Revised CONNECT Architecture

Emil Andriescu, Amel Bennaceur, Gordon S. Blair, Antonello Calabro, Roberto Speicys Cardoso, Luca Cavallaro, Nikolaos Georgantas, Paul Grace, Valerie Issarny, Yun Ma, et al.

Emergent Connectors for Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D1.3

Revised CONNECT Architecture

http://www.connect-forever.eu

Project Number	:	231167
Project Title	:	CONNECT – Emergent Connectors for Eternal Software Intensive Networked Systems
Deliverable Type	:	Report

Deliverable Number	:	D1.3
Title of Deliverable	:	Revised CONNECT Architecture
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	20
Contractual Delivery Date	:	1st February 2012
Actual Delivery Date	:	29th February 2012
Contributing WPs	:	WP1
Editor(s)	:	Paul Grace, Gordon S. Blair (LANCS)
Author(s)	:	Emil Andriescu (Inria-Ambientic), Amel Bennaceur (Inria), Gordon S. Blair (LANCS), Antonello Calabro (CNR), Roberto Speicys Cardoso (Ambientic), Luca Cavallaro (Inria), Nikolaos Georgantas (Inria), Paul Grace (LANCS), Valerie Issarny (Inria), Yun Ma(PKU), Maik Merten (TUD), Nicola Nostro(CNR), Vatsala Nundloll (LANCS), Pierre Guillaume Raverdy (Ambientic), Rachid Saadi (Inria), Daniel Sykes (Inria)
Reviewer(s)	:	CNR, Inria

Abstract

Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. This challenge is exasperated by the highly heterogeneous technologies employed by each of the interacting parties, i.e., in terms of hardware, operating system, middleware protocols, and application protocols. The key aim of the CONNECT *project is to drop this heterogeneity barrier and achieve universal interoperability. Here we report on the revised* CONNECT *architecture, highlighting the integration of the work carried out to integrate the* CONNECT *enablers developed by the different partners; in particular, we present the progress of this work towards a finalised concrete architecture. In the third year this architecture has been enhanced to: i) produce concrete* CONNECT*ors, ii) match networked systems based upon their goals and intent, and iii) use learning technologies to find the affordance of a system. We also report on the application of the* CONNECT *approach to streaming based systems, further considering exploitation of* CONNECT *in the mobile environment.*

Keyword List

Interoperability, middleware, middleware heterogeneity, service discovery heterogeneity, service interaction heterogeneity, application-level heterogeneity, data heterogeneity, non-functional properties, software architecture, connectors, semantics, ontologies.

Document History

Version	Type of Change	Author(s)
1	Document creation	LANCS
2	Section 3: Learning, Goals and Assembly	Inria
3	Section 1: Introduction (1st complete draft)	LANCS
4	Section 2	LANCS
5	Abstract to Concrete Section	LANCS
6	Enabler Architecture Section	LANCS
7	Enablers	CNR, Inria, TUD, LANCS, PKU
8	Section : Interoperable Live Streaming	Inria, Ambientic
9	Revision of chapter structure	LANCS
10	Synthesis Enabler	Inria
11	Implementation of Enabler architecture section	PKU
12	Revisions of Conclusions	Inria
13	Revisions of Contributions Chapter	Inria
14	Evaluation Chapter	Inria
15	Addition to Abstract to Concrete section	Inria
16	Addition to evaluation chapter	Inria
17	Enhancements and revisions of section 2	Inria
18	Enhancements and revisions of section on enablers and enabler architecture	Inria
19	Update of section on streaming and addition of streaming evaluation	Inria, Ambientic
20	Revisions according to internal reviews	CNR, Inria, TUD, LANCS, PKU, Ambientic

Document Review

Date	Version	Reviewer	Comment
07/02/2012	18	Antonello Calabro	This deliverable is a giant leap for the CONNECT architecture and its integration, well written and rich of contents. The main thing we noticed concerns the diagrams: according with SAC comments, we should improve standardization of diagrams with usage of strict UML. I noticed that the final part of the chapter 5 is not CONNECT related.
08/02/2012	19	Valerie Issarny	Abstract is to be revised to better match the structure. Chapter 1: I think the statement on adaptation is too much and raises too high expectation given that results so far is in WP5 and although it is already good, work is needed across the architecture; it should be made clear this is ongoing work and key aspect for final period. Chapter 2: I would change the title so as to make clear this is Y2 status. The statement on the use of semantic annotation in the learning enabler is wrong to me, and it is referred as future work in D4.3. Chapter 3: Introduce ”interaction enabler” that is new in Fig 3.1; also sec and trust enablers. I would add a statement about Starlink in Chapter 2. Overall a very good chapter but may give the wrong impression that we are advanced in terms of adaptation. Chapter 4: 4.6, 4.7 are quite long compared to the other enablers , I would summarize here, and have all the technical detail in specific deliverable. Chapter 5: Presentations of contributions to be revised. Chapter 6: Seems to be mostly evaluation of 5.4.5-6; I would move this under 5.4

Table of Contents

List of Figures

List of Acronyms

Acronym	**Meaning**
DSL	Domain Specific Language
eLTS	enhanced Labelled Transition System
LTS	Labelled Transition System
NS(s)	Networked System(s)
OSGi	Open Services Gateway Initiative
SOAP	Simple Object Access Protocol
WSDL	Web Service Description Language

1 Introduction

Pervasive computing is characterised by networked systems (NS) that are highly heterogeneous; this heterogeneity is characterised by differences in the application functionality provided by the two systems, and in the different middleware protocols they utilise. In pervasive computing systems, interactions are typically spontaneous, i.e., the networked systems only discover one another at runtime and then decide to interact. Interoperability is defined as the ability to exchange, understand and use data with another system; where systems are highly heterogeneous and spontaneously interact, interoperability remains an unsolved problem. It is precisely this problem that the CONNECT project is addressing.

Previously in Deliverable D1.1 [6] we identified five types of heterogeneity acting as a barrier to interoperability: i) discovery protocol, ii) middleware protocol, iii) data heterogeneity, iv) API heterogeneity, and v) heterogeneous non-functional properties. Our survey of existing research and industrial solutions [6] then showed that no approach proposed by either the middleware or the semantic interoperability communities achieved a complete solution that addressed each of these five types of heterogeneity in dynamic, pervasive computing environments.

The CONNECT project is investigating new approaches to this interoperability challenge; it aims to go beyond traditional approaches and solve the problem in a fundamentally different way. That is, rather than predefine a particular global standard or middleware solution, the interoperability software is created dynamically to meet the requirements of the interoperating systems. The structure and behaviour of networked systems is discovered and learned; based upon this information a CONNECTor is dynamically synthesised that will ensure the two systems will interoperate. The CONNECTor is then monitored to ensure it is maintaining the required non-functional properties (with respect to dependability, security and trust).

1.1 The Role of Work Package WP1

The aim of WP1 is to provide an overall architecture for CONNECT, defining and documenting the common architectural principles behind the CONNECT approaches to achieving extremely long-lived (eternal) networked systems. The original three tasks of WP1 as described in the description of work [10] are as follows:

Task 1.1: CONNECT architecture. Elaborating a technology-independent and eternal architectural framework for emergent CONNECTors.

Task 1.2: Eternal system semantics. Eliciting an ontology-based characterization of the semantics of connected systems.

Task 1.3: CONNECT realization. Developing key underlying systems principles and techniques to support the development of a practical, efficient and a self-sustaining CONNECT prototype.

Hence, this work package performs a central role for CONNECT as a whole: acting as a point of integration for the specialised work from each of the work packages: WP2 [11], WP3 [14], WP4 [13], and WP5 [12]. Importantly, WP1 provides the system prototypes necessary to directly support the experimentation and evaluation work of the project as a whole.

1.2 Summary of Achievements in Year One: The Initial CONNECT Architecture

In the first year of the project, the following key contributions where achieved [6]:

- Based upon analysis of typical complex pervasive systems, we identified the five important dimensions of heterogeneity (described previously) that are encountered and must be resolved by CONNECT.

- We produced a state of the art of middleware and data interoperability solutions; this showed that no solution achieves the interoperability proposed by CONNECT; and hence highlighted the key contributions that this project can make to the field.
- A description of the initial technology-independent and eternal architectural framework and architectural principles was presented; this was validated against a small number of case studies.

1.3 Summary of Achievements in Year Two: The Intermediary CONNECT Architecture

In the previous WP1 deliverable D1.2 [4], the intermediary version of the CONNECT architecture was presented; this refined the initial architecture and began to proceed towards a concrete realisation of the CONNECT vision. To do so, a number of core building blocks of the CONNECT architecture were highlighted:

- The CONNECT *Networked System Model* was defined to create a runtime software artefact that accurately described each networked system, such that the information could inform the CONNECT procedures. The model instances contain a rich semantic description of each individual networked system in terms of their: role, interface syntax, behaviour and non-functional properties.
- The *Enabler Architecture* which describes how enablers (software components that perform the CONNECT process of creating CONNECTors) are composed and co-operate was refined to provide a more concrete vision for how the different phases of CONNECT are put together.
- the CONNECTor architecture described how the software to connect two networked system is constructed.

Further achievements in year two, included: i) the concrete design and implementation of the discovery enabler, ii) the realisation of CONNECTors using two approaches (runtime interpretable domain specific models, and code generation), iii) the identification of how ontologies cross-cut the CONNECT architecture, and in particular how they are specified, also how they are utilised by the individual enablers.

1.4 Review Recommendations and Challenges for Year Three

The project reviewers proposed the following recommendations for both the WP1 work package and the project in light of the review of the work carried out during the second year of the project:

- *"There is clearly scope for improved integration and now the opportunity to do this is through the common case study"*. In this deliverable; we document the integration of the enablers; the advancement of each enabler implementation is presented and we then specify how these enablers co-operate. Further, we demonstrate the concrete realisation of CONNECTors as produced by the initial discovery, learning and synthesis phase. The initial abstract models that specify a CONNECTor (and were introduced in the second year of the project) are made concrete using a newly defined binding procedure, and can now be executed directly.
- *"We made the point last time about defining the intent of the system, i.e., what drives interconnection when it includes more than two participants. This has still not been satisfactorily addressed"*. The concepts of *networked system goal and intent* is introduced in this document (see Section 5.3).

- *“We would like to see how the architecture deals with adaptation for example to deal with changing quality of service. Consideration of the relationship between synthesis and adaptation would be beneficial”.* In the revised architecture we have introduced the conceptual methods to monitor and adapt a deployed CONNECTor at runtime when there are violations of the non-functional requirements of the networked systems. In the final year of the project we plan to concretely realise and test these concepts.

- *“The lifecycle of Enablers is not clear, specifically how they are instantiated and how their multiple instances cooperate should be explicitly stated”.* The lifecycle of enablers is specified in the revised version of the Enabler architecture (see Section 3.2). Here a single instance of each enabler is instantiated when the CONNECT architecture is created and deployed in a given environment.

- *“The means for expressing an intent to communicate and the procedure for reflecting in collaboration with Enablers need to be elaborated; also it is not fully clear what triggers an action of a Discovery enabler and what happens if there are multiple instance of it.”* The enhanced implementation of the Discovery enabler (as discussed in Section 4.1) illustrates how the Discovery enabler captures intent to communicate and also the trigger for discovery to take place.

1.5 Achievements in Year Three

The objective of this report is to provide a revised version of the CONNECT architecture, and in particular highlight the increasing integration work carried out in the third year, along with the novel research contributions that underpin both the architecture and the implementation of CONNECT enablers. In this report we highlight the following key contributions:

- *From abstract to concrete* CONNECT*ors* . In the second year, abstract LTS models specified the behaviour of a mediator that formed the potential functionality of a CONNECTor. However, this model that was produced by the synthesis enabler lacked the necessary concrete information to be executable. In the third year we have resolved this issue with a new approach to map abstract LTS models into concrete models that can be executed using an interpreter. We describe in Section 5.4 the concept of a *k-Coloured automaton* as the concrete model that specifies the mediation between two networked systems in terms of both application and middleware heterogeneity. The Synthesis enabler produces this model. This model is then interpreted; at interpretation time the application actions are bound to the annotated middleware protocol related to the action. This binding ensures that an application action transition in the k-Coloured automaton is transformed into the sending and receiving of middleware protocol messages.

- *Integration of learning, discovery and synthesis.* The Discovery and Learning enablers have been enhanced to include: affordance learning, and the discovery of networked system goals (see Sections 5.2 and 5.3). The synthesis enabler is integrated to receive this information and produces concrete CONNECTors according to the contribution above. A small case-study (see Section 5.4.5) illustrates a complete example of these enablers working together to address interoperability in a simple example: photo-sharing applications implemented using XML-RPC and CORBA.

- *The* CONNECT *architecture is enhanced to manage non-functional requirements.* We describe the conceptual revisions and contributions to the architecture concerning non-functional requirements; in particular we highlight how the CONNECTors are instrumented in order to inform the monitoring of behaviour (in Section 3.4.2). We then describe the integration plan for the non-functional enablers (DePer, Security and Trust) within the overall CONNECT architecture in Section 3.2.

- *Application of the* CONNECT *approach to media streaming applications in the mobile environment*. Networked systems providing streaming services, e.g., video streaming using a streaming communication protocol (e.g., RTSP) offer different challenges for CONNECTor construction in order to address interoperability. Section 5.5 explores streaming CONNECTors and discusses CONNECT solutions to achieve interoperable streaming by mobile devices.

1.6 Outline of Report

In Section 2 we describe the status of the CONNECT architecture at the end of the second year of the project. The goal of the section is to summarise the contributions presented in D1.2 [4] and provide the reader with a reminder of the previous progress. In particular, the concepts of Enablers, the Enabler Architecture, and the CONNECTor architecture are presented.

In Section 3 the revised CONNECT architecture is presented; this highlights the progress made during the third year beyond the Intermediate CONNECT architecture discussed in Section 2. The integration of enablers is specified in the Enabler architecture in Section 3.2. The specific connections between enablers are defined, and the behaviour sequences that achieve the co-ordination are specified. The implementation details of this integration are then presented in Section 3.3; here we discuss how the interface connections between enablers are implemented using different binding styles and messaging frameworks. Finally, in Section 3.4 the CONNECTor architecture describes the make-up of individual CONNECTors. The detailed internal interactions of CONNECTor elements are specified. Furthermore, the insertion of probes related to the maintenance of non-functional requirements is introduced.

The implementation progress of every enabler is then described in Section 4. Each enabler's functionality is described in further detail. Furthermore, we specify the interfaces that provide the enabler's service; these interfaces are used in the co-ordination of enablers to achieve the functionality of CONNECT.

The underlying conceptual research contributions are presented in Section 5. The work on enhancements towards learning goals and affordances, and also the specification of networked system intent is included. The enhancements to the Synthesis enabler to achieve the complete process from abstract to concrete CONNECTors is presented; a small case-study shows how the Synthesis enabler creates the required models that can be used to concretely construct a CONNECTor and deploy it. We further illustrate some performance measures of the concrete CONNECTors. Finally, the investigation of streaming CONNECTors in mobile environments is detailed, along with an evaluation of the performance of this type of CONNECTor.

Finally, in Section 6 we draw conclusions about the progress in year three and identify a roadmap for WP1 work in the remainder of the project.

2 The Intermediate Architecture Revisited: Year Two

2.1 Introduction

In this section we revisit the CONNECT architecture that was presented in Deliverable D1.2 [4], and which described the work carried out during the second year of the project. This summarises exactly what was stated in that document; we use this to describe the deficiencies of the architecture in order to highlight the improvements and refinements made during the third year. We also highlight the strengths of the intermediate architecture, which were in turn are carried forward without change.

The intermediate CONNECT architecture was previously presented to provide the first concrete vision of the architecture as a platform for development, integration and refinement. The primary elements of this architecture are:

- The CONNECTor architecture specified the structure of a CONNECTor in terms of the elements that implement and execute it. The intermediate CONNECTor architecture from year two is presented in Section 2.2.
- The Networked System Model defined the structure and behaviour of networked systems and served as the common information exchanged between CONNECT enablers. A detailed reminder is given in Section 2.3.
- The CONNECT Enabler Architecture defined how the core enablers were deployed and communicated with one another. This is summarised in Section 2.4.

2.2 The CONNECTor Architecture

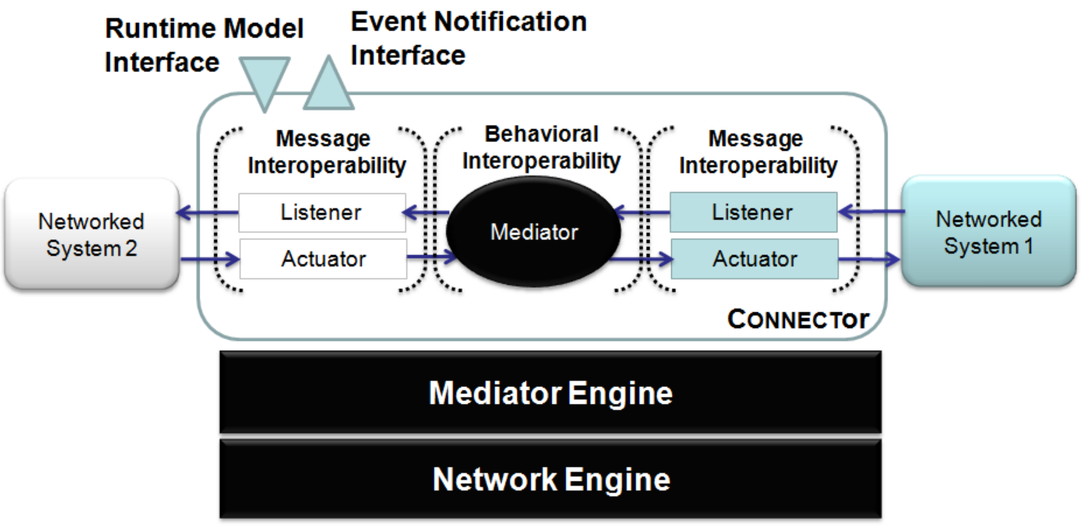


Figure 2.1: The Connector Architecture (Y2)

The CONNECTor architecture as presented in Deliverable D1.2 [4] is illustrated in Figure 2.1. The software elements that compose this key architectural element are described as follows:

- A *Listener* receives network messages (from the Network Engine) in the form of data packets and parses them according to the message format employed by the protocol that this message is specified by. Hence, each Listener parses messages from a single protocol, e.g., a SOAP listener parses SOAP messages. A listener produces an *Abstract Message* that contains the information found in the original data packet, providing a uniform representation that can be manipulated and understood by the other elements in the CONNECTor architecture.

- An *Actuator* performs the reverse role of a listener, i.e., it composes network messages according to a given middleware protocol, e.g., the SOAP Actuator creates SOAP messages. Actuators receive the Abstract Message and translate this into the data packet to be sent on the network via the Network Engine.
- The *Mediator* forms the central co-ordination element of a generated CONNECTor. Its role is to translate the content received from one protocol (in the form of an `AbstractMessage`) into the content required to send to the corresponding protocol. The mediator therefore addresses the challenges of: different message content and formats, and different protocol behaviour, e.g., sequence of messages.
- The *Network Engine* provides a library of transport protocols with a common uniform interface to send and receive messages. Hence, it is possible to receive messages and send messages from multicast (e.g. IP multicast), broadcast and unicast transport protocols (e.g. UDP and TCP).
- The *Mediation engine* is an optional element of the architecture depending upon the implementation approach taken for mediator. The behaviour of the mediator is determined by a high-level model determining the operations to take. In the case where this model is generated directly into code there is no need for a mediation engine. In the case where the mediator model is an executable model (e.g., a BPEL specification, or an alternative CONNECT mediator model) then it is the mediation engine which executes these scripts.
- Then *Event Notification Interface* outputs all important events that occur during the operation of the CONNECTor, e.g., the receiving of a network packet, the completion of parsing, etc. Other elements of the CONNECT architecture can then subscribe to receive these events; this behaviour is closely related to the Monitoring and Dependability enablers.
- The *Runtime Model Interface* is a Meta-Object Protocol based interface that supports introspection of the software elements that compose an individual CONNECTor, i.e., it is possible to observe at runtime what listeners, actuators and mediator are in operation. Furthermore, the interface supports the runtime adaptation of the CONNECTor architecture through the replacement of the prior described elements.

In Deliverable D1.2 [4], we also presented the Starlink framework [7] as a software framework to specify both middleware protocols, and protocol bridges using high-level specifications. Starlink dynamically interprets and executes these models to achieve both middleware and bridging functionality. Hence, Starlink was shown as an initial framework to implement the mediation engine, and also to realise CONNECTors concretely.

2.3 CONNECT Networked System Model

The role of the Networked System Model is to provide a formally-grounded, central specification that describes the syntax, behaviour and semantics of a networked system in a common description language. This common model then enables the CONNECT enablers to achieve their objectives. Importantly, the Networked System Model employs ontologies as the pillar to establish a common understanding of the specification of networked systems. Figure 2.2 highlights the key points of the model; the complete details of the languages employed are provided in Deliverable D1.2 [4]:

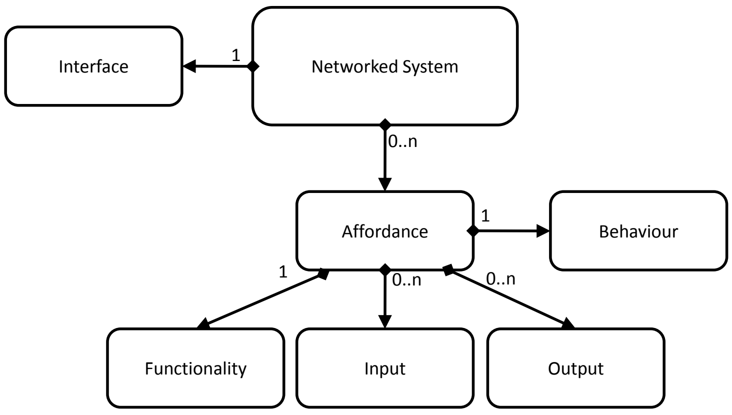


Figure 2.2: Overview of the Networked System Model. Diamonds denote compositions with the given arities.

- The *affordance* is a macroscopic view, or the quality of a feature, of a networked system. Essentially the affordance describes the high-level roles a networked system plays, e.g., 'prints a document', or 'sends an e-mail'. This allows semantically-equivalent interactions with another networked system to be matched; in short, they are doing the same thing. In Figure 2.2, the affordance is composed of one functional ontology concept, and any number of input and output concepts. *A detailed definition of affordance was given in [4], along with the reasons for embracing it within the* CONNECT *project.*
- *Interfaces* provide a refined or a microscopic view of the system by specifying finer actions or methods that can be performed by/on the networked system, and used to implement its affordances. Each networked system is associated with a unique interface. *The **xDL language** was presented as a language for specifying interfaces.*
- The *Behaviour description* documents the application behaviour in terms of how the actions of the interface are co-ordinated to achieve the system's affordance, and in particular how these are related to the underlying middleware functions. A BPEL-based specification language was employed to specify this behaviour.

In practice, textual networked system descriptions provide parts of the Networked System Model in various languages (the rest being provided by affordance or behaviour learning as described in Sections 5.2 and 4.2). Such textual descriptions are parsed to produce instances of the implementation-level CNSInstance shared representation described in Section 4.1.1. For instance BPEL is converted to a labelled transition system (LTS) using a library from WS-Engineer[1] and stored in the CNSInstance.

2.4 The CONNECT Enabler Architecture

The Enabler architecture is the configuration of the enabler components which are deployed in the network environment. Figure 2.3 illustrates how these combine to achieve the particular goal of CONNECT, i.e., to take two networked systems whose heterogeneity denies them from interoperating with one another, and then deploying the required CONNECTor. We now discuss the individual enablers in turn.

The ***Discovery Enabler*** receives advertisement messages that are sent within the network environment by the networked systems. The enabler obtains this input by listening on known multicast addresses (used by legacy discovery protocols). These messages are then processed; information from the legacy messages is extracted and the Networked System Models of the

[1] http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/

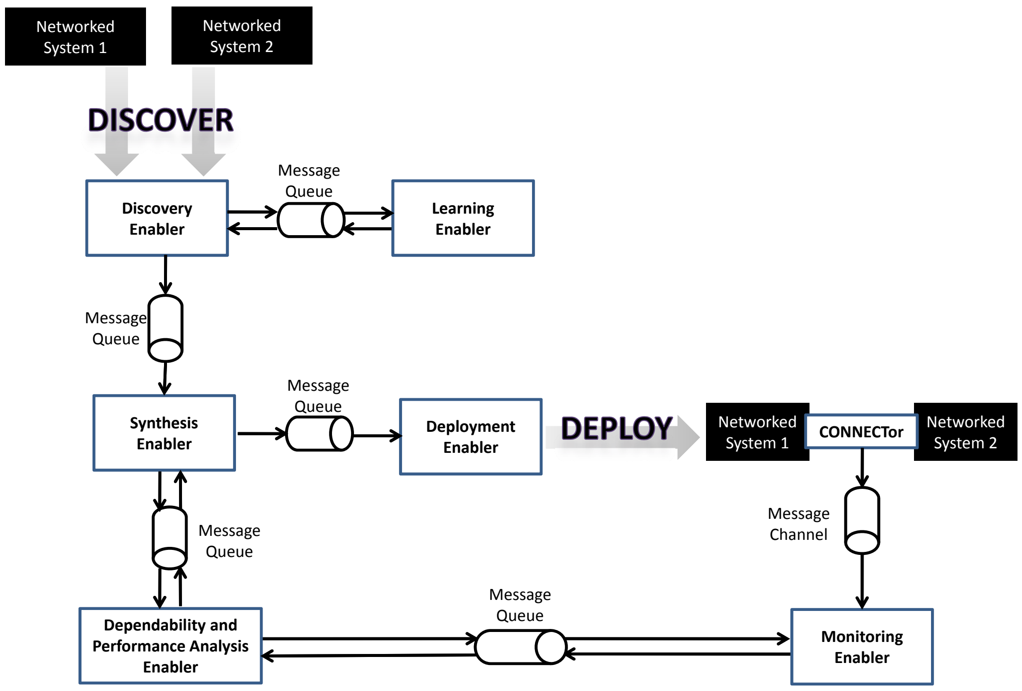


Figure 2.3: The CONNECT Enabler architecture (Y2)

systems in the environment are produced. At this stage the model is certain to contain an interface description. If no affordance is provided, *affordance learning* is used to infer one (see Section 5.2). If no behaviour is provided, the current model is sent to the Learning enabler, which adds the behaviour description. Initial matching is then performed to determine whether two networked systems are candidates to have a CONNECTor generated between them. On a match, the CONNECT process is initiated; the Discovery Enabler sends the model to the Synthesis enabler.

The ***Learning Enabler*** uses active learning algorithms to dynamically determine the interaction behaviour of a networked system from its intermediary representation and produces a model of this behaviour in the form of a labelled transition system (LTS); this employs methods based on monitoring and model-based testing of the networked systems to elicit their interaction behaviour. The implementation of the enabler is built upon the LearnLib tool [25]. It utilises the the interface description from the Networked System Model as an input, and an LTS describing the interaction behaviour is produced and added to the behaviour section of the Networked System Model, and the outcome is a complete - as far as possible - instantiated networked system model. This is sent back to the Discovery Enabler to complete the discovery of the description of networked systems.

The role of the ***Synthesis enabler*** is to take the Networked System Models of two systems and then synthesise the mediator component that is employed by the CONNECTor. For this it performs automated behavioural matching (as opposed to functional matching) and mediation of the two models [31]. This uses the annotated ontology information from the models to say where a sequence of messages is equivalent; based upon this, the matching and mapping algorithms determine an LTS model that represents the mediator. The Synthesis enabler can then output two alternatives (depending upon the style of CONNECTor in use):

- Mediator code. The Synthesis enabler generates the Java executable code that can be deployed directly as part of a CONNECTor configuration.

- An LTS model. The LTS model can be sent directly, in order for it to be used by the mediation engine of a CONNECTor.

Either of these two outputs is sent to the deployment enabler in order to complete the construction of the CONNECTor.

The ***Deployment Enabler*** receives as input the mediator code and the original Networked System Models; its objective is to finalise and then deploy the CONNECTor in this case. In order to do this, the enabler executes two important roles:

- It composes the required functionality to ensure that CONNECTors will communicate with the legacy networked systems, i.e., it will add the listeners and actuators to the mediator generated by the Synthesis Enabler.

- It deploys and manages the executable code of the CONNECTors in the network. For this, the enabler utilises OSGi techniques and available software.

Once a CONNECTor specification has been produced by the Synthesis enabler it sends it to the ***Dependability and Performance analysis*** enabler to determine if the non-functional requirements (as described in the Networked System Model of each networked system) are satisfied. If so, the enabler tells the Synthesis enabler to go ahead and deploy; otherwise, the Dependability enabler creates a set of enhancements to the specification and returns these to the Synthesis enabler. The Dependability enabler also continuously determines if the CONNECTor maintains its non-functional requirements. It receives monitoring data from the monitoring enabler and in the case where there is no longer compliance, the Dependability enabler sends a new specification to the Synthesis enabler to initiate redeployment of a suitable CONNECTor in the current conditions.

The ***Monitoring enabler*** receives requests concerning which CONNECTors to monitor and then collects raw information about the CONNECTors by monitoring data that this CONNECTor published to the monitoring channel. The derived data is passed to the dependability enabler to determine if the original non-functional requirements are being matched.

2.5 Ontologies in the CONNECT Architecture

During the second year of the project [4], ontologies were designed to help describe different aspects of networked systems, such that these ontologies could be leveraged by the enablers performing CONNECT functions. We described how ontologies describing the application domain could be used to support semantic matching and mapping of application behaviour. Where matching determines where application behaviour is equivalent and mapping describes the necessary mappings to achieve the equivalent behaviour. Further to this, ontologies were crucial to supporting the process to determine if the affordances of two networked systems matched. We also described ontologies of middleware and network protocols, i.e., domain ontologies describing the protocol behaviour, protocol messages and the packet content. Again the use of ontologies allowed semantic matching and mapping of protocols to support the interoperation between them.

Hence, it was shown in the intermediate architecture in year two that ontologies crosscut all CONNECT activities. This is applied from the perspective of both models and enablers. All Networked System Models refer to ontologies, i.e., the elements of the model are given meaning through a link to a domain specific ontology. Similarly, the enablers then use ontologies and their associated tools (e.g., reasoners) to perform specific actions; for example, improved discovery by the Discovery enabler, semantic matching and mapping of systems by the Synthesis enabler, and improved semantics-based learning by the Learning enabler.

2.6 Summary

To summarise, this section identified the CONNECT architecture at the end of the second year. The key strengths of this work which are carried forward and built upon in the third year of the project are as follows

1. The separation of the Enabler and CONNECTor architecture to compose the overall CONNECT architecture. This identified the different principles required by the architecture to realise a CONNECTor that achieves interoperability, and also the functionality required to employ the CONNECT functions in order to create such a CONNECTor.
2. The decomposition of key CONNECT elements into enablers. As can be seen in the revised architecture, only one further enabler (the Interaction enabler) has been added to this set (see Section 4.3).
3. The Networked System Model remains the key representation of CONNECT information that is exchanged between enablers.

The key weaknesses of the architecture, and those that we seek to refine in the revised architecture, are described as follows:

1. The architecture is immature with respect to the integration of the individual enablers with one another. Further, it is unclear as to how the architecture functions in order to achieve different tasks. For example, how does it initiate the process for creating a CONNECTor? How does it perform adaptation? In Section 3.2 the revised architecture presents a richer description of the integration in terms of the specific interfaces used by enablers with a clear illustration of how these enablers and their interconnections are implemented. We further show the flow of behaviour in the architecture by adding message sequence diagrams describing the enabler interactions.
2. The architecture does not address some of the key requirements of the CONNECT project. Namely, the ability to handle non-functional properties of networked systems. The architecture in 3.2 includes the initial description of the integration of enablers that achieve the requirements related to maintaining non-functional properties in the system. Further, in Section 3.4.2 we further discuss how (at a fine-grained level) the mechanisms to monitor and maintain non-functional requirements are inserted into the CONNECTor and Enabler architecture. These provide the basis of the work to be carried out in the final year of the project to concretely realise the non-functional behaviour.

While significant progress was made during the second year, there remained significant problems to tackle. The revised architecture described and evaluated in the following chapters highlights the progress made towards a fully integrated concrete architecture to serve as the building blocks for the final evaluation of the CONNECT project.

3 The Revised CONNECT Architecture

3.1 Overview

The Revised CONNECT Architecture is presented in this section. This concentrates on the integration work carried out to make the CONNECT vision concrete and highlights the revisions that improve upon the intermediate architecture.

Here we present the revisions in two sections:

- Section 3.2: the refinements to the Enabler architecture and in particular how enablers are integrated, and the implementation of this integration is introduced in Section 3.3.
- Section 3.4: the refinements to the CONNECTor architecture and the further details about how CONNECTors are implemented.

3.2 The Enabler Architecture: Refinements and Integration

The revised vision of the **Enabler architecture** takes into account the changes made during the third year of the project. Here we provide a complete picture of the architecture, emphasising in particular the integration of the enablers and the concrete realisation of the software elements that compose this architecture (detailed descriptions of each enabler is provided in Section 4). A key refinement is that we identify that there are two distinct phases of behaviour in the CONNECT Enabler architecture:

1. *Connection Time*. In this phase, the enablers of the CONNECT architecture identify networked systems to match, and then create the initial CONNECTor between them that is subsequently deployed. The Connection phase is described further in Section 3.2.1.
2. *Runtime*. In this phase, the enablers of the CONNECT architecture monitor the runtime behaviour of the CONNECTor and the connected networked systems in order to detect and adapt to changes that violate the non-functional requirements of the CONNECTor. Also, adaptation is enacted where there is a violation of the functional requirements due to an incorrect definition of the CONNECTor specification. Achieving Connectability is detailed further in Section 3.2.2.

We now discuss the details of the Enabler architecture in terms of these two phases. That is, which enablers connect together to achieve the functionality, and then how these interact with each other to do this. Subsequently, we then describe the overall implementation approach of the architecture, i.e. how the enablers and their connections are implemented.

3.2.1 Architecture at Connection Time

The configuration of the enablers to perform connection is illustrated in Figure 3.1. Here, software components are represented using the UML Component diagram notation (each enabler is a single instance of a software component). The interfaces provided by these components are represented by a line ending with a filled circle. Receptacles are represented by lines ending with a semi-circle arc and these represent the services required by a software component. A join of an interface and a receptacle defines a message-exchange binding, i.e, enablers communicate by sending messages to one another; bindings can be one of two styles: i) a request-response exchange where the request message contains input parameters and the response message contains the result data, ii) a message notification where a message is sent asynchronously and there is no response. Table 3.1 in Section 3.3 lists which style each of the bindings in the architecture is implemented as.

Importantly, Figure 3.1 introduces three new enablers into the CONNECT Enabler architecture (not included in the Intermediate architecture) whose role and functionality will be expanded upon in this section.

- *The Interaction enabler*. The Learning and Trust enabler need to interact directly with each networked system using that system's middleware protocol. The Interaction enabler provides this service, abstracting the heterogeneity of middleware protocols.
- *The Security enabler*. A separate enabler managing all functionality in the CONNECT Enabler architecture concerning the maintenance of the specific security requirements of a CONNECTor between two networked systems.
- *The Trust enabler*. The required levels of trust of each networked system must be maintained by a connection; the Trust enabler monitors and manages the maintenance of this property.

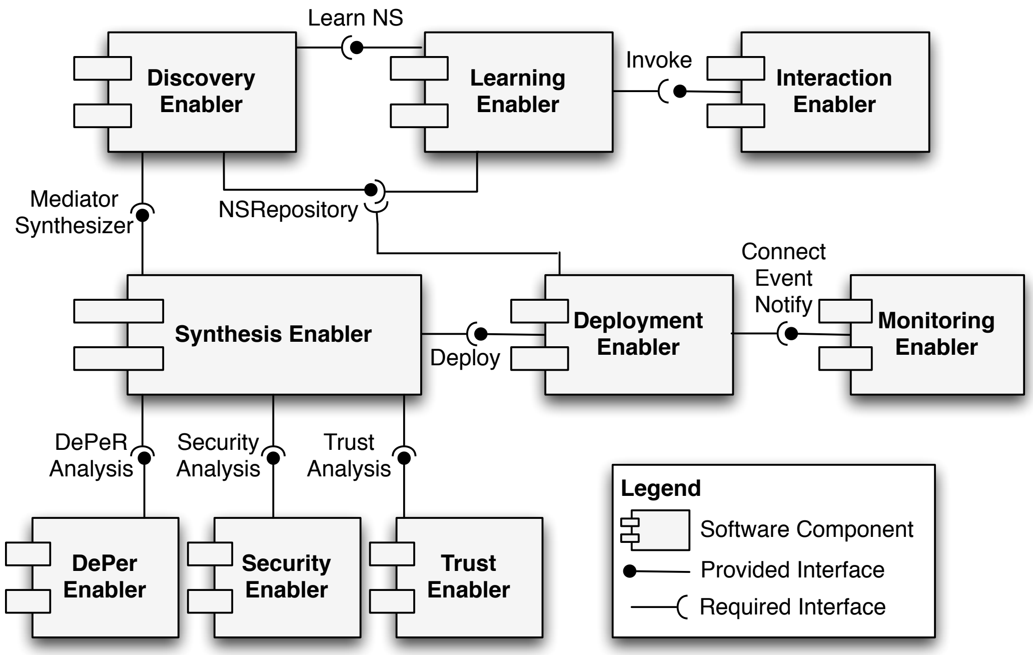


Figure 3.1: The configuration of the enabler architecture for the connection phase.

To explain the behaviour of the Enabler architecture in order to achieve connection, the sequence of messages exchanged by enablers (via their external interfaces) is shown in Figure 3.2. The specific details of how each enabler operates internally is provided in Section 4. Here the diagram follows the UML message sequence diagram notation (the legend in the figure defines the style of message exchange depicted by the notation).

Discovery and Learning of Networked Systems

In Figure 3.2 the *Discovery enabler* is the starting point for behaviour in the connection phase. Each networked system is discovered irrespective of the discovery protocol that is used for advertisement. When a networked system is discovered this produces the internal event (`DiscoverNS`); where an internal event is an occurence internal to a single enabler that triggers external events exchanged via the enabler's interfaces. This internal event of the Discovery enabler initiates the behaviour to build a Networked System Model that provides richer information about the system than is provided by the discovery protocol. The first task is to infer, if necessary, the affordance and to learn the behaviour of this networked system, and so the discovery enabler send a `learnCNSBehaviour` message to the Learning enabler's `LearnNS` interface (the message contains the identifier of the networked system to learn as the parameter). The Learning enabler then retrieves the interface of the networked system (by sending the `getNSInterface` message to the `NSRepository` interface of the Discovery enabler) and uses this to build a richer model of this networked system's behaviour. In order to do this it must interact directly with the networked system being learned, therefore it uses the Interaction enabler (sending an `invoke` message of the `Invoke` interface); here the abstract content describing the required invocation is mapped to the concrete legacy protocol used by the NS and the results of the invocation are returned in a message to the Learning enabler. Once learning is complete the NSModel is updated and then placed in the Discovery enabler repository by sending the `updateNSBehaviour` message to the `NSRepository` interface.

