RIVUS: A Template Language for Modelling Multimedia Streams

Donna Lindsey
(*email: D.L.Lindsey@ukc.ac.uk*)
University of Kent

Abstract

Streams have been included in the Reference Model for Open Distributed Processing (RM-ODP), in response to advances in distributed multimedia computing. They describe the complex interactions that can occur in such systems. The simple client-server model of interaction is insufficient to properly represent streams and therefore an explicit binding model has been introduced. Details of the communication are encapsulated in a binding object whose behaviour is defined in a template.

This report presents the ODP view of streams and then introduces the template language RIVUS. The language can be used to specify binding templates and thus allow classes of communication behaviour to be described. A syntax for the language is provided.

1 Introduction

In multimedia systems various forms of information, such as video, text and audio, are combined to produce a wide range of applications, for example video booths and tourist information packages. Powerful new applications are being developed to support video conferencing, distance learning and office information systems [11] by combining distributed system and multimedia technology.

With the rise of multimedia applications, there has come a need, in distributed systems, to support the transmission of new kinds of data. This data, for example audio and video, is time-based [3]; it is produced and consumed at continuous and steady rate. With such data comes issues of synchronization, for example lip synch, quality of service, for example what are acceptable loss rates on a video channel and configuration, for example how can we model the flow of information in an audio system?

An application programmer needs to be able to describe, with appropriate quality of service tags, the information flow in a system such as Fig.1. Management of the information also needs to be considered to ensure that the quality of service demands are met.

The traditional client and server model of interaction is insufficient for capturing the full complexity of the transmission of time-based media and so streams have been suggested [4, 10] as an effective way of modelling complex multimedia behaviour.

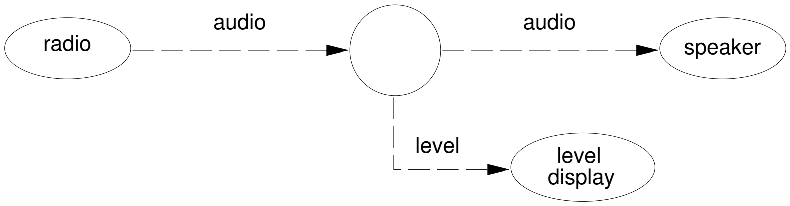


Figure 1: An audio system

This paper outlines the features that have been incorporated in the Basic Reference Model for Open Distributed Processing to support streams. Section 3 outlines a template language, RIVUS, which can be used to specify streams and presents the language syntax.

2 The ODP Basic Reference Model

The Reference Model has been developed by the ISO Open Distributed Processing standardisation process to address the problems of maintaining consistency in distributed systems [8].

The Basic Reference Model aims to simplify the development of distributed systems by allowing a system to be viewed and specified from five differing viewpoints. Each viewpoint focuses its attention on a particular set of design issues and can be applied to a system as a whole or components within a system. The viewpoints [6] are as follows:-

- Enterprise viewpoint considers the purpose, scope and management policies of the distributed system.
- Information viewpoint focuses on information modelling and processing in the system.
- Computational viewpoint considers the system in terms of objects and groups of objects and how they can be distributed.
- Engineering viewpoint focuses on the functions that are needed to support the distribution of a system.
- Technology viewpoint considers the technology used in a distributed system.

This paper will restrict its consideration of streams to the computational viewpoint.

2.1 Computational View of Streams

2.1.1 The binding model

The explicit binding model is used to describe streams, as illustrated in Fig.2. The binding is represented by a visible binding object which encapsulates the information flows between stream interfaces. The binding object allows configuration and quality of service details to be made explicit. It also provides a point at which flow management can be considered.

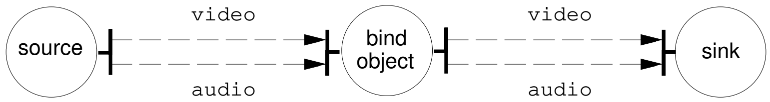


Figure 2: A stream binding

A stream interface is composed of a finite series of flows. A single flow is an abstraction of the sequence of interactions that can occur between a source and sink object. A flow has a type, for example video, audio or sensor data, and direction, for example source or sink. A stream interface can be described in terms of flows or in terms of pre-defined interfaces to reflect more complex communication activities, for example a co-operative working situation. A stream interface has a type which reflects the kind of interaction that the interface supports for example *radio* or *video conference*.

A binding object is instantiated from a stream binding object template by a create action. The template contains a number of stream interface signatures which define the flows and interfaces that comprise a particular stream interface. The signatures are represented by interface roles after an object is successfully instantiated. A stream interface is offered into a binding at a particular role thus allowing the binding to be considered at a single point. The binding action which links the stream binding object and the computational object supporting the stream interface is considered to be primitive.

