates a one to one mapping between a single instance of role r_1 and a single instance of role r_2, while <ONE-MANY,r_1,r_2,m> means that the binding object creates a one to many mapping between a single instance of role r_1 and all instances of role r_2. Finally, <MANY-MANY,r_1,r_2,m> means that the binding object creates a mapping between all instances of role r_1 and r_2.

Role cardinality is a specification of the number of instances of a particular role the binding object can support [9]. It is modeled as a pair <r,i> where r is a role and i is a range where the minimum value states the number of instances of the role that is needed for the binding object to make sense, while the maximum states the largest number of instances of the role the binding object is willing to support. An example is the specification <talk,2..10>.

Example: The following is an example of a specification of a binding type supporting audio/video conferencing. We will later refer to this specification by the name AVConf. The specification defines two role types AVConfProducer={AVTalk} and AVConfConsumer={AVListen} where

```
stream AVTalk =  {
        sink flow  a {
             audio a1 {encoding:{PCMA,GSM};
                       rate:{8000,16000};};};
        sink flow v {
             video v1 {encoding:H.261;
                       rate:(2..24);};
        constraint a|(a&v) }; //end stream

 stream AVListen {
        source flow a {
             audio a1 {encoding:{PCMA,GSM};
                      rate:{8000,16000};};};
        source flow  v {
             video v1 {encoding:H.261;
        constraint a|(a&v) }; //end stream
```

Note that the causalities of the flows are specified as they are provided by the binding. This means that a source flow of an audio conference participant (a talker) must locally bind to a corresponding sink flow offered by the binding (in this case to the flows of AVTalk)

The binding roles of the specification are

```
gen : AVConfProducer
rcv : AVConfConsumer
```

The role matching requirements are

```
<gen,fully_strict_compatible>
<rcv,fully_strict_compatible>,
```

while the role causality requirement of the binding type offer is

```
<MANY-MANY,gen,rcv,conv>,
```

and the role cardinality requirements are <gen,2..20> and <rcv,2..20>.

3 Trading Binding Types

In this section we present a trading model for selecting binding types from binding type requirement specifications. The trading model is based on the notion conformance between binding types.

3.1 Binding Type Conformance

An application selects a binding type by stating *binding type requirements* to a trader that compares the requirements to *binding type offers*. A binding type requirement specification is with one exception only, identical to a definition of a binding type as outlined above, while a binding type offer is simply a binding type specification. Selection is based on a conformance relationship between binding type requirements and binding type offers. The result of the selection is the identification of a set of binding factories that are all capable of instantiating binding objects with properties conforming to those specified by the client. A conformance relationship for binding types must be based on conformance of stream interfaces and notions of role matching, role causality, and role cardinality satisfaction.

While a role matching requirement of a binding type offer is specified as a pair $<r,m>$, the role matching requirement of a binding type requirement is specified as a triple $<r,m,\sigma>$ where σ indicates whether a stricter role matching requirement than m is acceptable (σ=narrow) or not (σ=no_narrow). It is easy to show that strict subtype logically implies ($\Rightarrow$) all other match kinds, relaxed subtype logically implies fully and partially relaxed compatibility, full compatibility logically implies partial compatibility, and strict compatibility logically implies relaxed compatibility. Thus, $<r,$ full_compatibility$>$ satisfies $<r,$ relaxed_compatibility, narrow $>$.

Definition 1 (*role matching satisfaction*) A role matching requirement $<r_1, m_1>$ of a binding type offer, satisfies a role matching requirement $<r_2, m_2, \sigma>$ of a binding type requirement if and only if $m_1=m_2$, or σ = narrow and $m_1 \Rightarrow m_2$. ❑

Definition 2 (*role causality satisfaction*) A role causality $<C,r_1,r_2,m>$ is satisfied by a role causality $<C',s_1,s_2,n>$ if and only if $C = C'$, s_1 conforms to r_1, s_2 conforms to r_2, and if $m \neq n$, then n=conv. ❑