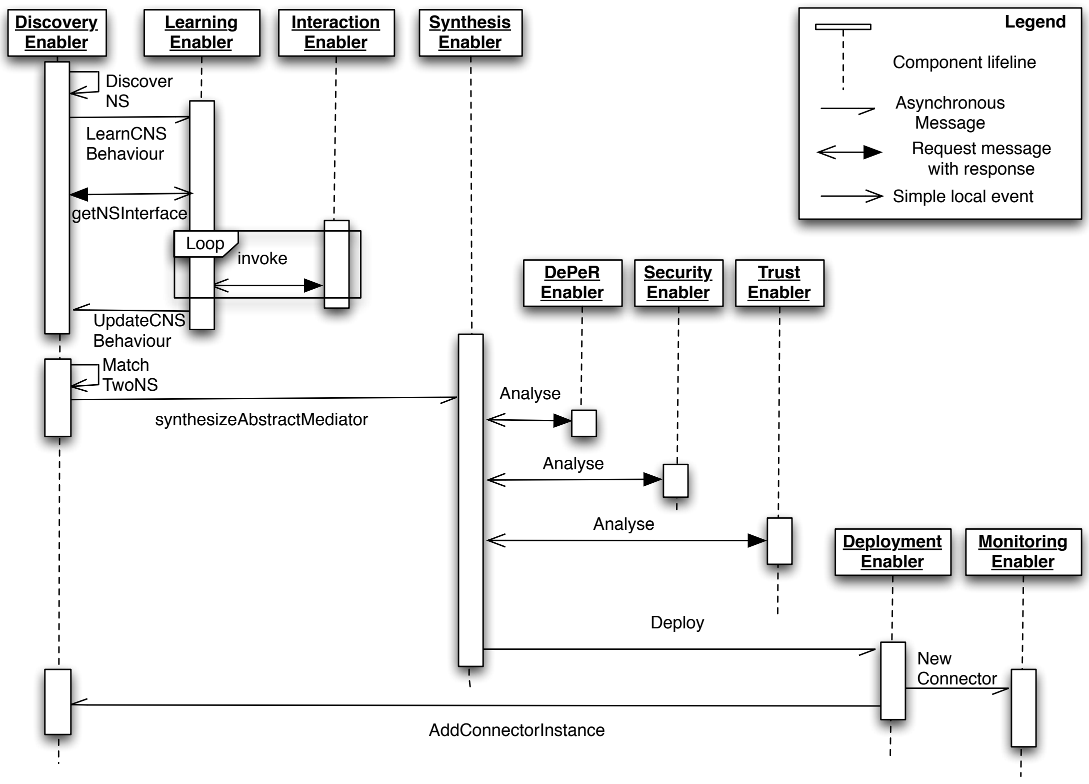


Figure 3.2: The sequence of messages exchanged by CONNECT enablers for connection.

Matching Networked Systems and Creating the CONNECTor

The Discovery enabler matches networked systems and decides where an interoperability solution is required. The result of this matching process is an internal event (`MatchTwoNS`) that initiates the construction of a CONNECTor between two networked systems. Here, the Discovery enabler sends the `synthesizeAbstractMediator` message to the `MediatorSynthesizer` interface (passing the models and information about the NS as parameters). The Synthesis enabler then constructs the initial abstract enhancedLTS model of the mediator in order for it to be analysed against the non-functional requirements. Subsequently, an `Analyse` message is sent to each of the three connectability enablers (i.e. the `DePerAnalysis`, `SecurityAnalysis`, and `TrustAnalysis` interfaces) and any potential refinements are returned as changes to this CONNECTor model. The CONNECTor is then deployed when the Synthesis enabler sends the `deploy` message to the Deployment enabler's `Deploy` interace. This creates and deploys the concrete CONNECTor specification and an instance of the Starlink interpreter [7] to execute the specification. Section 5.4 provides a rich description of the concrete specification, and how Starlink is used to execute a deployed CONNECTor. The Monitoring enabler is then notified of the created CONNECTor in order to notify subscribers of listening for deployment events (the Deployment enabler sends a `NewConnector` message to the Monitoring enabler's `ConnectEventNotify` interface. The Learning Enabler sends a `subscribe` message to the

Monitoring enabler's `Subscribe` interface to receive this notification. Similarly, the repository of the Discovery enabler is updated with information about the deployed CONNECTor by sending the `addConnectorInstance` message to the `ConnectorRepository` interface.

3.2.2 Runtime: Realising Connectability

Once a CONNECTor is deployed and executing, the configuration of enablers as seen in Figure 3.3 behaves so as to achieve the required connectability properties of the CONNECTor. Hence, this configuration concentrates on monitoring the behaviour of the networked systems and CONNECTors and dynamically adapts the CONNECTor when the non-functional requirements are no longer satisfied. Importantly, software elements are inserted into the CONNECTor (as described in Section 3.4.2) in order for CONNECTors to communicate with the enablers directly. These are software components composed directly within the CONNECTor instance (also a software component).

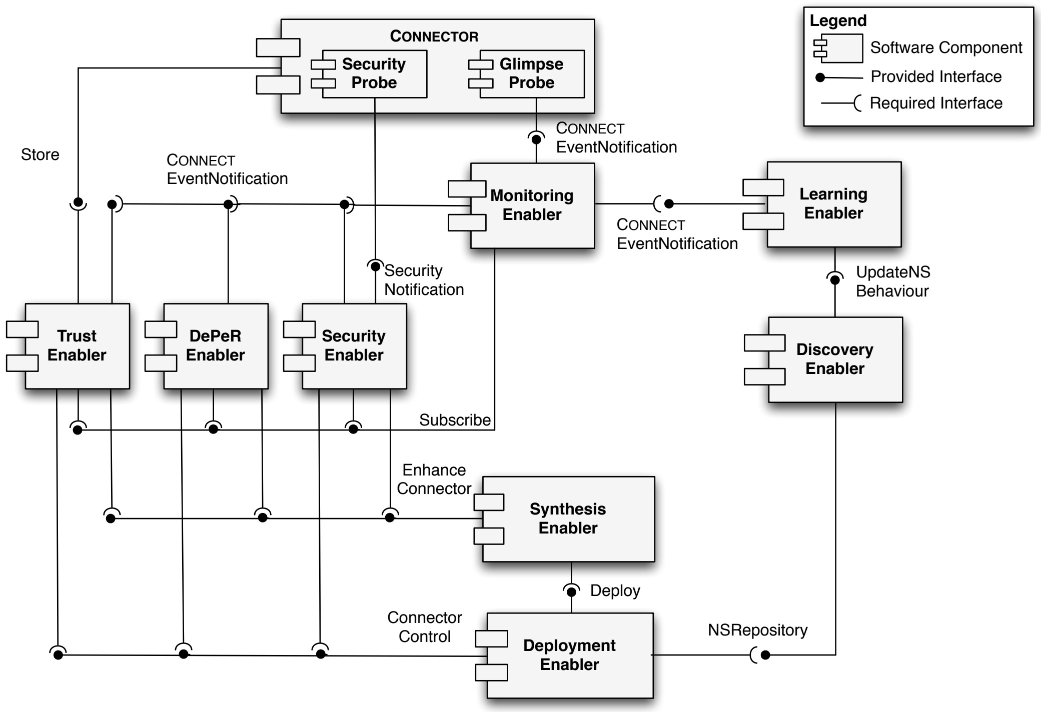


Figure 3.3: The configuration of the enabler architecture for the connectability phase.

Once again we use message sequence diagrams to highlight how the architecture is employed to achieve the dynamic monitoring and adaptation of CONNECTors. Figure 3.4 demonstrates the behaviour performed by the Monitoring and DePer enabler as part of the Connectability phase. The DePer enabler first subscribes to be notified of events of interest that are generated by the CONNECTor. Here, DePer sends a `subscribe` message (with the subscription filter as a parameter) to the `Subscribe` interface of the Monitoring Enabler.

All events generated by the CONNECTor are sent by the Glimpse probes embedded within the CONNECTor to the Monitoring enabler component via the `Publish` message of the `ConnectEventNotify` interface. Where these match a required subscription (of the DePer enabler) the Monitoring enabler then forwards the event message via the `GlimpseBaseEvent` message to the individual enabler.

The DePer enabler monitors and analyses the behaviour of the CONNECTor with respect to the non-functional requirements, starting the adaptation process where there is a violation of those requirements. Hence, Figure 3.4 illustrates the sequence of events where the

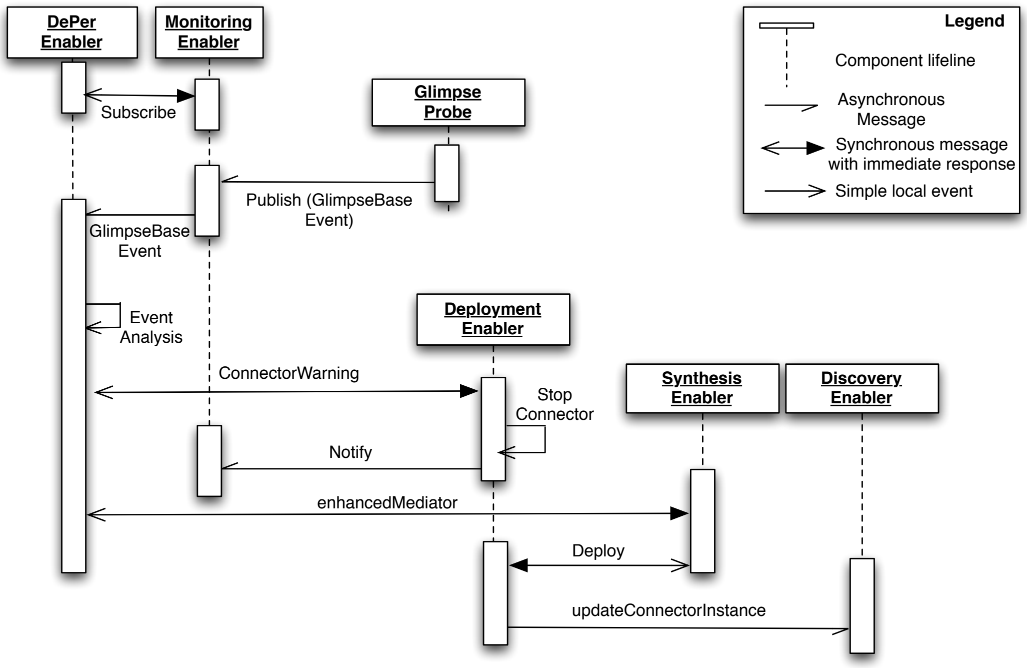


Figure 3.4: The sequence behaviour of monitoring and dependability adaptation

dependability enabler analyses the incoming CONNECTor events. When it detects a violation of the non-functional requirements it first sends a `ConnectorWarning` message to the `ConnectorControl` interface of the Deployment enabler. The Deployment enabler then performs a local `StopConnector` operation in order to place the CONNECTor in a safe state to adapt. Subsequently it sends an `enhancedMediator` message to the Synthesis enabler. The Synthesis enabler then constructs the new concrete CONNECTor based upon this model and then creates a new instance of the CONNECTor by sending the `deploy` message to the Deployment enabler. Finally, the CONNECT repository (which is part of the Discovery enabler and is described in Section 4.1.1) is updated with the new information about the CONNECTor instance when the Deployment enabler sends the `updateConnectorInstance` message to the Discovery enabler.

A similar adaptation process is enacted by both the Security and Trust enablers (as seen in Figures 3.5 and 3.6). This describes the planned integration of the to be completed Security and Trust enablers. The security enabler receives violation events directly from security probes inserted into the CONNECTor; these inform the decision to perform the adaptation loop. The Trust enabler is more proactive: it continuously monitors the trust of a networked system by using the Interaction enabler to test the provided service of each networked system. Based upon the result of this it calculates a trust value for the CONNECTor and stores this information as meta-data within the CONNECTor (sending the `PutMetaData` message to the CONNECTor's `Store` interface). Note, the current trust value is read by sending the `GetMetaData` message to the `Store` interface Once again, when there is a violation of trust the adaptation mechanism to stop, resynthesise and redeploy is initiated by the Trust Enabler.

3.3 Implementing the CONNECT Enabler Architecture

The previous section highlighted how enablers co-operate in order to provide the CONNECT functionality. In CONNECT, the different enablers are developed by different partners who use various implementation techniques and technologies. A detailed description of each enabler's implementation is described in Section 4. In order to implement the complete CONNECT ar-

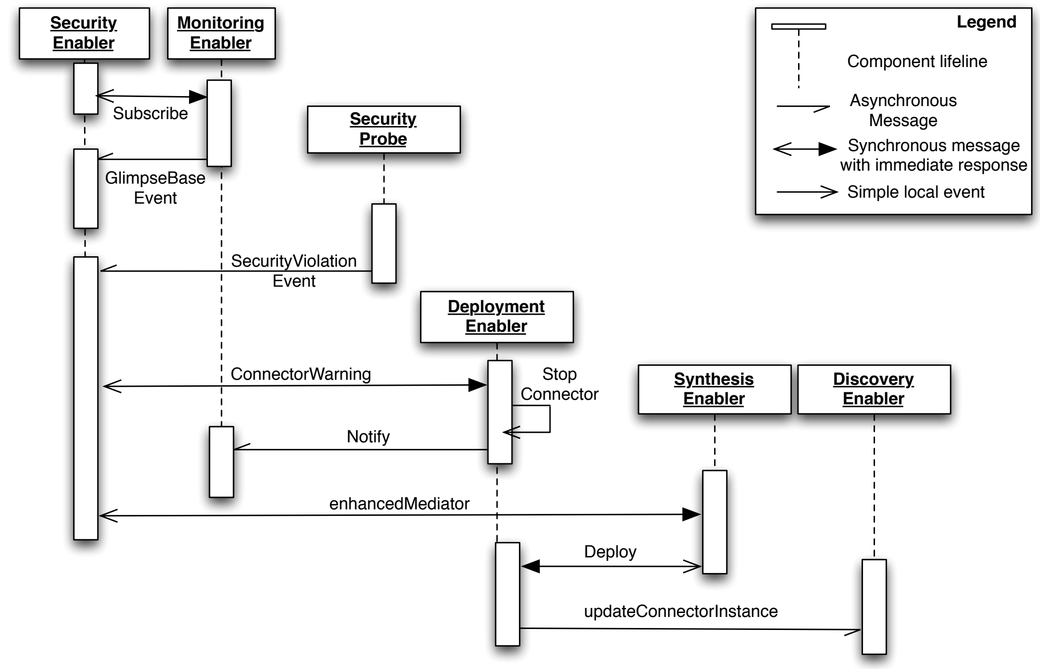


Figure 3.5: The sequence behaviour of monitoring and security adaptation

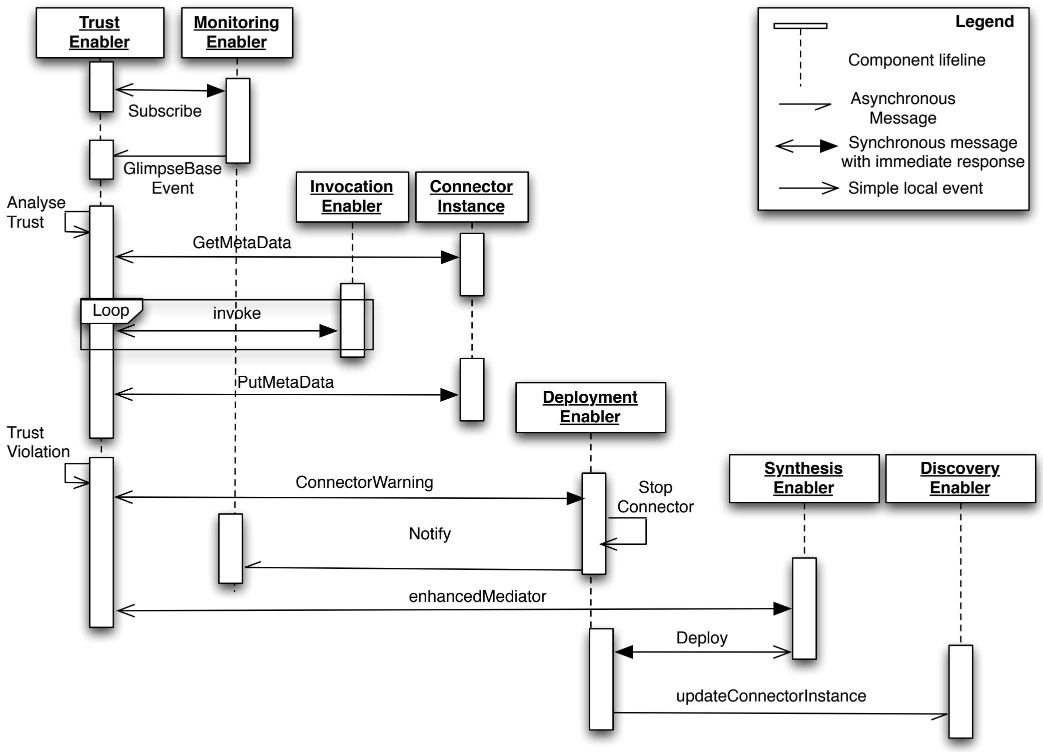


Figure 3.6: The sequence behaviour of monitoring and trust adaptation

chitecture and activate the CONNECT continuous process we now describe the implementation methods employed for each enabler's communication. That is to say how one enabler exchanges messages using the interface of another enabler.

The implementation of the enabler interfaces fall into two categories:

- **Java interfaces**. The enablers are implemented as plain Java objects (POJOs) and the methods they provide are defined in a Java interface. These methods can then be invoked using a Java method call.
- **Java messaging**. The enablers exchange asynchronous messages using a publish-suscribe message channel. That is, the enablers subscribe to the channel for messages of particular types, and the enablers publish messages to this channel.

In spite of the different styles of implementation, we chose the JMS (Java Message Service) API as the implementation solution to provide unified communication between enablers. The Java interface approach is implemented using JMS by creating an asynchronous request-response message exchange. A request message is sent corresponding to the method call, with a response message returned corresponding to the result of the method (for `void` methods this is a one-way message). The benefits of using solely JMS is that a single communication technology is used in the deployment of the CONNECT architecture, and hence simplifies the management and deployment of the architecture.

3.3.1 Usage of Java Message Service

The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. JMS is a part of the Java Platform, Enterprise Edition, and is defined by a specification developed under the Java Community Process as JSR 914. It is a messaging standard that allows application components based on the Java Enterprise Edition (JEE) to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous. JMS supports two different message delivery models:

- *Point-to-Point (Queue destination)*: In this model, a message is delivered from a producer to one consumer. The messages are delivered to the destination, which is a queue, and then delivered to one of the consumers registered for the queue. While any number of producers can send messages to the queue, each message is guaranteed to be delivered, and consumed by one consumer. If no consumers are registered to consume the messages, the queue holds them until a consumer registers to consume them.
- *Publish/Subscribe* (Topic destination): In this model, a message is delivered from a producer to any number of consumers. Messages are delivered to the topic destination, and then to all active consumers who have subscribed to the topic. In addition, any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribers. If there are no consumers registered, the topic destination doesn't hold messages unless it has durable subscription for inactive consumers. A durable subscription represents a consumer registered with the topic destination that can be inactive at the time the messages are sent to the topic.

Table 3.1 presents a list of the interfaces in the CONNECT architecture and describes their behaviour styles, either asynchronous messaging or request-response, and how they are to be implemented by JMS. We can see that there are three different ways of applying JMS to the implementation.

1. *Native JMS*. Some enablers directly use the programming model offered by JMS to implement their interface, providing some proxy classes that encapsulate the JMS entities and handle connection operations. For example, the interface provided by DePer enabler has a "DePerMessenger" class, whose functionality is to allow communications between the Synthesis enabler and the DePer enabler. The DePerMessenger is implemented with ActiveMQ JMS Broker which provides clients with connectivity and message storage/delivery functions.

Table 3.1: Implementation styles of CONNECT enabler interfaces

Interface Name	Style	Implementation
NSRepositoryInterface	Request-Response	CONNECT JMS Wrapper
ConnetorRepositoryInterface	Request-Response	CONNECT JMS Wrapper
LearningEnabler	Request-Response	CONNECT JMS Wrapper
MediatorSyhthesizerInterface	Request-Response	CONNECT JMS Wrapper
Invoke	Request-Response	CONNECT JMS Wrapper
DeploymentEnabler	Request-Response	CONNECT JMS Wrapper
DeployNotification	asynchronous messaging	Glimpse API (Native JMS)
ConnectEventNotification	asynchronous messaging	Glimpse API (Native JMS)
DePerMessenger	Request-Response	Native JMS

2. *Glimpse*. Glimpse is the monitoring framework developed by CONNECT to support behavioural learning, performance and reliability assessment, security, and trust management. Enablers relative to Connectability employ Glimpse for monitoring running CONNECTors. Glimpse is a system implemented atop native JMS.

3. *The* CONNECT *JMS Wrapper*. Some enablers provide request-response interfaces. We have pointed out that this style can be transormed to JMS messaging. The CONNECT JMS Wrapper performs this task. It contains a framework for transformation between method invocation and message sending, as well as a set of wrappers for Java interfaces.

3.3.2 CONNECT JMS Wrapper

The objective of the CONNECT JMS Wrapper is to develop a generic approach to implement method invocation of Java interfaces using the Java Message Service (JMS); subsequently this is applied to support distributed enabler communication. The JMS wrapper consists of two parts:

- A framework for the transformation of method invocations to message transfers. This framework is a generic approach which can be adopted in all the cases concerning the need to use message transfer to implement invocation.
- A set of wrappers implementing enablers' interfaces with the above framework by JMS. For each interface which features method invocation, there is a corresponding JMS wrapper. These wrappers are of specific use to the CONNECT architecture implementation.

The subsections describe the detail of the CONNECT JMS Wrapper implementation. The following terminologies are used for the convenience of the literature. *Caller* represents the component/enabler which produces the request to invoke a method of an interface provided by another component/enabler. *Callee* represents the component/enabler which provides interfaces for other components/enablers to invoke.

JMS Wrapper framework

The idea to implement remote method invocation by JMS is intuitively simple, since the parameters and return values naturally form messages to exchange. We open a channel for message exchange between caller and callee. When invoking a method, the caller creates a message including the method's name and parameters, sends it to the channel and waits for the reply. Callee listens to the channel and receives the request message. Then the callee invokes the method on the POJO, creates a message including the return values and sends this message back to the channel, where the caller listens to receive the return message.

We propose that the JMS wrapper framework must be: general, flexible, easy to use, and provide high performance. Concerning these requirements with respect to CONNECT, we design and implement the architecture of the CONNECT JMS wrapper framework as depicted in Figure 3.7.

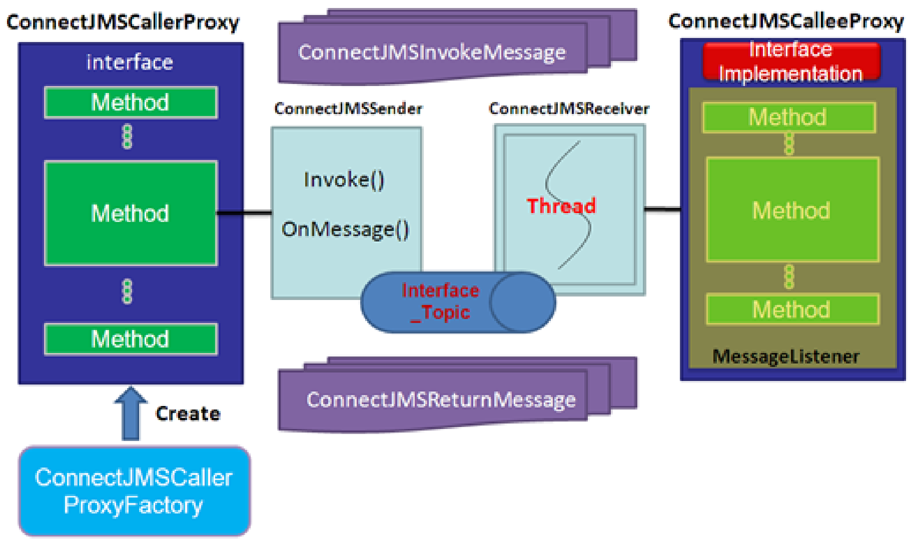


Figure 3.7: The CONNECT JMS Wrapper

We use the Publish/Subscribe model of JMS to implement the architecture. Currently it is assumed that there should be only one instance of a callee at a certain time on the network. In other words, all the requests to an interface will be handled by a unique instance of the enabler implementing the interface. However, since the Publish/Subscribe model supports multiple receivers, our framework can be extended to support multiple callees. We now discuss individual elements of the JMS wrapper framework in turn.

Interface Topic

Topics are administered objects offered by JMS for message transfer between publishers and subscribers. A topic is configured by the administrator of JMS clients, after which a publisher can publish message to the topic while all subscribers to this topic can retrieve the message. In the JMS Wrapper framework, we set up a dedicated topic for each enabler interface and the topic's name is the same as the interface's qualified name. Caller and callee of an interface are both publisher and subscriber to the corresponding topic. Callee subscribes to the topic and listens for method invocation messages. When caller invokes a method, it first publishes a message to the topic and then subscribes to the same topic waiting for return messages. When callee receives an invocation message, it first invokes the method and gets the return value and then publishes the return message to the topic. Note that we leverage the MessageID and CorrelationID offered by JMS to distinguish request and return messages of one topic. After caller publishes a request message, the JMS system will assign to this message a unique ID, which callee can use to fill the CorrelationID field of the return message. When caller subscribes to the topic, it filters message by the condition that the CorrelationID equals the messageID of the request message it publishes.

ConnectJMSInvokeMessage

When caller is to invoke a method, it creates a `ConnectJMSInvokeMessage` with required information and publishes it using the ObjectMessage offered by JMS to the interface topic. It is necessary to point out that only serializable objects can be transferred by JMS ObjectMessage, which means `ConnectJMSInvokeMessage` must implement the serializable interface. This requirement further indicates that all parameter types of the method have to be either primitive or serializable.

`ConnectJMSInvokeMessage` contains the following information: i) the name of the method to be invoked, ii) the types of each parameter, and iii) the concrete object realising each parameter value.

ConnectJMSReturnMessage

The outcome of a method invocation has three possibilities: non-return, return an object, or throw an exception. We abstract this information into the element `ConnectJMSReturnMessage`, and this consists of:

- *returnObj*. This object represents either the concrete return object of method return value or the exception thrown by the method. If returnObj is null, it means this is a non-return method.
- *isException*. This is a boolean value indicating whether the returnObj is an exception or a return value.

When a callee finishes handling a method invocation request, it creates a `ConnectJMSReturnMessage` with the needed information and publishes it using JMS ObjectMessage to the interface topic.

ConnectJMSCalleeProxy

ConnectJMSCalleeProxy is a proxy object for the callee to deal with the connection and subscription routines of JMS. When instantiated by the callee with an interface name and the concrete implemented object of the interface, CalleeProxy will connect to the JMS provider and subscribe to the corresponding topic. As CalleeProxy is implemented with the MessageListener interface offered by JMS, `onMessage()` will be invoked once messages are published to the topic. In `onMessage()`, CalleeProxy will use the ConnectJMSReceiver to handle the invocation requests.

ConnectJMSReceiver

This element is responsible for handling the method invocation request. It is implemented as a separate thread so that the callee can handle more than one request at the same time. Here we leverage the Java reflection mechanism to invoke a method by its name, parameter types and parameter concrete objects, which can be obtained from ConnectJMSInvokeMessage. After getting the result of the method invocation, ConnectJMSReceiver creates the ConnectJMSReturnMessage and publishes this to the interface topic.

ConnectJMSCallerProxy

A goal of the JMS Wrapper framework is to make a caller invoke remote methods the same way as local ones. ConnectJMSCallerProxy is the element provided to achieve this goal. For each interface there is a corresponding ConnectJMSCallerProxy object implementing it. In each method implementation of CallerProxy, a ConnectJMSInvokeMessage is created with the essential information for method invocation. Then the CallerProxy instantiates a ConnectJMSSender object for the interface qualified name. The invokeMessage method when called uses the `invoke()` method of ConnectJMSSender to do the remaining tasks. Meanwhile, CallerProxy itself will be blocked until `invoke()` returns. Note that CallerProxy has no relationship with JMS while ConnectJMSSender is performing the JMS operations.

ConnectJMSSender

ConnectJMSSender is responsible for sending invocation messages to the interface topic and receiving the return message. When instantiated by CallerProxy, it connects to the JMS provider and prepares for sending messages. When the `invoke()` method is called, it first publishes the ConnectJMSInvokeMessage to the interface topic and then subscribes to the same topic with a filtering condition. ConnectJMSSender implements the MessageListener interface so that once the expected ConnectJMSReturnMessage is returned to the interface topic, `onMessage()` will be invoked, after which ConnectJMSSender can pass the message back to CallerProxy.

Here we utilise the Java synchronize mechanism to realise blocking invocation. ConnectJMSSender has a "lock" object. When the `invoke()` method finishes publishing a message and subscribing to the interface topic, it will wait on this "lock" object. After `onMessage()` is executed, the "lock" object will be notified so that the `invoke()` method can continue and return the ConnectJMSReturnMessage.

ConnectJMSCallerProxyFactory

In order to facilitate the use of CallerProxy, the framework provides a ConnectJMSCallerProxyFactory element to create CallerProxy by interface name. When caller needs to invoke a method of some interface, it can use this factory to create the corresponding CallProxy and then invoke the method of this CallerProxy.

Figure 3.8 illustrates the whole process that caller employs within the JMS Wrapper Framework for remote method invocation.

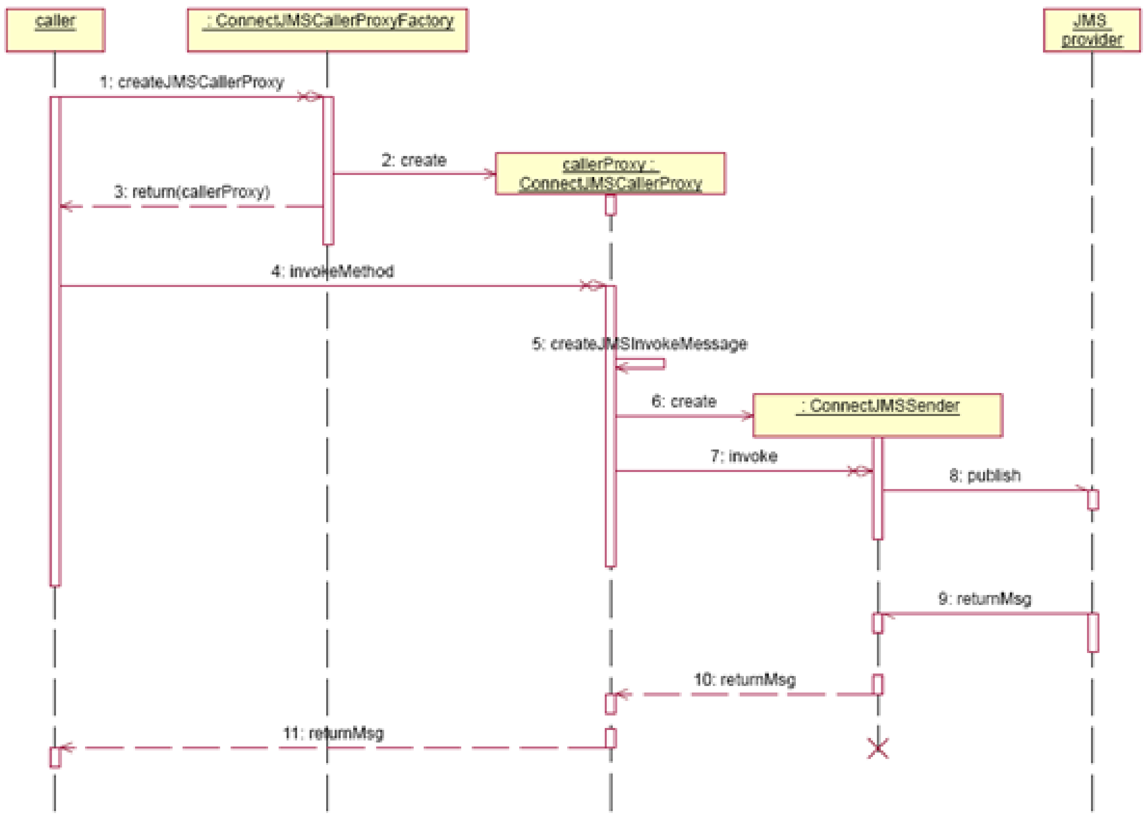


Figure 3.8: Message sequence for Callee interaction using the JMS Wrapper

3.4 Refining the CONNECTor Architecture

The intermediate CONNECTor architecture as defined in Y2 (and illustrated in Figure 2.1 in Section 2.2) illustrated the building blocks of each CONNECTor deployed between two networked systems. However, the focus was from a higher level and there remained open design questions to answer (in particular, related to how CONNECTors are created and executed). The following refinements were made during the third year:

- the CONNECTor architecture was refined to be underpinned by the interpretation of automata models using the Starlink framework [7] (this is fully documented in Section 3.4.1);

- the specification of the direct interactions between the CONNECTors and the enablers in order to perform the required functionality to achieve the non-functional requirements of the CONNECTor between two networked systems (these refinements are illustrated in Section 3.4.2).

3.4.1 Realising CONNECTors

Starlink (as introduced in Deliverable D1.2 [4]) is a software framework that takes as input a set of models that describe the required functionality of a CONNECTor:

- *Message Description Languages*. Three languages (Binary MDL, Text MDL, and XML MDL) are used to describe the content, format and encoding of protocol messages. These define the message types in terms of the fields and their types. These specifications are used to parse and compose messages; parsing creates the `AbstractMessage` data structure from the message content, and composing creates the byte packet from the `AbstractMessage`.

- *k-Coloured Automaton*. A coloured automaton specifies the transitions required to interact with a networked system in terms of sending and receiving messages only. The colour annotates the automaton with the concrete information required to communicate with systems, i.e., the colour adds information about the protocol employed by each transition, and also which MDL specification is required to parse a received message, or compose a message to send.

- *Merged k-Coloured Automaton*. A merged k-Coloured automaton describes the merge of two k-coloured automata. Bridge states are added to sending and receiving states to allow translation logic (e.g. the translation of data content) to be executed. This model is executed to perform mediation behaviour.

Starlink provides a small set of Java classes that interpret these models in order to support interoperability solutions. These classes are at the heart of implementing the concepts of the CONNECTor architecture that was previously re-introduced in Section 2, namely: listeners, actuators, bridges, the network engine and k-coloured automaton. The general architecture remains consistent with that presented in Y2 with minor refinements due to the implementation enhancements.

Figure 3.9 illustrates an implementation viewpoint of the CONNECTor architecture. Here, the Starlink classes that form a single, complete CONNECTor implementation instance are seen. Here, the diagram shows the Java objects and the method dependencies between them, i.e., a directed arrow is a method invocation (as labelled on the diagram) from one object to the other.

The central element in a CONNECTor instance is the **automata engine**. This is the Starlink interpreter that reads the models that specify the CONNECTor and execute the CONNECTor by co-ordinating the other Starlink objects (we describe this co-ordination later). Each CONNECTor has one instance of the automata engine. The code in Figure 3.10 demonstrates how each CONNECTor is instantiated and then executed. First, the CONNECTor requires three models:

1. The k-coloured automaton concrete specification of the first networked system. The description of k-coloured automata is provided in Section 5.4.1.

2. The k-coloured automaton concrete specification of the second networked system.

3. The k-coloured automaton describing the merge (i.e. the mediation) of the two previous automata.

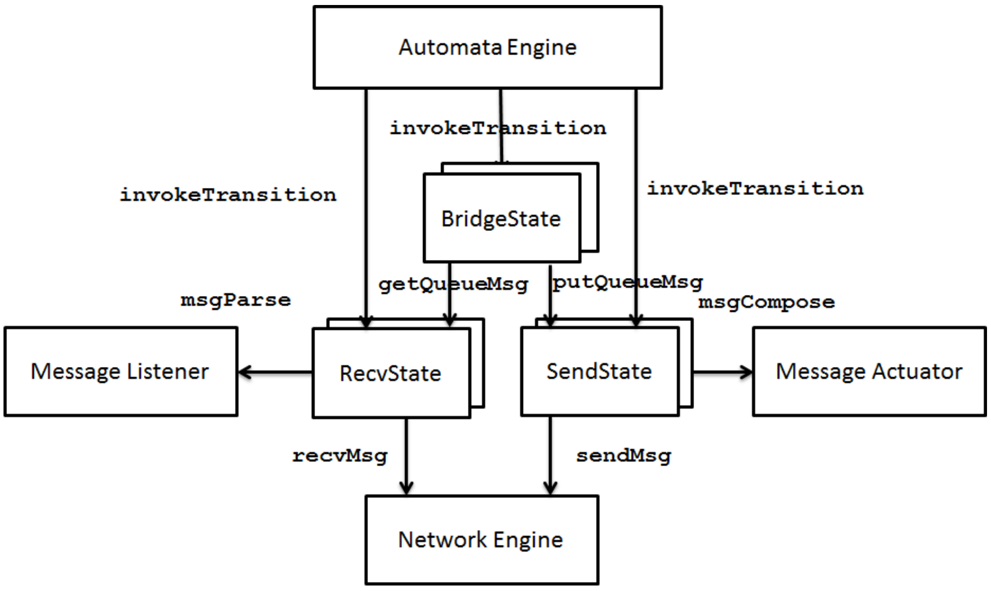


Figure 3.9: Starlink objects which compose a CONNECTor

```
/**
    * Reference to the three models that describe the application connector.
    */
        String automataModels = new String[3];
        automataModels[0] = ``XMLPhotoSharing.xml'';
        automataModels[1] = ``IIOPPhotoSharing.xml'';
        automataModels[2] = ``XMLIIOPPhotoSharingMediator'';

        /**
        * Create the connector of type XMLPhotoIIOPPhoto.
        */
        AutomataEngine r = new AutomataEngine(``XMLPhotoIIOPPhoto'');
        for (String str : automataModels) {
            r.addAutomata(str);
        }

        /**
        * Create a unique instance of the connector using the given id.
        */
        r.execute( "XMLPhoto1IIOPPhoto1");
```

Figure 3.10: Complete code listing for constructing a CONNECTor

Hence, to create a CONNECTor the `AutomataEngine` class is instantiated and the three models are added using the `addAutomata` method. This operation instantiates a set of objects related to the states that are parsed in the models; these objects are of three class types: `recvState`, `sendState` and `bridgeState`. For example, if a mediator constists of a state that receives a message, followed by a state that bridges, followed by a state that sends a message, then one instance of each is created. This forms a complete CONNECTor (as shown in Figure 3.9). We now illustrate how a CONNECTor executes using these objects.

When the automata engine interprets a state transition in the model specification it invokes the invokeTransition method implemented by the State object at the current position in the automaton. Figure 3.11 highlights the behaviour when `invokeTransition` is called on a `recvState` object. First, the `recvState` object listens for incoming connections and messages. For this it calls the `Listen` method on the Network Engine object. Subsequently it calls the `recvMessage` method of the network engine for which it receives the message sent from a networked system in the original network packet format. The `recvState` object then invokes the `messageParse` operation on a given Message Listener instance (if the packet type is SOAP an instance for parsing SOAP messages is utilised, hence there can be multiple instances of listeners and actuators in any given CONNECTor). The abstract message representing the packet is returned and stored locally at the message queue of the state object. The `recvState` then selects the next transition to take (based upon the abstract message information, i.e., message type) and returns a reference of the state to move to.

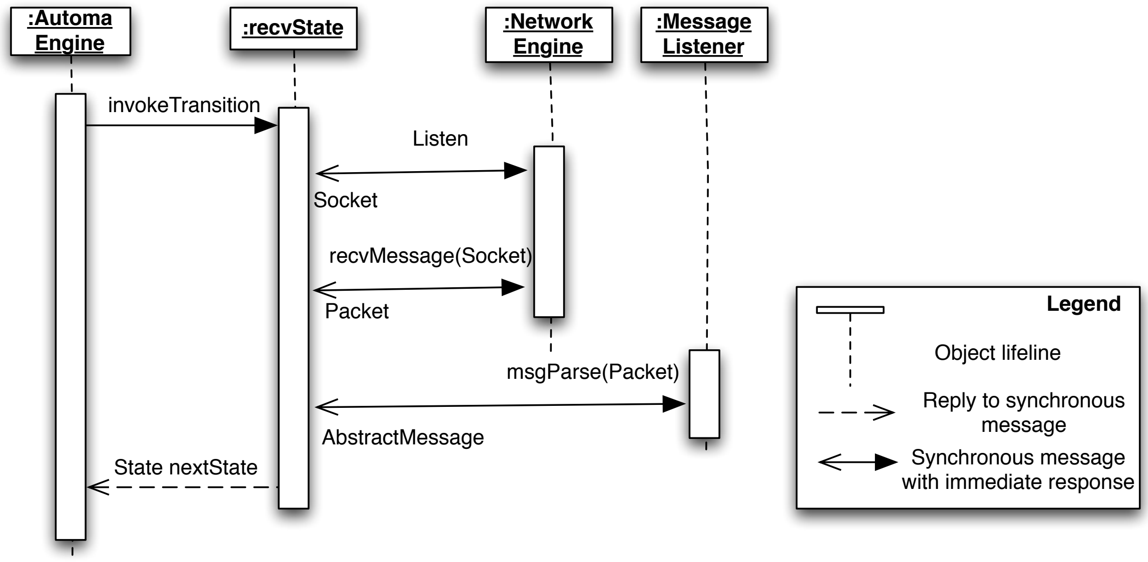


Figure 3.11: The message sequence when invokeTransition is called at a recvState

When `invokeTransition` is called on a `BridgeState` (seen in Figure 3.12) it executes the Message logic specified at the state in the Message Translation Language specification of the merged k-Coloured Automaton. For example, If one of the translation logic operations is a field assignment operation then the `BridgeState` reads an Abstract message from a specified `RecvState` (using the `getMessageFromQueue` method), performs the translation and then places this in the required `SendState` message queue by invoking the `putMessageinQueue` method. Hence a translated message is ready to be sent to a networked system when the `invokeTransition` of the `SendState` object is called.