It is possible for the binding object to support a set of control interfaces which allow for the management of the binding. Operations are defined at these interfaces which are used to change the membership of the binding and also to provide management functions. Such operations can be used to monitor quality of service levels and react to any deterioration of the service or to collect data for charging purposes.

2.1.2 Stream types and bindings

The binding object acts as an intermediary between bound objects. A binding object will therefore only allow a binding to exist between interfaces if a meaningful communication is possible. A binding is possible where two interfaces are complementary, (Fig.3).

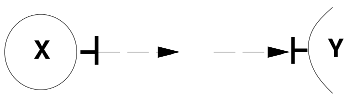


Figure 3: Complementary interfaces

The properties of complementary interfaces are that there exists some type relationship, either subtype or partial match, and that the flow direction in one interface is reversed by the flow direction in the other, ensuring the flow direction over the whole binding is the same. The binding object interrogates a type repository to determine the relationships between interface types.

Subtypes

A subtype relationship exists between and interface X and a role Y if, for all flows which have the same names:

- the flow type in X is a subtype of the flow type in Y, if X is producing the flow.
- the flow type in Y is a subtype of the flow type in X, if X is consuming the flow.

A further assumption is made that X is a subtype of Y, if X can be substituted [1, 9] for Y without loss of functionality; that is X offers either the same or more capabilities than Y.

For example, if an interface has type *tv* and a role has type *radio* defined as follows:

radio = < a1 : (audio, SOURCE) >
tv = < a1 : (audio, SINK), a2: (video, SINK) >

then the *tv* interface can be considered a subtype of the *radio* role. They have a complementary audio flow and the *tv* interface can be substituted for the *radio* role without any loss of functionality. It is the binding object's responsibilty to handle the extra capabilities provided by the *tv interface*.

Partial Match

An interface X is said to partially match a role Y if:

- at least one flow in X corresponds to a flow in Y.
- where a correspondence exists, a subtype relationship exists.

The partial match relationship allows interfaces to be accepted when the binding object only needs a subset of the role's defined functionality to create a meaningful binding, for example using a *telephone* interface in a video conference binding. Again it is the responsibility of the binding object to handle the flows for which no corresponding flow exists.

3 RIVUS - A Stream Template Language

A binding object allows quality of service, management and configuration issues to be made explicit. The application programmer is thus given more control over the communication activity that he is trying to model. However the programmer needs some method for describing these details such that a binding object can be created with the required behaviour.

The stream template language, RIVUS, allows a programmer to do this. The programmer creates a template with the behaviour of the binding object described in the language constructs and then allows the supporting infrastructure to instantiate an object from this template.

The language can be used to state:

- the resources that the binding object requires.

- the functionality required for named roles.
- rules for binding complementary interfaces.
- any necessary type conversions.

3.1 Binding Templates

A binding template is a computational specification of a stream binding object. It is a formal definition of the complex behaviour of the binding object. The template of a video conference binding may be structured as follows:

video_conference = *[*

ROLETYPES
...
BINDROLES
...
BINDREQ
...
BINDBEHAVE
...
BINDCARD
...
]

Each of the template components will be briefly described in the following sections.

3.2 Binding Types

The template includes type definitions of the roles that binding object supports in the binding. The definition is given as a list of uniquely named components and their types.

A role type can be defined as a list of flows. Each flow description contains the data type and the direction of the flow of information.

$$
\begin{aligned}
\mathit{videoconf\_cons} &= \langle\, a1 : (audio, SINK), a2 : (video, SINK) \,\rangle \\
\mathit{videoconf\_prod} &= \langle\, a1 : (audio, SOURCE), a2 : (video, SOURCE) \,\rangle
\end{aligned}
$$

Or in terms of already predefined stream types:

$$\mathit{videoconf\_party} = \langle\, b1 : \mathit{videoconf\_prod}, b2 : \mathit{videoconf\_cons} \,\rangle$$

This a grouping can be explicitly created:

videoconf_party = < b1 : < a1 : (...), a2 : (...) >, b2: < a1 : (...), a2 : (...)> >

The grouping of flows in a stream interface can also be removed:

tv = < COMPONENTS OF videoconf_prod >

Any combination of the above can be used to define a stream interface type.

3.3 Binding Roles

Roles define the type and behaviour of a stream.

if_α *: videoconf_prod;*
if_β *: videoconf_cons;*

The type of if_α and if_β signifies that the stream is a video conference where communication is in a single direction.

3.4 Binding Requirements

The role type defines the optimum functionality expected from an interface. The template language defines two constraints which allow the programmer to make the functionality requirements explicit.

If no constraint is applied to a role then the assumption is that the offered interface must match the role functionality exactly.