Definition 3 (*role type conformance*) A role type $\tau=\{T_1,\ldots,T_n\}$ conforms to a role type $\sigma=\{S_1,\ldots,S_n\}$ if and only if there exists a bijection β between τ and σ such that for all $(T_i,S_j) \in \beta$, T_i conforms to S_j. ❑

Definition 4 *(binding type conformance)* A binding type offer $\mathbf{B_1} = \langle T_1,P_1,M_1,\Delta_1,E_1\rangle$ conforms to a binding type requirement $\mathbf{B_2} = \langle T_2,P_2,M_2,\Delta_2,E_2\rangle$ if and only if there exists a bijection β between the sets of role causalities Δ_1 and Δ_2 such that for all $(\delta,\varepsilon) \in \beta$ with $\delta = <C_1,r_1,s_1,m_1>$ and $\varepsilon = <C_2,r_2,s_2,m_2>$, δ satisfies ε, and the role cardinality requirements $<r_1,i_1>$ and $<s_1,j_1>$ in E_1 satisfies the corresponding role cardinality requirements $<r_2,i_2>$ and $<s_2,j_2>$ in E_2 such that $i_2 \subseteq i_1$ and $j_2 \subseteq j_1$, and the role matching requirements $<r_1,m_1>$ and $<s_1,n_1>$ in M_1 satisfies the corresponding role matching requirements $<r_2,m_2,\sigma_2>$ and $<s_2,n_2,\mu_2>$ in M_2. ❑

The kind of binding type conformance described above may be characterized as *structural* since conformance largely is determined by comparing the syntactic structure of binding type specifications (although for flows, attribute values are also compared). The analogy to this approach in the world of operational interfaces is signature matching [20]. A drawback of pure signature matching is that we might get false positives since semantics is not taken into account. This can be compensated in

the case of stream bindings by "standardizing" the names of generic element types and their attributes. This is the approach taken, for example, by the Internet Engineering Task Force on a real time transport protocol [23] in which profiles standardize sets of attributes for certain media types and specific payload types such as audio and video encodings, are assigned unique names by an appropriate Internet authority.

3.2 Example

The following is a simple specification of a binding type requirement for an audio conference binding. We will later refer to this binding type by the name `audioConf`. We define two role types `audioConfProducer={audioTalk}` and `audioConfConsumer= {audioListen}` where

```
stream  audioTalk {
       sink flow a {
         audio a1 {encoding:{PCMA};rate: 8000};};
        constraint a }; //end stream

stream audioListen {
       source flow a {
         audio a1 {encoding:{PCMA};rate: 8000};};
       constraint a }; //end stream
```

The corresponding binding roles are

```
talk : audioConfProducer
listen : audioConfConsumer
```

The role matching requirements are

```
    <talk,partially_relaxed,narrow>
  <listen,partially_relaxed,narrow>,
```

while the role causality required by the audio conference application is

```
  <MANY-MANY,talk,listen,no_conv>,
```

and the role cardinality requirements are `<talk,2..6>` and `<listen,2..6>`.

Taken as a binding type offer, it is easy to show that the binding type specification referred to as `AVConf` in section 2.3, satisfies the above binding type requirement `AudioConf` according to definition 4. We basically need to show that the role causality requirements of the binding type offer, `<MANY-MANY,gen,rcv,conv>`, satisfies the role causality requirements of the binding type requirement, `<MANY-MANY,talk,listen,conv>`, and that role cardinality and role matching requirements of corresponding roles are satisfied.

3.3 Architecture of Trading Binding Types

Architecturally a binding trader can be considered as an object service of middleware platforms. The basic idea is that developers of binding factories register their implementations at the binding trader. The information that must be registered includes a specification of the binding type offer together with information on how to activate the corresponding binding factory.

An application may interrogate the trader to inquire about binding factories that may create bindings satisfying the requirements of the application. For example, a multimedia database may automatically generate the specification of the binding type requirement based on meta-data describing the result of the query and QoS requirements of the corresponding database clients [18]. Parameters of the inquire operation to the trader must include a specification of a binding type requirement.