Finally, when `invokeTransition` is called by the automata engine on a `SendState` object instance. First, the `SendState` object invokes the `MessageCompose` method of a message actuator instance. This converts the abstract message to the network packet format of the required protocol. The object then interacts with the network engine to open a connection to the networked system and then send the network packet bytes directly. The `SendState` finishes by returing the next state to be invoked to the automata engine.

3.4.2 Underpinning non-functional concerns

The management of non-functional requirements by the CONNECT architecture was introduced in the refinements to the Enabler architecture, in particular the behaviour performed during the Connectability phase. The key pattern is observation and adaptation, i.e., the behaviour of the CONNECTor is monitored by the enablers, and when the enablers detect a non-functional requirement not being met then they adapt the CONNECTor.

In the previous sections we saw how adaptation is performed by the enablers. They receive event data and make decisions about when to change the CONNECTor and produce the necessary specifications to resynthesize a new CONNECTor that replaces the original. The original is safely stopped and then the new one is deployed and started.

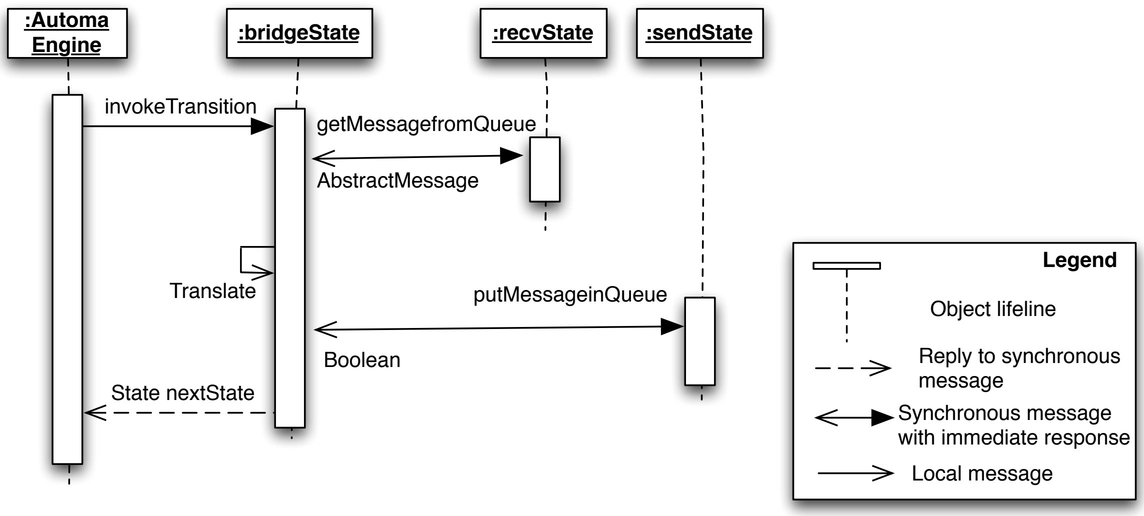


Figure 3.12: The message sequence when invokeTransition is called at a bridgeState

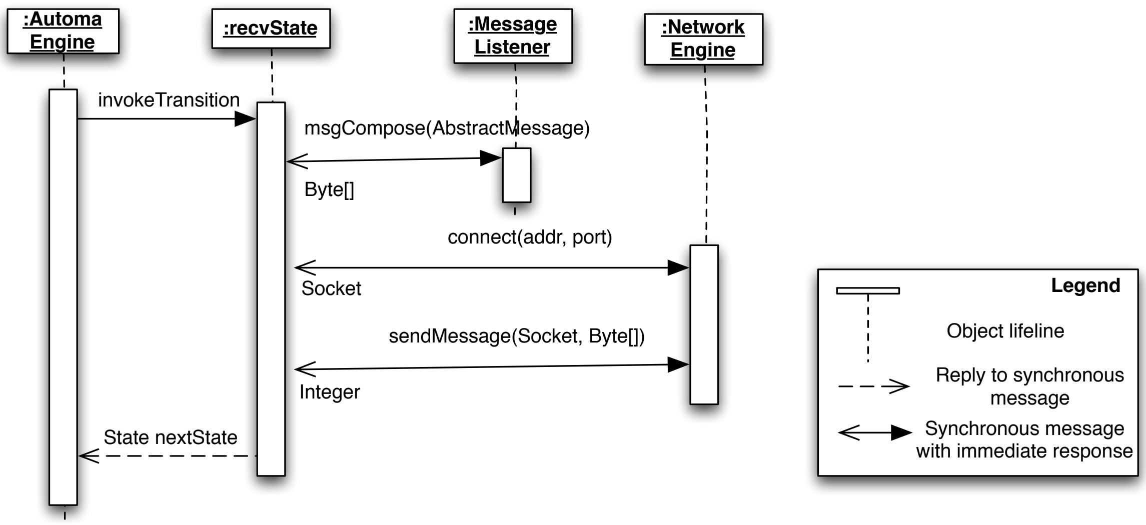


Figure 3.13: The message sequence when invokeTransition is called at a sendState

However, we have yet to discuss how the enablers interact with the CONNECTors. That is, how are the enablers informed of the events that offer important information about the context of CONNECTor invocation? Further, how can code be inserted into the CONNECTor to enforce non-functional policies, e.g., the insertion of security probes that enforce security policies?

To answer both of these questions, we utilise a simple approach that inserts *probes* (small pieces of java code) at *join points* within the CONNECTor architecture. Where a join point describes a method invocation (or its exception handler) of one object within the CONNECTor to another.

There are two types of probe:

- *A Glimpse probe*. An example of the code for this probe is shown in Figure 3.15. This constructs an event using the information from the method invocation, i.e., that this is an `invokeTransition` and hence an there has been a state transition in the automaton. The message is then sent to the Glimpse monitoring bus so that it can be received by the monitoring enabler.

- *A Security Probe*. This is a similar piece of code to the Glimpse probe. It is similarly inserted at join points in the CONNECTor architecture. This code will be produced in the final year of the project, and then integrated into the running CONNECTors.

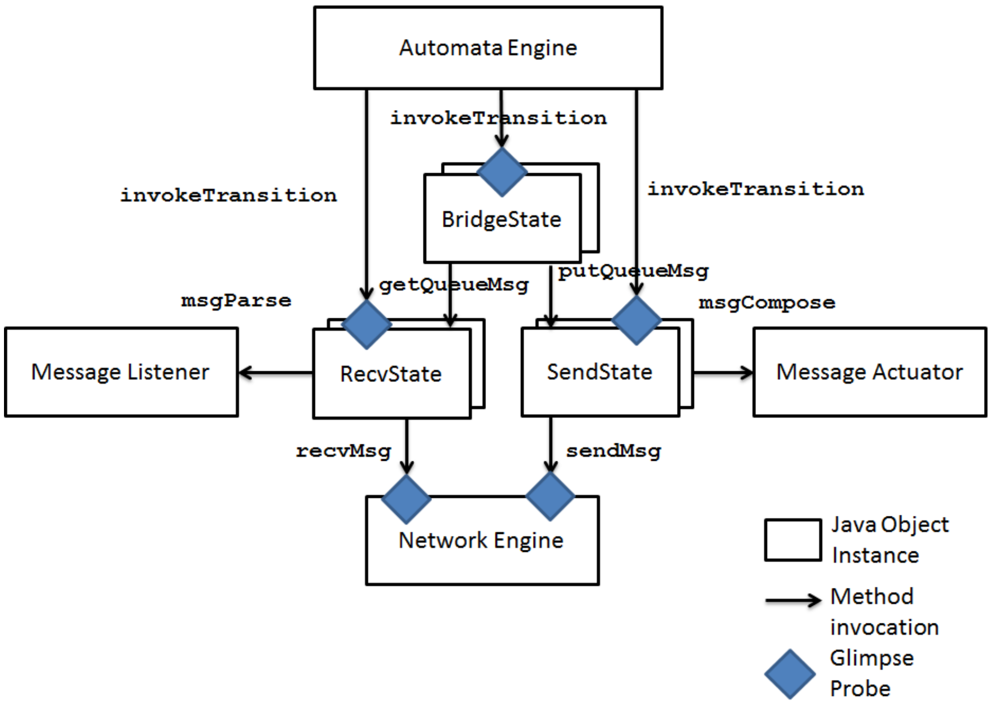


Figure 3.14: Join points where Glimpse probes are inserted into the CONNECTor architecture

Figure 3.14 demonstrates the join points in the CONNECTor architecture where the Glimpse probes are inserted. Note insertion means the code is embedded into the source of the actual method (i.e. it is an invasive insertion). These methods are the three `invokeTransition` methods of the different state type objects. Hence, the probes monitor when the state transitions occur. Further probes are inserted into the Network Engine object on the send and receive methods in order that the physical messages can also be monitored.

3.5 Summary

In this section we have introduced the revised CONNECT architecture. This highlights the progress that has been made during the third year towards producing a fully integrated and concrete architecture. The key contributions of this section in achieving this progress are:

- *An integrated Enabler architecture*; we have documented the interactions between the independently developed enablers and discussed the implementations of these connections. The further implementation details of each individual enabler within this integrated architecture is presented in Section 4.

- *An integrated* CONNECT*or architecture*; we have highlighted the behaviour and concrete implementation of each CONNECTor. In particular, we have shown how the Starlink framework is applied to create and execute a single CONNECTor instance. Furthermore, we have introduced the methods to ensure that CONNECTors can communicate with enablers in order to perform tasks related to the monitoring and future adaptation.

```
MyGlimpseProbe aGenericProbe = new MyGlimpseProbe(Manager.createProbeSettingsPropertiesObject
                ("org.apache.activemq.jndi.ActiveMQInitialContextFactory",
                "tcp://atlantis.isti.cnr.it:61616","system","manager","TopicCF",
                 "jms.probeTopic",false, "probeName", "probeTopic"));

try {
  //to send a generic String GlimpseBaseEvent, the data field should be the label of the LTS
  GlimpseBaseEvent<String> message = new GlimpseBaseEventImpl<String>("asd", "connector1","
      connInstance",                                          "conninstexec",123,122,System.
      currentTimeMillis(),"NS1", false);
  message.setData("transition:"+this.getStateLabel());
  aGenericProbe.sendMessage(message, false);
}
catch(IndexOutOfBoundsException asdG) {
  GlimpseBaseEvent<Exception> exceptionEvent = new GlimpseBaseEventImpl<Exception>(
        "e", "connector1","connInstance", "conninstexec",123,122,System.currentTimeMillis(),"NS1
            ", true);

  exceptionEvent.setIsException(true);
  exceptionEvent.setData(asdG);

  aGenericProbe.sendMessage(exceptionEvent, false);

}
```

Figure 3.15: Code listing of a Glimpse probe

4 The Enablers

During the third year of the project, further progress has been made towards the finalised implementations of the enablers that were originally outlined in the second year of the project (and described in Section 2.4). In this section, we discuss the individual details of each enabler; exploring further how each enabler operates in order to achieve the required behaviour as outlined in the Enabler architecture described previously. Importantly we also present the specifications of each of the interfaces either provided or required by these software components These interfaces form the basis of the integration of the enablers into the concrete Enabler Architecture as specified in Section 3.2.

4.1 The Discovery Enabler

Calls	`learnCNSInformation` method of the **Learning enabler** to acquire behavioural descriptions, and the `synthesizeAbstractMediator` method of the **Synthesis enabler** in order to initiate synthesis after finding a potentially compatible pair of NSs.
Called by	The **Learning enabler** returns an updated model using the `updateNetworkedSystemBehaviour` method. The **Deployment enabler** calls repository methods such as `getConnector`.

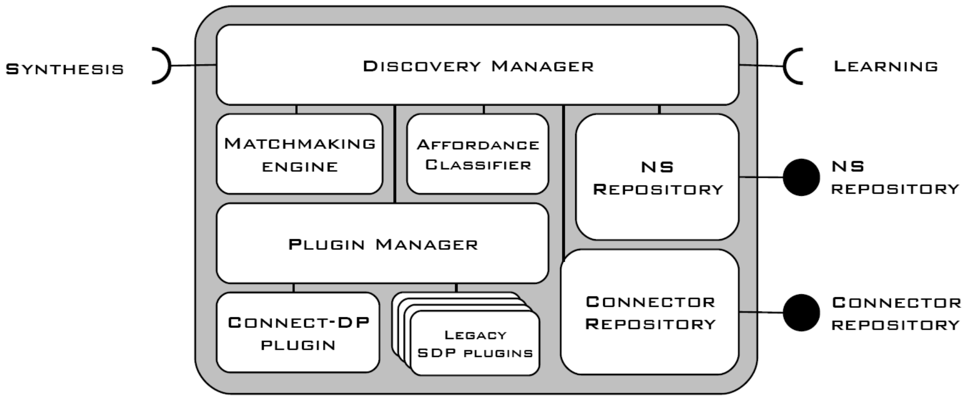


Figure 4.1: Discovery enabler architecture.

While the overall design and purpose of the CONNECT discovery enabler remains that described in Deliverable 1.2, the prototype implementation has been enhanced in several respects:

1. Integration with the synthesis and learning enablers according to their declared interfaces (as above).
2. Implementation of a *connector repository* and exposure of an additional interface to support the querying of that repository.
3. Means to provide a goal (Section 5.3.1).
4. The internal representations of networked systems and CONNECTors have been refined in order to facilitate their reuse throughout the CONNECT architecture.
5. An *affordance classifier* has been integrated to facilitate the work described in Section 5.2. When a networked system without an affordance description is discovered, the discovery enabler passes the interface description of the NS to the classifier, which then provides an affordance.

Points 2 to 5 are detailed in the subsequent sections. First we briefly recall the functionality and architecture of the Discovery enabler, as shown in Figure 4.1. The discovery manager consists of plugins and a plugin manager which implement various legacy (and CONNECT-specific) discovery protocols such as DPWS-WSDD; repositories to store NS and CONNECTor descriptions; a matchmaking engine to find compatible pairs of NSs; and an affordance classifier that infers the affordance of an NS from its interface description. The new components with respect to D1.2 are the affordance classifier and the CONNECTor repository. The Discovery enabler now performs several steps upon the (passive) discovery of a new NS:

1. if the NS description does not include an affordance, the affordance categoriser is invoked to infer one;
2. if the description does not include a behavioural model, the Learning enabler is invoked to provide one (see Section 4.2 and D4.3 [13]);
3. affordances are compared using the domain ontology to find pairs of NSs which are likely to interoperate (this search can be refined by only considering affordances related to a user-provided goal);
4. the set of pairs of NSs that have matching affordances is further restricted by taking only those for which a mediator can be synthesised;
5. the fully matching pair is composed using the synthesised concrete CONNECTor, and a CONNECTor description (ConnectorInstance) is inserted into the CONNECTor repository by the deployment enabler.

4.1.1 Repositories and shared representations

In order to facilitate smooth integration and reduce unnecessary duplication of effort, it was decided that the internal implementation-level representations (that is, Java objects) of networked systems and of CONNECTors would be available for use where relevant throughout the architecture. In particular the Learning enabler is concerned with extracting the interface description from the NS representation, and adding a behavioural description to it, while the deployment enabler is concerned with the runtime status of connectors.

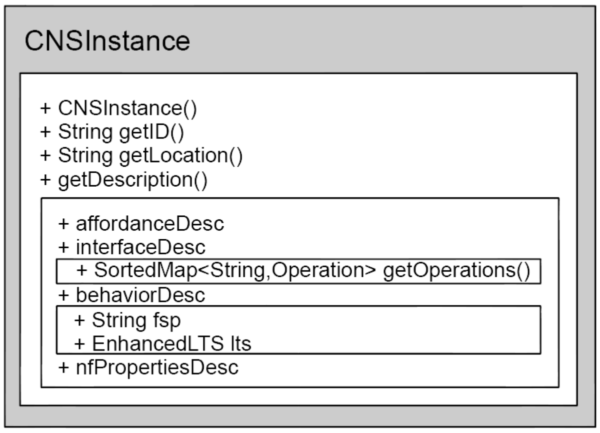


Figure 4.2: CNSInstance shared representation.

The shared representation of a networked system is the CNSInstance class, shown in Figure 4.2. It contains an ID and a network address, along with representations for the four parts of the NS model. In particular, the interface description stores a list of operations, while the behaviour is stored as an EnhancedLTS (see Section 4.4). CNSInstances are stored in the NS repository component of the Discovery enabler and can be accessed through the NSRepositoryInterface:

```
public interface org.connect.enablers.discovery.repository.NSRepositoryInterface
{
  public abstract CNSInstance getNSInstance(String nsID);
  public abstract CNSAffordanceDesc getNSAffordance(String nsID);
  public abstract CNSInterfaceDesc getNSInterface(String nsID);
  public abstract CNSBehaviorDesc getNSBehaviour(String nsID);
  public abstract CNSNFPropertiesDesc getNSNFProperties(String nsID);
  public void updateNetworkedSystemBehaviour (CNSInstance ns);
  public void updateNetworkedSystemBehaviour (String nsID, EnhancedLTS lts);
}
```

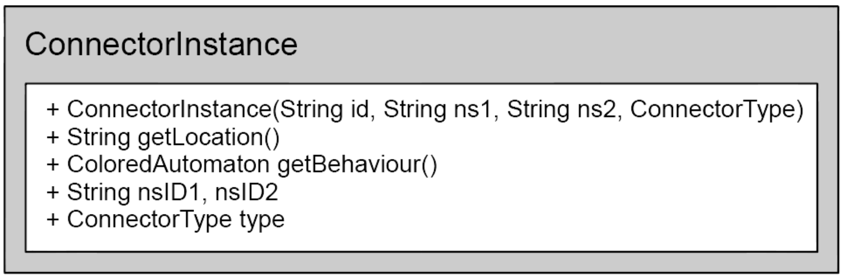

ConnectorInstance
+ ConnectorInstance(String id, String ns1, String ns2, ConnectorType)
+ String getLocation()
+ ColoredAutomaton getBehaviour()
+ String nsID1, nsID2
+ ConnectorType type

Figure 4.3: ConnectorInstance shared representation.

The shared representation of a CONNECTor is the ConnectorInstance class, shown in Figure 4.3. The ConnectorInstance contains the IDs of the two NSs it connects, a ColoredAutomaton representation of the synthesised mediator (see Section 5.4.1), and a network address. ConnectorInstances are stored in the CONNECTor repository component of the Discovery enabler and can be accessed through the ConnectorRepositoryInterface:

```
public interface org.connect.enablers.discovery.repository.ConnectorRepositoryInterface
{
  public ConnectorInstance getConnector(String connID);
  public ConnectorInstance  getConnector(String nsID1, String nsID2);
  public void add(ConnectorInstance c);
  public void getConnectorType(String nsType1, String nsType2);
}
```

Both CNSInstances and ConnectorInstances have associated types. In the case of CNSInstance, the type is accessible through the `getDescription()` method. In the ConnectorInstance the type is accessible through the `type` field. The key difference between instances and types is that instances represent actual (running) artefacts with a network address. There may be several NSs at different addresses which 'implement' the same type. Equally there may be several connectors at different addresses which are of the same type. The relationships between types and instances are shown in Figure 4.4. When a new NS is discovered, address information is removed from the description, making it a type description rather than an instance description. This description is hashed to produce a `descriptionID`. Two NSs of the same type therefore have the same `descriptionID`. In contrast, the output of synthesis (see the `synthesizeAbstractMediator` method of the Synthesis enabler) is in fact a generic ColoredAutomaton which is packaged as a ConnectorType. It is then the responsibility of the deployment enabler to instantiate the type to create a ConnectorInstance on a particular network address, interacting with two particular CNSInstances.

4.1.2 Goal

The GUI of the discovery enabler has been extended to enable the entry of a user-defined goal. This goal is expressed in the temporal logic language defined in Section 5.3.1. When considering candidate NSs for connection, the Discovery enabler finds the subset of NSs in the repository that provide or require the affordance associated with the goal, then queries the

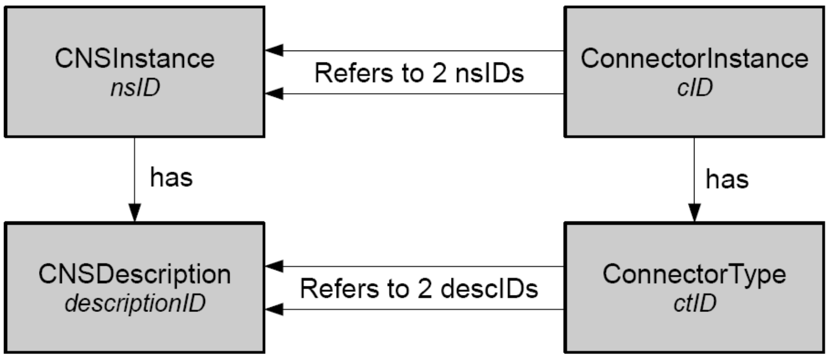


Figure 4.4: Relationships between types and instances.

Synthesis enabler for all pairs in this set to determine which pairs can be mediated in a manner that achieves the goal. The set of pairs to be checked can be reduced further by restriction to NSs with alphabets containing the alphabet of the goal (modulo ontological differences).

4.1.3 Affordance classifier

The affordance classifier (an instance of class `Classifier` developed by the University of Trento[1]) is loaded by the Discovery enabler in order to assign the functional concept of an affordance to a networked system that lacks this part of its description. Having been trained beforehand, the classifier uses the interface description of the NS in order to infer its affordance. The classifier is trained offline according to the process outlined in Section 5.2 in which the classifier generalises from examples consisting of WSDL interface descriptions that have been manually assigned affordances.

4.2 The Learning Enabler

Calls	`invoke` method of the **Interaction enabler** to execute methods on a remote networked system in order to test the behaviour. The **Discovery enabler** method `updateNetworkedSystemBehaviour` in order to add newly learned behaviour about a networked system to the networked system model.
Called by	The **Discovery enabler** to run the `learnCNSInformation` method in order to start learning the behaviour of a particular networked system.

The Learning Enabler uses active learning algorithms to dynamically determine the interaction behaviour of a networked system from the initial CNSInstance representation and produces a model of this behaviour in the form of a labeled transition system (LTS); this employs methods based on monitoring and model-based testing of the networked systems to elicit their interaction behaviour. The implementation of the enabler is built upon the LearnLib tool. It utilises the interface description from the Networked System Model as an input, also exploiting the semantic annotations, and an LTS describing the interaction behaviour is produced. This behavioral model is sent back to the Discovery Enabler by using the `updateNetworkedSystemBehaviour` one way message exchange to complete the discovery of the description of the NSs.

To perform active learning, the enabler must interact with the NSs directly, i.e. use the legacy protocols to invoke operations on particular application services. The Learning enabler uses the invoke interface provided by the Interaction enabler to perform this behaviour.

[1] http://disi.unitn.it/~johansson/wsdl/

Finally, the Learning enabler monitors the deployment and operation of CONNECTors to continuously learn about the behaviour of the networked system. For this it subscribes to the global Deployment and Monitor information channel and the events are received on the corresponding interfaces. A detailed description of the implementation of the Learning enabler and its functionality is provided in Deliverable D4.3 [13].

```
public interface LearnNS {
  /**
   * Starts the learning of a networked system's behavior. The learning process is
   * started asynchronously, hence this method returns immediately. Upon completion,
   * the behavioral model will be stored into the CNS repository of the discovery
   * enabler.
   * @param cnsId the identifier of the networked system to be learnt.
   */
  public void learnCNSBehavior(String cnsId);

}
```

4.3 The Interaction Enabler

Calls	The Interaction Enabler does not call any other Enabler.
Called by	Both the **Learning Enabler** and the **Trust Enabler** call the `invoke` method of the `Invoke` interface to interact directly with a networked system. Hence, the Interaction enabler can call operations provided by each networked system's application interface.

The Interaction enabler provides an interface (`Invoke`) that is listed below. This interface consists of the `invoke` method which provides the capability to interact directly with the services provided by a single networked system. For example, if a networked system provides an operation `print(document)` through its remotely accessible API then the Interaction enabler is able to invoke such an operation. The `invoke` method takes a `URL` as the first parameter; this contains the network location of the provided service and the concrete protocol required to interact with it. The second parameter is the `actionlabel`, which is the application label to be performed by the interaction (e.g. `print`). The `parameters` parameter is a list of Objects that map onto the parameters of the application interaction (e.g. `document`), and finally the `returnTypes` parameter is the list of return types that are required for each of the result parameters returned by the method invocation.

```
public interface Invoke {

    /**
     * Dynamic invocation interface to invoke services implemented using
     * different middleware protocols.
     * @param URL The address of the network system to invoke dynamically.
     * @param operation The label of the operation to invoke e.g. method name of an rpc call.
     * @param parameters The list of input parameters for the operation
     * @param returnTypes The required types of the returned parameters as an ordered list.
     * @return the outcome of the invocation as a list of Java objects
     */
    public Object[] invoke(URL serviceID, String operation, Object[] parameters, String[]
        returnTypes) throws Exception;
}
```

The key feature of the Interaction enabler is that is can interact with networked systems irrespective of the middleware protocols they utilise for interaction. For example, if a networked system provides `print` as a SOAP service, the Interaction enabler will translate the information from the `invoke` call into a SOAP request. If a different networked system provides `print` as a CORBA service, the Interaction enabler will operate using a CORBA request. Hence, the Interaction enabler determines the legacy protocol (from the information in the `URL` parameter) and then constructs the legacy protocol request from the information in the `actionlabel` and `inputParameters` parameters. After an interaction with a networked system has been performed, the result of the interaction (as typically returned in a service response) is mapped onto the return parameter (which is an Object[] list) of the `invoke` method.

The implementation of the Interaction enabler is based upon the capabilities provided by the Starlink framework [7]. Starlink is an interpreter that executes application actions defined as a coloured automaton; that is, it performs an application request upon a given middleware protocol based upon the information provided in the automaton. A detailed account of the procedure is provided in Section 5.4.3 and hence is not duplicated here.

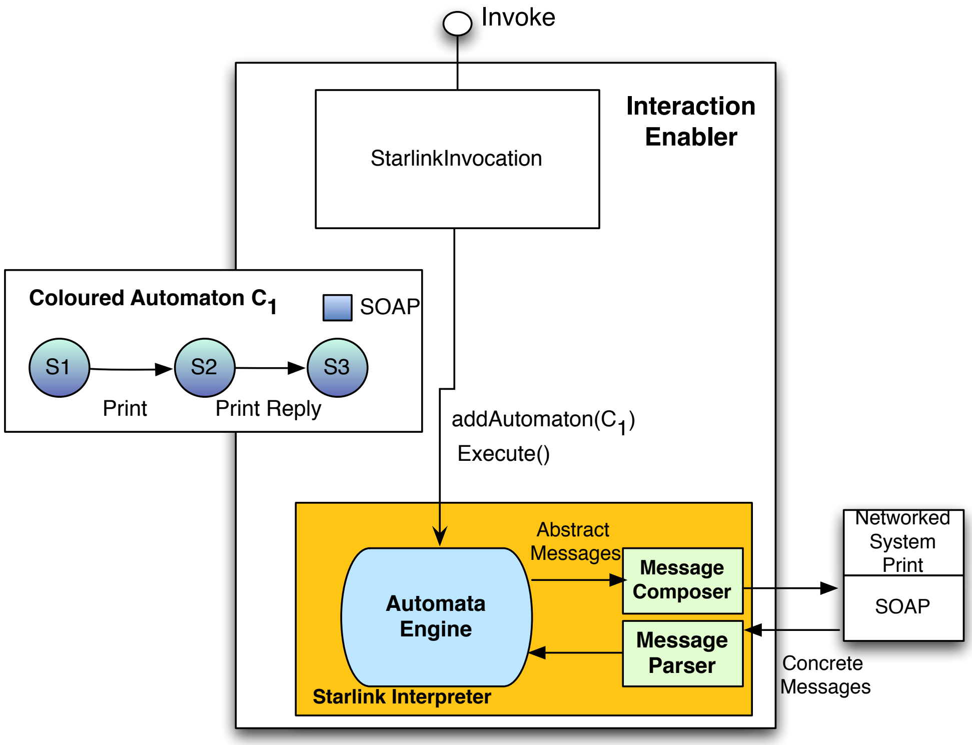


Figure 4.5: Implementation of the Interaction Enabler

Figure 4.5 illustrates the implementation of the enabler, and indicates how it operates in practice. The `Invoke` interface is implemented by an object called `StarlinkInvocation`. This implementation simply takes the data from the `invoke` method and constructs a new coloured automaton to be executed. For example, where the `print` operation is implemented as a SOAP service the invocation must be a request-response sequence, and hence a coloured automaton (C_1) with a *send* transition followed by a *receive* transition is constructed. The labels of the transitions carry the information about the call to be made (e.g. print). Starlink is then used to execute the created coloured automaton, which results in the networked system being utilised. The `addAutomaton` method is called with the newly created colour automaton C_1, and then the `execute` operation is called to run it. On completion, the `StarlinkInvocation` object extracts the result information from the final automaton transition and returns this information as the result of the `invoke` method call.

The StarlinkInvocation object also performs basic resource management in order that constructed coloured automaton can be re-used where invocations are repeated. The `URL` and `actionLabel` parameters identify the instance of an invoke call and are used to retrieve a previously constructed automaton. Where an automaton remains unused for a specified length of the time (a defined parameter of the Interaction enabler) it is released from the memory store.

4.4 The Synthesis Enabler

Calls	`Deploy` method of the **Deployment enabler** to executes the mediator model. **DePer**, **Security** and **Trust enablers** in order to analyse the produced mediator model.
Called by	the **Discovery enabler** to launch the `synthesiseAbstractMediator` method in order to initiate synthesis after finding a potentially compatible pair of NSs.

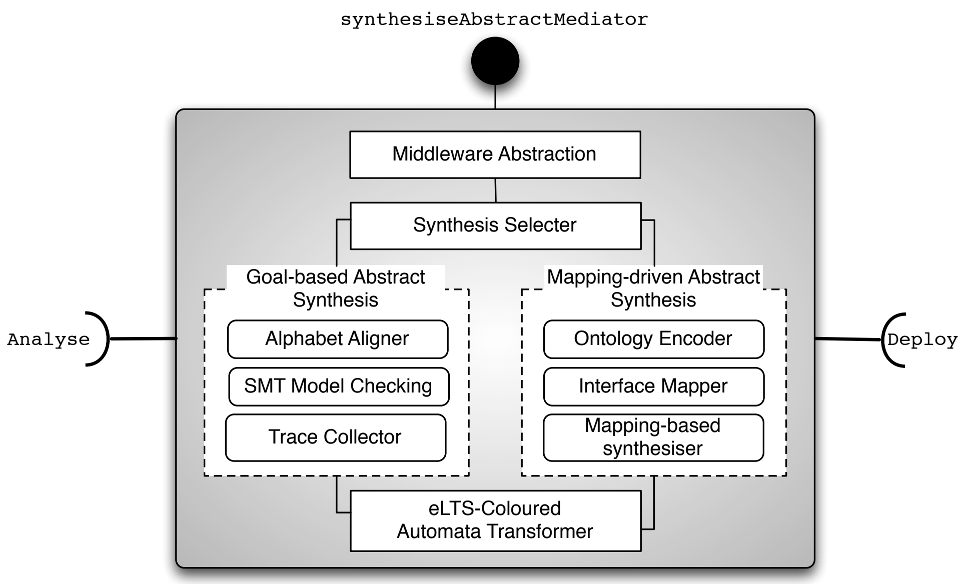

synthesiseAbstractMediator
Middleware Abstraction
Synthesis Selecter
Goal-based Abstract Synthesis
Alphabet Aligner
SMT Model Checking
Trace Collector
Mapping-driven Abstract Synthesis
Ontology Encoder
Interface Mapper
Mapping-based synthesiser
eLTS-Coloured Automata Transformer
Analyse
Deploy

Figure 4.6: Synthesis enabler architecture.

The Synthesis enabler (as shown in Figure 4.6) takes the Enhanced LTS models of a pair of networked systems as provided by the Discovery enabler and constructs a CONNECTor model. Two approaches to mediator synthesis are implemented. The *Synthesis Selector* chooses the most appropriate one, at the moment this is based on the existence of the goal. These two approaches relies on the same modules for middleware abstraction and to transform the mediator model into a k-coloured automata.

The Synthesis enabler verifies that this CONNECTor meets the non-functional requirements of the interoperation between the two NSs by calling the `Analyze` method as provided by the three Connectability enablers, i.e., DePer, Security and Trust.

Then, it sends this model to the Deployment enabler in order for the connection between the networked systems to be realised. A detailed description of the implementation and interface of each individual block is included in D3.3 [14].

The method to be invoked in order to synthesise the CONNECTor is the `synthesiseAbstractMediator` of the interface `Syhthesise`:

```
public interface synthesise{
 /**
 * Generates the mediator models, sends it to the deployment enabler.
 * @param The eLTS representation of NS1
 * @param The eLTS of NS2
 * @param A list of LTL formulas that specifies the goals that the mediator should verify
 * @return The The XML model of the mediator between NS1 and NS2
 * @throws if fails to generate the mediator
 */
 public ColouredAutomaton synthesiseAbstractMediator(EnhancedLTS ns1, EnhancedLTS ns2, List<
     LTLFormula> goal)
```

```
  throws NoMediatorException;
}
```

The `EnhancedLTS` and the `ColouredAutomaton` are the Java objects implementing the eLTS and the k-coloured automaton respectively, which are both defined in Section 5.4.2. If the synthesis is unable to generate the mediator or if the Connectability analysis reveals possible failures regarding non-functional properties, then the enabler raises a `NoMediatorException` exception.

In the current stage, the synthesis only sends the CONNECTor to the Connectability enablers in order to analyse it. Nevertheless, we envision the integration between synthesis and the Connectability enablers so as to enable the incremental re-synthesis of CONNECTors.

4.5 The Deployment Enabler

Calls	the `notification` mechanism of the **Monitoring enabler** to notify subscribers of a newly deployed CONNECTor. the `synthesiseAbstractMediator` method of the Synthesis enabler in order to create a replacement CONNECTor.
	the `add` method of the **Discovery enabler's** repository interface to add a record about a newly created `ConnectorInstance` describing the CONNECTor.
Called by	the **Synthesis enabler** to deploy a synthesised CONNECTor using the `Deploy` method.
	the **DePer, Trust or Security** enablers with a warning that the CONNECTor has violated a non-functional property.

The Deployment enabler takes the concrete CONNECTor models produced by the Synthesis enabler and constructs a CONNECTor that can be executed by the Starlink framework. The CONNECTor is a set of Java classes that are bundled using OSGi. These classes are described in further detail in the description of the CONNECTor architecture in Section 3.4. The OSGi bundle is then distributed to a CONNECT enabled host (i.e. one that is running an OSGi engine and can be connected to by the Deployment enabler) and the CONNECTor instance can be created and executed using the code in Figure 3.10. Information about this CONNECTor (ConnectorInstance info) is constructed and then added to the repository in the Discovery enabler.

The interface providing the deployment service is listed below:

```
public interface Deploy{
  /**
  * Construct a connector and deploy, deposit the ConnectorInstance into the repository
  * and return the ID of this.
  * @param The XML model of the concrete coloured automaton of NS1
  * @param The XML model of the concrete coloured automaton of NS2
  * @param The XML model of the concrete mediator between NS1 and NS2
  * @return The unique id of the connector instance for this given connector type
  */
  public ConnectorID Deploy(String NS1, String NS2, String mediator);

  /**
  * Construct a new connector and redeploy, deposit the ConnectorInstance into the repository
  * and return the ID of the new connector.
  * @param The unique identifier of the connector to redeploy
  * @param The XML model of the new mediator between NS1 and NS2
  * @return The unique id of the connector instance for this given connector type
  */
  public ConnectorID Redeploy(ConnectorID id, String newMediator);
}
```

The Deployment enabler also serves as the central management point of created and running CONNECTors. That is, it is the central point to which other enablers can make lifecycle management requests such as start, stop and destroy operations. It is also the enabler that manages requests to replace a CONNECTor with an improved version (e.g. through re-synthesis). The

Control interface provides operations to start, stop and destroy deployed CONNECTors, updating the repository in accordance with the subsequent status. Whenever the status of a CONNECTor changes it is also announced on the global channel (Deployment Announcement) for monitoring deployment events. When another enabler wishes to replace the CONNECTor it calls the Redeploy operation on the Deploy interface listed previously.

```
public interface Control{
  /**
  * Start a deployed Connector i.e. begin listening for the first received msg from an NS.
  * @param ConnectorID The unique ID of the Connector to start.
  */
  public void Start(ConnectorID);

  /**
  * Stop a deployed Connector i.e. Stop the current thread of execution and terminate
  * safely.
  * @param ConnectorID The unique ID of the Connector to stop.
  */
  public void Stop(ConnectorID);

  /**
  * Undeploy a Connector and remove the instance data from the repository
  * @param ConnectorID The unique ID of the Connector to destroy.
  */
  public void Destroy(ConnectorID);

  /**
  * Notification to suspend operation of a Connector instance
  * @param ConnectorID The unique ID of the Connector to stop.
  * @param WarningType The type of violation that has occured.
  */
  public void ConnectorWarning(ConnectorID, WarningType);
}
```

4.6 The Monitoring Enabler

Calls	the **Learning, DePer, Trust, Security** enabler to notify the occurrence of a requested pattern of events, or violation of NF-property.
	the **Glimpse Probes** for the notification of an event occurrence on the CONNECTor.
Called by	the **Learning, DePer, Trust, Security** enabler to Subscribe a request for monitoring of NF-property or a pattern of events.

The Monitoring Enabler (that we called GLIMPSE) is in charge to check and notify violation of NF-Properties of the CONNECTor. We propose a model-driven approach to monitoring that is based on two key elements: a coherent set of domain-specific languages, expressed as metamodels, that enable us to exploit the support to automation offered by model-driven engineering techniques, and a generic monitoring infrastructure that offers the greatest flexibility and adaptability. The latter supports all main functionalities of any monitoring infrastructure, i.e., data collection, local interpretation, data transmission, aggregation, and reporting.

To provide a better communication decoupling, the communication paradigm adopted by the monitoring enabler is publish-subscribe. Through the Manager component, in fact, CONNECT enablers (which are the consumers of the monitoring) can instrument a knowledge base with inference rules. Complex patterns can also be matched by using the Complex Event Processor module.

The main components of Glimpse are shown in figure 4.7, in the following sections, all of these components and objects that have a role in the interaction are explained further.