ISSUBTYPE if_α

> The subtype constraint states that only subtype interfaces are acceptable in the named role.
>
> The assumption is that the constraint also applies at the stream component level. The role, if_α, with a component of type *audio_mono* will accept a subtype interface with a component of type *audio_stereo*.

ISMATCH if_β

The partial match rule states that an interface is acceptable if it matches at least one component in a named role.

It is possible that an interface that partially matches a role is only acceptable if it has a certain functionality. In most cases it makes little sense to participate in a video conference with only video capabilities. An optional argument to the constraint allows the programmer to list the components that are necessary for a meaningful binding.

ISMATCH if_β *a1*

ISSUBTYPE if_α*.a1*

It is assumed that the constraint applied to a role also applies to its components. If if_α has type *videoconf_prod* and its audio component, *a1*, has stereo properties then the constraint

ISMATCH if_α

allows an interface with mono audio properties into the binding. Constraints can be applied at the flow level to maintain quality of service at the lower levels.

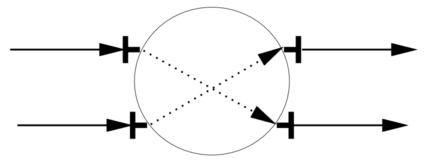
ISSUBTYPE $if_\alpha.a1$

3.5 Binding Behaviour

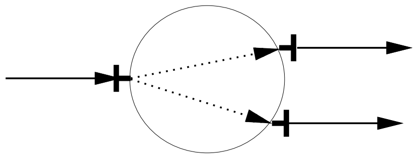
The binding object needs to know which roles, defined in the binding complement each other. The programmer has three options for defining the binding object's mapping behaviour. These behaviour definitions ensure that a variety of communications are catered for.

SINGLECAST if_α if_β

The binding object creates a one to one mapping between a single instance of role if_α and a single instance of role if_β.



Such a binding might model a telephone call. The binding object may support several one to one bindings for example, to model a telephone exchange.



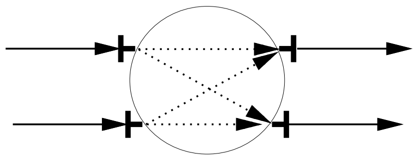
BROADCAST if_α if_β

The binding object creates a one to many mapping between a single instance of if_α and all instances of if_β.

This object behaviour can be used to model a video lecture.

MULTICAST if_α if_β

The binding object creates a mapping between all instances of if_α and if_β. This behaviour supports streams that model multi-party conferences.



3.6 Type Conversion

Producer and consumer objects can be combined where the type of the information produced is different from the type of the information consumed. The binding object needs to convert, transparently, the information supplied by the producer to a form that the consumer understands. The template language is used to state the type conversions that the binding object supports.

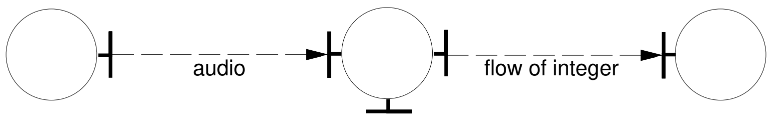


Figure 4: Computational Type Conversion

In Fig. 4, a stream binding models the flow of information between a microphone, producing audio, and a graphic equalizer display which expects a series of integers relating to the level of bass, treble etc in the audio flow. In this case the roles, if_α and if_β have the types

microphone	=	*< a1 : (audio, SOURCE) >*
equalizer	=	*< b1 : (sensor, SINK) >*

The programmer states that type conversion is necessary between certain flows in the complementary roles if_α and if_β.

CONVERT $if_\alpha.a1$ *TO* $if_\beta.b1$

3.7 Binding Cardinality

Binding cardinality is a statement of the number of a particular interface the binding object wishes to support. If no explicit statement is made then the cardinality defaults to one.

REQUIRE min .. max OF if_α

The cardinality is given as a range. The minimum value states the number of a particular interface that the binding object needs for the stream binding to make sense. If the underlying system is unwilling to support this number then the binding object can not be instantiated. The maximum states the largest number of interfaces that the binding object is willing to support.