The result of the inquire operation will typically be a reference to (the service interface of) a binding factory that is capable of instantiating bindings with properties conforming to the specified requirements. The application may now call the `create` method of the binding factory. Parameters of the method call include specifications of the interfaces to be bound, and the roles in which the interfaces are offered by the application. Upon completing the execution of the method, the binding factory returns a reference to the control interface of the binding.

3.4 Trader Implementation Issues

The main challenge of our approach is its computational complexity. Although a full implementation of a binding trader has not been made yet, some earlier results might indicate its complexity. In [8] is presented an algorithm that determines compatibility and subtype relationships between flows. The algorithm has polynomial complexity in the number of element types of a flow. On a Sun Ultra 2 workstation running Solaris 2.5.1 the execution time is demonstrated to be in the order of 1 ms to determine the presence of a flow type relationship for flow types with 5 element types or less, while flow types with 30 element types require about 50 ms execution time. It is expected, though, that flow types with more than 5 element types will not be very common.

Determining binding type conformance means matching $r \times s$ binding roles for "correspondence" where r and s are the number of different role types in each binding type. For each pair of role types to be compared, $m \times n$ flow types need to be compared for some flow type relationship where m and n are the number of flows in each stream interface. In an attempt to estimate the required execution time on the Sun Ultra for determining the presence of binding type conformance, suppose each binding type is composed of 4 different role types, and each role type is composed of one stream interface having 4 flow types. Then a rough estimate of the required execution time is in the order of a few hundreds of ms.

The performance of the trader task of finding a first conforming binding type offer now largely depends on the efficiency with which binding type offers likely to conform to the requirements can be located in the trader's database. This will narrow down the set of candidates that will be considered in detail such that all members of the set have a similar structure as the binding type requirement. In our future work we will investigate whether this can be efficiently achieved through proper indexing based on a classification of the most discriminating properties of stream bindings.

An alternative to comparing syntactic structure as part of the trader `look_up` operation, is *declared conformance*. This is the approach taken by the ODP/CORBA

Trader [22] in which service type offers and corresponding interface types are (manually) registered in a service type repository as unique *names*. The registration also encompasses information about which already registered service types the new service type conforms to. One advantage of this approach is the obvious reduced computational complexity. However, disadvantages are that only pre-registered conformance relationships can be detected by the trader, and that importers can only refer to registered service type offers in the trader `look_up` method, i.e. all applications have to know in advance the kind of stream bindings they potentially may need.

In our future work we will therefore look for ways to combine the above two approaches to trader implementations.

4 Negotiating Local Binding Behaviour

After having traded a binding type and corresponding binding factory (BF) as outlined in section 3.2, the BF type checks each application interface and the corresponding role of the binding type. The type checking is performed by computing the common flow properties of each pair of corresponding flows in the two interfaces. The result is a new interface specification representing the common behaviour supported by both interfaces [8]. We refer to this interface as the *intersection interface*.

In those cases where the intersection interface specifies alternative behaviours, it becomes necessary to choose the (initial) interface behaviour to be used for each local binding. The intersection interface may specify alternative behaviours with respect to stream interface configurations (see section 2.2), for each flow alternative flow configurations (see section 2.1), and for each possible flow configuration alternative quality behaviour (c.f. set valued attributes described in section 2.1).

If the selected binding type does not support conversion between causally related interfaces, the negotiations at one local binding will be constrained by the alternative behaviours that are possible at causally connected interfaces. Otherwise, it is a matter of BF policy how the negotiation at causally related interfaces are mutually constrained.

In the following we focus on the issue of negotiating individual flow quality behaviour.

4.1 Policy Specification

In order to support *automatic* negotiation of flow quality behaviour, we extend the flow type model of [6] with *policy specifications* that can be associated to each flow of a stream interface. A policy specification effectively specifies an order on the quality interpretation of a flow type. This ordering can be taken to represent user priorities with respect to desirable properties of the flow. The ordering is used as a basis for negotiation of the (initial) flow quality behaviour to be used for the local binding.

The alternative quality behaviours of a flow configuration is given by the set of quality attribute values associated to the flow elements of the flow. Given a set of attributes A_1, .., A_n such that the value of A_i is a set of values $\{v_1, .., v_m\}$. The Cartesian product $A_1 \times \ldots \times A_n$ gives the total set of possible behaviours of the flow with respect to the properties A_1, ... , A_n as a set of n-tuples. This we refer to as the interpretation of A_1, .., A_n . A policy specification specifies a priority order on this set of n-tuples.