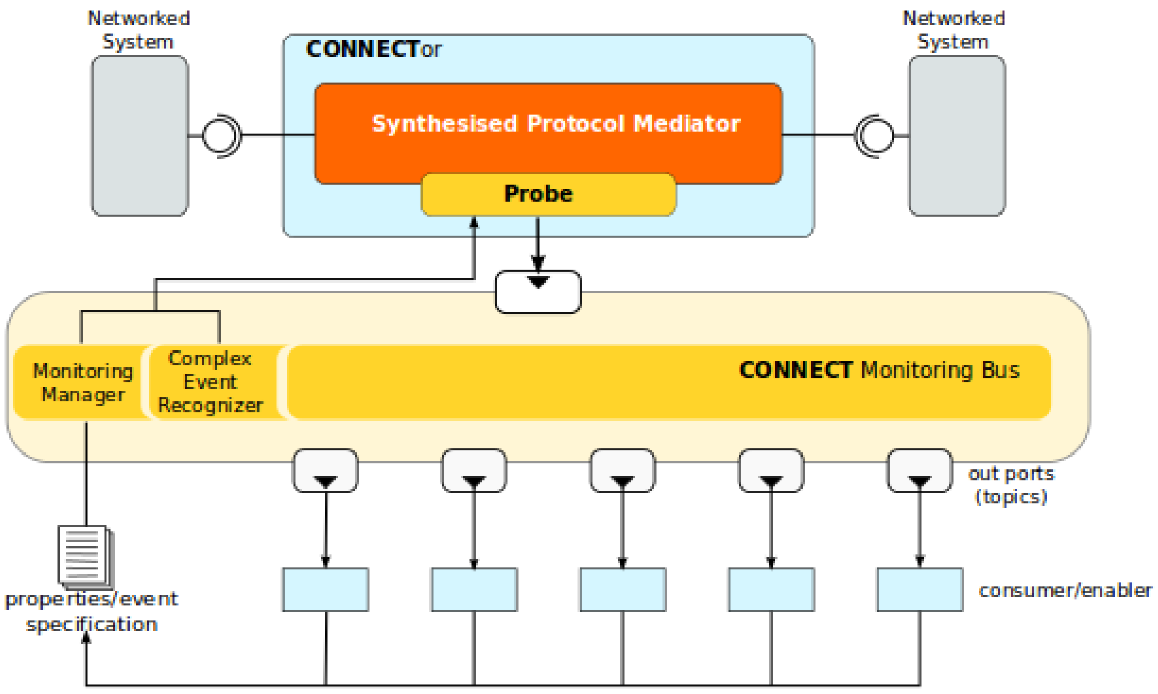


Figure 4.7: The Glimpse infrastructure providing the monitor enabler behaviour

4.6.1 Probe

The probe component instruments the CONNECTor component to send or to locally filter raw data from the execution of the observed NSs. Each deployed CONNECTor contains Glimpse probes that capture and then send messages (events) of the following types to the Glimpse Monitoring Bus (a description of the instrumentation is provided in Section 3.4.2):

- Events: these describe the transitions executed by the CONNECTor as messages are received from and sent to the NSs. An Event in the Glimpse Infrastructure is represented by a GlimpseBaseEvent<T> object that contains all the information useful for the Complex Event Processing. For more information, see [13].
- Exceptions: all errors occurring during the execution of the CONNECTor are trapped and the Exception object is sent to the Monitoring Bus by the probe. The Monitoring infrastructure will analyze the Exception and forward the useful information to the Enabler that requested it's notification using the ComplexEventException Object, whose xsd is listed below.

ComplexEventException

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://labse.isti.cnr.it/glimpse/xml/ComplexEventException"
  xmlns:tns="http://labse.isti.cnr.it/glimpse/xml/ComplexEventException"
  elementFormDefault="qualified">
    <element name="ComplexEventExceptionList" type="tns:ComplexEventException" />
    <complexType name="ComplexEventException">
  <sequence>
    <element name="CauseClassName" type="string" maxOccurs="1" minOccurs="1" />
    <element name="ClassName" type="string" maxOccurs="1" minOccurs="1" />
    <element name="Message" type="string" maxOccurs="1" minOccurs="1" />
    <element name="StackTrace" type="string" maxOccurs="1" minOccurs="1" />
    <element name="LocalizedMessage" type="string" maxOccurs="1" minOccurs="1" />
  </sequence>
  </complexType>
```

```
</schema>
```

4.6.2 Monitoring Manager component

The Manager component is the orchestrator of the Glimpse architecture. It manages all communications between the Glimpse components. Specifically, the Manager fetches requests received on the serviceTopic channel from Enablers, analyzes them and instructs the CEP Evaluator. It then creates a dedicated channel on which it will provide the results produced by the CEP Evaluator, and then notifies the Enablers about this channel (through a JMS Message).

4.6.3 Complex Event Recognizer

The Complex Event Processor (CEP) is the rule engine which analyzes the primitive events, generated from the probes, to infer complex events matching the consumer requests. There are several rule engines that can be used for this task (like Drools Fusion, RuleML). In the current implementation, we adopt the Drools Fusion rule language that is open source and can be fully embedded in the realised Java architecture. Note that, the proposed flexible and modular architecture allows for easily replacing this specific rule language with another one.

4.6.4 Monitoring Bus

The Monitoring Bus is the communication backbone where all information (events, questions, answers) is sent by: Probes, Consumers, Complex Event Processor and by all the services querying information.

In this prototype, we adopt a publish-subscribe paradigm devoting the communication handling to the Manager component; this communication pattern allows more consumers to fetch the same CEP evaluation results and offers data dissemination at low computational cost. This component is implemented by means of ServiceMix4, an open source Enterprise Service Bus, used to combine advantages of event-driven architecture and service- oriented architecture functionality. We chose ServiceMix4 because it offers a Message Oriented Bus and is able to run an open source message broker like ActiveMQ of which different implementations exist for several platforms.

The monitoring enabler receives requests concerning which CONNECTors to monitor; these are subscriptions and rules applied on the Glimpse message bus describing which content should be forwarded to that enabler's monitoring interface connected to the bus (behaviour illustrated by the Learning, DePer, Security and Trust enablers reception interfaces from the monitoring channel).

4.6.5 Properties specification

The properties object provided to the monitoring infrastructure, is a set (composed of 1 or more) of rules that an enabler wants to be monitored. A property message, is a JMS ObjectMessage that must respect only three requisites:

1. the contained object must be a ComplexEventRequest object;
2. the ObjectMessage must have a property "DESTINATION" sets to "monitor";
3. the ObjectMessage must have a property "SENDER" sets with the consumerName.

ComplexEventRequest The ComplexEventRequest object is specified as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule"
    xmlns:tns="http://labse.isti.cnr.it/glimpse/xml/ComplexEventRule"
    elementFormDefault="qualified">
```

```
  <element name="ComplexEventRuleActionList" type="tns:ComplexEventRuleActionType" />
  <complexType name="ComplexEventRuleActionType">
     <sequence>
       <element name="Insert" type="tns:ComplexEventRuleType" maxOccurs="unbounded" minOccurs="0
            " />
       <element name="Delete" type="tns:ComplexEventRuleType" maxOccurs="unbounded" minOccurs="0
            "/>
       <element name="Start" type="tns:ComplexEventRuleType" maxOccurs="unbounded"  minOccurs="0
            "/>
       <element name="Stop" type="tns:ComplexEventRuleType" maxOccurs="unbounded" minOccurs="0"
            />
       <element name="Restart" type="tns:ComplexEventRuleType" maxOccurs="unbounded" minOccurs="
            0"/>
     </sequence>
  </complexType>
  <complexType name="ComplexEventRuleType">
      <sequence>
        <element name="RuleName" type="string" maxOccurs="1" minOccurs="1" />
        <element name="RuleBody" type="string" maxOccurs="1" minOccurs="0" />
      </sequence>
      <attribute name="RuleType" type="string" />
    </complexType>
</schema>
```

The specification of properties like Latency, Coverage or others kind of Non-Functional properties, may also be exploited using the CONNECT Property Meta Model (CPMM). This model has been developed using the Eclipse Modeling Framework and allows the user to define generic and specific models of a NF-property. In order to have all operations strictly connected, we are developing a tool to be able to convert the generated EMF model of the NF-property into a Drools Rule.

4.6.6 Event specification: the GlimpseBaseEvent<T> Object

The events fired from a probe, are JMS ObjectMessages whose payload is a GlimpseBaseEvent<?> object.

The parameters of the GlimpseBaseEvent provides the information that is useful for complete complex event processing, including:

- NetworkedSystemSource: the id of the networked system that fires the event;
- ConnectorID: the identifier of a CONNECTor
- ConnectorInstanceID: the identifier of a specific instance of a CONNECTor
- ConnectorInstanceExecutionID: the identifier of a specific iteration of an instance of the CONNECTor
- Data: the payload of the event, usually the label of the LTS transition

4.6.7 APIs

The Monitoring Enabler provides a set of APIs that allow users to implement a Probe or a Consumer with a specific behaviour, or just to use the default one contained in the example package.

The activity diagram of a basic interaction between an Enabler and the Monitoring is shown in figure 4.8

To publish to the channel (from the Connector):

```java
public void sendMessage(GlimpseBaseEvent<?> event, boolean debug);
```

To add a subscription to the channel and send a request for evaluation to the Monitoring the GlimpseAbstractConsumer is extended; or the constructor of the MyGlimpseConsumer class is used.

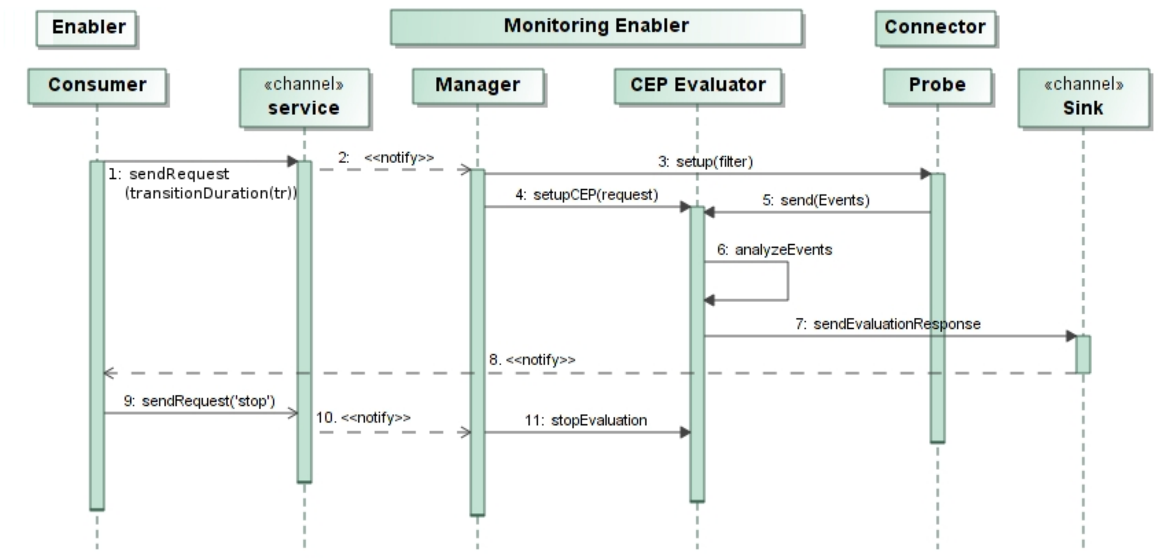


Figure 4.8: The activity diagram of the Monitor Enabler

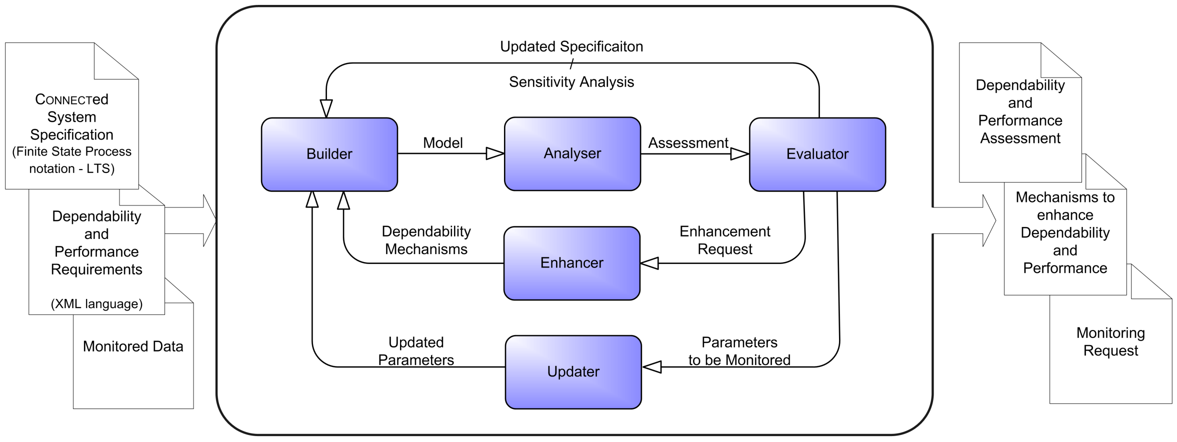
```
public class MyGlimpseConsumer extends GlimpseAbstractConsumer {
  /**
  * @param settings can be generated automatically using {@link Manager#
      createConsumerSettingsPropertiesObject(String, String, String,
  *         String, String, String, boolean, String)}
   * @param plainTextRule a plain text rule is a String containing the Drools<br />
   * (or other cep engine implemented) rule that can be generated<br />
   * using the {@link ComplexEventRuleActionListDocument} classes.<br />
   * For a rule example see the exampleRule.xml file.
   */
  public MyGlimpseConsumer(Properties settings,String plainTextRule) {
    super(settings, plainTextRule);
  }

  public MyGlimpseConsumer(Properties settings,ComplexEventRuleActionListDocument
      complexEventRuleXML) {
    super(settings,complexEventRuleXML);
  }
```

4.7 The Dependability and Performance Enabler

Calls	the **Monitoring enabler** method `subscribe` in order to subscribe and filter events related to the dependability and performance analysis. the **Deployment enabler** to warn that a CONNECTor violates the required non-functional properties. the **Synthesis enabler** after a CONNECTor violation with an enhanced model to address the problems.
Called by	the **Synthesis enabler** to run the `Analyze` method to verify the synthesised CONNECTor against the required non-functional properties. the **Monitoring enabler** to call the `notify` method in order to be informed about newly CONNECTor events of interest.

The DePer Enabler provides support to the definition of a CONNECTor that allows NSs to interoperate with a given level of dependability and performance, if such non functional requirements have been expressed by the involved NSs. Its intervention is both at pre-deployment time

and at run-time, after CONNECTor deployment.
Once a CONNECTor specification has been produced by the Synthesis enabler it sends it to the DePer enabler to determine if the dependability and performance requirements (as described in the Networked System Model of each networked system) are satisfied. If so, DePer tells the Synthesis enabler to go ahead and deploy; otherwise, it evaluates a set of enhancements to the specification and returns feedback on these enhancements.



CONNECTed System Specification (Finite State Process notation - LTS)
Dependability and Performance Requirements (XML language)
Monitored Data
Updated Specificaiton / Sensitivity Analysis
Builder
Model
Analyser
Assessment
Evaluator
Dependability Mechanisms
Enhancer
Enhancement Request
Updated Parameters
Updater
Parameters to be Monitored
Dependability and Performance Assessment
Mechanisms to enhance Dependability and Performance
Monitoring Request

Figure 4.9: Architecture of the Dependability and Performance (DePer) Analysis in CONNECT

When the CONNECTor has been deployed, the DePer enabler also continuously determines if the CONNECTor maintains its dependability and performance requirements. It receives monitoring data from the Monitoring enabler and in the case where there is no longer compliance, DePer communicates to the Deployment enabler (via the `control` interface) that the current instance of the CONNECTor should be stopped and proper actions should be taken towards definition and deployment of a new CONNECTor.

The architecture of DePer, shown in Figure 4.9, is composed of five modules: *Builder*, *Analyser*, *Evaluator*, *Enhancer* and *Updater*. The Builder module produces a probabilistic model for the system specification. The Analyser module computes quantitative assessment of the dependability and performance metrics. The Evaluator module checks the quantitative assessment of the metrics under evaluation and reports the results to the Synthesis Enabler. The Enhancer module provides a support for enhancing the CONNECTor specification through dependability mechanisms, should Evaluator's result reveal that non-functional requirements are not satisfied. Finally, the Updater module exploits run-time data on the CONNECTed system executions from the monitor Enabler, to update the accuracy of model parameters. Full details on the description of the individual models, as well as implementation aspects of the developed prototype, are in deliverable D5.3 [12], Chapter 3.

4.8 Further Connectability Enablers

Two final enablers will be elaborated on and their implementations finalised in the final year of the project. These are the Security and Trust enablers that form part of the set of enablers to achieve Connectability properties in the CONNECT architecture. While not fully ready for concrete integration we provide a brief description of their behaviour here (in Section 3.2 we provided the plan for them to be fully integrated into the CONNECT Enabler architecture).

4.8.1 The Security Enabler

The behaviour of the Security enabler is two-fold. First, it receives an analysis request from the Synthesis enabler and returns an enhanced LTS instrumented with appropriate security policies which will meet the security requirements of the mediated systems. Secondly, the instrumented CONNECTor interacts directly with the Security enabler through embedded security monitors. These monitors can directly enforce security policies on the execution of a deployed CONNECTor. As with the DePer enabler, the Security enabler can co-ordinate a resynthesis of the CONNECTor.

4.8.2 The Trust Enabler

The Trust enabler analyses and monitors a CONNECTor to determine if the level of trust provided by networked systems meets the trust requirements. To do this, the trust enabler analyses the model provided by the Synthesis enabler. Further the enabler interacts directly with a CONNECTor; once it has calculated the current level of trust provided by the networked systems it invokes the `setMetaData` method on the open interface of a specific CONNECTor to store the current trust values. Subsequently, the corresponding `getMetaData` method can be used to read the current trustvalues.

The Trust enabler also utilises the interaction enabler to invoke operations on the networked system and hence build the trust values; this uses the same approach as the Learning enabler. Hence, the Trust enabler is continuously monitoring the levels of trust provided by these networked systems.

4.9 Summary

This section has presented the complete list of enablers used in the CONNECT architecture to carry out the functionality required to produce interoperability solutions on-the-fly. This includes the introduction of a new enabler not envisioned in the Y2 architecture, i.e., the Interaction enabler, which provides a key building block in the understanding and learning of networked systems; this provides the services required to allow the enablers to interact directly with these legacy systems.

Table 4.1 documents the implementation progress of these enablers and their integration into the concrete CONNECT architecture. Here the table describes the completeness of the implementation of the enabler itself (in the second column), i.e., in terms of the functionality it provides to CONNECT. The table further reports on the integration progress (in the third column), i.e., in terms of the enabler implementation communicating with other enablers to achieve co-ordinated functionality.
It is clear from this section that advancement has been made during the third year in terms of:

- *Implementation*. Implementations of each enabler has advanced in order to better support the architecture and also to interact with other enablers.
- *Integration*. The interface specifications for each enabler have better supported the overall integration of the enablers into a co-ordinated architecture.

Table 4.1: Implementation and integration progress of CONNECT enablers

Enabler	Implementation Status	Integration Status
Discovery	Complete	Integration with synthesis conceptually complete (coding ongoing). Integration with learning conceptually complete (coding ongoing). Integration with deployment conceptually complete (coding ongoing).
Learning	Complete	Integration with discovery complete. Integration with interaction conceptually complete and implementation being finalised.
Interaction	Complete	Integration with Learning conceptually complete and coding is being finalised; integration with Trust to be started.
Deployment	To be revised	Integration with synthesis complete. Integration with discovery conceptually complete (coding ongoing). Integration with monitoring in progress. Integration with Connectability to be started.
Synthesis	Conceptually complete, development in progress	Integration with Discovery conceptually complete, implementation (in progress). Integration with Deployment conceptually complete and implementation (in progress). Integration with Connectability enablers to be started.
Monitoring	Complete	Integrated with Deper, Security, Learning and CONNECTors.
DePer	Under refinement	Integration with Monitor complete; integration with synthesis in progress.
Security	Conceptually complete, development in progress	Integration to be started.
Trust	Conceptually complete development in progress	Integration to be started.

5 Underlying Conceptual Contributions

5.1 Introduction

This section highlights the important contributions made by WP1 with respect to the progression of the architecture in the third year. These contributions underpin the functionality provided in different parts of the CONNECT architecture, and examine the application of the architecture to solve different interoperability problems:

- We present how affordances are learned in Section 5.2 and then we introduce in Section 5.3 how the goals and intent of given networked systems are captured during discovery and then utilised by the CONNECT architecture.
- The concepts underpinning the integration of the synthesis process with the deployment of a concrete CONNECTor are shown in Section 5.4, and we qualitatively and quantitatively evaluate these methods using a simple photo-sharing application.
- The CONNECT architecture is applied to the domain of mobile systems, and in particular to interoperability between systems performing media streaming. Section 5.5 presents the AmbiStream architecture as a CONNECTor architecture for mobile interoperable live streaming. Again this architecture is qualitatively and quantitatively evaluated.

5.2 Affordance learning

One of the key challenges in discovery, and thus in achieving interoperability, is to ensure that all the necessary parts of the description of networked systems is available. In Deliverable 1.2 we put forth an NS model (reflected in Figure 2.2) comprising four kinds of description: the interface, the behaviour (process), non-functional properties, and the affordance. Often traditional discovery protocols provide only an interface description, and so we must find other means to enrich the NS description with the additional parts. In the case of behaviour, this is achieved through automata learning. In the case of the affordance (a triple of ontology concepts describing the functional capability of the NS), we use text categorisation techniques from statistical machine learning to perform inference from the text of the interface description. The work was done in collaboration with the University of Trento under the EternalS co-ordination action[1], and further detail can be found in [5, 3].

The intuition behind using text categorisation is that interface descriptions usually contain many human-readable terms, such as method names and type identifiers, and sometimes even documentation in free natural-language text. An engineer looking at an interface with many instances of the word "weather"—as in Figure 5.1—might conclude that the networked system belongs to a category of systems dealing with the weather. It is this process we sought to automate.

```
<wsdl:message name="GetWeatherByZipCodeSoapIn">
    <wsdl:part name="parameters"
          element="tns:GetWeatherByZipCode" />
</wsdl:message>
 <wsdl:message name="GetWeatherByZipCodeSoapOut">
   <wsdl:part name="parameters"
         element="tns:GetWeatherByZipCodeResponse" />
</wsdl:message>
```

Figure 5.1: WSDL fragment.

[1] https://www.eternals.eu/

5.2.1 Background

Standard text categorisation approaches (such as support vector machines [16]) require us first to provide a set of categories, and a set of training instances where text samples are manually associated with a category. This data is used to train a classifier which then is able to categorise new, previously unseen text automatically. In the case of affordance learning, the categories are provided by the set of possible functional capabilities listed in the domain ontology, such as "WeatherService". The text to be categorised is the interface description in a format such as WSDL.

In the approach we have applied within CONNECT [5], linear support vector machines are used. This approach requires us to define informative *features* that will indicate when a document belongs to a certain category. The features in this case are the natural-language words used in the documentation sections of the interface description, and the (usually camel-case) identifiers for operations and data types. The description is encoded following the *bag-of-words* [27] method. The interface is tokenised and represented as a vector in which each element is the frequency of the token (word) associated with that element. For example, an interface consisting of "getWeather, getStation" could be represented as the vector $(2, 1, 1, \ldots)$ where, e.g., 2 represents the frequency of the "get" token, while the frequencies of "Weather" and "Station" are both 1. When presented with the bag-of-words encoding of an unseen interface description, the trained classifier suggests the category that it belongs to.

5.2.2 Experiments

The LibLinear[2] implementation of linear support vector machines was used to perform two experiments. The first experiment used a small set of WSDL descriptions manually assigned to two categories ("Stock" and "Weather"), and the second used a larger set of WSDL descriptions available on the web[3]. On the small set, the classifier was able to achieve 100% accuracy (using three-fold cross-validation), while on the larger set of 402 documents in 10 categories (using eight-fold cross-validation) the accuracy was 58%. Table 5.1 shows the results for each category. P indicates the *precision*, which is the number of documents correctly assigned to a category compared to the number that are correctly or incorrectly assigned to that category (a precision of 1 means there are no false positives). R indicates the *recall*, which is the number of documents correctly assigned to a category compared to the number that should be assigned to that category (a recall of 1 means there are no false negatives). For example, the "CountryInfo" category has a recall of 0.83, meaning that few documents of that category were falsely assigned to another. n indicates the number of documents manually assigned to each category while F indicates the F-measure. These results are published in [3].

Category	P	R	F	n
Mathematics	0.29	0.20	0.24	23
Business	0.17	0.08	0.11	46
Communication	0.71	0.80	0.75	49
Converter	0.57	0.63	0.61	65
CountryInfo	0.64	0.83	0.72	38
Developers	0.18	0.11	0.14	46
Finder	0.55	0.59	0.57	10
Money	0.72	0.72	0.72	56
News	0.70	0.63	0.67	30
Web	0.47	0.46	0.47	39

Table 5.1: Performance by category.

[2] http://www.csie.ntu.edu.tw/~cjlin/liblinear/

[3] http://www.andreas-hess.info/projects/annotator/ws2003.html

Table 5.2 shows the most highly weighted features (words) for each category, confirming the intuition that the words used in the interface description are a good indication of the NS's category.

Category	Features
Mathematics	calculator, previous, at, value
Business	description, chart, parent, n
Communication	send, message, email, subject
Converter	to, translate, unit, my
CountryInfo	country, state, zip, postal
Developers	reverse, text, case, generate
Finder	whois, who, iwhois, results
Money	stock, amount, card, currency
News	news, quote, day, daily
Web	key, name, valid, d

Table 5.2: Most highly weighted features in the ten-category experiment.

Given the reasonable accuracy achieved by the text categorisation approach, we have incorporated it into the discovery enabler, which will allow NS descriptions without affordances to be completed, and allow us to evaluate the approach in the CONNECT context. The risk associated with using a statistical learning approach to provide information used as the basis for later processes such as synthesis is that the information gained is rarely certain. In particular, incorrect categorisations can lead to false positives, where two NSs have been assigned the same affordance when in fact they do not match, and false negatives, where two matching NSs are assigned different affordances and hence no attempt to connect them will be made. Minimising the number of false negatives (i.e. maximising recall) is hence critical for useful application within CONNECT. Greater accuracy may be achieved by finding more nuanced features, such as the structure of the document, on which to base the categorisation. For example, the "value" token could be relevant to both the "Money" and "Mathematics" categories above. However, considering "value" in the proximity of "card" may be sufficient to distinguish between the two categories.

5.3 Goals and Intent

5.3.1 Goal specification

There has thus far been little said about the intent or goal that is in mind when pairs or larger groups of networked systems are connected according to the principles and processes detailed in this project. Without such a specification from the user, it may be possible to apply the CONNECT processes to many different networked systems resulting in many different connections (and hence compositions), but there is no way to determine which from those possibilities would be beneficial to the end user. One way to restrict the choice of NSs and then to restrict their behaviour to useful operations is to have the end user specify a goal in temporal logic. Such formulas use the ontology concepts that define operations and data, allowing goals that state, for example, that an operation must eventually be performed. Each operation concept is related to another concept representing some high-level functionality, which identifies an affordance. Affordances (defined in Section 2.3) and goals are closely related, and in some cases they coincide. In addition to a concept describing its functionality, an affordance *describes* an associated behaviour, while a goal *prescribes* a behaviour (see Figure 5.2). More generally a goal will be a restriction of the behaviour of an affordance to certain traces in which the user is interested. Therefore we say that the affordance associated with a goal is the one having a behaviour that the goal restricts. For example, if a goal states that the *GetFlightQuote* operation must eventually be performed, and this operation is part of the behaviour of the *TravelTickets* affordance, then we say that the goal is associated with the *TravelTickets* affordance[4].

Definition 1 (Goal) *A goal G is a formula of temporal logic (CLTLB(D)) over the behaviour of the connected networked systems such that $S_1 \| M_G \| S_2 \models G$ for some networked systems S_1, S_2 and synthesised mediator M_G. The goal language makes use of standard temporal logic operators:*

- ***`[]` f** – formula f must hold in all (future) states.*
- ***`<>` f** – formula f must eventually hold in some (future) state.*
- ***`X` f** – formula f must hold in the next state.*
- ***f `U` g** – formula f must hold until formula g holds.*
- ***`||`, `&&`, `!`, `->`, `<->`** – with their usual meanings in propositional logic.*

Three predicates can be combined with the above operators:

- ***`executed(c)`** – the operation represented by concept c must have been executed.*
- ***`received(c)`** – the data (input or output) represented by concept c must have been received.*
- ***`sent(c)`** – the data (input or output) represented by concept c must have been sent.*

The concepts c above are defined in the domain ontology as data or operations. Each such operation is associated (in the ontology) with another concept that determines the affordance associated with the goal.

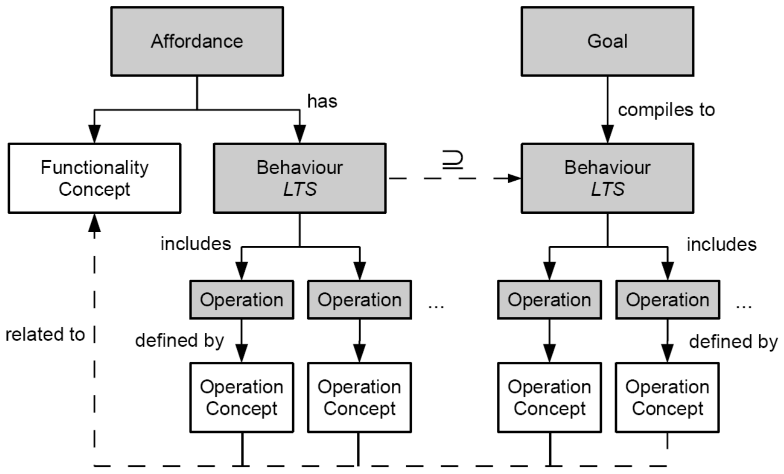


Figure 5.2: The relationship between user-specified goals and NS-specified affordances.

For example, consider a client and server for booking trips. The client and server respectively require and provide a *TravelTickets* affordance. Operations such as *SetDestination* and *GetFlightQuote* and data types such as *FlightQuote* are also defined in the ontology and pertain to this affordance. One possible goal is `<> received(FlightQuote)` with the meaning that the *FlightQuote* data should eventually be received. Another possibility is `<> executed(GetFlightQuote)`. This goal, along with the behavioural models of the pair of NSs, is provided to behavioural synthesis to find a mediator to enable interaction between the

[4]At present we do not consider the case when a goal uses concepts combining several affordances since synthesis produces connectors for a single affordance, in other words, a choreography.

pair in pursuit of this goal. For instance, the operations of the client may include *GetDevisVol* which is mapped to *GetFlightQuote* in the ontology. Mediator synthesis must therefore produce a mediator which ensures that the *GetDevisVol* operation is eventually performed. Goal-based mediator synthesis (D3.3 chapter 5) provides the means to deal with user goals during synthesis. If no goal is given by the user, then the interface mapping-based synthesis technique (D3.3 chapter 4) may instead be used.

Goals defined as above state a precise *functional* requirement on the connection and consequently on the composition of a pair of NSs. We are investigating means to combine this with security, dependability and trust requirements which can further limit the compositions that are acceptable to the end user. Moreover, the functional goal may be used to direct the automata learning implemented by the learning enabler, such that a decision about whether the discovered network system supports the goal can be made more rapidly. In addition, we envisage incorporating work from the area of requirements engineering, such as [1] or [18], which would allow a temporal logic goal to be elicited from a higher-level representation.

5.3.2 Choreography versus orchestration

We can consider two methodologies for introducing a functional goal into the CONNECT architecture:

- In the first case, a user specifies a goal G_{Σ_U} through some interface to the Discovery enabler, and the Discovery enabler actively determines (using the synthesis enabler) what connections need to be established for this goal to be satisfied, while the networked systems are passive until the point of connection. This can be seen as a top-down approach. The corresponding synthesis problem can be characterised as a composition of the behaviours of many NSs S_i and many mediators M_j which together achieve the goal:

  $$S_1 \| \dots \| S_n \| M_1 \| \dots \| M_m \models G_{\Sigma_U}$$

  In other words, mediator synthesis should produce a set of mediated connections established between pairs of NSs that ensure a goal given in terms of operations (or data) Σ_U specified in the domain ontology.

- In the second case, one or more networked systems have goals $G_{\Sigma_{S_i}}$ associated with them (by one or more users), and the networked systems are active in requesting connections from the Discovery enabler that will satisfy their goals[5] (the request being implicit in the act of being discovered). This can be seen as a bottom-up approach. The corresponding synthesis problem can be characterised as a composition of a mediator and a pair of NSs which together achieve the goal:

  $$S_1 \| M_G \| S_2 \models G_{\Sigma_{S_i}}$$

  In other words, mediator synthesis should produce a mediated connection between a pair of NSs that ensures a goal given in terms of the operations Σ_{S_i} of one of the pair.

The second case is in effect a specific case of the first, and they can in fact be treated in the same manner when we have $G_{\Sigma_U} = \bigwedge_i G_{\Sigma_{S_i}}$ (modulo ontological differences).

It might be expected that G_{Σ_U}, being specified in terms of the (global) domain ontology, could express global goals, such as may be achieved by the creation of several mediators that synchronise their behaviour. This however would likely require communication and synchronisation between mediators and result in an *orchestration* (the composition of mediators forming an orchestrator). Since the CONNECT project considers only pair-wise *choreographies*, we assume that the goal, if it pertains to more than one connection, is a conjunction separable into sub-goals for each connection.

[5]We assume that the goals do not conflict.

Once a goal is specified, and a pair of NSs which can satisfy it through interaction are identified, synthesis can often be directly initiated. However, there are cases in which an NS has requirements, expressed as required affordances, that must be satisfied in order for the NS to function correctly. Such requirements may not be directly related to the goal in question. For example, a travel booking client may require a storage service to store the user's travel preferences, while the likely goal of its interaction with a booking service is to have booked a trip.

Tackling these sorts of dependencies requires us to consider the structure of a composition of NSs—in terms of the involved NSs and the connections between them—in addition to the behaviour of the composition, which is handled through mediator synthesis. Hence, we have introduced a step before mediator synthesis called *assembly* (or structural synthesis) based on existing work [32]. We refer to the assembled structure as a *configuration*.

In the following section we describe the assembly process and the future research questions it poses for the CONNECT approach.

5.3.3 Assembly

Definitions

Recalling the definition of networked system descriptions given in D1.2 [4], we introduce here some more formal definitions to aid the explanation of assembly.

We define a networked system description as follows:

$$\mathit{NSDesc} \subseteq \mathit{Interface} \times 2^{\mathit{Affordance}} \times \mathit{NFProp}$$

where $\mathit{Interface}$ denotes the NS interface consisting of a set of *operations* and

$$\mathit{Affordance} \subseteq \{+,-\} \times \mathit{Concept} \times \mathit{Concept} \times \mathit{Concept} \times \mathit{LTS}$$

where for some $c \in \mathit{Affordance} = \langle t_c, op_c, i_c, o_c, p_c \rangle$, op_c refers to an ontology concept describing the functionality the affordance represents, i_c and o_c are input and output data concepts, $t_c \in \{+,-\}$ denotes whether the affordance is provided or required, and p_c denotes the *labelled transition system* (LTS) describing the protocol for that affordance as defined in D3.3. We use the additional notation $prov(s)$ and $req(s)$ to denote the set of provided and required affordances of NS s, respectively. We do not elaborate the definition of *NFProp* here.

Notice that these definitions permit an NS, with a single interface, to provide or require a number of different affordances, each with an associated protocol. We interpret a required affordance as meaning that the NS cannot (at least not safely) provide any functionality unless the requirement is satisfied. This is necessary because we treat the NS as a black box, and in general the various different affordances of a single NS will nevertheless be related (in some unobservable way) in its internal implementation. The consequence for assembly is that we must include within configurations additional NSs that satisfy such required affordances.

Once a description is discovered, it is stored in the *repository* of the Discovery enabler: $\mathit{Repository} \in 2^{\mathit{NSDesc}}$.

Assembly

Given a repository as above, the Discovery enabler must choose a structural configuration of NSs and connections between them, which can be used to achieve the goal. We define a structural configuration as:

$$\mathit{SConfiguration} \subseteq 2^{\mathit{NSDesc}} \times 2^{\mathit{Connection}}$$

i.e. a configuration is a set of NSs and connections[6], where

$$\mathit{Connection} \subseteq \mathit{NSDesc} \times \mathit{Concept} \times \mathit{Goal} \times \mathit{NSDesc}$$

[6]To simply the notation we do not distinguish between *instances* and *types* (respectively CNSInstance and CNSDescription in the implementation), and use NSDesc for both. Strictly a configuration contains instances.

i.e. a connection exists between two NSs to achieve a particular goal with its associated affordance. In all configurations $config = \langle S, K\rangle$, $\forall\langle s_1, a_G, G, s_2\rangle \in K : s_1 \in S \wedge s_2 \in S$, and of course $S \subseteq Repository$. Configuration $config$ is called *complete* if the following additional conditions hold:

$$\bigcup_{s\in S} prov(s) \supseteq \bigcup_{s\in S} req(s)$$

$$\exists k \in K = \langle s_1, a_G, G, s_2\rangle : M_k \models G$$

for some mediator M_k and goal G. The mediator synthesis process for producing M_k (for every $k \in K$) is described in D3.3 [14]. If no goal is provided, the interface mapping-based synthesis technique may be employed (D3.3 chapter 5), otherwise goal-based synthesis is necessary (D3.3 chapter 4).

The process of choosing a structural configuration is called assembly and is based on previous work [32, 33]. A configuration is assembled by recursively adding NSs to provide affordances that are required by the NSs that have already been selected. In the initial step, two NSs are selected, one providing and one requiring a_G, the affordance related to G. They are added to the configuration, and selection is repeated for all their required affordances.

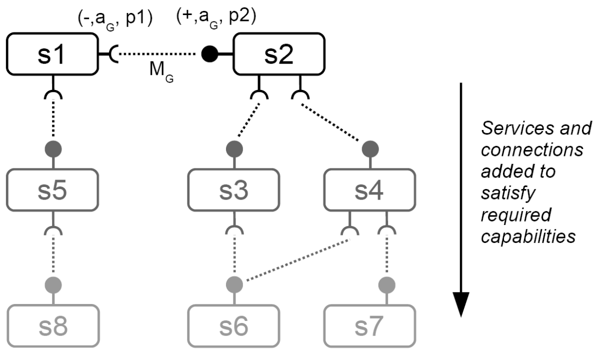

(-,a_G, p1)
(+,a_G, p2)
s1
s2
M_G
s5
s3
s4
s8
s6
s7
Services and connections added to satisfy required capabilities

Figure 5.3: Structural configuration assembly.

Selection of an NS s_2 to satisfy a required affordance $\langle -, a_1, i_1, o_1, p_1\rangle$ (of NS s_1) is according to two criteria:

1. s_2 must provide an affordance $\langle +, a_2, i_2, o_2, p_2\rangle$ where $a_2 \sqsubseteq a_1$, $i_1 \sqsubseteq i_2$, $o_2 \sqsubseteq o_1$ (in the ontology), and
2. it must be possible to synthesise a mediator between protocols p_1 and p_2.

When s_2 has been selected, it is added to the configuration, along with a new connection $\langle s_1, a, s_2\rangle$, and the process repeats for the requirements of s_2. The goal associated with this new connection is the full behaviour of p_1[7]. Figure 5.3 illustrates the process graphically.

Discussion

The result of the assembly process is a configuration consisting of a set of connections between networked systems, each potentially involving a mediator, all of which operate independently and can be treated as such by all the CONNECT enablers. Any co-ordination between the connections is assumed to be taken care of inside the NS implementations. The configuration, once constructed, is thus decentralised and has no orchestrator. One root mediator is specifically concerned with achieving the goal.

[7]Recalling that a goal given in LTL is compiled to a Büchi automaton for goal-based synthesis. The compilation step is omitted for connections where the goal is already a process rather than an LTL formula. If interface mapping-based synthesis is used, no goal need be considered.

Treating the generation of a structural configuration as a separate problem to that of mediator synthesis provides a performance benefit compared to approaches that synthesise orchestrators, such as Pistore *et al.* [24]. The state space for synthesis of orchestrators grows exponentially when the behaviours of many NSs are considered together. In contrast we only ever consider synthesis between a pair of NSs, thus making the approach more amenable to runtime application. In addition, keeping the structure separate from the co-ordination provides opportunities for structural reasoning (e.g. for adaptation) that are difficult to perform when the structure is implicit in an orchestrator specification.

Assembly (without synthesis) has been studied extensively by CONNECT participants and others, such as Cavallaro *et al.* [8], who considers cost; Sykes *et al.* [33], and [34] in which the assembly process is decentralised; while other approaches include Rainbow [9] and Aura [30]. This background provides a solid footing on which to consider the characteristics of the CONNECT context that may require refinements to the assembly process, and what features assembly can contribute to the CONNECT approach. At present this work is at an early stage, and research questions include whether assembly can be influenced by the security, dependability and trust properties considered elsewhere in the architecture and whether the adaptations envisaged by feedback from the monitoring enabler can be handled at a purely structural level. Moreover, ontological information, such as part-whole relations, may provide a guide for assembling configurations that combine several affordances without demanding the full strength of an orchestration.