4 RIVUS Syntax

The syntax of the template language is defined using Backus-Naur Form. The following notation is used:

- All names in **bold** are terminals and all others are non-terminals.
- The symbol | is used to group rules with the same left hand side non-terminal and has the meaning 'or'.
- ϵ stands for the empty string.
- The following types are used throughout the language:

integer	::=	integer digit | digit
digit	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
idname	::=	string
string	::=	string followingchar | letter
followingchar	::=	letter | integer | **_**

Template Structure

template	::=	**templatename** **=** **[** templatebody **]**
templatebody	::=	**ROLETYPES** roletypes **BINDROLES** roles **BINDREQ** requirements **BINDBEHAVE** behaviour **BINDCARD** cardinality

Role Types

roletypes	::=	roletypes streamtype | streamtype
streamtype	::=	idname **=** streambody **;**
streambody	::=	**<** streamsig **>**
streamsig	::=	streamsig **,** sigcomponent | sigcomponent
sigcomponent	::=	**COMPONENTS OF** idname | idname **:** componenttype
componenttype	::=	idname | flowdef | streambody
flowdef	::=	(idname **,** direction)
direction	::=	**SOURCE** | **SINK**

Binding Roles

roles	::=	roles roledef | roledef
roledef	::=	idlist **:** idname **;**
idlist	::=	idlist **,** idname | idname

Binding Requirements

requirements	::=	requirements requirement | requirement
requirement	::=	subtyperel **;** | matchrel **;**
subtyperel	::=	**ISSUBTYPE** flowitem | **ISSUBTYPE** idname
matchrel	::=	**ISMATCH** flowitem | **ISMATCH** idname expectflows
expectflows	::=	{ flowlist } | ϵ
flowlist	::=	flowlist **,** idname | idname | ϵ
relitem	::=	flowitem | idname

Binding Behaviour

behaviour	::=	mappings conversions
mappings	::=	mappings mapbehave | mapbehave
mapbehave	::=	behaveop idname idname **;**
behaveop	::=	**SINGLECAST** | **BROADCAST** | **MULTICAST**

Cardinality

cardinality	::=	cardinality cardinal | cardinal
cardinal	::=	**REQUIRE** min **..** max **OF** idname **;**
min, max	::=	integer

Type Conversion

conversions	::=	conversions convert | ϵ
convert	::=	**CONVERT** flowitem **TO** flowitem **;**
flowitem	::=	idname **.** idname

5 Conclusion

Streams have an important role to play in distributed multimedia systems. They provide a formal specification of the transfer of information between a source and sink objects. Tools need to be made available to the multimedia applications designer to enable them to model the complex behaviour inherent in such applications.

RIVUS captures much of the behaviour of multimedia applications and thus allows the programmer to specify binding templates which model such behaviour.

Currently the language is being extended to support type inheritance. Further work is aimed at constructing an infrastructure which will allow streams to be instantiated directly from templates described in RIVUS and the introduction of a type repository to support this process.

References

[1] A. Berry and K. Raymond. The A1 Architecture Model. In K. Raymond and L. Armstrong, editors, *Proceedings of the International Conference of Open Distributed Processing*, pages 47–52, Brisbane, Australia, February 1995.

[2] W. Brookes J. Indulska A. Bond and Z. Yang. Interoperability of Distributed Platforms: a Compatibility Perspective. In K. Raymond and L. Armstrong, editors, *Proceedings of the International Conference of Open Distributed Processing*, pages 53–64, Brisbane, Australia, February 1995.

[3] S. Gibbs C. Breiteneder and D. Tsichritzis. Modelling of Audio Video Data. *Lecture Notes In Computer Science*, 645:323–339, 1992.

[4] G. Coulson G. S. Blair J. B. Stefani F. Horn and L. Hazard. Supporting the Real-Time Requirements of Continuous Media in Open Distributed Processing. *Computer Neworks and ISDN systems*, 27(8):1231–1246, July 1995.

[5] W. Brookes A. Berry A. Bond J. Indulska and K. Raymond. A Type Model Supporting Interoperability in Open Distributed Systems. In *Proceedings of the First International Conference on Telecommunications Information Networking Architecture*, pages 275–289, Melbourne, Australia, February 1995.

[6] ISO/IEC. ITU Recommendation X.902 | ISO/IEC 10746-2, Open Distributed Processing - Reference Model - Part 2: Foundations. Technical Report ISO/IEC 10746-2: 1995, International Standards Organization, Central Secretariat , Geneva, Switzerland, 1995.

[7] ISO/IEC. ITU Recommendation X.903 | ISO/IEC 10746-3, Open Distributed Processing - Reference Model - Part 3: Architecture. Technical Report ISO/IEC 10746-3: 1995, International Standards Organization, Central Secretariat , Geneva, Switzerland, 1995.

[8] P.F. Linington. RM-ODP: The Architecture. Presentation at International Conference of Open Distributed Processing, February 1995.

[9] B. Liskov and J.M. Wing. A Behavioural Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[10] P.Leydekkers V.Gay and L.Franken. A Computational and Engineering View on Open Distributed Real-time Multimedia Exchange. In *Proceedings of NOSSDAV'95*, pages 53–64, Boston, USA, April 1995.

[11] N. Williams and G. S. Blair. Distributed Multimedia Applications - A Review. *Computer Communications*, 17(2):119–132, 1994.