For example, suppose after type checking two interfaces to be bound, the intersection includes a flow having the following alternative behaviours with respect to the attributes `depth`, `framerate`, and `size`:

```
depth {24,16,8};
framerate {30,25,20,15};
size {800x600,640x480,320x200};
```

A language for specifying policies of flow quality should allow the specification of arbitrary orderings of the Cartesian product of `depth`, `framerate` and `size`. On the other hand, such a language should not force a user to enumerate explicitly the ordering of all possible combinations of attribute values. The Cartesian product of the above attributes, for example, will give a total of 36 possible unordered (or arbitrary ordered) combinations.

Hence such a language should allow the users to specify orderings in a simple, yet expressive way. Our policy specification language is an attempt to achieve this. The language specifies value ranges for attributes assuming that attribute domains are totally ordered (either implicitly or explicitly specified). If no policy is specified for a flow, a default ordering is assumed derived from the ordering of the attribute domains. Otherwise the default ordering can be overridden with fine granularity, by specifying short attribute ranges, or with coarser granularity by specifying larger attribute ranges. In the extreme case a user could explicitly specify the complete ordering by consistently applying value ranges of length 1.

In our first version of a flow policy language, a policy specification is given by a list of selection statements. Each selection statement defines an ordered partition of the interpretation of the attributes. The total ordering is then achieved by concatenating each partition in the order given by the list of selection statements. A typical policy specification will have a structure as shown below.

```
A11r11, A12r12,..., A1jr1j;
A21r21, A22r12,..., A2jr2j;
...
Ai1ri1, Ai2ri2,..., Aijrij;
```

A_{ij} denotes an attribute name, while r_{ij} denotes an attribute value range. A range is written as `(v,w)` where `v` and `w` are atomic attribute values. The position of the same attribute name might vary from selection element to selection element in the list.

The ordering implied by each selection statement is obtained by looping through a set of nested loops, where the left most attribute given corresponds to the outer most loop, and the right most corresponds to the inner most loop. Thus, all attribute values

of $A_{i,j+1}$ will be used before starting to using "lower" values of $A_{i,j}$. The range `(v,w)` for a given attribute, specifies the range of values to be used for this attribute in this selection. For example, the selection statement

```
Depth(24, 16),size(800x600,800x600), framerate(25,20)
```

specifies that we first want to select qualities with depth between 24 and 16, size of 800x600 and framerate between 25 and 20. Since `framerate` is listed rightmost, the selector will first try out combination with lower frame rates, before starting to reduce the depth.

Example: Below we show an example of a policy specification and the corresponding ordering of the interpretation of the flow attributes given above.

```
1: size(800x600, 640x480), depth(24, 16),
                             framerate(30, 20);
2: size(640x480, 320x200), framerate(30, 15),
                             depth(24, 16);
3: depth(16, 8), framerate([20, 15),
                             size(320x200, 320x200);
```

Gives,

```
     depth       framerate   size
1:   24          30          800x600
     24          25          800x600
     24          20          800x600
     16          30          800x600
     16          25          800x600
     16          20          800x600
     24          30          640x480
     24          25          640x480
     24          20          640x480
     16          30          640x480
     16          25          640x480
     16          20          640x480
2:   24          30          640x480
     ...
3:
     ...
     8           15          320x200
```

4.2 Negotiation

In this section we consider policy specifications as a foundation for QoS. In general, a variety of possible QoS negotiation protocols can be considered. In the MULTE-ORB architecture, QoS negotiation protocols are embedded within binding factories. Thus different BFs might support different negotiation protocols. In the following we discuss policy specification in the context of a simple, hypothetical negotiation

protocol. For the sake of the discussion, we do not pay any attention to the efficiency of the protocol, but rather approach the issues in a principled manner. Possible optimizations are addressed in section 4.3 below.