5.4 Synthesis and Deployment Integration: From Abstract to Concrete CONNECTors

The role of synthesis is to produce the CONNECTor between two networked systems that can then be realised and deployed. This process is carried out in two stages: first, the synthesis enabler produces an abstract representation (the enhanced LTS) of the CONNECTor; second, this abstract CONNECTor is made concrete by producing the necessary k-coloured automaton.

In the previous Deliverable D1.2 [4] we proposed two alternative methods for realising concrete CONNECTors, where a concrete CONNECTor is an executable software artefact that is deployed to connect two systems and successfully support interoperation. The first approach advocated code generation techniques whereby the source code of the CONNECTor is fully generated from an LTS specification of the mediator (this can then be compiled and executed dynamically). The second approach proposed to make the LTS model specification concrete in order that it can be interpreted dynamically; here the use of k-coloured automata [7] was proposed, where the concept of colour adds concrete protocol information to a specification in order that it can be executed as part of the CONNECTor.

Based upon the work carried out on the implementation of the synthesis enabler (by WP3) and on the implementation of CONNECTors and deployment enablers (by WP1) the most suitable choice for integration was the use of k-coloured automata as a common representation. This was largely due to the inherent similarity of the two approaches and the existence of tool support (in the form of the Starlink framework developed under the CONNECT project), both of which added to the ease of integration.

Here we illustrate the integration of synthesis through to deployment by stepping through this abstract to concrete procedure using a simple example to help illustrate the process.

5.4.1 Synthesising Abstract Models

Networked systems are interoperable if they are both *functionally* and *behaviourally compatible*. They are functionally compatible if they have semantically matching affordances, i.e., at a high enough level of abstraction the functionality provided by one application is semantically equivalent to that required by the other. Existing formal notions of behavioural compatibility [22] assume a closed-world setting, i.e., the use of the same actions to define the behaviour of the systems. What is needed is a notion of compatibility that further takes into account the semantic compatibility of actions and relies on an intermediary system, i.e., the mediator, to address their syntactic differences. Towards this aim, we investigated two directions for the synthesis of mediators (both detailed in Chapters 4 and 5 of Deliverable D3.3 [14]):

- *Goal-based abstract mediator synthesis.* This approach considers the protocols of two networked systems models NS_1 and NS_2 and produces the mediator protocol that allow them to interact so as to satisfy user-specified goals (see Section 5.3). More specifically, the alphabet (actions) of the two protocols are mapped into a common one using ontology reasoning. This mapping is then used to project the protocol onto a common alphabet, i.e., both protocols are rewritten using the same alphabet. Then, the projected protocols as well as the user goals are encoded as a satisfiability problem. The Zot model checker[8] solves this problem (if possible) and produces a possible feasible interaction satisfying user goals. The model checking process is reiterated so as to discover all the feasible satisfying traces, which are finally concatenated to build the mediator.

- *Mapping-driven abstract mediator synthesis.* This approach first focuses on the interfaces of both networked systems and combines ontology reasoning and constraint programming to identify the semantic correspondence between networked systems' interfaces, i.e., *interface mapping*. The interface mapping defines the correspondence between the actions of the systems' interfaces so as to generate the mapping processes that perform the necessary translations between actions. Then, we generate the parallel composition of the mapping processes and verify that the overall system successfully terminates using the LTSA (Labelled Transition System Analyser) model checker. Furthermore, we are improving the algorithm so as to deal with ambiguous mappings, i.e., when an action from one system may semantically be mapped to different actions from the other system. The synthesised mediator model ensures the *safe* interaction between the networked systems, i.e., it ensures deadlock-freedom and the absence of unspecified receptions (since we assume blocking sends), whatever goal is considered.

Both approaches are performed in two steps: first we use ontology reasoning to establish the correspondence between actions based on their semantics (see Figure 5.4-❶). The ontology-based action mapping is performed at runtime and does not require *a priori* knowledge about the systems; the description of the interface actions only need to adhere to the same ontology, which is defined to reflect the shared understanding of the application domain. In a second step, the action mappings are used to generate the mediator that performs the necessary co-ordination enabling the two networked systems to interoperate (see Figure 5.4-❷). However, there are many distinguishing properties that make each approach appropriate in certain cases. First, the presence of the goal allows the synthesis approach to consider only goal-related behaviours and hence to mediate between systems whose behaviour only intersects on the specific case of the goal. Consequently, the goal-driven synthesis procedure may give fewer false negatives. Section 5.3.1 discusses how to discover and specify this goal and whether different goals can be maintained among different networked systems.

On the other hand, the mapping-driven synthesis can detect the impossibility of mediation while building the ontology-based action mapping thanks to the constraints that these mappings need to satisfy, and in this case avoid expensive behavioural checking. An interesting direction would also be to drive the action-mapping search using a specific goal.

Furthermore, mapping-driven synthesis succeeds only if each input action has all its outputs available at the time of occurrence, which is essential to prove the correctness of the mediator. Hence, many-to-many mismatches with asynchronous semantics cannot be handled, whereas the goal-driven synthesis manages both synchronous and asynchronous semantics for actions.

Finally, the mapping-driven synthesis considers the whole behaviour of systems whereas the goal-based synthesis requires the mapping to be performed on a trace and then composed to generate the mediator. While the former can go faster as the model checking is performed only

[8] http://home.dei.polimi.it/pradella/Zot/

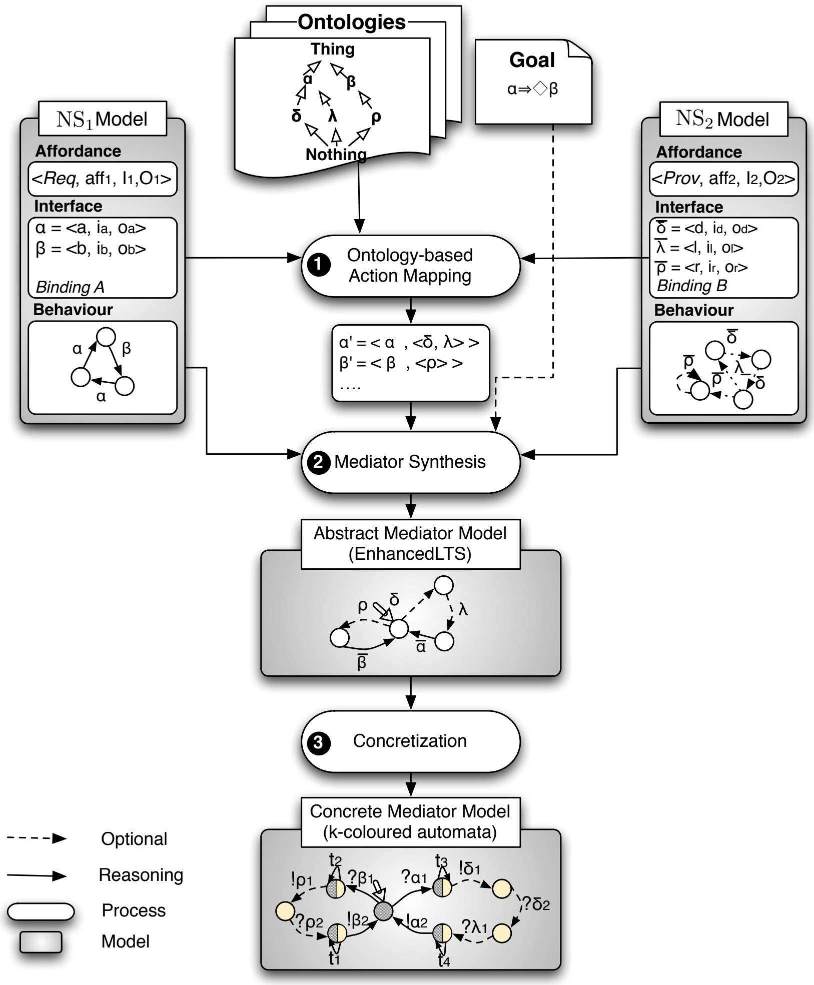


Figure 5.4: Overview of Abstract Synthesis

once, it imposes exact compatibility between the networked systems, i.e., any possible execution of one system can be mapped to an execution of the other system.

The last step of the synthesis of mediators consists of transforming the obtained Enhanced LTS model of the mediator into a k-coloured automaton that can be executed by Starlink (see Figure 5.4-❸). In the following section, we define both Enhanced LTS and k-coloured automaton and describe the steps to transform the former to the latter.

5.4.2 From Enhanced LTS to k-coloured Automaton

An *Enhanced Labelled Transition System (eLTS)* (also defined in Chapter 3 of Deliverable D3.3 [14]) representing either the networked systems or the mediator is defined as a tuple $\langle S, Act, \rightarrow, F, s0\rangle$ where:

- S is a finite set representing the states of the system,
- Act defines the set of observable *actions* that the component requires/provides from its running environment. An *input action* $\alpha = <op,\ i,\ o>$ $(op, i, o \in \mathcal{O}$ where $\mathcal{O}$ is a common domain-specific ontology) requires an operation op for which it produces some input data i and consumes the output data o. Its dual *output action*[9] $\overline{\beta} = <\overline{op},\ i,\ o>$ uses the inputs and produces the corresponding outputs.
- s_0 is the initial state from which the system begins its execution,
- F is the set of states indicating a successful termination of the system, and
- $\rightarrow \subseteq S \times Act \times S$ is the transition relation indicating the change of the system state after performing an action.

We distinguish between two *types of k-coloured automata* [7] : the one representing a networked system and the one representing the mediator. The former refines only the actions of the eLTS associated with the same networked system according to the abstraction relation (see Figure 5.5) and has a unique colour to mark all its states.

The k-coloured automaton (k=2) representing the mediator allowing communication and coordination between two networked systems $NS_i = \langle S_i, M_i, s_{i_0}, F_i, \rightarrow, \emptyset, \emptyset, \emptyset\rangle$, denoted $NS_1 \bigoplus NS_2$, is defined as a tuple $\left\langle S, M, s_0, F, \rightarrow, \xrightarrow{\gamma}, P, \cong \right\rangle$ where:

- S maps to the union of the states of each system,
- M maps to the union of the messages that can be sent to (denoted !) or received from (denoted ?) each system. A message is made up of many fields; some are mandatory, others are optional and others can be calculated (e.g., message length).
- s_0 is the initial state of NS_1,
- F maps to the union of the final states of both systems,
- $\rightarrow$ maps to the union of transitions of both systems,
- $\xrightarrow{\gamma}$ defines the transition between states belonging to different systems while performing a translation between the data received from one system, transformed and sent to the other, that is, and $\xrightarrow{\gamma} \subseteq S_{i=\{1,2\}} \times P \times S_{3-i}$ messages according to the $\cong$ relation. Each data transformation λ between a message $n \in M_{i=\{1,2\}}$ and a sequence of messages $m_1 ... m_k, m_{j=1...k} \in M_{3-i}$ consists of assigning to each mandatory field of n a value of a field of $m_{j=1...k}$ or the result of a translation function applied to the fields of these messages.

[9]Note the use of an overline as a convenient shorthand to denote output actions.

- $\cong$ is a partial order relation specifying the semantic equivalence between a message $n \in M_{i=\{1,2\}}$ and a sequence of messages $m_1...m_k, m_{j=1...k} \in M_{3-i}$, i.e., whether the content of n's fields can be obtained out of the information of the $m_{j=1...k}$'s fields.

To concretise the mediator produced by the abstract mediator synthesis, i.e., to make it executable by Starlink, we need to *transform the associated eLTS to a k-coloured automaton model*. Although there are many similarities between the two models, there is a need for many transformations to translate the eLTS to a k-coloured automaton:

1. The states must be marked so as to distinguish between the states of each system and the merging states, that is the *marking phase*. This marking is straightforward when the correspondence between actions is maintained. More precisely, the ontology-based action mapping produces pairs $\langle\langle\alpha_1, ..., \alpha_m\rangle, \langle\overline{\beta_1}, ..., \overline{\beta_n}\rangle\rangle$ where $\alpha_{j=1..m}$ are actions of $NS_{i=\{1,2\}}$ and $\alpha_{k=1..n}$ are actions of NS_{i-3}, The associated mapping process is: $Map = \beta_1 \rightarrow ... \rightarrow \beta_n \rightarrow \overline{\alpha_1} \rightarrow ... \rightarrow \overline{\alpha_m} \rightarrow Map$ After refinement, all the states traversed when performing α_j (reps. β_k) are marked with the colour of NS_i (resp. NS_{i-3}) while the state between β_n and $\overline{\alpha_1}$ transitions is marked with both colours.

2. The actions of the eLTS need to be refined into messages that can be handled by the k-coloured automaton. This can lead to the translation of one action of the eLTS to a sequence of messages in the k-coloured automaton, which we refer to as the *expansion phase*. The expansion phase performs the reverse of abstraction, i.e., transforms the middleware-agnostic actions of eLTS into actions specific to a middleware type (see Figure 5.5). The constraints on the action mappings guarantee the substitutability between the data flowing in the mediator. These constraints need to be maintained during the refinement. Finally, the binding to a given instance of a middleware type is then directly performed by Starlink (see Section 5.4.3).

3. The abstract synthesis verifies the substitutability of the data concepts that are transformed by the eLTS but does not specify how to actually perform this transformation. We identified two alternatives: we can either annotate the mandatory fields of the messages of each system or specify the transformations using lowering and lifting transformations. The former is more efficient since it performs the assignments/translations atop of the execution engine but requires a finer-grained specification of the messages. The latter has long been used in the field of Semantic Web Services, it is easy to use but relies on heavy XSL transformations[10].

Discussion and Limitations. The k-coloured automaton (k=2) manages the co-ordination and communication flow from one system to another only, whereas the eLTS of the mediator manages both directions. Therefore, one eLTS might produce many k-coloured automata. There also remains a need for rigorous data transformations and better techniques and tools to realise the automated translation between semantically equivalent actions and messages. Finally, we also need to maintain the traceability relationships among the various models so as to be able to manage the evolution of the networked systems and the associated mediator.

5.4.3 Binding k-Coloured Automata to Middleware Protocols

We now describe how the colours of the **CONNECTor model** are used to bind application actions to specific middleware protocols; this process creates a k-coloured protocol automaton that can be directly executed to perform the interaction between two legacy systems.

To bind to a particular protocol we first require: i) the k-coloured protocol automaton of the middleware protocol to tell us what sequences of messages can perform application actions (e.g. the k-coloured protocol automata for CORBA and SOAP are shown in Figure 5.6) n.b. k-coloured protocol automata were first introduced in Deliverable D1.2 [4], ii) the MDL specification

[10]http://www.w3.org/TR/xslt

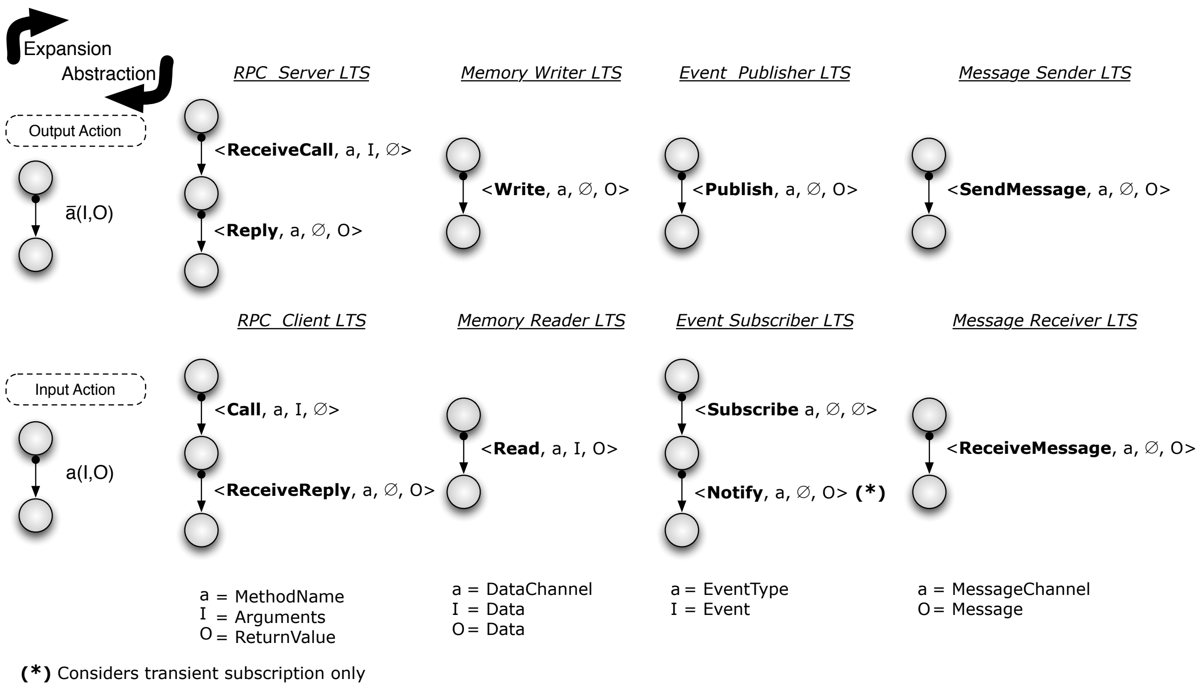
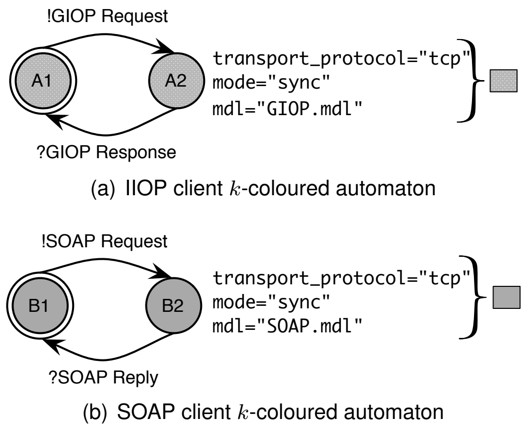


Figure 5.5: Abstraction/Expansion on Application-Middleware LTSs



(a) IIOP client k-coloured automaton

(b) SOAP client k-coloured automaton

Figure 5.6: Examples of concrete k-coloured automata

of that protocol's messages (e.g. Figure 5.7) and iii) the set of rules that describe how a particular protocol (e.g. GIOP) is bound to the application automata concepts (i.e. the action labels, and the parameters).

Figure 5.8 highlights the exact binding procedure. Here the k-coloured application automaton consists of the sending of an add action followed by the reception of the response. This is then shown to be bound to two concrete middleware protocols (SOAP and IIOP)–this highlights the flexibility of application automaton supporting binding to multiple protocols. First, each action transition is bound to the sequence of messages of the middleware protocol. IIOP and SOAP are both RPC protocols and hence the actions correspond to the request and response messages of each protocol, as seen by the corresponding k-coloured sequence.

Each middleware protocol has a set of *binding rules* (shown at the bottom of the figure) that describe how the data in the action transitions are mapped onto the protocol messages.

```
<Message:GIOPRequest>
<Rule:MessageType=0>
<RequestID:32><Response:8>
... <ObjectKeyLength:32><ObjectKey:ObjectKeyLength>
... <OperationLength:32><Operation:OperationLength>
... <align:64><ParameterArray:eof>
<End:Message>

<Message:GIOPReply>
<Rule:MessageType=1>
<RequestID:32><ReplyStatus:32><ContextListLength:32>
...<align:64><ParameterArray:eof>
<End:Message>
```

Figure 5.7: MDL specification of the GIOP message format

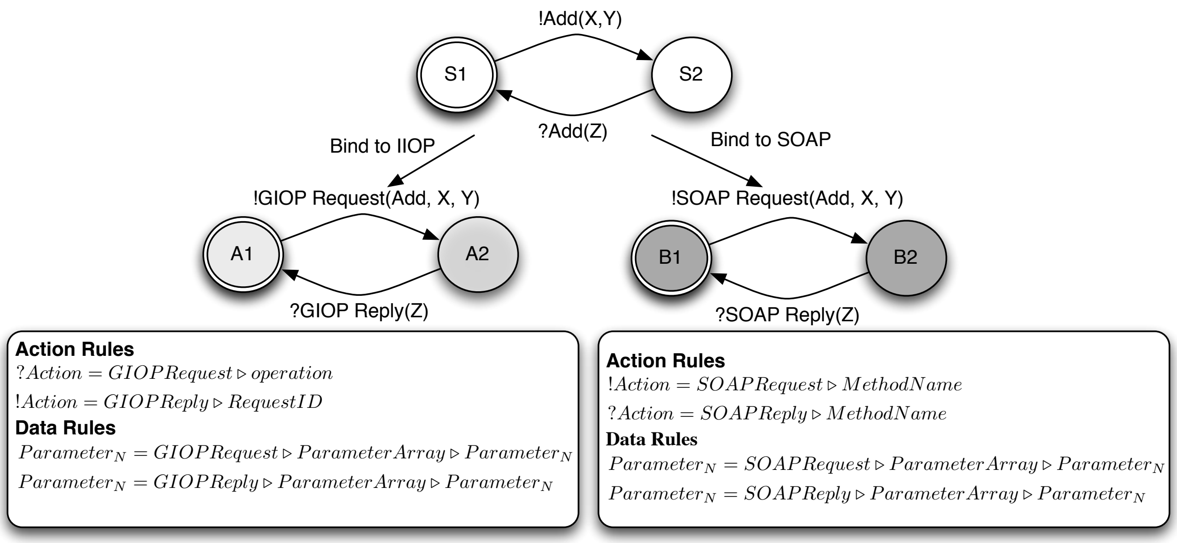

!Add(X,Y)
S1
S2
?Add(Z)
Bind to IIOP
Bind to SOAP
!GIOP Request(Add, X, Y)
A1
A2
?GIOP Reply(Z)
!SOAP Request(Add, X, Y)
B1
B2
?SOAP Reply(Z)
Action Rules
$?Action = GIOPRequest \triangleright operation$
$!Action = GIOPReply \triangleright RequestID$
Data Rules
$Parameter_N = GIOPRequest \triangleright ParameterArray \triangleright Parameter_N$
$Parameter_N = GIOPReply \triangleright ParameterArray \triangleright Parameter_N$
Action Rules
$!Action = SOAPRequest \triangleright MethodName$
$?Action = SOAPReply \triangleright MethodName$
Data Rules
$Parameter_N = SOAPRequest \triangleright ParameterArray \triangleright Parameter_N$
$Parameter_N = SOAPReply \triangleright ParameterArray \triangleright Parameter_N$

Figure 5.8: Binding to concrete application-middleware automata

First, the *action rules* state how the action label (e.g. `add`) maps onto the field content within a protocol message described by the protocol's MDL. For example, the `operation` field of the GIOP Request message, and the `methodname` field of the SOAP request envelope. Secondly, the *data rules* state how the data parameters (input and output) map onto the field content of the protocol messages; for example the request action parameters (the X and Y integers) relate to the first two parameters in the ParameterArray field of the GIOP Request message. The return value parameter (the Z integer value) relates to the first parameter of the GIOP reply ParameterArray.

We finally illustrate the application of this binding procedure to the specification of a merged coloured application automaton. When executed the solution resolves both application and middleware heterogeneity. For this example, we continue with the simple addition example. However, in this case NS2 is a SOAP service that provides an add operation with an `int Plus(int, int)` signature whereas the IIOP client (NS1) interface signature is `int Add(int, int).` Hence, there is application heterogeneity in terms of the operation name. Figure 5.9 shows the result of the merged application automaton when bound to the concrete protocols. On the left side of the figure is the specified application merge, with the bi-coloured states representing the translation of parameters between actions. On the right side is the concrete merged k-coloured automaton, where the action transitions are bound to specific middleware protocols (the operation name difference is overcome by this, after an Add action is received a Plus action is sent). Note, the application translations are bound to the specific MTL translations based upon the previously specified binding rules for SOAP and CORBA (in Figure 5.8).

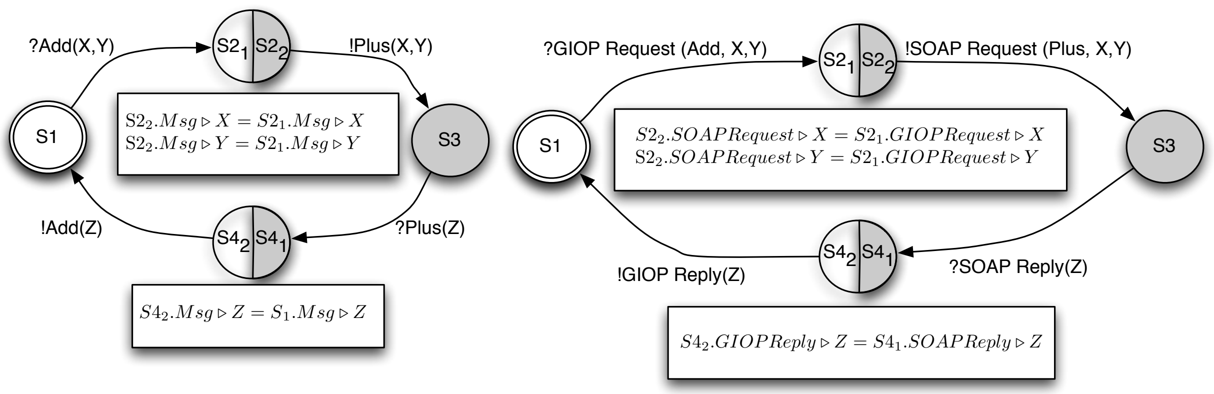

?Add(X,Y)
S2_1 S2_2
!Plus(X,Y)
S1
S2_2.Msg ▷ X = S2_1.Msg ▷ X
S2_2.Msg ▷ Y = S2_1.Msg ▷ Y
S3
!Add(Z)
S4_2 S4_1
?Plus(Z)
S4_2.Msg ▷ Z = S_1.Msg ▷ Z
?GIOP Request (Add, X,Y)
!SOAP Request (Plus, X,Y)
S2_2.SOAPRequest ▷ X = S2_1.GIOPRequest ▷ X
S2_2.SOAPRequest ▷ Y = S2_1.GIOPRequest ▷ Y
!GIOP Reply(Z)
?SOAP Reply(Z)
S4_2.GIOPReply ▷ Z = S4_1.SOAPReply ▷ Z

Figure 5.9: Construct a concrete merged application automaton

5.4.4 The Starlink Framework: dynamically interpreting middleware models

As illustrated in Figure 5.10, the Starlink framework interprets the previously described concrete models at runtime in order to support the necessary middleware behaviour on demand and essentially build and deploy a running CONNECTor.

The *network engine* sends and receives physical messages (i.e. data packets) to and from the network. A transition in the k-coloured automaton attaches network semantics to describe the requirements of the network. The network engine then provides the services to meet these requirements, which could include different types of transport or multicast behaviour. The current implementation of the network engine provides traditional TCP and UDP services for infrastructure networks. However, the architecture is configurable so that if Starlink were to be deployed in more heterogeneous environments, e.g. *ad hoc* networks, this network engine could be replaced with configurable services for *ad hoc* routing [26].

The *message parsers* read the contents of a network packet and parse them into the *AbstractMessage* representation such that the data can be manipulated during the mediation process. For example, if a HTTP message is received a HTTP parser reads all the fields of the header and body. Correspondingly, *message composers* construct the data packet for a particular protocol message, e.g. constructing the content for a HTTP GET message. Importantly, the message composers and parsers are generic reusable software elements that interpret high-level specifications of message content. The Message Description Language (MDL) specification (as described previously) specialises these generic components at runtime to create a specific protocol parser or composer.

5.4.5 CONNECT in Action: The Photosharing Application

We use a simple case-study to evaluate a subset of the integrated elements of the architecture; this evaluates the work related specifically to the progress of WP1. Note, that the architecture as a whole is evaluated against the common GMES case-study, and this evaluation is reported in Deliverable D6.3 [19].

The goal of this simple case-study is three-fold:

1. To demonstrate that the elements of the CONNECT architecture can work together to produce a CONNECTor that will connect two heterogeneous systems. Importantly, these systems differ in terms of their application and middleware behaviour; this is notably beyond what has been previously evaluated in Deliverables D1.1 [6] and D1.2 [4], where heterogeneity was resolved in a single dimension, i.e., either application of middleware heterogeneity.

2. To demonstrate that the production of a CONNECTor is automated and does not involve any “hand-coded” solutions. For this, the Synthesis enabler, given the specification of two

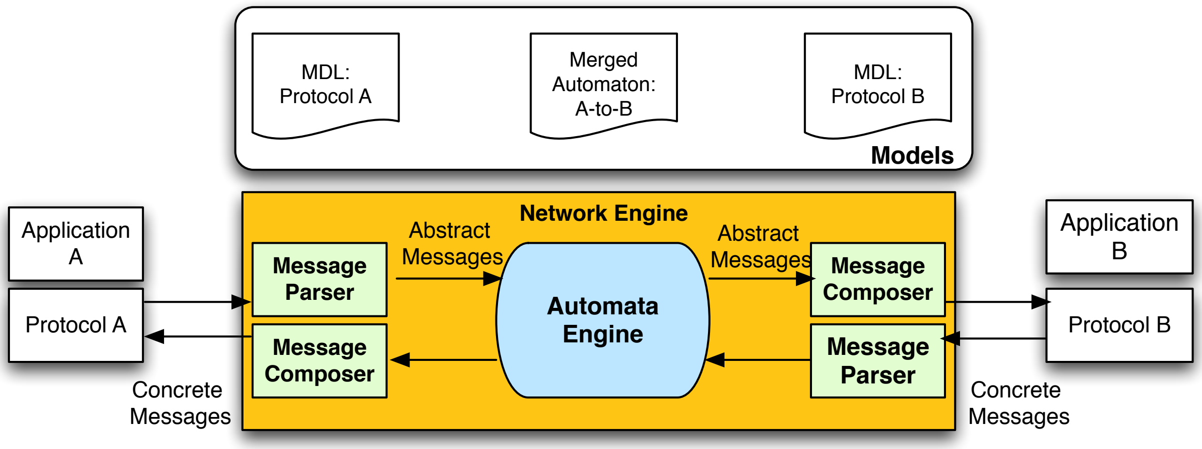


Figure 5.10: Architecture of the Starlink framework

networked systems produces the concrete specifications which are used to construct and deploy an instance of a CONNECTor. This is then deployed to connect the two systems such that they successfully interact with one another.

3. To investigate the performance measures of this particular CONNECT solution. For this purpose, we investigate the throughput measures of created CONNECTors in comparison to traditional middleware bindings, e.g. against the equivalent binding in IIOP.

The Case Study

The context of the case study is as follows. There are two networked systems exhibiting the following behaviour and requirements:

- **NS1: XML-RPC Photo Sharing Client**. This is a networked system looking for a photo-sharing service in order to search for photographs, share photographs, and read or add to photograph comments from other users. This client is implemented using the XML-RPC protocol and it performs lookup using the the WS-Discovery protocol. The operations available are listed in Figure 5.11.

- **NS2: CORBA Photo Sharing Service**. This is a networked system providing a photo-sharing service with functions such as uploading and commenting of photographs, and also keyword photograph search. This service is implemented using the CORBA protocols (specifically IIOP/GIOP) and it is advertised using the CDL description in the WS-Discovery protocol. The operations available are listed in Figure 5.12.

The heterogeneity between the two networked systems can be summarized as follows:

- The syntax of the application interfaces, e.g., the label of the search operation is `photoMetaData` in NS1 and `SEARCHPHOTOS` in NS2. Hence, there exist application differences.

- The two middleware protocols are XML-RPC by NS1 and IIOP by NS2 and hence the two cannot interoperate.

Constructing the NS Models

A concrete model is created of each networked system. This consists of a k-coloured automaton representing the behaviour of the system. The two systems are visually represented in figure 5.13. The complete XML specification of networked system one is found in Appendix 7.1, while the complete XML specification of networked system two is found in Appendix 7.2.

```
public interface Photo
  {
    // Search for Photos that match a keyword query
    // Return a list of PhotoMetaData describing each picture
    String[] photoMetaData (String query)

    // Retrieve the URL of the Photo to download
    String PHOTOFILE (String PhotoID)

    // Download the comments added to a Photo
    String[] PHOTOCOMMENT (String PhotoID);

    // Add a new comment to a given photo
    int AddCOMMENT (String PhotoID, String comment);
  };
};
```

Figure 5.11: Interface description of the XML-RPC implemented Photosharing application client

```
module PhotoSharing
{
    typedef sequence<string> PhotoMetaDataList;
    typedef sequence<string> PhotoCommentList;

  interface Photo
  {
    // Search for Photos that match a keyword query
    // Return a list of PhotoMetaData describing each picture
    PhotoMetaDataList SEARCHPHOTOS(in string query);

    // Retrieve the URL of the Photo to download
    string DOWNLOADPHOTO(in string PhotoID);

    // Download the comments added to a Photo
    PhotoCommentList DOWNLOADCOMMENT(in string PhotoID);

    // Add a new comment to a given photo
    oneway void COMMENTPHOTO(in string PhotoID, in string comment);
  };
};
```

Figure 5.12: Interface description of the CORBA implemented Photosharing application service

Here is can be seen that the transitions represent the sequence of operations that make up the application behaviour. These are then annotated with additional information describing which middleware protocol performs these transitions. In NS1 there is first the sending of the `photoMetaData` action and then the reception of the result as a list of strings. NS1 then downloads the photocomments of one photo identifier using the `PHOTOCOMMENT` action. The client then downloads the URL (performing the `PHOTOFILE` action so the photograph can be displayed. Finally, a new comment is added to the photograph using the `AddCOMMENT` action. The automaton is annotated as XML-RPC and hence the client executes these actions as XML-RPC request-response invocations.

Synthesizing and deploying the CONNECTor

Given the two NS models from the previous section as inputs, the synthesis enabler produces the mediator that merges the two together. To output is again a k-coloured automaton. This is visualised in Figure 5.14 and the concrete specification is given in Appendix 7.3. It can be seen that the specification mediates by translating between the two systems. There are two colours: one for the XML-RPC client and one for the IIOP server. Hence, the mediator executes

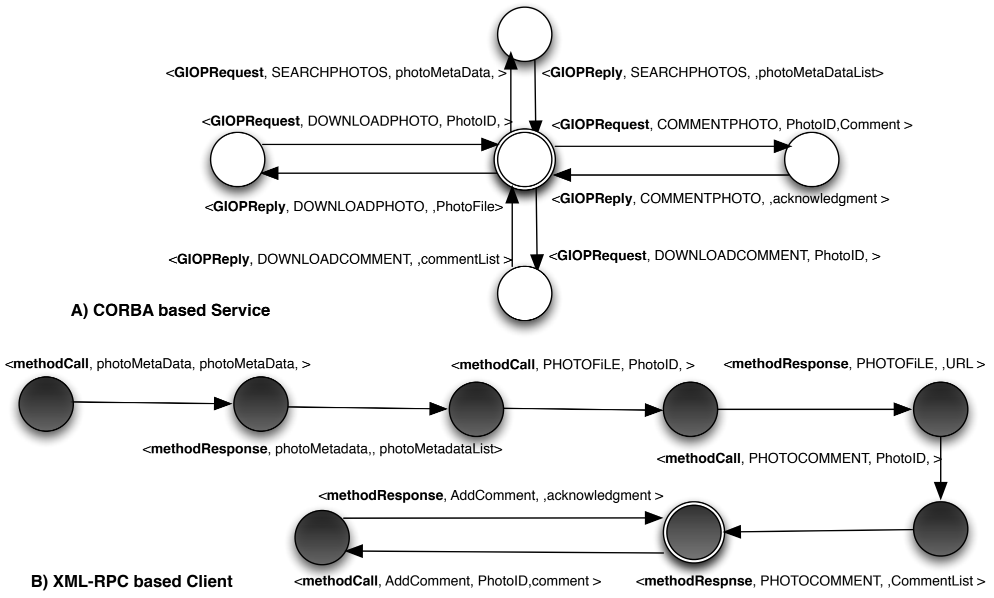
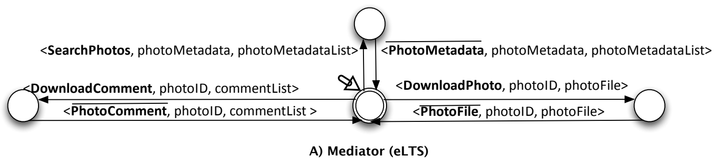

<GIOPRequest, SEARCHPHOTOS, photoMetaData, >
<GIOPReply, SEARCHPHOTOS, ,photoMetaDataList>
<GIOPRequest, DOWNLOADPHOTO, PhotoID, >
<GIOPRequest, COMMENTPHOTO, PhotoID,Comment >
<GIOPReply, DOWNLOADPHOTO, ,PhotoFile>
<GIOPReply, COMMENTPHOTO, ,acknowledgment >
<GIOPReply, DOWNLOADCOMMENT, ,commentList >
<GIOPRequest, DOWNLOADCOMMENT, PhotoID, >
A) CORBA based Service
<methodCall, photoMetaData, photoMetaData, >
<methodCall, PHOTOFiLE, PhotoID, >
<methodResponse, PHOTOFiLE, ,URL >
<methodResponse, photoMetadata,, photoMetadataList>
<methodCall, PHOTOCOMMENT, PhotoID, >
<methodResponse, AddComment, ,acknowledgment >
B) XML-RPC based Client
<methodCall, AddComment, PhotoID,comment >
<methodRespnse, PHOTOCOMMENT, ,CommentList >

Figure 5.13: Behavioural description of the heterogeneous photo-sharing networked systems


<SearchPhotos, photoMetadata, photoMetadataList>
<PhotoMetadata, photoMetadata, photoMetadataList>
<DownloadComment, photoID, commentList>
<DownloadPhoto, photoID, photoFile>
<PhotoComment, photoID, commentList >
<PhotoFile, photoID, photoFile>
A) Mediator (eLTS)

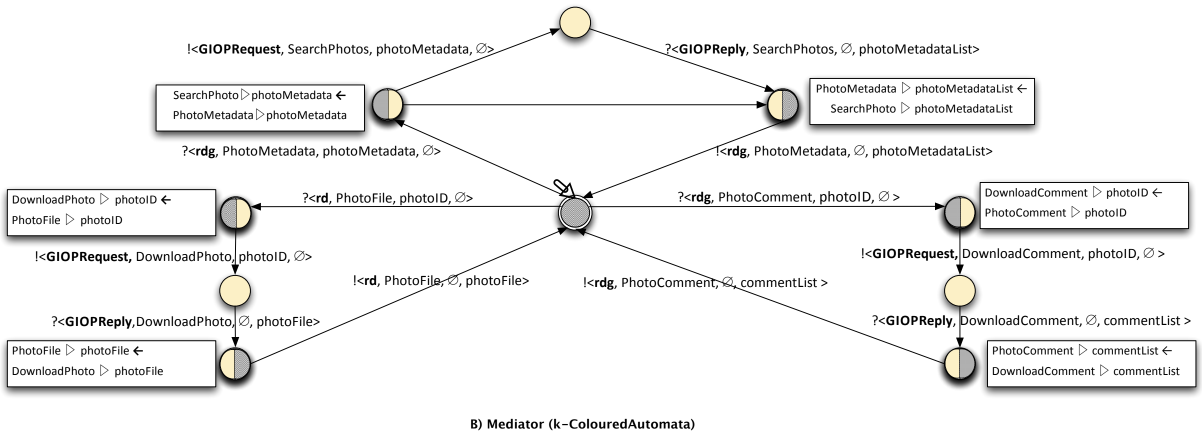

!<GIOPRequest, SearchPhotos, photoMetadata, ∅>
?<GIOPReply, SearchPhotos, ∅, photoMetadataList>
SearchPhoto ▷ photoMetadata ←
PhotoMetadata ▷ photoMetadata
PhotoMetadata ▷ photoMetadataList ←
SearchPhoto ▷ photoMetadataList
?<rdg, PhotoMetadata, photoMetadata, ∅>
!<rdg, PhotoMetadata, ∅, photoMetadataList>
DownloadPhoto ▷ photoID ←
PhotoFile ▷ photoID
?<rd, PhotoFile, photoID, ∅>
?<rdg, PhotoComment, photoID, ∅ >
DownloadComment ▷ photoID ←
PhotoComment ▷ photoID
!<GIOPRequest, DownloadPhoto, photoID, ∅>
!<GIOPRequest, DownloadComment, photoID, ∅ >
!<rd, PhotoFile, ∅, photoFile>
!<rdg, PhotoComment, ∅, commentList >
?<GIOPReply,DownloadPhoto, ∅, photoFile>
?<GIOPReply, DownloadComment, ∅, commentList >
PhotoFile ▷ photoFile ←
DownloadPhoto ▷ photoFile
PhotoComment ▷ commentList ←
DownloadComment ▷ commentList
B) Mediator (k-ColouredAutomata)

Figure 5.14: Behavioural description of the mediator

an action with the appropriate colour for the particular system and at the important bridge states it translates the application and middleware differences.