Our approach to QoS negotiation is to take the ordering of alternative flow QoS behaviours implied by a policy specification as the users priorities in the negotiation. In the example above, the user gives priority to 24 bits pr pixel, 30 frames pr second, and a frame size of 800x600 pixels. The main principle of the QoS negotiation protocol is first to suggest a QoS level corresponding to the users first priority of QoS. If this can not be achieved, then the second priority of QoS is tried, and so on.

This simple protocol is sufficient for application scenarios where a single user retrieves a stream form a server, e.g. a video on demand server. In this case alternative QoS parameter configurations will be tried. Configurations might be rejected due to lack of resources. Alternatives are tried until either one configuration achieves enough resources to create the binding, or all the configurations fails. In the latter case, the binding attempt fails.

In other situations, when there are multiple receivers, there has to be a negotiation in order to agree on a common QoS parameter configuration (assuming no conversion such as scaling or transcoding of flows is supported). Below we outline such a negotiation protocol.

The negotiation protocol aims at finding a QoS parameter configuration that satisfies all the participants of the binding. In general, the various binding participants will specify different policies of flow QoS behaviour. Hence, in this case the goal of the protocol should be to find a QoS parameter configuration that is a "best fit" according to some metric. Again one might consider many different metrics for balancing the QoS parameter configuration between conflicting requirements. In the following we describe one possible metric that could be used as a basis for the negotiation.

Given that the negotiation is to be over a set of QoS attributes A_1, .., A_n , the starting point for the negotiation is the interpretation of A_1, .., A_n. Additionally, each participant specifies its own ordering of this interpretation as a flow QoS policy. Each QoS parameter configuration has a distance from the top of the list. A given QoS parameter's aggregated distance, is the sum of its distances from the top of the priority list over all participants lists. One possible metric to determine the "best fit" QoS parameter configuration, is to choose the configuration for which the aggregated distance to the top of the priority list is the shortest for all participants. If some configurations have the same aggregated distance, then the aggregated relative distance from the top is computed for these configurations. Relative distance is computed as a configuration's distance from the top in percents. This relative distance is used to find the inter distance between the configurations. The configuration with the shortest inter distance is considered the best. This corresponds to those configurations which have a relative height closest to each other. If there still are more than one candidate, one of them can be selected at random.

Below we give an example of the negotiation protocol for two receivers. Suppose user `A` and user `B` have the following flow QoS policies:

```
       A                          B
depth rate  size          depth rate  size
24    30    800x600       16    30    640x480
24    25    800x600       16    25    640x480
16    30    800x600       16    20    640x480
16    25    800x600       16    30    320x200
16    30    640x480       16    25    320x200
16    25    640x480       16    20    320x200
...
```

From the above priorities and a "best fit" metric as described above, we see that the best configuration is (`16,30,640x480`), having an aggregated height of 6 while the second best is (`16,25,640x480`), having an aggregated height of 8. Thus, a binding supporting the quality (`16,30,640x480`) will be tried created first. If the BF is unable to create this binding, due to lack of available resources, a binding supporting (`16,25,640x480`) will then be tried created. This will continue until either the BF is able to create the binding, or it fails to create any binding due to lack of resources.

4.3 Design Issues of Negotiation Protocol

The above approach to a QoS negotiation protocol for a single flow did not consider efficiency. The protocol as it stands might result in several rounds of message exchanges in order to find a QoS parameter configuration. The reason for this is the way resources are handled. The protocol finds a possible candidate configuration, and then tries to allocate resources to support the binding for each of the binding parties. If there is insufficient resources at any of the participants, the binding attempt for this configuration will fail, and a new attempt has to be made.

The scalability of the negotiation protocol can be measured by the complexity of the algorithms and the number of messages which have to be sent. The number of messages exchanged will depend on how fast a "best fit" can be found. For this reason, it will be important to develop a negotiation protocol which takes the current resource situation into account when creating priority lists. Thus, an optimization of the above protocol would be to have participants reserve sufficient resources before they announce their policies. This might lead to participants having to remove some of their configurations, due to lack of resources, before they start the negotiation protocol. We may refer to the resulting policy at a *resource adapted policy* (RAP). The result of this requirement will be that once the participants find a configuration, they will have enough resources to create the binding.

With this new approach, a two-party negotiation for a single flow requires a two-way handshake. One participant announces its RAP, and the other participant subsequently intersects its RAP with the received RAP and communicates back its selected configuration provided the intersection was non-empty. For a multi-party negotiation a three-way handshake is required. First the initiator has to ask participants for their RAPs. Once the result of all participants have been collected, the initiator will select the configuration that is the "best fit" for all of the participants, if such a

configuration can be found. Then it will inform the participants of the selected configuration, or it will inform that it failed to create the binding. Thus, using RAP specifications, the protocol will scale linearly to the number of participants, with regard to message exchanges. A challenge for further work will be to develop an efficient algorithm to calculate RAP specifications. Since the calculation of RAPs is done at each participant's node, scalability will not depend on this algorithm.

The scalability of the protocol will also depend on the efficiency of the "best fit" algorithm. Thus, it is important to design an algorithm that scales well with the number of participants. We are currently in the process of designing such an algorithm. The results of this work will be reported elsewhere.

5 Related Work

Stream interfaces have been adopted in the work on Open Distributed Processing [11], TINA-DPE [12] and OMG [13]. Compatibility and subtyping rules for stream interfaces, however, have been deemed outside the scope of the RM-ODP standard [11]. In the work of TINA-C, the need for a compatibility relationship for stream flows that is more relaxed than equivalence, is recognized, but no definition is offered [12]. A recent proposal for audio/video support in CORBA [14], also introduces the notion of flow end-point compatibility. QoS parameters beyond media encoding are not considered.

Microsoft's ActiveMovie framework [10] also includes the notion of "compatibility negotiation" between "pins" (connection points that carry flows between different processing objects). The subject of this negotiation is data compatibility rather than QoS. There is no support for distribution.

QML is a recent proposal for a QoS specification language [5]. The semantics of QML is similar to our stream and flow type model, and from our judgement should be capable of specifying quality properties of stream interfaces. Its applicability has been demonstrated for operational interfaces only. This work does not consider automatic support of QoS negotiation from QML specifications.

Other work that considers QoS specifications and/or negotiations includes [16], [2], [4], [19] and [21]. However, the focus of our work is different. These works do not provide anything corresponding to a type model of streams and bindings, including type relationships such as subtype, compatibility and conformance, and the derivation of automatic systems support such as QoS negotiation from high-level interface specifications.

6 Conclusions and Future Work

In this paper we introduced a trading model for selecting appropriate explicit stream bindings based on statements of binding type requirements provided by the application. We showed how an earlier proposed type model for stream flows can be

extended to support binding type selection based on a notion of binding type conformance. In this scheme a set of binding factories are located based on a specification of the required properties of the binding. The located binding factories are all capable of instantiating binding objects with properties conforming to those specified by the client.

Furthermore, in those cases where binding objects specify alternative behaviour at its supported interfaces, we also introduced the notion of *policy specification* supporting automatic negotiation to choose the actual interface behaviour to be used at each interface. Finally, we demonstrated the usefulness of a trading facility and policy specifications as indicated above, through a number of examples.

In our future work we will address some of the limitations of the current model. In particular this includes automatic negotiation of stream interface configurations and flow configurations. Furthermore, the integration of resource management into the binding framework is a matter of high priority. In our current work we assume the availability of a resource manager that only supports simple reservation requests that can either be accepted or rejected depending on the availability of resources. This might force binding factories to make repeated reservation requests corresponding to different QoS parameter configurations. When one request is rejected, the binding factory will have to try again with a different QoS requirement. In future work we will give binding factories the possibility to examine the resource situation through the resource managers. Knowledge of available resources can be used by binding factories to reason about which QoS parameter configurations can currently be supported before making reservation requests.