Finally, the deployment enabler takes the three XML models and uses the exact code specification given in Figure 3.10 to construct a new CONNECTor instance which is executed. From this point forward, when the networked systems execute they directly and automatically interact with this created CONNECTor.

Table 5.3: Comparison of Latency and Throughput of a CONNECTor

Implementation Case	Throughput (msgs/sec)
XML RPC	205
CORBA	1090
IIOP to XML-RPC CONNECTor	7

Table 5.4: Resource costs of a CONNECTor

Measure	Value
Time to create	978 ms
Dynamic Memory Footprint	708 KBytes

CONNECTor Performance Measures

To measure the performance overheads of the constructed CONNECTors we created three implementations of the photosharing application. In the first case, we executed both client and server networked systems implemented in XML-RPC (for the Photosharing interface). We then executed 1000 executions of the client and measured the median throughput of the connection. We used the Apache implementation of XML-RPC[11] to implement the photosharing application in XML-RPC. In the second implementation, we executed a client and server both implemented in CORBA; for this we used the ORB provided with Java version 1.6. Finally, the third implementation is the CONNECT produced version. All experiments were run on a machine running Windows 7 using Java 1.6; the machine specifications were: 4Gb memory and an Intel Core 2 Duo 2.97 GHz. The results of these experiments are shown in Table 5.3. Table 5.4 also shows the overhead of one instance of a CONNECTor in terms of the time to create and also the memory cost at runtime.

Analysis

It can be seen that (as expected) the CONNECT version's throughput is an order of magnitude slower than the legacy protocols. This is to be expected; indeed its maximum is bound by the XML-RPC throughput (this is the server the CONNECTor is using). Further the memory costs and time to create a CONNECTor are high. However, the prototypes are experimental; the first aim in the third year of the project was to achieve working interoperability solutions. Now that this has been achieved, but with low performance measures, one goal in the final part of the CONNECT project is to optimise the operation of these CONNECTors.

5.5 Interoperable Live Streaming

The role of this section is to discuss the challenges of applying the CONNECT interoperability approach for the particular application domain of Live Multimedia Streaming in relation to technologies developed by Ambientic for Collaborative Mobile Applications.

The CONNECT architecture enables incompatible Networked Systems to seamlessly communicate by means of runtime-deployed mediators (or CONNECTors). A particular class of NSs where the CONNECT approach is beneficial is represented by mobile devices. Mobile networked devices such as smartphones or tablets now support a large array of applications. Such devices also have a more particular architecture than other types of networked systems, for example the ability to communicate over multiple radio links. Thus, the deployment of CONNECT-based interoperability solutions in mobile environments should also take mobility-related aspects into account. Further, focusing on Live Multimedia Streaming, efficiency plays a key role in supporting interoperability solutions. This is mainly because multimedia streaming applications are resource-intensive, and commonly rely on hardware optimizations to run on resource-

[11] http://ws.apache.org/xmlrpc/

constrained devices. At the same time, node mobility requires applications to be infrastructure-independent in order to function in fully-distributed environments(such as ad hoc networks).

The next sections elaborate on the application of CONNECT interoperability solutions to enabling Live Multimedia Streaming among heterogeneous mobile devices. Section 5.5.1 first highlights the specific challenges of Live Mobile Streaming and especially the ones related to interoperability. Section 5.5.2 then assesses the CONNECT architecture to meet such challenges, leading to introduce the AmbiStream architecture in Section 5.5.3. AmbiStream revisits the CONNECT architecture for supporting interoperable live streaming in the mobile environment. Further, in Section 5.5.4, we present iBICOOP a partial and lightweight implementation of CONNECT Enablers for mobile platforms (currently available for Android, iOS, Blackberry, Windows Phone 7 and J2ME platforms). iBICOOP is a technology intended to facilitate the creation of *Collaborative Mobile Applications* by providing a mobile-specific CONNECT Discovery Enabler together with an Enabler for communication in heterogeneous multi-network environments. In addition to iBICOOP, in Section 5.5.5 we discuss its integration with the AmbiStream CONNECTor. We then also extend the iBICOOP multi-radio Communication Enabler to allow its use with multiple *Multimedia Live Streaming Applications* (since it was originally designed for discrete communications) and also improve streaming scalability for *Group Communication*. Finally, in Section 5.5.6, we assess the applicability of AmbiStream on concrete mobile devices, by an initial series of experiments on Android and iOS, targeting multi-platform and multi-protocol live streaming.

5.5.1 Challenges for Live Mobile Streaming

The new generation mobile phones provide the means of exchanging high volumes of data over multiple wireless links, as well as capturing, managing and producing multimedia content using on-board cameras. All mentioned technologies should enable mobile phones to support richer multimedia interactions and applications. Applications such as *Videoconferencing* and *Live Video Broadcast* over the Internet require that audio and video data are transported over various networks, with strict requirements. But node heterogeneity and network heterogeneity raise the complexity of designing interoperable applications on existing mobile devices. Solving this complexity will enable mobile applications using streaming media to be designed, implemented and deployed much easily.

Architectures for multimedia streaming on the Internet such as [15, 35] assume the existence of powerful servers that can adapt content on behalf of clients, which is infeasible in fully-distributed environments, where the streaming server is a resource-constrained smartphone. Existing mobile platforms such as Android, iOS, Blackberry and Windows Phone 7 provide support for multimedia streaming either through platform-specific APIs or system services. However, they use heterogeneous protocols and multimedia container formats, thus compromising interoperability. Interoperability solutions for mobile streaming in ad hoc networks [21] mainly follow a cross-layered design to achieve content adaptation. Such approaches, where cooperation between different system layers is required are not feasible on most existing smartphone platforms.

We organize the challenges of enabling *Live Interoperable Streaming* on mobile platforms, at the same time aligning to the overall challenges of CONNECT presented in Deliverable 1.2 [4], as:

1. *Interaction protocol heterogeneity.* Multiple incompatible multimedia streaming protocols exist today, and each mobile platform supports one or a small subset of them. As a result, smartphones must overcome the streaming protocol heterogeneity problem to be able to exchange multimedia streams with heterogeneous devices.

2. *Data heterogeneity.* Each smartphone platform generates, manages and streams multimedia data using some specific multimedia container formats (e.g., RIMM streaming file format for Blackberry smartphones), usually depending on the streaming protocols it supports. These data cannot be directly transmitted through a different streaming protocol because the multimedia container format is specific to the protocol. Smartphones, then,

must also adapt the media container format to enable translation from the native streaming protocol to non-native protocols supported by other peers.

3. *Application heterogeneity.* Multimedia streaming is used in a number of types of applications such as video broadcast, video conferencing and place shifting each having different requirements in terms of quality. For example videoconference applications require very low delay and precise stream synchronization (e.g., less than 200ms of latency between audio and video streams to enable lip synchronization).

4. *Mobile communication heterogeneity.* Mobile applications are designed to function in changing contexts and environments. Thus, smartphones are equipped with multiple network interfaces such as Cellular (3G/4G), Bluetooth, WLAN and even software abstractions over physical links such as WiFi Direct (allowing multiple WiFi devices to communicate simultaneously without the need of an access point). Application developers have to consider aspects such as multiple data links, communication restrictions and reachability issues (i.e., devices under NATs or behind Firewalls).

5. *Mobile streaming scalability.* Another recurring problem with streaming media is scalability. Video broadcast services must provide excellent scalability while using the least amount of bandwidth. Addressing this challenge, UDP Multicast allows sending one packet to multiple destinations subscribing to a group. But multicast over the Internet is still not supported by most ISPs. Creating multiple unicast connections is not an option, since current mobile phones can only handle a few unicast connections and are restricted in terms of bandwidth achievable by radio links. Most NAT traversal solutions based on relays (e.g., SOCKS5 [20]) are mechanisms designed to achieve transparent unicast, and thus fail to provide a solution for resource efficient group communication.

5.5.2 CONNECT Architecture vs. Interoperable Live Streaming in the Mobile Environment

The exploitation of the CONNECT architecture for enabling interoperable live streaming in the mobile environment decomposes into two complementary issues: (i) deploying the CONNECT architecture in mobile environments, and (ii) applying the CONNECTor architecture to enable mediation across heterogeneous live streaming solutions.

CONNECT in the mobile environment

CONNECT is constructed in the form of multiple *Enablers* collaborating to produce on the fly *Mediators* between heterogeneous networked systems. The architecture does not specify the place where the *Enablers* (such as *Discovery* and *Synthesis*) are deployed, but requires that communication between enablers is possible by the means of *Message Channels* and *Queues*. Smartphones are networked systems running in heterogeneous environments, where context changes are frequent. In the context of collaborative applications, mobile communities can appear and dissolve in very short time frames, thus providing an *infrastructure-less* solution is important, as illustrated in the example below.

Example: A group of tourists, participating in a guided tour, are using their mobile phones to CONNECT to a *Simultaneous Interpretation (SI)* service depending on their preferred language. Given that tours are organized in various (and possibly remote) places, access to infrastructures can not be guaranteed. This is why the necessary interoperability *Enablers*, or at least the concrete CONNECTors, should be deployed on specific devices. In this case, a valid choice might be to use the interpreter's smartphones, since they are in close proximity of the tourists.

An important aspect with respect to the place of deployment of CONNECT is scalability. Returning to the previous example, and supposing that the subgroup of people interested in listening to the English comments has 50 participants, the Interpreter's phone must be able to instantiate 50 CONNECTors (for a number of protocols).

On the fly synthesis and deployment are also hard to achieve on mobile platforms, mostly because of resource constraints, platform restrictions and unexpected context changes. A common restriction is that code can not be compiled on the device on most existing mobile platforms, and this will restrict the deployment to interpreted only CONNECTors. This will result in poor performance on restricted devices, or in the failure to deploy CONNECTors for resource-intensive applications.

While modern smartphones run heterogeneous operating systems, there is also a high fragmentation in terms of features offered by devices running specific platforms. Furthermore, given that smartphones are equipped with multiple wireless communication links with different restrictions and transport types, assuring interoperability between mobile devices requires at least one more *Enabler* to assure seamless communication. The iBICOOP middleware technology for mobile platforms presented in Section 5.5.4 implements a *Communication Enabler* dedicated to the heterogeneous mobile environment. The communication enabler can seamlessly manage connections on multiple network interfaces, transports, and also resolve network reachability issues such as NAT traversal.

Live streaming applications are resource intensive and some require real-time responsiveness. This constraint requires that for specific applications such as live multimedia streaming, CONNECTors to be compiled. Because on-board compilation is not supported by mobile platforms, CONNECTors need to be compiled and deployed in advance on such devices.

CONNECT and Interoperable Live Streaming

Writing protocol mediators for each existing streaming protocol implies a high development effort given the important number of protocols and mobile platforms. CONNECT enables automated mediation between different protocols, but does not currently take into account the challenges introduced by mobility and particularly live streaming on mobile platforms. We base our solution for streaming protocol interoperability, called AmbiStream (which is presented in Section 5.5.3) on Starlink (Deliverable D1.2 [4]) and further adapt (and improve) it with respect to our specific scenario.

Live streaming protocols, either real-time or non real-time, commonly use two communication flows: one for assuring *Stream Control*, and another for *Media Transport*. Multimedia container formats are multi-layer wrappers for the transport of audio and video streams. Multimedia containers are required for any live streaming protocol because Audio and Video frames (or samples) cannot be directly transferred over IP networks, since they lack the necessary meta-information required to facilitate correct decoding and presentation at the receiver side. The entire process of wrapping and unwrapping elementary stream packets is sometimes referred to as multiplexing and respectively demultiplexing (or demuxing). This results from the fact that some container formats (e.g., MPEG-TS) frames, samples, or sample fragments from multiple audio and/or video tracks are interleaved. Such packets also include mechanisms for synchronization control and signalling.

In some cases, parsers and composers for low complexity message formats can be easily synthesized using application interface descriptions provided by discovery protocols (e.g., UPnP). On the other hand, multimedia container formats cannot be synthesized in such a manner given their increased complexity. However, since that they are specific to streaming protocols, the CONNECTor must be able to interpret them.

The CONNECT Synthesis Enabler produces MDL descriptions which are passed to Starlink to generate parsers and composers of protocol messages. Data values contained within message fields are transformed into primitive types or sequences of primitive types. Depending on the application, message formats can be of low complexity (e.g., text, XML) or higher complexity which is the case of multimedia container formats. Given that application data formats can not be easily synthesized, a solution is to enable an "assisted synthesis", where the CONNECT Synthesis Enabler would create a partial mediator, to be completed with an application data description. Such

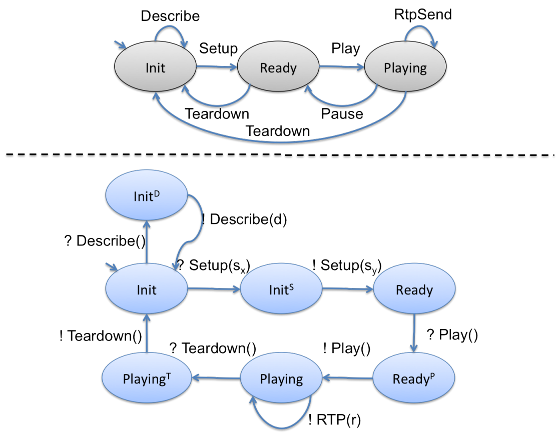


Figure 5.15: Simplified LTS describing the behaviour of an RTSP Server

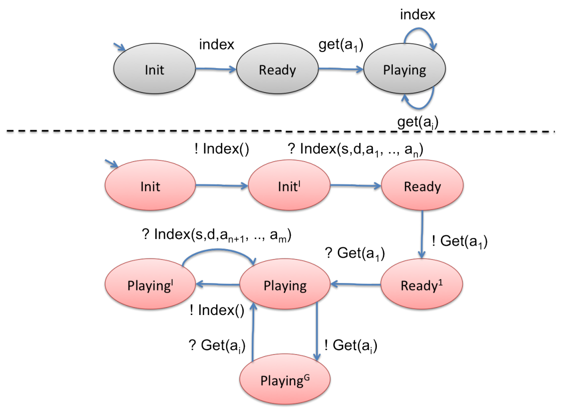


Figure 5.16: Simplified LTS describing the behaviour of an HLS Client

a model could be obtained at run-time from a database. However, as observed with concrete protocols there isn't a clear separation between application data formats and protocol message formats. In the worst scenario, the two layers are highly interleaved.

Another requirement in order to enable interoperability between live streaming protocols is to manage timing from two perspectives: first, real-time streaming protocols impose temporal constraints on the arrival and inter-arrival of packets, and secondly, streaming protocols manage flows of data, and in order for the content to pass over packet networks buffering is required. Buffering techniques depend with protocols, and it is thus the role of the mediator to solve this type of heterogeneity.

5.5.3 The AmbiStream Architecture: CONNECTor Architecture for Mobile Interoperable Live Streaming

AmbiStream is designed to overcome interoperability challenges of live multimedia streaming applications on mobile phones. Contrary to other existing interoperability approaches which rely on powerful centralized infrastructures (e.g., Wowza Media Server [12]), AmbiStream follows a fully-distributed architecture (deployed on mobile devices) enabling interoperability without depending on existing network infrastructures.

AmbiStream provides multimedia streaming interoperability amongst heterogeneous mobile devices with the following assumptions: (i) Both the Source and the Destination support a common pair of audio/video codecs and (ii) the codec pair used is compatible with the Destination's (client-side supported) streaming protocol. Multimedia codec translation is not necessary in most cases since a small set of encoders/decoders are available on most mobile platforms and are compatible with many of the existing streaming protocols. For example, the video codec H.264/MPEG-4 AVC (ISO/IEC 14496-10 MPEG-4 Part 10, Baseline profile) is supported on Android (RTSP), iOS (HLS), Windows Phone 7 (IIS MSS) and Blackberry (RTSP).

Further in this section we motivate the rationale behind our design solution based on a concrete example of live streaming protocols: RTSP and HLS. Then, we approach the challenges introduced by the chosen example. In the final part of Section 5.5.3 we describe the overall architecture of the AmbiStream and present the DSLs introduced for modelling the mediator.

From CONNECT to AmbiStream using a concrete example

We choose Real Time Streaming Protocol [29] as the source (or server) protocol and HTTP Live Streaming [23] as the client. The simplified behaviour of the RTSP server is presented in the

[12]http://www.wowza.com/

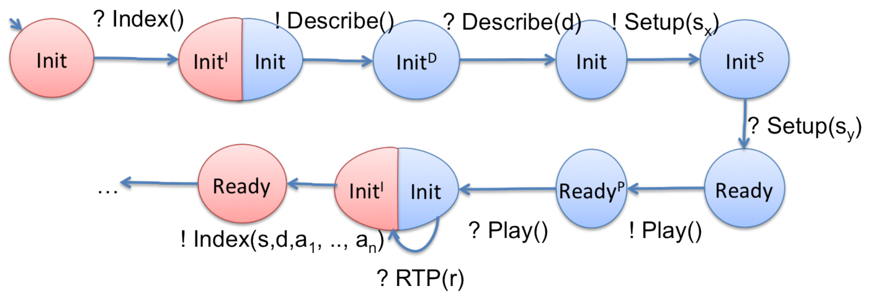


Figure 5.17: Fragment of the k-Coloured Automaton of the RTSP-HLS mediator

top part of Figure 5.15 and the behaviour of the HLS client in the top part of Figure 5.16 in the form of labelled transition systems. As it can be easily observed, although the two protocols are quite different in design, the first being real-time and the second high-latency, the application states are identical. Both protocols mainly follow a request-response messaging pattern, thus each transition must be duplicated in request and response actions. This is shown in the bottom part of the two figures, and further detailed below. The LTSs share the same definition as the k-Coloured Application Automaton presented in Section 4.3.3 of Deliverable D1.2 [4]. In brief, a k-Coloured Application Automaton defines the sequences of ordered application actions, where an application action is either a *send (!)* or a *receive (?)* action, while the colour is used to identify the application.

RTSP Server: The RTSP server receives a *setup* message from the client meant to describe the client's connection request. The server responds with a *setup* response containing information such as a session ID and transport ports. While in *Ready* state, the server can receive a *play* request. For simplicity we chose the TCP interleaved transport mode, where multimedia packets are sent in-band over the same TCP connection as the control messages. In *Playing* state, the server will constantly send one-way RTP messages. The stream is interrupted when the server receives a *teardown* message. RTP is a lightweight wrapper for audio or video samples. RTP messages are sent at a frequency chosen by the server. The message frequency is based on the media sampling frequency (e.g., 90000Hz in most video codecs). The RTP frequency is not equal to the sampling frequency because large samples are fragmented in multiple RTP messages sharing the same *Timestamp*.

HLS Client: The HTTP Live Streaming Protocol uses HTTP as a transport protocol for both session control and stream data transport. The basic message flow of this protocol is as follows: the client application sends an HTTP request to download an extended M3U playlist (*! Index()*). The playlist received contains a sequence number, a list of stream *chunks* (i.e., $a_1, a_2, .., a_n$) and also a chunk duration, representing the play-time in seconds of one chunk. The client starts downloading stream chunks in order (*? Get(a_i)*). Intuitively, HLS is actually breaking the overall stream into a sequence of small HTTP-based file downloads. When then list of chunks is completed, the client requests a new, updated *Index*, from the server. This process can be repeated, supporting in this manner unbounded transport streams.

Analysing Protocol Heterogeneity

A number of important differences with respect to multimedia streaming can be observed between the two protocols:

1. HLS chunks are much larger than RTP messages, and thus contain a greater stream duration. Common values are 5 to 30 seconds for a HLS audio/video chunk, and less than 30ms for an RTP video packet[13].

[13]Configurations commonly observed in existing applications

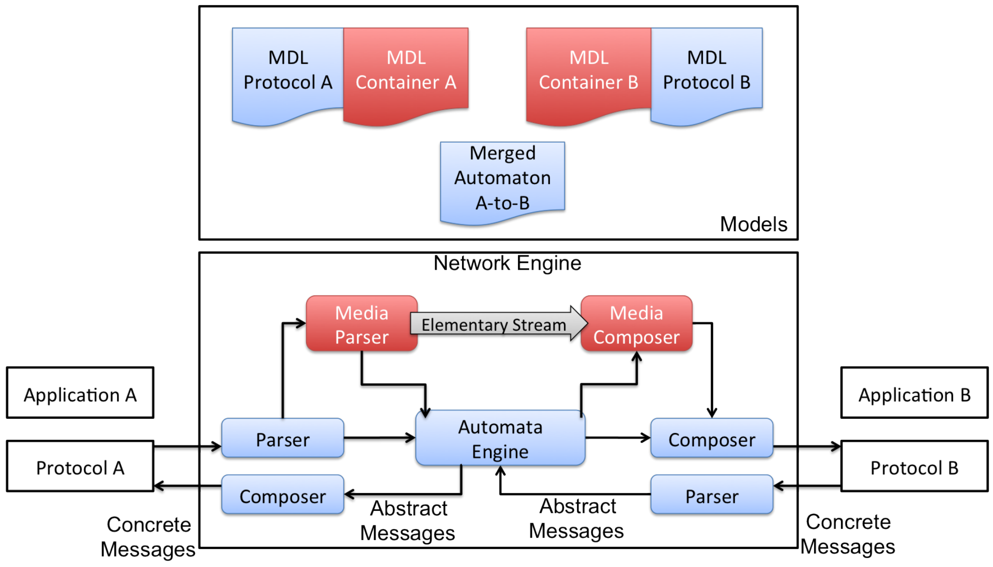


Figure 5.18: Starlink architecture extension for live streaming

2. HLS is not a real-time streaming protocol. Live streams are delayed by at least three times the duration of a chunk (e.g., 3 x 5 seconds) and in case of network congestion, this delay can further increase (this delay can be bounded by server configuration by reducing the number of buffered chunks).

3. The description of the transported stream in terms of: audio/video codec, decoder profile, sampling frequency and channel identification are provided as a text description for RTP (using Session Description Protocol (SDP), IETF Proposed Standard as RFC 4566/2006). For HLS such information is not available at protocol level, but can be extracted from the MPEG-TS multimedia container used to wrap the stream chunks.

4. The HLS client uses a request-response pattern to obtain stream data while RTSP uses one-way RTP messages.

5. An HLS client requires a large media buffer (which should be available for download from the server in the form of chunks, e.g., 15 seconds) before starting presentation, while RTSP only requires a small buffer (usually $<$1s) on the client side in order to eliminate packet inter-arrival jitter.

Improving the CONNECTor architecture for live streaming support

Considering the architecture of Starlink, presented in Figure 5.10, we see that only message formats are managed (by the means of MDL Models) and not application data formats. This is because Starlink was not initially designed to deal with application data heterogeneity. In the particular case above for RTSP to HLS, the stream description is provided at protocol messaging level (using SDP) for RTSP and at the application data level (inside the MPEG-TS multimedia container) for HLS. To better illustrate this case, we consider the data field identifying the audio (or video) codec used in a streaming session for RTSP and HLS (Figure 5.19). For RTSP this information is in the form of a string-encoded field (“H264”) of the *Describe* message. In the case of HLS, the same codec is identified by the one byte code $0x1B$ of a Packetized Elementary Stream Message, wrapped inside a sequence of MPEG-TS Packets and further contained in an HTTP response message.

To address this cross-layer dependency, in Figure 5.18 we suggest an improved framework structure, by adding two more abstract models for multimedia container formats, and an extension to the *Automata Engine* to allow the transfer of messages between the application data and

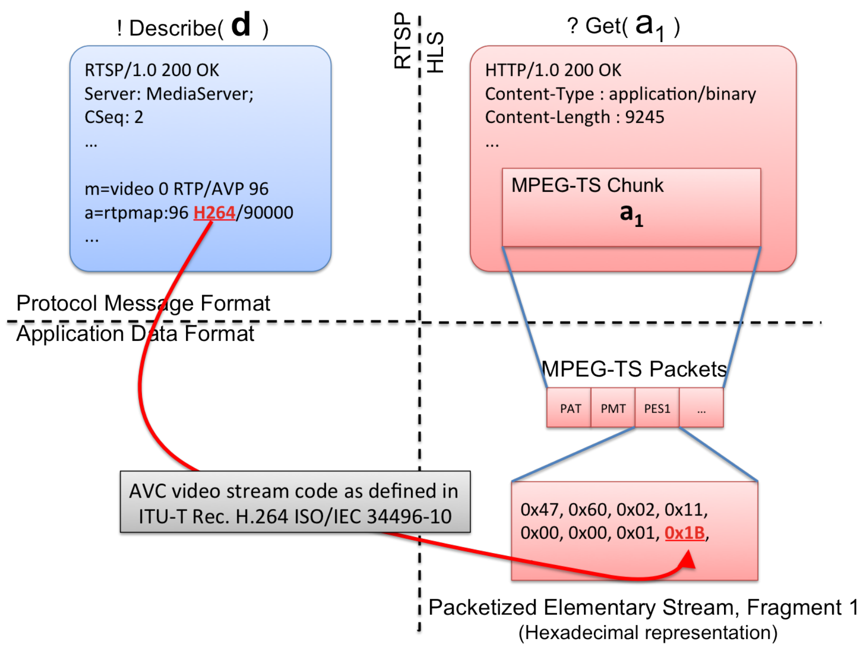


Figure 5.19: Example of cross-layer message field mapping

the protocol layer. As opposed to the protocol message descriptions (MDL Protocol A, MDL Protocol B in Figure 5.18), and as stated in the previous section, application specific data formats can not be automatically synthesised by the *Synthesis Enabler*. In order to support the definition of such models, further in this section, we present the AmbiStream *Multimedia Container Format* DSL. This language can be used to specify the application data formats of multimedia streaming applications. As seen in Figure 5.18, one multimedia container description will be used to generate a *Media Parser* for incoming stream data packets (in the case presented above, RTP), and another for the *Media Composer* of the second protocol's data format (MPEG-TS). While multimedia flows are only transmitted in one direction, applications such as Videoconferencing need to manage two flows in opposite directions. This requires to mirror the newly introduced components as a means to add a second flow. In the case of multimedia container formats there is no need to synthesise a mapping between the input messages of the multimedia container format and the messages of the output multimedia container format. This is mainly because any multimedia stream can be transformed to a common elementary format called *Elementary Stream*. An *Elementary Stream (ES)* as defined by Moving Picture Experts Group[14] is usually the raw output of an audio or video encoder. In Figure 5.18 the multimedia flow is shown in the form of an arrow labelled *Elementary Stream*. Going back to the concrete example above, while both the server and client are in *Playing* state, the *Media Parser* should produce audio/video samples at the content sampling frequency (e.g., PCM audio encoding at 8,000 Hz). The *Media Parser* should also pass all stream description data to the *Automata Engine* to be used at protocol message level, and the *Automata Engine* should as well return multimedia specific information to the *Media Composer*.

Earlier, in Section 5.5.2, we mentioned that timing is an important dimension for assuring interoperability between streaming protocols. Message timing is not currently addressed in the CONNECT Mediator model. This is because most protocols have loose requirements with respect to message timing, with the sole role of assuring that opened network connections (which consume resources such as processing time and memory) do not persist indefinitely. Such enforced time-out events (e.g., the default connection timeout for persistent connections of Apache

[14]http://mpeg.chiariglione.org/

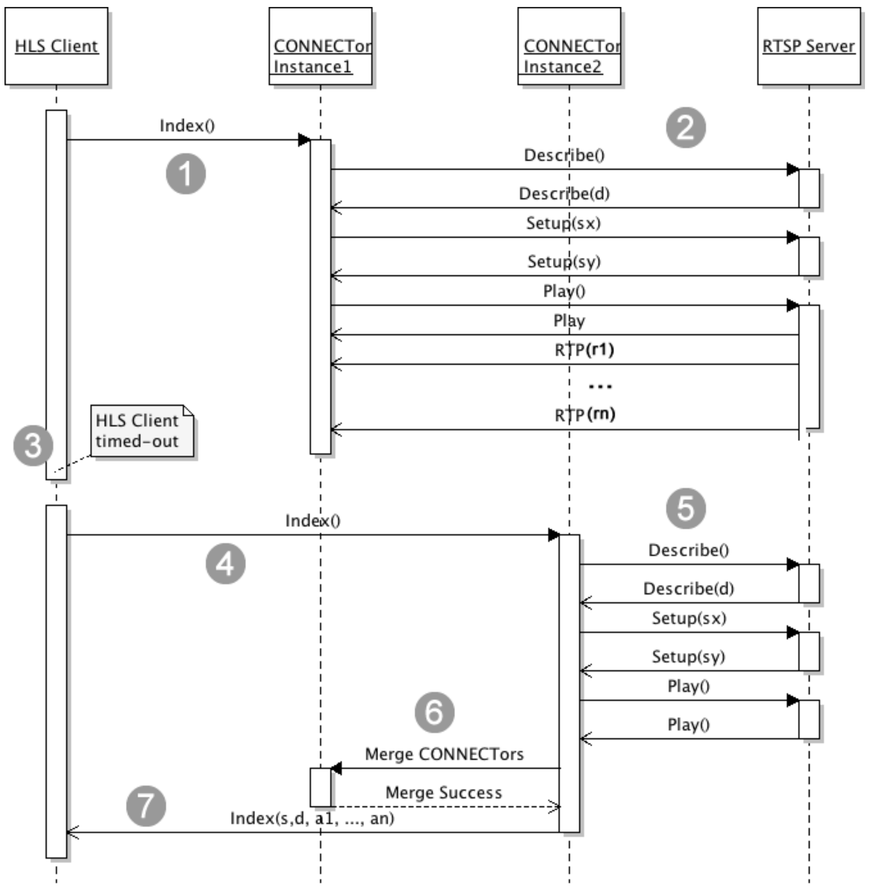


Figure 5.20: Sequence diagram illustrating the merge of two RTSP-to-HLS CONNECTors

2.0 httpd server is 15 seconds [15]) do not necessarily pose an obstacle to interoperability.

On the contrary, real-time streaming protocols impose strict requirements on packet arrival and also on the inter-arrival variation (or jitter) as a means to assure the quality of service. This is why any processing done in-the-middle by a mediator should treat such requirements explicitly. To assure real-time streaming, multimedia packets (Figure 5.18, Elementary Stream) must be processed with respect to timing. First, packets which arrive late or are delayed during processing should be dropped at any phase by the mediator. This QoS policy takes place inside the *Media Parser* and *Media Composer*.

In the RTSP-to-HLS example, mediation is done between a real-time protocol and a non real-time protocol. This application driven heterogeneity leads us to the second timing related challenge of streaming protocols, which we call *application buffering requirements*. Buffering techniques differ depending on the streaming protocol either being done on the client side, on the server side or both. This is why buffering requirements of each protocol should be managed by the mediator. In the RTSP-to-HLS example, we see that the first request of the HLS client *!Index()* (Figure 5.17) triggers an *?Index(s, d,* $a_1, a_2, .., a_n$*)* response from the server. The fields of this message are: the sequence number *s*, the duration *d* in seconds of each chunk, and a list of stream chunk URLs $a_1, a_2, .., a_n$. The mediator is free to choose any chunk duration, but according to the HLS specification [23] “the client SHOULD NOT choose a segment which starts less than three target durations from the end of the Playlist file”. In other words, at least three chunks should have already been cached by the server (in our case by the mediator) by the time of the *Index* request. Since the RTSP server does not buffer data, the mediator must assure it, but in this case, delaying the HLS *Index(s, d,* $a_1, a_2, .., a_n$*)* response by three chuck durations will

[15]http://httpd.apache.org/docs/2.0/mod/core.html#keepalivetimeout

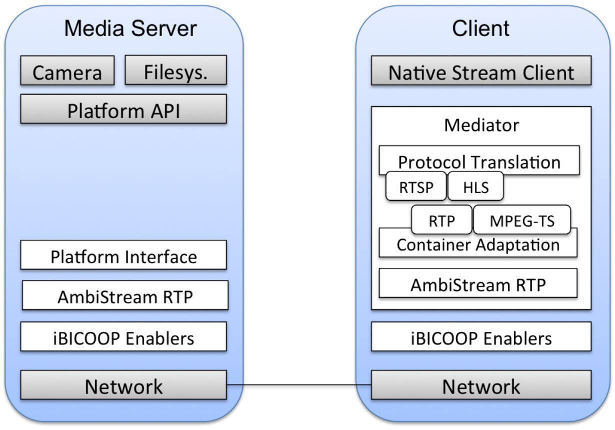


Figure 5.21: The AmbiStream architecture

exceed the HTTP response time-out of the client application.

Based on the CONNECT Mediator definition (refer to Section 4.3.3 of Deliverable 1.2 [4]) the two presented networked systems are semantically equivalent, that is, there exists a mapping to merge their respective colored LTS (in Figures 5.15 and 5.16) into a k-colored automaton (represented in Figure 5.17). However, the CONNECTor will fail at run-time because the *?Index()* request (in Figure 5.17) will trigger a time-out. This problem was anticipated in Section 4.4.3 of Deliverable 1.2 [4] where, in the case of the Bonjour-to-SLP experiment, mediation presents a 600 percent increase in response-time while still being low enough not to trigger time-out.

We propose a solution, for solving streaming application heterogeneity with respect to buffering requirements, based on the principle of locality. The principle of locality is widely used in many areas of computer science for a number of optimizations of systems, like: caching, pipelining, instruction prefetch, etc. If buffering (or in the more general case, a long operation) done by a CONNECTor triggers a time-out event on one side, the CONNECTor should be kept active with the other NS and not take any transition involving the (disconnected) NS. We first employ the principle of temporal locality. If, shortly after a first session triggering a response time-out, a second mediation session is initiated between two NSs, and the systems reach the same protocol state (that previously triggered the time-out), the two CONNECTor instances can be merged. By doing so, the time-out will not occur in the second session, because the first CONNECTor instance was able to complete the long operation (multimedia buffering in our case) during the elapsed time. Of course, this solution also assumes branch locality, that is, the second session between the two NSs will follow the same transition sequence as the first one. We conclude that reaching an equivalent state is necessary to merge two CONNECTor instances.

We illustrate the presented solution for the case of the RTSP-HLS example in the form of a sequence diagram (in Figure 5.20). The flow of messages used in the sequence diagram is given by the merged k-coloured automaton in Figure 5.17. At phase 1 the HLS client sends an *(*Index()) request to the CONNECTor. Next, in 2, the CONNECTor opens a connection to the RTSP server and initiates a streaming session by the sequence of request-response messages: *Describe*, *Setup*, *Play*. At this point, the RTSP server will start sending RTP messages, that will be buffered by the mediator in order to meet the requirements of the HLS client in terms of duration of the initial stream "chunk". Because the buffering period is greater than the response time-out enforced by the client, the HLS client will disCONNECT at step 3. We assume that the HLS client will retry to establish the connection by making an identical request *Index()* (marked as point 4). This request is treated by a second CONNECTor instance, which follows the same transition path, and eventually arriving in the same system state that triggered the time-out of the client. At this point 6, the CONNECTor should verify that merging is possible (i.e., the states are indeed equivalent). Because the stream "chunks" $a_1, a_2, .., a_n$ were pre-buffered by the first CONNECTor instance, at point 7 the *Index(s, d,* $a_1, a_2, .., a_n$*)* response is delivered immediately and not triggering time-out.

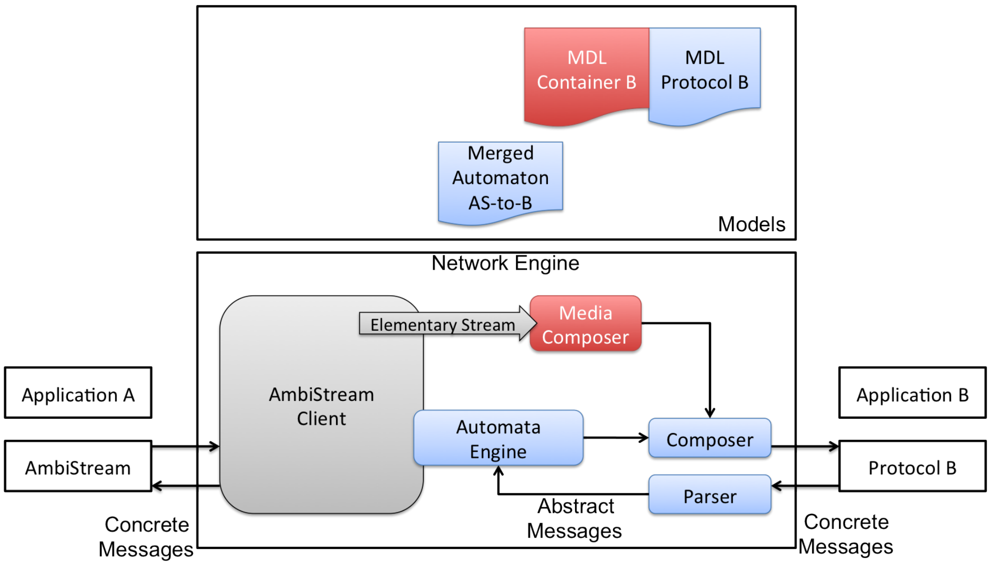


Figure 5.22: The AmbiStream CONNECTor Architecture

AmbiStream CONNECTor Architecture

The overall architecture of AmbiStream is presented in Figure 5.21. It includes a compile-time CONNECTor which can be deployed on the streaming client device. AmbiStream enables multimedia streaming protocol interoperability in two directions: *first*, it enables the translation between real-time and on-demand streaming protocols, which requires buffering, dropping and combination of messages to deliver time-sensitive data at the right intervals, *second*, we support adaptation of container formats, which in the case of multimedia are dependent on the streaming protocol. Both operations are done by the CONNECTor, which is specified by the means of high level models using a set of Domain Specific Languages described below.

The AmbiStream CONNECTor is a simplified version of the one introduced above following the RTSP-to-HLS example (Figure 5.18). The CONNECTor in the form it is currently implemented by AmbiStream is presented in Figure 5.22. One of the changes is the use of an intermediary server protocol (AmbiStream Protocol) and *RTP* [28] as an intermediate media transport ("AmbiStream RTP"). As a consequence, AmbiStream translates from the intermediate protocol to each existing streaming protocol, thus considerably reducing the total number of CONNECTors required. However, this does not imply that any of the mobile platforms or devices should support this intermediate protocol natively.

The AmbiStream CONNECTor can be also to solve interoperability for legacy streaming-enabled devices (Figure 5.23), which do not allow software (or firmware) extensions (e.g., televisions). This functionality was possible because the CONNECT architecture is flexible enough to allow any deployment place for CONNECTors. In this case, the CONNECTor is not deployed on the server or on the client, but on another smartphone which we call an "AmbiStream mediator support node". We have successfully tested this technique using Android smartphones as *server* and *mediator support node*, and an array of phones, running on different platforms, as legacy devices (including iPhone 3G, Nokia N8, Sony-Erricson W715, etc).

On-the-fly vs design-time mediator. Although on the fly mediation, as supported by CONNECT, is more flexible and enables adapting protocols that are unknown at compile-time, in our case we decided to use compile-time mediators, given the resource-restrictions of mobile devices. In addition, the availability of a specific protocol is only subject to the support of mobile platforms, thus making it possible to know in advance the adaptation requirements of each mobile device. Also, the adaptation only concerns the client-side since, on the server side we use an intermediate protocol. Given the newly introduced challenges with respect to live streaming, we did not

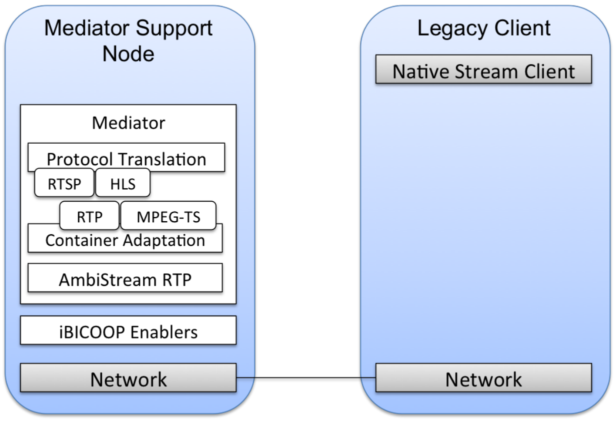


Figure 5.23: AmbiStream mediator support node

yet experiment with the CONNECT Synthesis Enabler.