References

1. Blair, G. S. et al. (1997) Adaptive Middleware for Mobile Multimedia Applications. *Network and Operating System Support for Digital Audio and Video (NOSSDAV '97),* St Louis, USA, 1997.
2. Campbell, T. (1996) A Quality of Service Architecture. *PhD Thesis*, Lancaster University.
3. Coulson, G., Blair G. S., Stefani, J. B., , Horn, F., Hazard, L. (1992) Supporting the Real-Time Requirements of Continuous Media in Open Distributed Processing. *Technical Report MPG-92-35*, Lancaster University.
4. Dini, P., Hafid, A. (1997) Towards Automatic Trading of QoS Parameters in Multimedia Distributed Applications, *In proceedings of IEEE/IFIP ICODP/ICDP Conference*, Toronto, Canada, 166 – 179.
5. Frølund,, S., Koistinen, J (1998) Quality-of-Service Specification in Distributed Object Systems, *Distributed Systems Engineering Journal*, Vol.5, No.4
6. Eliassen, F., Nicol, J. R. (1996) Supporting Interoperation of Continuous Media Objects. *Theory and Practice of Object Systems: special issue on Distributed Object Management* (ed. G. Mitchell), Vol.2, No.2, Wiley, 1996, 95-117.

7. Eliassen, F. (1997) A Conformance Relationship for Stream Interfaces, 2nd Int'l Conf on Formal Methods in Open Object-based Distributed Systems (FMOODS'97), Canterbury July 21-23, Chapman & Hall.
8. Eliassen, F., Mehus, S. (1998) Type Checking Stream Flow Endpoints. *Middleware'98*, The Lake District, England, 16-18 Sept, Chapman & Hall, 305 - 322.
9. Lindsey, D., Linington, P.F. (1995) RIVUS: A Stream Template Language for Capturing Multimedia Requirements*, Lecture Notes in Computer Science (LNCS 1052)*, Springer Verlag, pp. 259 – 277.
10. Microsoft (1996), Microsoft ActiveMovie: Software Development Kit, Beta Release, June 1996.
11. ITU-T X.901 | ISO/IEC 10746-1 (1995) ODP Reference Model Part 1: Overview. *Draft International Standard.*
12. TINA-C (1995) TINA Object-Definition Language, Version 1.3. *TINA-C Deliverable*.
13. Object Management Group (1996) Control and Management of A/V Streams Request for Proposal. *OMG Document*: telecom/96-08-01.
14. IONA Technologies, Plc, Lucent Technologies, Inc, Siemens-Nixdorf, AG (1997) Control and Management of A/V Streams Request for Proposal. OMG RFP Submission, *OMG Document*: telecom/97-05-07.
15. Kristensen, T., Plagemann, T. (1999) Extending the Object Request Broker COOL with Flexible QoS Support, Technical Report UniK – Center for Technology, University of Oslo.
16. Nahrstedt, K., Smith, J. M. (1995) The QoS Broker, *IEEE Multimedia*, 2(1), pp. 53-67.
17. Plagemann, T. (1994), A Framework for Dynamic Protocol Configuration", *Dissertation at Swiss Federal Institute of Technology*, Computer Engineering and Networks Laboratory, Zurich, Switzerland, Sept. 1994.
18. Plagemann, T., Eliassen, F., Goebel, V., Kristensen, T., Rafaelsen, H. O. (1999), Adaptive QoS Aware Binding of Persistent Objects, in IEEE *Proceedings of International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland.
19. Vogt C., Wolf, L. C., Herrtwitch, R. G., Wittig, H. (1998), HeiRAT - Quality of Service management for distributed multimedia systems, *Multimedia systems*, 6(3), ACM/Springer, pp. 152-166.
20. Zaremski, A.M., Wing, J. M. (1995), Signature matching: a tool for using software libraries, ACM Trans. Softw. Eng. Methodol., Vol.4, No.2, pp. 146-170.
21. Zinky, A., Bakken, D.E., Schantz, R.D. (1997), Architectural Support for Quality-of-Service for CORBA Objects, *Theory and Practice of Object Systems,* Vol.3, No.1, Wiley.
22. ISO/IEC 13235-1 (1998) Information technology - Open Distributed Processing - Trading function: Specification.
23. Schulzrinne, H., Casner, R., Frederick, R., Jacobsen, V. (1996), RTP: A transport protocol for real-time applications, *IETF*, rfc 1889.