Currently, the mediator is specified by a developer in the form of three DSL-based models, as illustrated in Figure 5.22:

- **Protocol Message Format DSL** (MDL Protocol B) which is based on the Starlink MDL, describes the format and structure of message fields of the streaming client protocol. This model is used to synthesize message parsers for incoming messages and composers for outgoing messages. The XML based DSL is defined in the form of an XML schema in Appendix Section 7.4.

- **Multimedia Container Format DSL** (MDL Container B), defined in Appendix Section 7.5., is used to specify the multimedia container format used by "Protocol B". It is different to the first language, as it is specifically designed to capture aspects of live multimedia streaming, by adding support for common operations such as message timing, fragmenting and multiplexing.

- **Merged Automaton DSL** (Merged Automaton AS-to-B), is used to specify the protocol mediation in the form of a k-coloured automaton. This language is also inspired by the Starlink Mediator Engine. Although currently used in relation to an intermediary protocol, the DSL is not restricted to this usage. The definition of the DSL is included in the Appendix Section 7.6.

The models are passed on to a compiler that produces multi-language (Java, C and C#) CONNECTors.

Protocol Message Formats. An example of a message description is shown in Figure 5.24. As a difference to Starlink MDL, the AmbiStream description is divided in *Input* and *Output* to better differentiate between incoming messages that should be parsed into structured data types and outgoing messages that are composed. This distinction is important with text protocols, where messages have loose requirements in terms of line order, optional parameters, delimiters, spacing characters and so on. The DSL proposed here supports protocols that use either binary, text or XML message formats. To assure a sufficiently expressive message description, we extract the required fields using value capture patterns defined using Posix[16] regular expressions for text protocols, XPath for XML and based on field size and location for binary protocols. The choice of Posix regular expression for text protocols was driven by its availability on most of the platforms, most notably that it is part of the GNU C library and is compatible with the regular expressions integrated in Java standard library (java.util.regex).

[16] Portable Operating System Interface, is a family of standards specified by the IEEE

```
<Protocol type="text">
  <Input>
    <Header name="http_head">
        <Var name="Url" type="String"/>
        <Rule test="capture_order(Url)">1</Rule>
        <Capture var="Method"> [RegEx] </Capture>
        <Finish test="empty_line"/>
    </Header>

    <Message name="GET_IDX">
        <Insert>http_head</Insert>
         ...
    </Message>
    ...
  </Input>
  <Output>
    <Message name="IDX">
       <Var name="$TargetDuration" type="Integer"/>
       <Line>#EXTM3U</Line>
        ...
    </Message>
    ...
  </Output>
</Protocol>
```

Figure 5.24: DSL-based specification of message formats for the HLS protocol

Multimedia Container Formats. The format in which audio/video content is wrapped differs depending on the protocol. As a difference to message formats, multimedia container formats present a much higher complexity in terms of: dependence between messages (order, timing, fragmenting), buffering requirements, and multiplexer logic required to interleave multiple media tracks inside single packets/messages.

As presented in the definition of the Multimedia Container Format DSL (Appendix Section 7.5), we divide the media container adaptation into four distinct phases: sample fragmenting, fragment packaging, multiplexing and final adjustment. Similarly to the synthesis of models for protocol message formats, the description of multimedia container formats is compiled to be deployed to designated platforms. To simplify the description, a number of media packet-related parameters are exposed through the DSL. Parameters include: the length of the media payload, media encoding, fragmenting flag, sampling frequency, sequence number, inner frame sequence number and first/last fragment flag. The components for protocol message description and container format adaptation are independent, thus allowing, for example, an application to choose between multiple supported data formats.

Because we intend to support real-time protocols (i.e., RTP), the problem of timing should also be taken into account. To do so, we add a time-stamp reference to each packet resulting from any of the four phases of multimedia container format format adaptation. Fragments of one frame share the same time-stamp information, while messages composing multiple frames contain the time-stamp of the first frame and their duration. The time required for a frame to pass through all of the phases required by the format, should not exceed the sampling interval of the content. Failing to assure this property can cause the client to run out of buffered data, resulting in playback stalls. In order to prevent such behaviour, frames should be explicitly dropped such that the output of the conversion is completed at the right time to assure a fluent playback.

The fragmenting step defines the way large audio or video samples are divided into smaller segments according to the limits imposed by the streaming protocol, by the media container or by the network configuration. For example, in the case of MPEG-TS, the samples are split into fragments which are inferior in size to 184 bytes, such that they can be correctly wrapped inside the standard 188 byte packets. For RTP fragmenting follows the standard RTP Payload Format depending on the codec used. We note that a simplified description of the packet format is very useful in the case of RTP, where there are multiple payload formats depending on the media encoder used. In the case where media content is composed of multiple tracks (i.e., one video and one audio track), two separate fragmenting units are used. The number of fragments created from single frames is variable. Each fragment contains a reference to the time-stamp of its originating frame. The time required for fragmenting one frame should never exceed the sampling interval of the content. In case this requirement can not be respected, the quality of the stream should be degraded by dropping frames.

The packaging stage adds individual packet headers. This transformation conforms to for RTP packets and for MPEG-TS. Depending on the protocol, the resulting packets are passed to the multiplexer or sent directly to the protocol translator.

The multiplexing phase assures time-division multiplexing for a set of given fragments or frames of multiple data tracks. Depending on the format, the multiplexing is done at a frame level or at a frame-fragment level. In order to achieve multiplexing at frame level, phase one of the adaptation should be skipped. This phase outputs only at a given time or data limit. Such a limit is necessary to be able to produce media fragments of specified duration or size. The split is always done at random access points of the stream, such that no reference between frames is lost.

The final transformation adds extra headers or packets, such that the resulting fragment is recognised as valid by standard client protocol implementations.

Many existing media container formats also contain a number of specific fields which are particularly hard to model. One example is the MPEG2 Transport Stream, which requires a 32-bit cyclic redundancy check value to be added to the Program Association Table package. In this a case we offer the possibility to implement such transformations outside the DSL. This feature can be used in any of the four steps defined earlier.

Streaming Protocol Translation and Multimedia Container format adaptation done by AmbiStream is further detailed in [2].

Towards Broader Interoperability. As mentioned in the beginning of this section, AmbiStream alone cannot solve interoperability in all cases. Furthermore, the native support of some platforms might be too restrictive for some application types. For example Apple iOS platform offers a native HTTP Live Streaming Client, but also requires any application using cellular data to use the mentioned protocol, among other restrictions. Since HLS is a high latency protocol targeted at scalability and low infrastructure costs rather than real-time communication, it is infeasible for applications requiring low latency (i.e., less than 200ms for Videoconferencing). On the other hand, in other applications, such as video broadcast, time shifting and so on, interoperability solutions between real-time and non real-time protocols is definitely possible.

A way to extend multi-application streaming media interoperability to various heterogeneous mobile platforms is to use third-party clients, integrated in the form of *applications*, *libraries* or *system intent*[17]. We have added support for this type of extensibility within AmbiStream. We tested the approach using FFmpeg [18] library on Android and obtained promising results for its use in a Videoconferencing application built on top of AmbiStream. The choice for the client in use at runtime can be made by the user or automatically, by AmbiStream, by analysing the constraints specified by the protocols implemented by each client application (or library).

5.5.4 iBICOOP middleware implementing CONNECT Enablers for Mobile Environments

iBICOOP is an Ambientic middleware technology for enabling the development of Collaborative Mobile Applications on heterogeneous platforms. As illustrated in Figure 5.25, iBICOOP provides the following functionalities:

- The **Discovery Enabler**, is a mobility-aware solution to enable transparent discovery of resources on mobile nodes, either on the same network, using a multi-cast mechanism

[17]In AndroidOS, an Intent is an abstract description of an operation to be performed. Intents specify actions that can be executed by the system or by third-party applications. It is basically a runtime binding between the code in different applications.

[18]FFmpeg is a cross-platform library for recording, converting and streaming audio and video. http://ffmpeg.org/

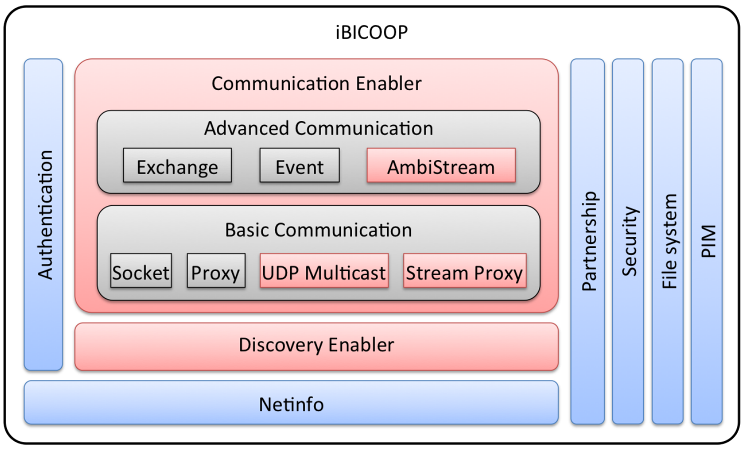


Figure 5.25: iBICOOP architecture

(similar to SLP and SSDP) or over the Internet using a centralized repository.

- The **Communication Enabler** enables seamless communication between heterogeneous devices independently of the underlying network infrastructure. It is organized in two layers. The *Basic Communication* provides an abstraction over the existing transport modes such as: TCP/IP Socket Communication, HTTP proxy-relayed Communication. A second communication layer called *Advanced Communication* provides higher level communication such as *File Exchange* and *Event dispatching*.
- A set of modules that provide the necessary base functionalities for building collaborative mobile applications on heterogeneous devices:
  - The Local File Manager supports synchronization, replication, and sharing among different users and devices.
  - The Security Manager is an interface to a set of cryptographic primitives and libraries intended to overcome the challenge of platform heterogeneity with respect to security.
  - The Partnership Manager provides context-aware sharing and trusted interactions.
  - Personal Information Management is also a source of heterogeneity, as each mobile platform treats the user's data differently. Data usually managed by native applications such as contacts and calendar can be seamlessly managed by iBICOOP applications by the means of the PIM interface.

Since mobile services are scattered over heterogeneous devices, running in different networks, a transparent way to locate resources accurately is required. Resources are identified using a URI scheme combining the names of protocol, device, service, email. Discovery can be done using a centralized internet repository, on the local network or using a local cache to improve performance.

The *Communication Enabler* provides an extensible architecture to support various physical network links. This communication abstraction allows services and applications running on mobile nodes to seamlessly communicate, without having to internally adapt to changes in network restrictions, reachability, available network interfaces and so on. The integration with AmbiStream extends the functionality with an new *Advanced Communication* component for interoperable live streaming. Its place in the iBICOOP architecture is illustrated in Figure 5.25. Given the architecture, AmbiStream mediator will be deployed on the same devices as the iBICOOP Enablers. Considering the example in Figure 5.26, NS1 represents the media server. NS2 can execute its deployment-time AmbiStream mediator to connect to NS1. At the same time NS3 can act as a third party mediator and execute an AmbiStream mediator instance compatible with the application supported on NS4.

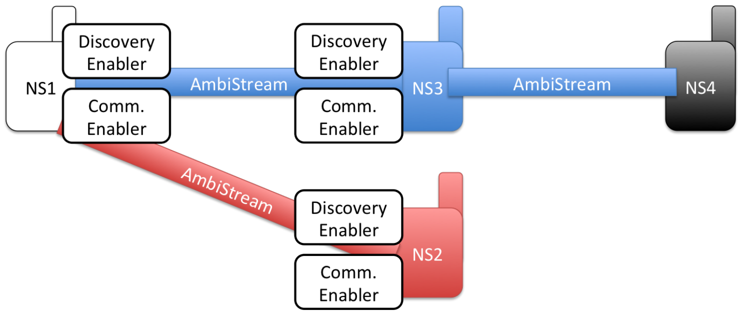


Figure 5.26: AmbiStream deployment using iBICOOP Enablers

If the iBICOOP communication abstraction over multiple radio-links and multiple transport protocols is used for multimedia transport, certain properties can not be met in all configurations. In such cases, the middleware should select the best radio interface and communication protocol depending on the application requirements. Taking the example of a Videoconference application; if a device is equipped with Wi-Fi and a 3G radio links, iBICOOP should use Wi-Fi as it offers a better bandwidth. If all Videoconference members are on the same network, then UDP Multicast should be used with priority. In the case where a node is unreachable, a relayed communication transport can also be used.

5.5.5 Integrating the AmbiStream CONNECTor with iBICOOP

iBICOOP was not originally designed to provide multimedia live streaming support. Depending on the application, streaming protocols have specific quality requirements in terms of latency, jitter, delay, buffering and message size. Traditionally such properties are obtained by requiring communication to be done out-of-band, using a specific transport protocol (i.e., RTP/UDP) and with restrictions regarding the sent messages (e.g., RTP messages sent over UDP should be smaller in size than the MTU, to prevent fragmenting, and thus reduce latency).

Multimedia Streaming using iBICOOP Enablers

In-band multimedia transport and encoding. One of the first necessary improvements to allow iBICOOP to support multimedia streaming on mobile phones was to reduce the overhead of message encoding, decoding and transmission. At the same time, we relied on the in-band data transport to benefit from the communication abstraction of iBICOOP. The core implementation of iBICOOP was extended to support a second, more efficient binary encoding of iBICOOP messages. The new encoding format follows the same logical structure of the other existing format which was XML based. Each packet contains fields representing: the *Source* and *Destination* using the iBICOOP resource URI pattern, the type of operation and a set of key-value pairs containing the actual payload data.

In Figure 5.27 we observe that packet loss was reduced significantly, now allowing streams of up-to 1500Kbps (on the same Android phone) with less than 2% loss. In contrast, the initial version could only transfer up to 500Kbps. At higher bit-rates, the test could not be realized, since the stream quality was too low for playback on the client device. Along with *Packet-Loss*, *Inter-arrival Jitter* was also observed (Figure 5.28) as it is an important metric for real-time multimedia streams. A high packet delay variation will cause packets to be discarded and thus reduce the playback quality. The improved implementation (as tested on the Android device) maintains a low packet delay variation at about 5ms for all tested video bit-rates.

While initially, the low performance was attributed to the Base64 encoding, necessary to send in-band data over XML encoding, our experiments showed that memory management has a much more important overhead. Memory allocation was improved to enable streaming media and cope with higher data rates and strict timing constraints. Memory management for live multimedia streaming has to be precisely crafted even on systems that rely on Garbage-

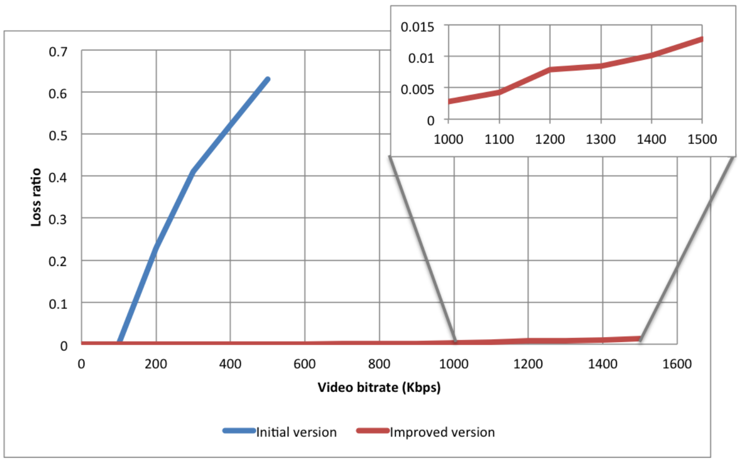


Figure 5.27: Packet loss for iBicoop in-band RTP Stream (Socket transport)

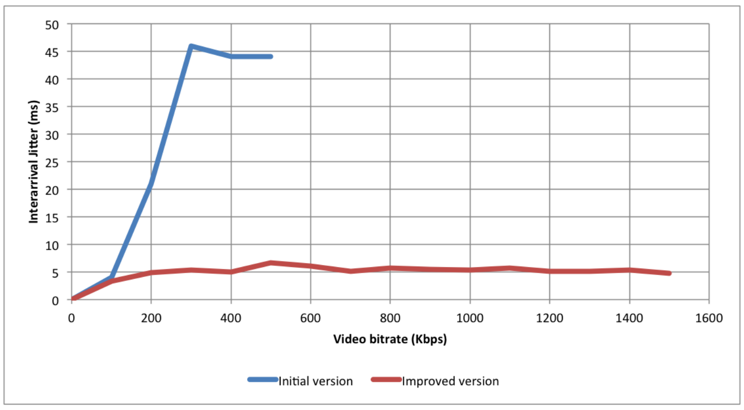


Figure 5.28: Inter-arrival jitter for iBicoop in-band RTP Stream (Socket transport)

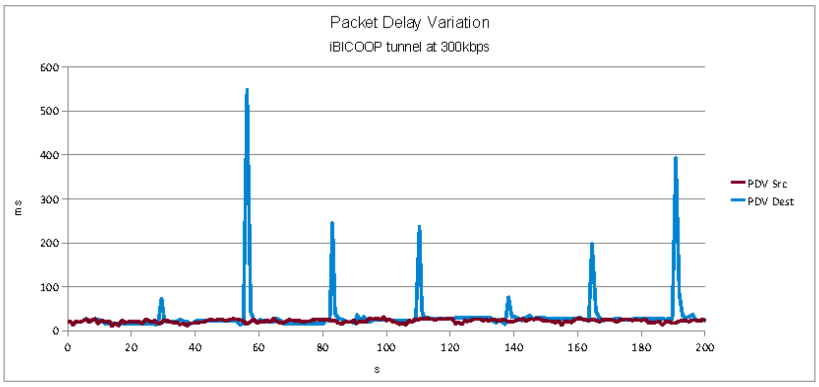


Figure 5.29: Interarrival jitter due to GC on Android

collection. This is because frequent interventions of the Garbage Collector to free up memory will introduce periodic delays and thus reduce the quality of the stream. Such a case is presented in Figure 5.29 for a 300Kbps RTP Stream with in-band transport over the initial version of iBICOOP on an Android mobile phone. Almost periodical spikes in packet delay (measured on the receiver side) can be observed, some reaching 500ms. Confronting this observation with the system logs we can confirm that the delays are caused by the intervention of the Garbage Collector. Given that the system we used in this test has a single-core CPU, the communication thread is periodically interrupted for long periods, thus decreasing quality.

Relayed streaming

iBICOOP uses proxy relays as a fall-back communication transport when two nodes are unreachable directly (usually caused by Network Address Translators) and Firewalls. An implementation of an Open HTTP proxy, that users can deploy (to avoid sending data through untrusted relays), is provided by iBICOOP. The existing proxy implementation relies on an *HTTP Polling* mechanism to receive messages. While HTTP transport is used by many applications, it is known to add a non negligible overhead in the exchange of messages. In contrast, this approach is likely to pass through all known Firewall configurations which allow outbound HTTP access.

Live streaming media for application types requiring low latency cannot follow the current *HTTP Polling pattern* because it introduces important delays even if techniques to reuse the TCP connection such as HTTP keep-alive are used. To address this problem, the proxy implementation was extended to support binary message encoding of iBICOOP messages and Chunked Transfer Encoding [17]. Chunked Transfer Encoding offers the following benefits: persistent connection, manages dynamically generated content (such as video streams) and can use on the fly compression (gzip). Our experiments done over Wi-Fi and 3G with the improved iBICOOP Proxy server deployed on a different network, showed satisfactory performance (and most importantly low latency).

Relayed group communication

We further extended the relay proxy to support group communication. By doing so, the task of handling multiple connections and replicating messages on the fly is delegated to a more resource-rich system. To integrate this new transport mode with iBICOOP, the *Stream Proxy* plug-in (Figure 5.25) interface was implemented at the *Basic Communication* layer.

To better present the achievable communication scenarios, we consider the example in Figure 5.30. *Source 1* wants to distribute a live video stream to a number of clients. Since *Source 1* is located under a Network Address Translator, it cannot actively connect provide such a service to clients outside its network. Using *AmbiStream Service* in the *Advanced Communication* layer of iBICOOP, it transparently registers to a desired proxy relay on the Internet. Even though multiple

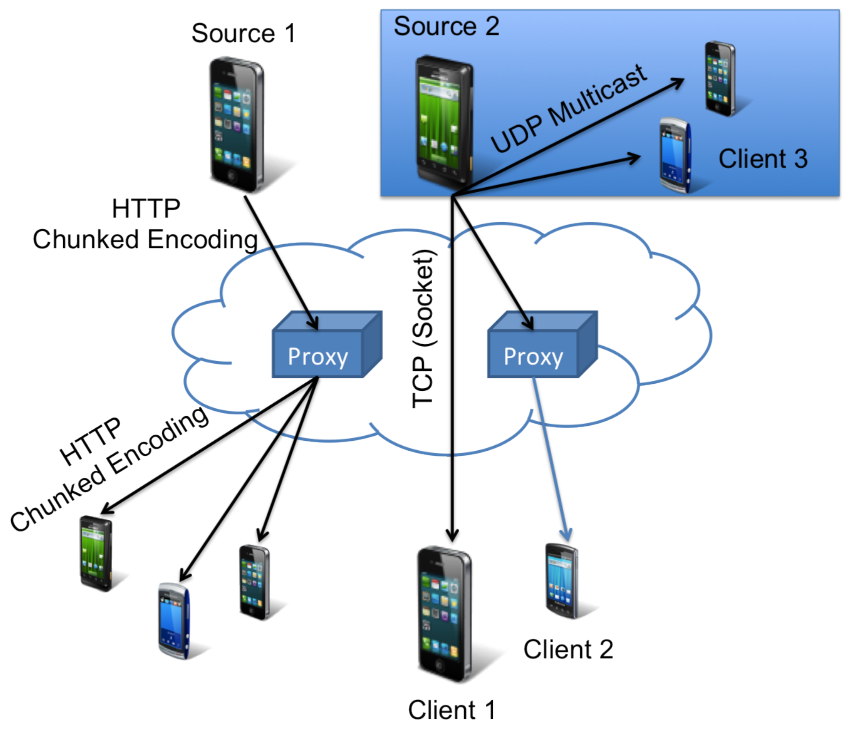


Figure 5.30: iBICOOP live streaming scenario

Device	Samsung GT-I9000	Google Nexus One	iPhone 3G
Role	Server	Client	Client
Platform	Android 2.2.1	Android 2.3.4	iOS 4.2.1
CPU	1 GHz (S5PC110)	1 GHz (QSD8250)	412 MHz
Memory	512 MB	512 MB	128 MB
Media framework	PV OpenCORE	Stagefright	AV Foundation
Stream support	RTSP	RTSP/HLS	HLS

Table 5.5: Test smartphones used

clients connect to watch the live feed, *Source 1* only maintains one connection with the iBICOOP proxy. Proxy connections are authenticated at both the source and destination ends. *Source 1* can decide at any time to add or remove a client from the communication group by requesting this to the Proxy.

However, a streaming source is not constrained to using only one transport at a time. For example, *Source 2* is able to communicate with *Client 2* over a relay, with *Client 1* using TCP Socket transport over the Internet and with *Client 3* on the Local Network using UDP Multicast transport. At the application level, each destinations is identified by an unique iBICOOP URI, thus reducing application complexity.

5.5.6 Experiments

As a means to evaluate the presented solution, we have implemented AmbiStream in Java and Objective-C and tested it on AndroidOS and iOS. The goal of the experiments presented here is to evaluate the overall performance of AmbiStream and to assess the achievable stream quality. The experiments were performed on both Android and iPhone smartphones.

In both of the experiments presented below, the same set of source media files was used. The test files have a duration of 210 seconds, are encoded with a single (H.264-avc video) track, have a CIF frame-size (352 by 288), and a frame-rate of 30 fps. The test is conducted for 16 different bit-rates between 50kbps and 1500kbps using the mentioned file format and content. Each set of tests is repeated at least three times, so each of the metrics presented

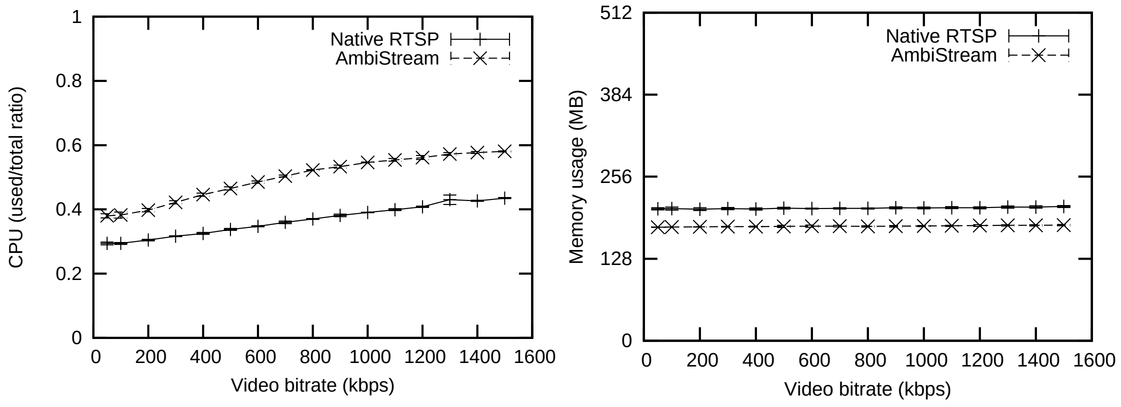


Figure 5.31: AmbiStream performance on Nexus One (RTSP)

is characterized by 168 minutes of video streaming to each client device. In total, more than 16 hours of streaming between smartphones were necessary. The mobile phones used are mentioned in Table 5.5. The first two (Samsung GT-I9000 and Google Nexus One) are used in the first experiment, and all three in the second one.

Collecting mobile device performance data

Although RTSP provides out-of-band feedback of stream quality through RTCP, we have decided not to use this feature to obtain information related to the quality of service. This is due to the fact that in the case of the media framework Pocket Video OpenCore (used by Android platform in versions preceding 2.3) the information provided is not sufficiently precise. For example, the interval jitter value reported, used to observe the effect of network packet delays, is usually ten times higher than what we found at network level or on the client device. Furthermore, on the newer Stagefright media framework the feedback always reports no packet loss and inter-arrival jitter equal to zero. AndroidOS also provides an information callback from the media player service. Unfortunately, this information is limited to a small set of event codes and do not include any metric.

We have chosen to favour system-wide metrics to more specific ones (i.e., metrics of the application process) because we also make use of native system services and because mobile platforms do not frequently provide equivalent metrics. We use as metrics for device performance: the total CPU utilization and the system-wide used RAM memory. Quality of service metrics considered are the packet delay variation (also referred to as inter-arrival jitter, described in [28]) and packet loss ratio. The quality metrics are only provided for the case where the protocol is adapted. The values are obtained should indicate the maximum bit-rate achievable while still providing satisfactory quality. The reference test cases, used to compare the overall performance, make use of system media services directly.

On Android mobile phones, the CPU and memory information is obtained by accessing the *proc* filesystem, used as an interface to the operating system kernel on most Linux based distributions. The logs are stored in the internal memory of both Android phones. To avoid that the access to the filesystem and data parsing are influencing the final results, the access to the */proc/stat* and */proc/meminfo* is done every five seconds, and the same file-descriptors are reused multiple times until the end of the test. On the iOS platform, system performance information was collected using the tools integrated with the development kit.

Translating to RTSP between AndroidOS smartphones

In this first experiment, we show that adaptation from AmbiStream intermediary protocol to RTSP/RTP/UDP is sufficiently efficient to be used in mobile multimedia-enabled applications. Since in this case the message format used by the client protocol is equivalent to the intermediary

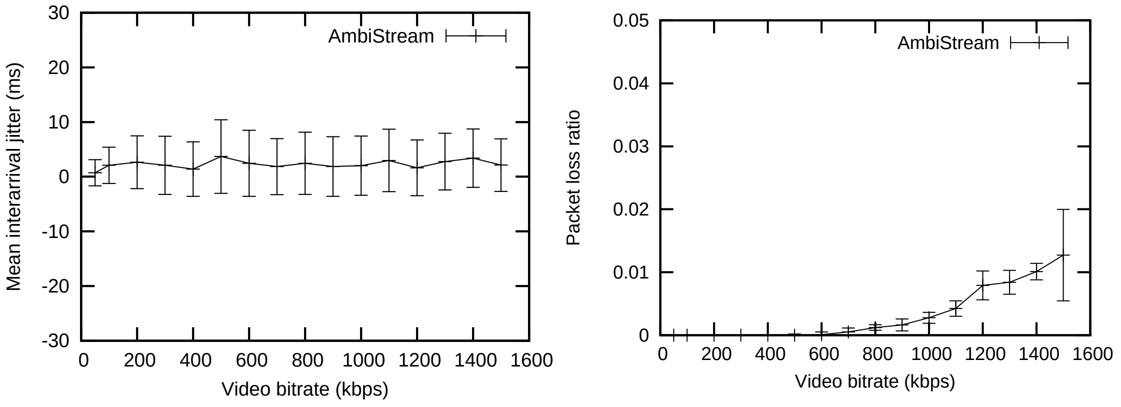


Figure 5.32: Adapted stream quality (RTSP)

transport protocol, the wrapping and unwrapping of messages is simpler than in other cases. Nevertheless, this client protocol is the only real-time streaming protocol currently available on mobile phones, and is thus interesting to analyse the feasibility of streaming real-time multimedia data. Another experiment involving a more complex media format adaptation is presented at the end of this section. In this test we use two server implementations: one using the AmbiStream intermediary protocol and the other using RTSP. The RTSP server is not part of the solution but it is used in this experiment to determine the overhead of the adaptation (on the client-side) with reference to the native RTSP support.

In the case of protocol translation to RTSP, the performance of the client device (realising the content and protocol adaptation) is not badly affected, with a processing overhead of less than 20% compared to a native RTSP session (see Figure 5.31). As with all of the experiments conducted, the memory usage remains constant, or increases slightly because of buffers required for higher data-rates (Figure 5.31). The fact that our solution uses slightly less memory than the reference one is due to the way jitter buffers are managed internally by the RTSP client, most probably being influenced by the different transport protocols (UDP and TCP). The quality of the stream remains within acceptable limits in terms of inter-arrival jitter (see Figure 5.32) and packet loss, for all the test cases (from 50 to 1500kbps) considered.

Translating to HLS between AndroidOS and iOS smartphones

The second experiment consists of translating AmbiStream intermediary protocol to HTTP Live Streaming, using two different client platforms: AndroidOS 2.3.4 and iOS 4.2.1. The choice of the smartphones is motivated by their native support of HLS. This way we can reason about the overhead introduced by our mediator with two different devices. Contrary to the first experiment, this one requires data conversion between RTP and MPEG-TS. MPEG-TS is one of the most used multimedia formats, most notably for digital television. The conversion from RTP to MPEG-TS requires a large number of transformations, thus providing a good impression of achievable on-the-fly conversion limits of media formats on current generation smartphones.

Because HLS protocol requires the existence of a cached amount of content on the server-side before a client can connect (and begin playback), while the intermediary AmbiStream protocol does not, a 30s start-up delay is introduced by the mediator to allow protocol translation. During this period, less memory and CPU are used. To better evaluate the performance of the devices, we divide the experiment run in four periods (e.g., as shown in Figure 5.33 for CPU utilisation): *(I) the buffering period* (only multimedia data adaptation is performed), *(II) the media-player start-up* (causes a short increase in CPU usage), *(III) the streaming period* (both data adaptation and playback are performed) and *(IV) the stream-end* (the source has finished streaming, but the playback is continued until buffer depletion). Thus, only the part *(III)* of the observation was used to produce the results presented in Figures 5.35.

As expected, the difference in container formats (RTP and MPEG-TS), increases the over-

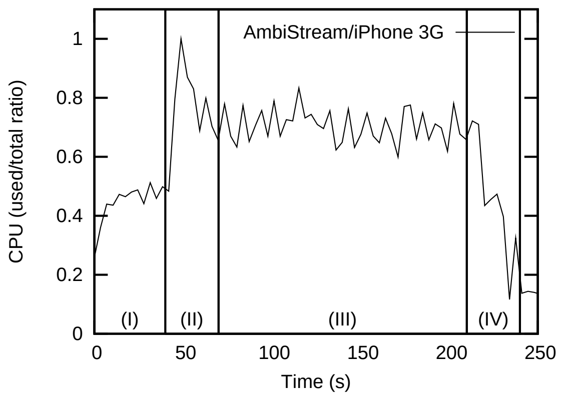


Figure 5.33: Data capture (HLS)

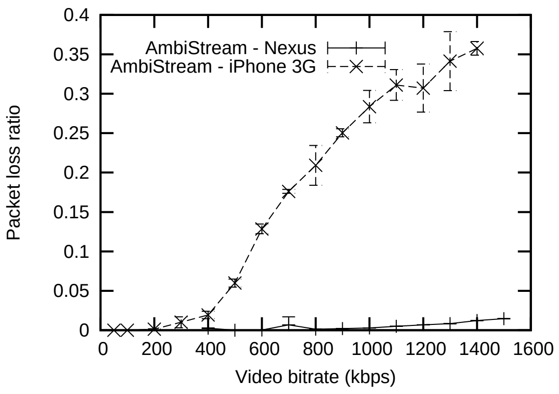
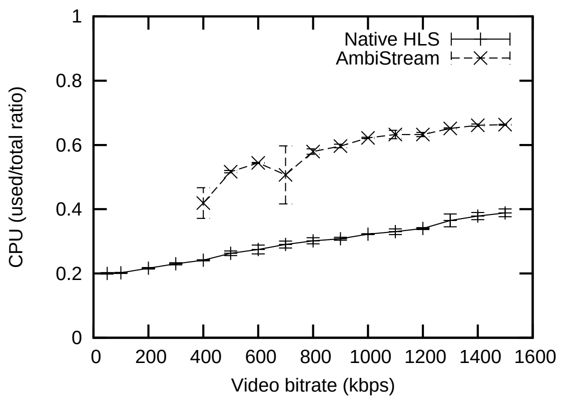


Figure 5.34: Packet loss (HLS)



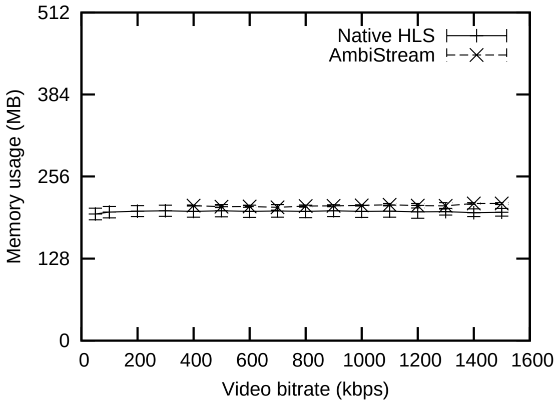


Figure 5.35: AmbiStream performance on Nexus One (HLS)

head of AmbiStream. For Android platform, the tests for bit-rates inferior to 400kbps (in Figures 5.35) were discarded due to the existence of a minimal caching size, requiring a longer start-up delay. While on the Nexus One, the overhead introduced does not reach a quality limit for bit-rates below 1500kbps, the iPhone 3G is only able to adapt streams of up to 400kbps. Above this limit, the packet loss (see Figure 5.34) becomes noticeable and the media-player suffers playback stalls. The results on the iPhone are worse due to the significantly lower processing power and memory (see Table 5.5). Nevertheless, according to the mobile platform providers, a 400kbps video bit-rate is considered to be medium/high quality for smartphones[19] [20]. Considering the results in Figure 5.35, we see that the memory usage is decreasing (in the case of AmbiStream) for higher video bit-rates. This behaviour is normal considering the packet loss (see Figure 5.34).

5.5.7 Conclusion

In this section we addressed the challenges of enabling Live Multimedia Streaming on heterogeneous mobile devices with reference to the CONNECT architecture. Further, we presented AmbiStream, a compile-time, multi-platform CONNECTor that can be deployed in fully-distributed mobile environments. We motivated our design choices with respect to the CONNECTor architecture using a concrete live streaming interoperability example. Then, we introduced iBICOOP middleware technology, a partial and lightweight CONNECT Enabler implementation intended to simplify the development of Collaborative Mobile Applications on heterogeneous devices. iBICOOP was then improved and integrated with AmbiStream CONNECTor in order to fulfil the goal of interoperable live streaming on current generation mobile platforms. Finally, we validated the applicability of the AmbiStream CONNECTor solution by a series of experiments on two mobile platforms.

The AmbiStream live streaming interoperability solution was designed to function in fully-distributed environments. This characteristic is vital for the deployment of networked systems in remote places and a priori unknown environments. A good example of such a circumstance is presented in Deliverable 6.3 as The Joint Forest-Fire Operation, where an number of systems of key importance (such as IP cameras and UAVs) are enabled with multimedia streaming capabilities.

[19] http://developer.apple.com/library/ios/# technotes/tn2224/_ index.html
[20] http://developer.android.com/guide/appendix/media-formats.html

6 Conclusions

6.1 Concluding Remarks

The overall aim of the CONNECT project is to address the interoperability challenges that result from the use of different data and protocols by the different entities involved in the software stack such as applications, middleware, platforms, etc. This is to be performed on-the-fly in order to succeed in highly heterogeneous, dynamic environments where systems must interact spontaneously i.e. they only discover each other at runtime.

This deliverable has focused on the revision of the CONNECT architecture with the goal of becoming concrete, and integrating the work of the individual Work Packages. In this report we first presented the previous intermediate CONNECT architecture (as reported in D1.2 [4]) and subsequently discussed the key refinements that have been made to the architecture in the third year of the project; these include:

- The enhancement of the architecture to drive the CONNECT process according to goals (intent), in particular requiring changes to the Discovery and Synthesis enablers. While at present we consider a goal to be local to a pair of interacting networked systems, assembly nevertheless provides means to create larger compositions of services. This further enhances the opportunities for adaptation in response to feedback from runtime.
- The management of non-functional CONNECTor properties and requirements through the improved integration of the DePer, Security and Trust Enablers. We have outlined a control loop for performing dynamic adaptations of CONNECTors through resynthesis and redeployment. In the final year of the project we aim to fully realise the implementation and integration of these adaptations.
- The extension of the Discovery enabler to learn the affordance of a system; this better informs the matching of potential networked systems to interoperate.
- The exploration of CONNECT solutions for streaming-based systems. The solutions to connect heterogeneous streaming protocols highlight the role CONNECT can achieve beyond simple message-based platforms.
- The specification of concrete CONNECTors using k-coloured automata supporting the realisation and deployment of interoperable software capable of addressing both application and middleware heterogeneity.

The other important contribution of this report is to highlight the integration work carried out during the the third year of the project which led to the revised CONNECT architecture as presented in Section 3. Here it can be seen that the connection-time process from discovery through to deployment is well integrated, while the integration of runtime feedback from dependability, security and trust enablers is progressing. A number of prototype software solutions have been created to complement the realisation of the CONNECT architecture. Those highlighted in this deliverable and described in the accompanying appendix (Deliverable D1.3 Appendix - Prototype) are:

- *Starlink* is a complete software framework for realising concrete interoperability CONNECTors. It offers support to interpret both application and middleware behaviour which is described in k-coloured automaton. It also supports the generation of network packet parsing and composing behaviour based upon message description specifications.
- *iBICOOP* is a middleware technology for enabling collaborative applications on heterogeneous mobile platforms, and is a partial implementation of the CONNECT architecture for fully-distributed mobile environments. iBICOOP has been extended to support AmbiStream for the mediation of multimedia streams. AmbiStream leverages iBICOOP communication and discovery enablers to provide interoperable streaming across heterogeneous mobile devices and networks.

- *The* CONNECT *JMS wrapper* provides a solution to wrap the Java interfaces of POJOs such that a distributed request-response message exchange can be achieved using JMS. Hence, enablers implemented as POJOs can be integrated into the JMS-based CONNECT architecture.
- *Enablers*. The prototypes for the Discovery, Learning, Interaction and Deployment enablers are described in the D1.3 Appendix - Prototype).

These prototypes have been or are being evaluated using small application or middleware case-studies; these show that the prototypes successfully achieve the objectives of the CONNECT project (namely long-lived interoperability) and conform to the specification of the CONNECT architecture. The prototypes also form the foundation of the implementation and evaluation of the case-study as described in Deliverable D6.3 [19].

6.2 Future Activities for WP1

The focus in our ongoing work is the full integration of the runtime feedback loop between monitoring (of security, dependability and trust) and the various steps of synthesis and deployment phase. Such feedback should enable the CONNECTor, or potentially the wider group of interacting systems, to react and adapt to the actual behaviour and properties observed at runtime. Key questions include what information it is useful to monitor and how such information can be used to inform the synthesis process. Moreover, it will be necessary to investigate the non-trivial relationship between the non-functional properties considered by the Connectability enablers and the functional properties (and goals) critical for synthesis.

Another important related issue is that of groups (configurations) of networked systems assembling to achieve an overall goal. We have outlined a structural assembly process that can derive configurations in which one synthesised connector effectively achieves the goal. However, high-level goals may require the co-ordinated interaction of several systems, and continuing to satisfy the goals at runtime may require a structural change as opposed to a behavioural change in a connector.

Evaluation is an important area of future work for the final stage of the project. Essentially this will involve the further application of the CONNECT architecture to the common GMES case study. In particular, this will focus on the development of a complete working demonstrator that can then be used to assess the benefits and impact that the CONNECT architecture provides against both the interoperability challenges faced and the alternative solutions that are available to address interoperability.

7 D1.3 Appendix

7.1 XML RPC PhotoSharing Networked System

```
<?xml version="1.0"?>
<protocol xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ApplicationAutomata.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <protocolName>XMLRPC_PhotoSharing</protocolName>

    <attributes>
        <ProtocolBinding>XMLRPCServerBinding.xml</ProtocolBinding>
        <ServerHost>127.0.0.1:8081</ServerHost>
    </attributes>

    <automaton>
        <start>S1</start>
        <end>S9</end>
        <state>
            <label>S1</label>
            <transition>
                <action>Photo.photoMetaData</action>
                <operation>send</operation>
                <Outputs>
                    <Output>
                        <simpleType>
                            <name>PhotoMetaData</name>
                            <type>String</type>
                        </simpleType>
                    </Output>
                </Outputs>
                <toState>S2</toState>
            </transition>
        </state>

        <state>
          <label>S2</label>
          <transition>
            <action>Photo.photoMetaData</action>
            <operation>recv</operation>
            <toState>S3</toState>
            <Inputs>
                <Input>
                    <complexType name="PhotoMetaDataList">
                        <name>PhotoMetaDataList</name>
                        <type>stringarray</type>
                         <xsd:sequence>
                                <xsd:element name="item" type="String"
                                maxOccurs="unbounded" minOccurs="0" />
                        </xsd:sequence>
                    </complexType>
                </Input>
            </Inputs>
          </transition>
        </state>

        <state>
          <label>S3</label>
          <transition>
                <action>SEARCHPHOTOS</action>
                <operation>noaction</operation>
                <toState>S4</toState>
            </transition>
        </state>

        <state>
```

```
            <label>S4</label>
            <transition>
                <action>Photo.PHOTOCOMMENT</action>
                <operation>send</operation>
                <Outputs>
                    <Output>
                        <simpleType>
                            <name>photoid</name>
                            <type>string</type>
                        </simpleType>
                    </Output>
                </Outputs>
                <toState>S5</toState>
            </transition>
        </state>

        <state>
          <label>S5</label>
          <transition>
            <action>Photo.PHOTOCOMMENT</action>
            <operation>recv</operation>
            <toState>S6</toState>
            <Inputs>
                <Input>
                    <complexType name="CommentList">
                        <name>CommentList</name>
                        <type>stringarray</type>
                         <xsd:sequence>
                                <xsd:element name="item" type="String"
                                maxOccurs="unbounded" minOccurs="0" />
                        </xsd:sequence>
                    </complexType>
                </Input>
            </Inputs>
          </transition>
        </state>

        <state>
          <label>S6</label>
          <transition>
                <action>SEARCHPHOTOS</action>
                <operation>noaction</operation>
                <toState>S7</toState>
            </transition>
        </state>

        <state>
            <label>S7</label>
            <transition>
                <action>Photo.AddCOMMENT</action>
                <operation>send</operation>
                <Outputs>
                    <Output>
                        <simpleType>
                            <name>photoid</name>
                            <type>string</type>
                        </simpleType>
                    </Output>
                    <Output>
                        <simpleType>
                            <name>comment</name>
                            <type>string</type>
                        </simpleType>
                    </Output>
                </Outputs>
                <toState>S8</toState>
            </transition>
        </state>

        <state>
          <label>S8</label>
          <transition>
                <action>Photo.AddCOMMENT</action>
                <operation>recv</operation>
                <Inputs>
                    <Input>
                        <simpleType>
                            <name>result</name>
                            <type>integer</type>
                        </simpleType>
                    </Input>
                </Inputs>
                <toState>S9</toState>
            </transition>
        </state>
        <state>
          <label>S9</label>
        </state>
   </automaton>
</protocol>
```

7.2 CORBA Photosharing Networked System

```
<protocol xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="ApplicationAutomata.xsd"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <protocolName>IIOPPhotoSharing</protocolName>

    <attributes>
        <ProtocolBinding>IIOPClientBinding.xml</ProtocolBinding>
    </attributes>

    <automaton>
        <start>B1</start>
        <end>B6</end>
        <state>
            <label>B1</label>
            <transition>
                <action>SEARCHPHOTOS</action>
                <operation>recv</operation>
                <Inputs>
                    <Input>
                        <simpleType>
                            <name>PhotoMetaData</name>
                            <type>java.lang.string</type>
                        </simpleType>
                    </Input>
                </Inputs>
                <toState>B2</toState>
            </transition>
        </state>

        <state>
            <label>B2</label>
            <transition>
                <action>SEARCHPHOTOS</action>
                <operation>send</operation>
                <Outputs>
                    <Output>
                        <complexType name="PhotoMetaDataList">
                            <name>PhotoMetaDataList</name>
                            <type>stringarray</type>
                             <xsd:sequence>
                                    <xsd:element name="item" type="java.lang.string"
                                    maxOccurs="unbounded" minOccurs="0" />
```

```
                              </xsd:sequence>
                          </complexType>
                      </Output>
                  </Outputs>
                  <toState>B3</toState>
              </transition>
          </state>

          <state>
              <label>B3</label>
              <transition>
                  <action>DOWNLOADCOMMENT</action>
                  <operation>recv</operation>
                  <Inputs>
                      <Input>
                          <simpleType>
                              <name>PhotoID</name>
                              <type>java.lang.string</type>
                          </simpleType>
                      </Input>
                  </Inputs>
                  <toState>B4</toState>
              </transition>
          </state>

          <state>
              <label>B4</label>
              <transition>
                  <action>DOWNLOADCOMMENT</action>
                  <operation>send</operation>
                  <Outputs>
                      <Output>
                          <complexType name="CommentList">
                              <name>CommentList</name>
                              <type>stringarray</type>
                               <xsd:sequence>
                                      <xsd:element name="item" type="java.lang.string"
                                      maxOccurs="unbounded" minOccurs="0" />
                              </xsd:sequence>
                          </complexType>
                      </Output>
                  </Outputs>
                  <toState>B5</toState>
              </transition>
          </state>

          <state>
              <label>B5</label>
              <transition>
                  <action>COMMENTPHOTO</action>
                  <operation>recv</operation>
                  <Inputs>
                      <Input>
                          <simpleType>
                              <name>photoid</name>
                              <type>java.lang.string</type>
                          </simpleType>
                      </Input>
                      <Input>
                          <simpleType>
                              <name>comment</name>
                              <type>java.lang.string</type>
                          </simpleType>
                      </Input>
                  </Inputs>
                  <toState>B6</toState>
              </transition>
          </state>

```

```
        <state>
            <label>B6</label>
            <transition>
                <action>COMMENTPHOTO</action>
                <operation>send</operation>
                <toState>B1</toState>
            </transition>
        </state>

        <state>
          <label>B7</label>
        </state>

  </automaton>
</protocol>
```

7.3 Photosharing Mediator

```
<?xml version="1.0"?>
<merge xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="
     Automata.xsd">
  <label>MEDIATOR</label>
  <start>
    <node>
      <protocol>IIOPPhotoSharing</protocol>
      <label>B1</label>
    </node>
  </start>

   <split>
    <node>
      <protocol>IIOPPhotoSharing</protocol>
      <label>B2</label>
    </node>
  </split>

  <split>
    <node>
      <protocol>IIOPPhotoSharing</protocol>
      <label>B4</label>
    </node>
  </split>

  <bridge>
    <from>
      <node>
        <protocol>IIOPPhotoSharing</protocol>
        <label>B2</label>
        <branch>1</branch>
      </node>
    </from>
    <to>
      <node>
        <protocol>XMLRPC_PhotoSharing</protocol>
        <label>S1</label>
      </node>
    </to>
    <translationlogic>
      <assignment>
          <field>
              <state>B1</state>
              <message>SEARCHPHOTOS</message>
              <label>PhotoMetaData</label>
        </field>
        <field>
            <state>S1</state>
```

```
                <message>photoMetaData</message>
                <label>PhotoMetaData</label>
            </field>
        </assignment>
    </translationlogic>
</bridge>
<bridge>
    <from>
        <node>
            <protocol>XMLRPC_PhotoSharing</protocol>
            <label>S3</label>
        </node>
    </from>
    <to>
        <node>
            <protocol>IIOPPhotoSharing</protocol>
            <label>B2</label>
            <branch>2</branch>
        </node>
    </to>
    <translationlogic>
        <assignment>
            <field>
                <state>S2</state>
                <message>photoMetaData</message>
                <label>PhotoMetaDataList</label>
            </field>
            <field>
                <state>B22</state>
                <message>SEARCHPHOTOS</message>
                <label>PhotoMetaDataList</label>
            </field>
        </assignment>
    </translationlogic>
</bridge>

<bridge>
    <from>
        <node>
            <protocol>IIOPPhotoSharing</protocol>
            <label>B4</label>
            <branch>1</branch>
        </node>
    </from>
    <to>
        <node>
            <protocol>XMLRPC_PhotoSharing</protocol>
            <label>S4</label>
        </node>
    </to>
    <translationlogic>
        <assignment>
                <field>
                    <state>B3</state>
                    <message>DOWNLOADCOMMENT</message>
                    <label>PhotoID</label>
            </field>
            <field>
                <state>S4</state>
                <message>PHOTOCOMMENT</message>
                <label>photoid</label>
            </field>
        </assignment>
    </translationlogic>
</bridge>

<bridge>
    <from>
        <node>
```

```
          <protocol>XMLRPC_PhotoSharing</protocol>
          <label>S6</label>
        </node>
      </from>
      <to>
        <node>
          <protocol>IIOPPhotoSharing </protocol>
          <label>B4</label>
          <branch>2</branch>
        </node>
      </to>
      <translationlogic>
        <assignment>
          <field>
              <state>S5</state>
              <message>PHOTOCOMMENT</message>
              <label>CommentList</label>
          </field>
          <field>
              <state>B42</state>
              <message>DOWNLOADCOMMENT</message>
              <label>CommentList</label>
          </field>
        </assignment>
      </translationlogic>
    </bridge>

    <bridge>
      <from>
        <node>
          <protocol>IIOPPhotoSharing </protocol>
          <label>B6</label>
        </node>
      </from>
      <to>
        <node>
          <protocol>XMLRPC_PhotoSharing</protocol>
          <label>S7</label>
        </node>
      </to>
      <translationlogic>
        <assignment>
            <field>
                <state>B5</state>
                <message>COMMENTPHOTO</message>
                <label>comment</label>
          </field>
          <field>
              <state>S7</state>
              <message>AddCOMMENT</message>
              <label>comment</label>
          </field>
        </assignment>
        <assignment>
            <field>
                <state>B5</state>
                <message>COMMENTPHOTO</message>
                <label>photoid </label>
          </field>
          <field>
              <state>S7</state>
              <message>AddCOMMENT</message>
              <label>photoid </label>
          </field>
        </assignment>
      </translationlogic>
    </bridge>
</merge>
```

7.4 XSD definition of the AmbiStream message format DSL for text-based protocols

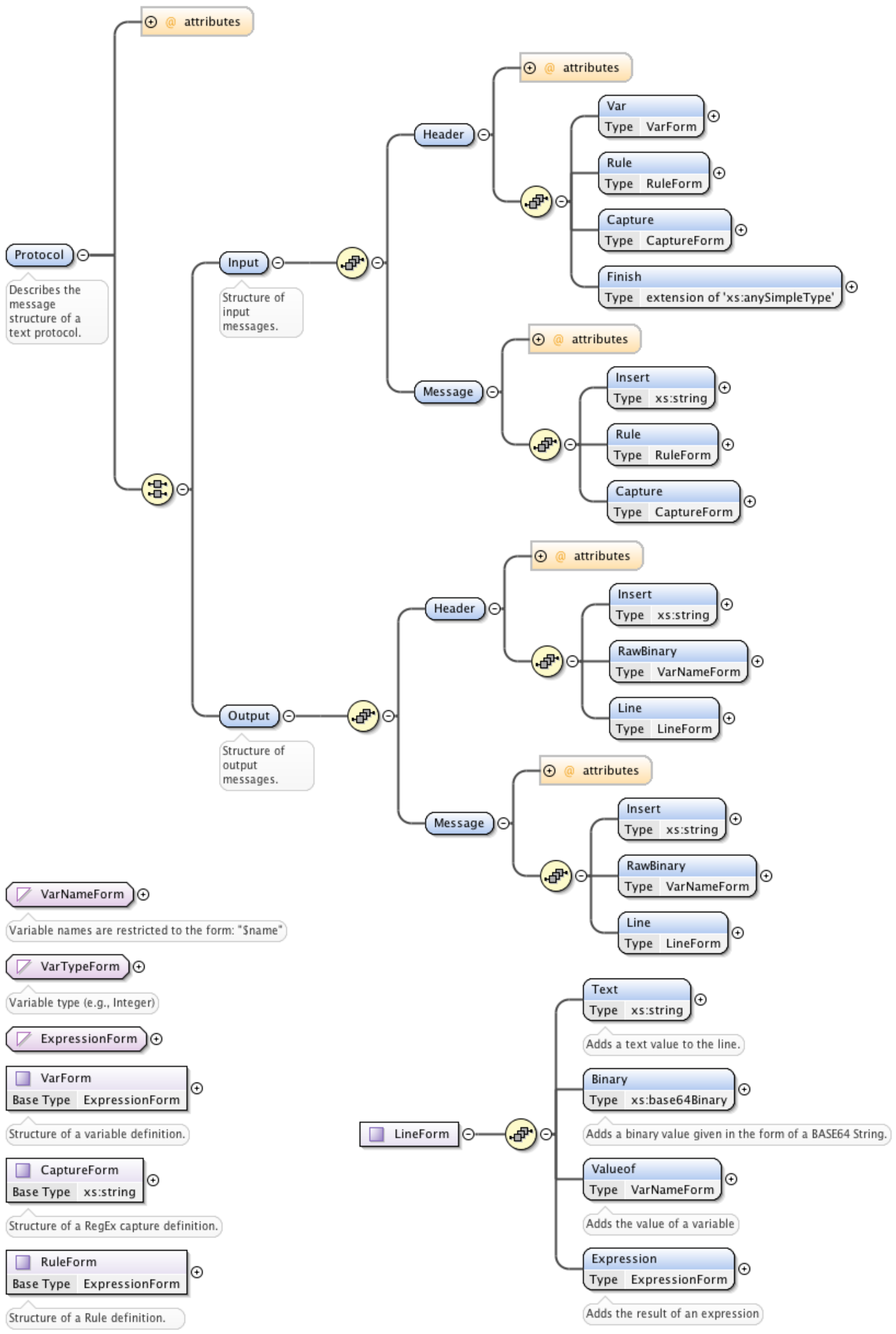


Figure 7.1: XSD definition of the AmbiStream message format DSL for text-based protocols

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.ambientic.com/AmbiStream"
    targetNamespace="http://www.ambientic.com/AmbiStream" elementFormDefault="qualified">
    <xs:simpleType name="VarNameForm">
        <xs:annotation>
            <xs:documentation>Variable names are restricted to the form: "$name"</xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="$[0-9a-zA-Z]+"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="VarTypeForm">
        <xs:annotation>
            <xs:documentation>Variable type (e.g., Integer)</xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:enumeration value="String"/>
            <xs:enumeration value="Integer"/>
            <xs:enumeration value="Bytes"/>
            <xs:enumeration value="Base64String"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="ExpressionForm">
        <xs:annotation>
            <xs:documentation>e.g., $var1 + $var2</xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string"/>
    </xs:simpleType>
    <xs:complexType name="VarForm">
        <xs:annotation>
            <xs:documentation>Structure of a variable definition.</xs:documentation>
        </xs:annotation>
        <xs:simpleContent>
            <xs:extension base="ExpressionForm">
                <xs:attribute name="type" default="String" use="optional" type="VarTypeForm">
                    <xs:annotation>
                        <xs:documentation>The type of variable (e.g., String, Integer)</xs:documentation>
                    </xs:annotation>
                </xs:attribute>
                <xs:attribute name="name" use="required" type="VarNameForm">
                    <xs:annotation>
                        <xs:documentation/>
                    </xs:annotation>
                </xs:attribute>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:complexType name="CaptureForm">
        <xs:annotation>
            <xs:documentation>Structure of a RegEx capture definition.</xs:documentation>
        </xs:annotation>
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="var" type="VarNameForm" use="required">
                    <xs:annotation>
                        <xs:documentation>Variable to store the RegEx captured value.</xs:documentation>
                    </xs:annotation>
                </xs:attribute>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:complexType name="RuleForm">
        <xs:annotation>
            <xs:documentation>Structure of a Rule definition.</xs:documentation>
        </xs:annotation>
        <xs:simpleContent>
            <xs:extension base="ExpressionForm">
                <xs:attribute name="test" use="required">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="msg_line_delimiter"/>
                            <xs:enumeration value="msg_max_lines"/>
                            <xs:enumeration value="msg_min_lines"/>
                            <xs:enumeration value="var_capture_order"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
                <xs:anyAttribute>
                    <xs:annotation>
                        <xs:documentation>A rule can have a variable number of arguments.</xs:
                            documentation>
                    </xs:annotation>
                </xs:anyAttribute>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:complexType name="LineForm">
        <xs:sequence>
            <xs:element name="Text" type="xs:string">
                <xs:annotation>
                    <xs:documentation>Adds a text value to the line.</xs:documentation>
                </xs:annotation>
            </xs:element>
            <xs:element name="Binary" type="xs:base64Binary">
                <xs:annotation>
                    <xs:documentation>Adds a binary value given in the form of a BASE64 String
                        .</xs:documentation>
                </xs:annotation>
            </xs:element>
            <xs:element name="Valueof" type="VarNameForm">
                <xs:annotation>
                    <xs:documentation>Adds the value of a variable</xs:documentation>
                </xs:annotation>
            </xs:element>
            <xs:element name="Expression" type="ExpressionForm">
                <xs:annotation>
                    <xs:documentation>Adds the result of an expression</xs:documentation>
                </xs:annotation>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="Protocol">
        <xs:annotation>
            <xs:documentation>Describes the message structure of a text protocol.</xs:
                documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:all maxOccurs="1" minOccurs="1">
                <xs:element name="Input">
                    <xs:annotation>
                        <xs:documentation>Structure of input messages.</xs:documentation>
                    </xs:annotation>
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="Header">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="Var" type="VarForm"/>
                                        <xs:element name="Rule" type="RuleForm"/>
                                        <xs:element name="Capture" type="CaptureForm"/>
                                        <xs:element name="Finish">
                                            <xs:complexType>
                                                <xs:simpleContent>
```

```
                          <xs:extension base="xs:anySimpleType">
                          <xs:attribute name="test" use="required">
                          <xs:simpleType>
                          <xs:restriction base="xs:string">
                          <xs:enumeration value="empty_line"/>
                          <xs:enumeration value="text_sequence"/>
                          <xs:enumeration value="regex_match"/>
                          <xs:enumeration value="byte_length"/>
                          </xs:restriction>
                          </xs:simpleType>
                          </xs:attribute>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                </xs:complexType>
              </xs:element>
              <xs:element name="Message">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Insert" type="xs:string"/>
                    <xs:element name="Rule" type="RuleForm"/>
                    <xs:element name="Capture" type="CaptureForm"/>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Output">
          <xs:annotation>
            <xs:documentation>Structure of output messages.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Header">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Insert" type="xs:string"/>
                    <xs:element name="RawBinary" type="VarNameForm"/>
                    <xs:element name="Line" type="LineForm"/>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                </xs:complexType>
              </xs:element>
              <xs:element name="Message">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Insert" type="xs:string"/>
                    <xs:element name="RawBinary" type="VarNameForm"/>
                    <xs:element name="Line" type="LineForm"/>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:all>
      <xs:attribute fixed="text" name="type"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

7.5 XSD definition of the AmbiStream DSL for multimedia container formats

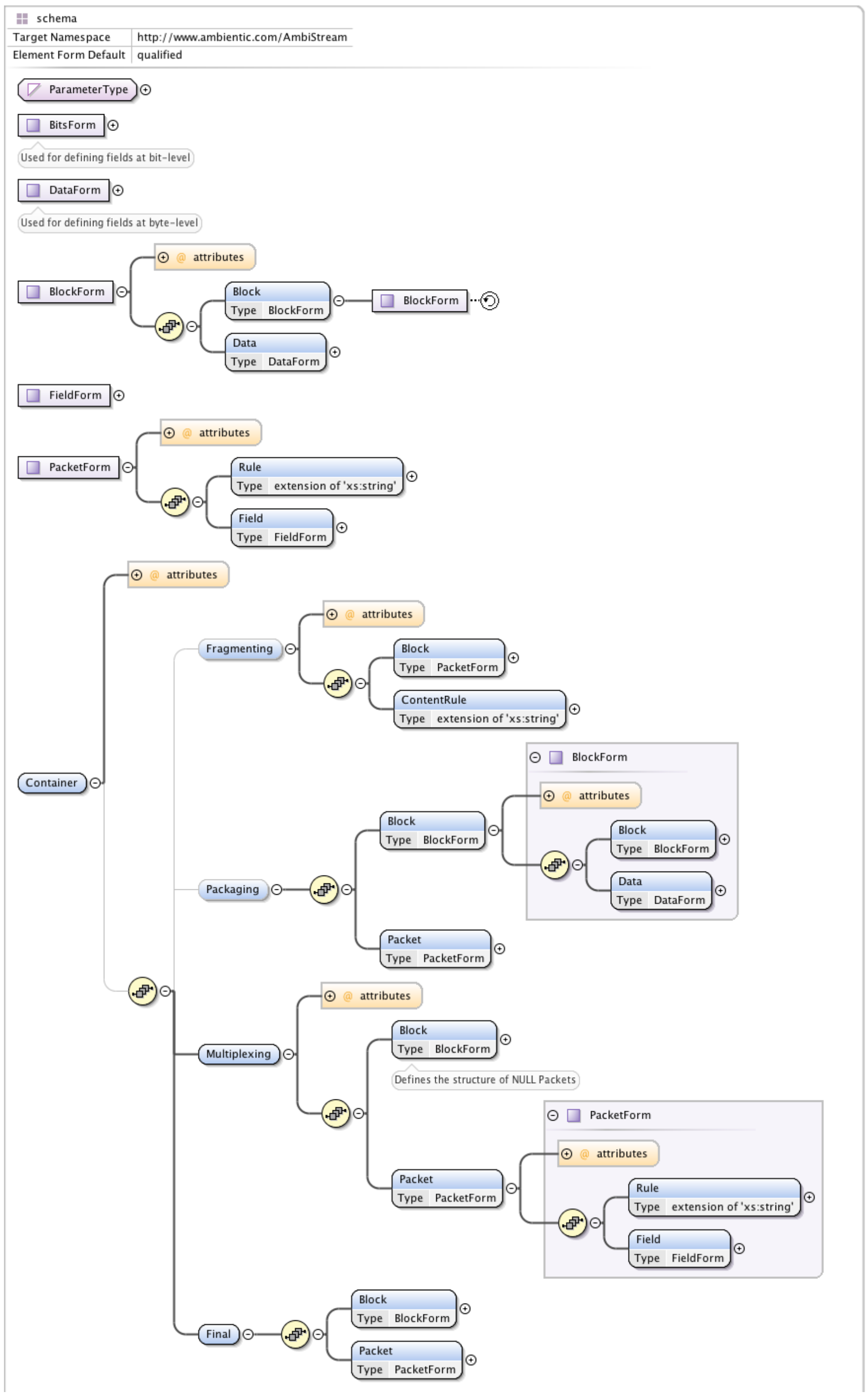

schema
Target Namespace http://www.ambientic.com/AmbiStream
Element Form Default qualified
ParameterType
BitsForm
Used for defining fields at bit-level
DataForm
Used for defining fields at byte-level
BlockForm
attributes
Block Type BlockForm
Data Type DataForm
FieldForm
PacketForm
Rule Type extension of 'xs:string'
Field Type FieldForm
Container
Fragmenting
Block Type PacketForm
ContentRule Type extension of 'xs:string'
Packaging
Packet Type PacketForm
Multiplexing
Defines the structure of NULL Packets
Final

Figure 7.2: XSD definition of the AmbiStream DSL for multimedia container formats

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.ambientic.com/
    AmbiStream"
    targetNamespace="http://www.ambientic.com/AmbiStream" elementFormDefault="qualified">
    <xs:simpleType name="ParameterType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="media_input_length"/>
            <xs:enumeration value="sampling_frequency"/>
            <xs:enumeration value="fragment_index"/>
            <xs:enumeration value="fragment_number"/>
            <xs:enumeration value="fragment_first"/>
            <xs:enumeration value="fragment_last"/>
            <xs:enumeration value="sequence_number"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="BitsForm">
        <xs:annotation>
            <xs:documentation>Used for defining fields at bit-level</xs:documentation>
        </xs:annotation>
        <xs:attribute name="name" type="xs:string"/>
        <xs:attribute name="bitlength" type="xs:integer"/>
        <xs:attribute name="value" type="xs:hexBinary"/>
        <xs:attribute name="parameter" type="ParameterType"/>
    </xs:complexType>
    <xs:complexType name="DataForm">
        <xs:annotation>
            <xs:documentation>Used for defining fields at byte-level</xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="Bits" type="BitsForm"> </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string"/>
        <xs:attribute name="bytelength" type="xs:integer"/>
        <xs:attribute name="value" type="xs:hexBinary"/>
        <xs:attribute name="parameter" type="ParameterType"/>
    </xs:complexType>
    <xs:complexType name="BlockForm">
        <xs:sequence>
            <xs:element name="Block" type="BlockForm"/>
            <xs:element name="Data" type="DataForm"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>
    <xs:complexType name="FieldForm">
        <xs:sequence>
            <xs:element name="Field" type="FieldForm"/>
            <xs:element name="InputRange">
                <xs:complexType>
                    <xs:attribute name="startbyte" type="xs:integer"/>
                    <xs:attribute name="bytelength" type="xs:integer"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="value" type="xs:hexBinary" use="optional"/>
    </xs:complexType>
    <xs:complexType name="PacketForm">
        <xs:sequence>
            <xs:element name="Rule">
                <xs:complexType>
                    <xs:simpleContent>
                        <xs:extension base="xs:string">
                            <xs:attribute name="type" type="xs:string"/>
                        </xs:extension>
                    </xs:simpleContent>
                </xs:complexType>
            </xs:element>
            <xs:element name="Field" type="FieldForm"/>
        </xs:sequence>
        <xs:attribute name=”name” type=”xs:string”/>
        <xs:attribute name=”block” type=”xs:string”/>
    </xs:complexType>
    <xs:element name=”Container”>
        <xs:annotation>
            <xs:documentation/>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence maxOccurs=”1” minOccurs=”0”>
                <xs:element name=”Fragmenting” maxOccurs=”1” minOccurs=”0”>
                    <xs:annotation>
                        <xs:documentation/>
                    </xs:annotation>
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name=”Block” type=”PacketForm”/>
                            <xs:element name=”ContentRule”>
                                <xs:complexType>
                                    <xs:simpleContent>
                                        <xs:extension base=”xs:string”>
                                            <xs:attribute name=”type” type=”xs:string”/>
                                        </xs:extension>
                                    </xs:simpleContent>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                        <xs:attribute name=”method”>
                            <xs:simpleType>
                                <xs:restriction base=”xs:string”>
                                    <xs:enumeration value=”length”/>
                                    <xs:enumeration value=”content”/>
                                    <xs:enumeration value=”packaged_length”/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:attribute>
                    </xs:complexType>
                </xs:element>
                <xs:element name=”Packaging” maxOccurs=”1” minOccurs=”0”>
                    <xs:annotation>
                        <xs:documentation/>
                    </xs:annotation>
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name=”Block” minOccurs=”1” type=”BlockForm”> </xs:
                                element>
                            <xs:element name=”Packet” type=”PacketForm”/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name=”Multiplexing”>
                    <xs:annotation>
                        <xs:documentation/>
                    </xs:annotation>
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name=”Block” minOccurs=”1” type=”BlockForm”>
                                <xs:annotation>
                                    <xs:documentation>Defines the structure of NULL Packets</xs:
                                        documentation>
                                </xs:annotation>
                            </xs:element>
                            <xs:element name=”Packet” type=”PacketForm”/>
                        </xs:sequence>
                        <xs:attribute fixed=”time_division” name=”type” type=”xs:string”/>
                        <xs:attribute name=”constantbitrate” type=”xs:boolean”/>
                        <xs:attribute name=”timeslot” type=”xs:integer”/>
                        <xs:attribute name=”maxtimesloterror” type=”xs:integer”/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Final">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="Block" minOccurs="1" type="BlockForm"> </xs:
                                element>
                            <xs:element name="Packet" type="PacketForm"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute fixed="binary" name="type"/>
            <xs:attribute name="name" type="xs:string"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

7.6 XSD definition of the AmbiStream Mediator DSL

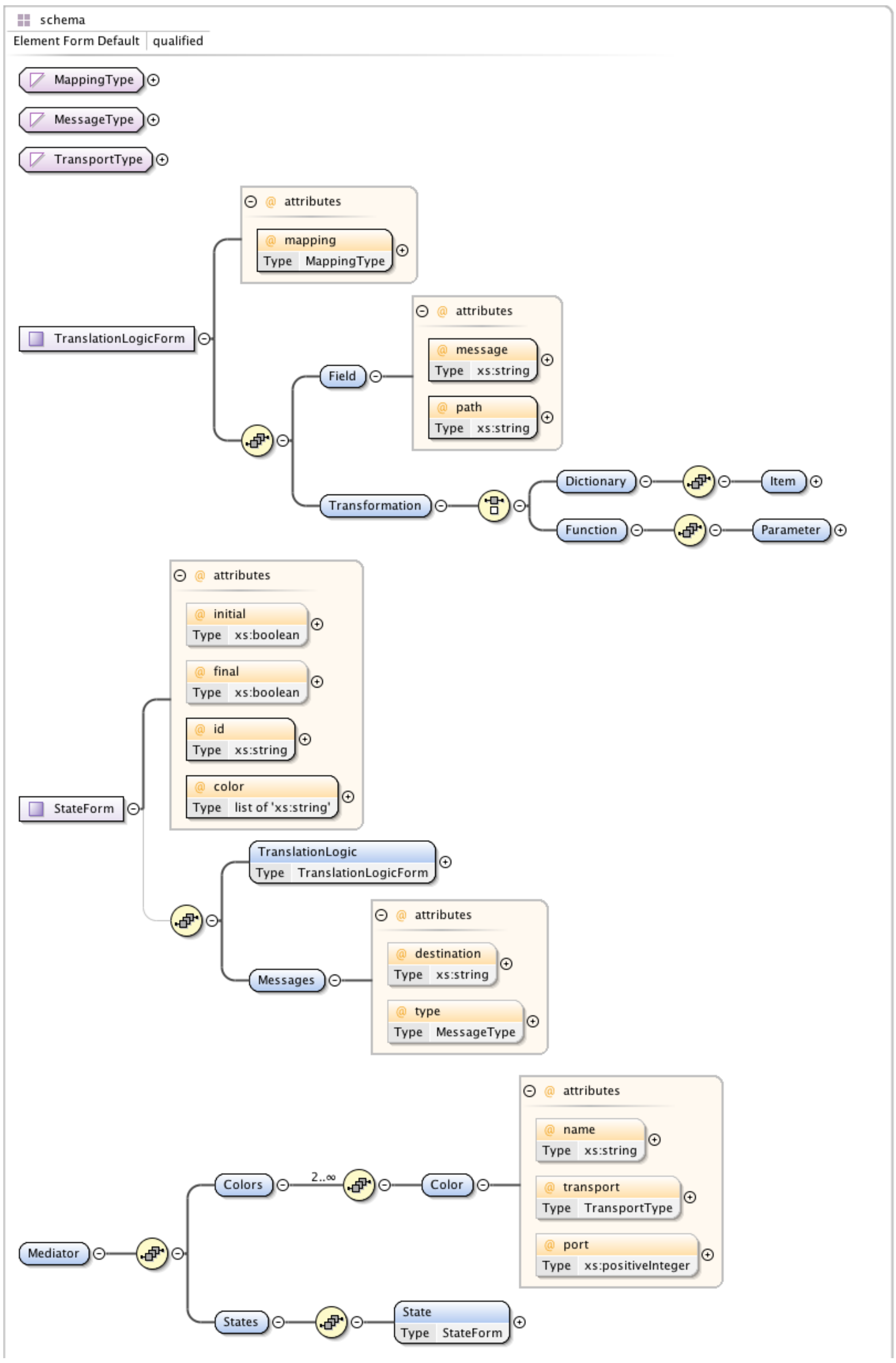


Figure 7.3: XSD definition of the AmbiStream Mediator DSL

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:simpleType name="MappingType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="assignment"/>
            <xs:enumeration value="dictionary"/>
            <xs:enumeration value="function"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="MessageType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="send"/>
            <xs:enumeration value="receive"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="TransportType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="UDP"/>
            <xs:enumeration value="TCP"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="TranslationLogicForm">
        <xs:sequence>
            <xs:element name="Field">
                <xs:complexType>
                    <xs:attribute name="message" type="xs:string" use="required"/>
                    <xs:attribute name="path" type="xs:string" use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="Transformation">
                <xs:complexType>
                    <xs:choice>
                        <xs:element name="Dictionary">
                            <xs:complexType>
                                <xs:sequence>
                                    <xs:element name="Item">
                                        <xs:complexType>
                                            <xs:attribute name="key" type="xs:string" use="
                                                required"/>
                                            <xs:attribute name="value" type="xs:string"/>
                                        </xs:complexType>
                                    </xs:element>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="Function">
                            <xs:complexType>
                                <xs:sequence>
                                    <xs:element name="Parameter">
                                        <xs:complexType>
                                            <xs:attribute name="name" use="required"/>
                                            <xs:attribute name="path" use="required"/>
                                        </xs:complexType>
                                    </xs:element>
                                </xs:sequence>
                            </xs:complexType>
                        </xs:element>
                    </xs:choice>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="mapping" type="MappingType" use="required"/>
    </xs:complexType>
    <xs:complexType name="StateForm">
        <xs:sequence maxOccurs="1" minOccurs="0">
            <xs:element name="TranslationLogic" type="TranslationLogicForm"/>
            <xs:element name="Messages">
                <xs:complexType>
```

```
                    <xs:attribute name="destination" type="xs:string"/>
                    <xs:attribute name="type" type="MessageType"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="initial" type="xs:boolean"/>
        <xs:attribute name="final" type="xs:boolean"/>
        <xs:attribute name="id" type="xs:string" use="required"/>
        <xs:attribute name="color" use="required">
            <xs:simpleType>
                <xs:list itemType="xs:string"/>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
    <xs:element name="Mediator">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Colors">
                    <xs:complexType>
                        <xs:sequence maxOccurs="unbounded" minOccurs="2">
                            <xs:element name="Color">
                                <xs:complexType>
                                    <xs:attribute name="name" type="xs:string"/>
                                    <xs:attribute name="transport" type="TransportType"/>
                                    <xs:attribute name="port" type="xs:positiveInteger"/>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="States">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="State" type="StateForm"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Bibliography

[1] D. Alrajeh, A. Russo, and S. Uchitel. Inferring operational requirements from scenarios and goal models using inductive learning. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 29–36. ACM, 2006.

[2] E. M. Andriescu, R. Speicys Cardoso, and V. Issarny. AmbiStream: A Middleware for Multimedia Streaming on Heterogeneous Mobile Devices. In A.-M. Kermarrec and F. Kon, editors, *ACM/IFIP/USENIX 12th International Middleware Conference*, volume 7049 of *Lecture Notes in Computer Science*, Lisbon, Portugal, Dec. 2011. Springer.

[3] A. Bennaceur, G. Blair, N. Georgantas, P. Grace, F. Howar, V. Issarny, R. Johansson, M. Merten, A. Moschitti, B. Steffen, and D. Sykes. The role of machine learning in achieving and sustaining interoperability in extreme distributed systems. 2011. Under submission.

[4] A. Bennaceur, G. S. Blair, F. Chauvel, N. Georgantas, P. Grace, V. Nundloll, M. Paolucci, R. Saadi, and D. Sykes. Intermediate CONNECT architecture. Technical Report D1.2, CONNECT ICT FET IP Project, February 2011.

[5] A. Bennaceur, J. Richard, M. Alessandro, S. Romina, D. Sykes, R. Saadi, and V. Issarny. Inferring Affordances Using Learning Techniques. In *International Workshop on Eternal Systems (EternalS'11)*, Budapest, Hongrie, 2011.

[6] A. Bertolino, G. Blair, F. Chauvel, C. F. Cortes, N. Georgantas, P. Grace, F. Howar, T. Huyn, B. Jonsson, M. Paolucci, A. Pathak, B. Souville, and M. Tivoli. Initial CONNECT architecture. Technical Report D1.1, CONNECT ICT FET IP Project, February 2010.

[7] Y.-D. Bromberg, P. Grace, and L. Réveillère. Starlink: runtime interoperability between heterogeneous middleware protocols. In *Proceedings of 31th International Conference on Distributed Computing Systems, ICDCS (IEEE).*, pages 446 –455, June 2011.

[8] L. Cavallaro, E. Di Nitto, C. Furia, and M. Pradella. A tile-based approach for self-assembling service compositions. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 43–52. IEEE, 2010.

[9] S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, 2008.

[10] CONNECT Consortium. Emergent connector for eternal software intensive networked systems. Fet proactive 6: Ict forever yours description of work, CONNECT, 2008.

[11] CONNECT Consortium. Automating interoperability checking. Technical Report D2.3, CONNECT, February 2011.

[12] CONNECT Consortium. Consolodated dependability framework. Technical Report D5.3, CONNECT, February 2011.

[13] CONNECT Consortium. Deployment of learning techniques. Technical Report D4.3, CONNECT, February 2011.

[14] CONNECT Consortium. Dynamic connector synthesis - revised prototype implementation. Technical Report D3.3, CONNECT, February 2011.

[15] R. S. Cruz, M. S. Nunes, and J. E. Gonalves. A personalized HTTP adaptive streaming WebTV. In *User Centric Media*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Dec. 2010.

[16] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.

[18] H. Goldsby, P. Sawyer, N. Bencomo, B. Cheng, and D. Hughes. Goal-based modeling of dynamically adaptive system requirements. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 36 –45, 31 2008-april 4 2008.

[19] T. Huynh and A. L. (eds.). Experiment scenarios, prototypes and report iteration 2. Technical Report D6.3, CONNECT ICT FET IP Project, February 2012.

[20] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. Socks protocol version 5, 1996.

[21] M. Lindeberg, S. Kristiansen, T. Plagemann, and V. Goebel. Challenges and techniques for video streaming over mobile ad hoc networks. *Multimedia Systems*, 2011.

[22] M. Ouederni and G. Salaün. Tau be or not tau be? - a perspective on service compatibility and substitutability. In *WCSI*, 2010.

[23] R. Pantos and W. May. HTTP Live Streaming. (Internet-Draft), Mar. 2011.

[24] M. Pistore, P. Traverso, and P. Bertoli. Automated composition of web services by planning in asynchronous domains. 2005.

[25] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In *FASE*, pages 377–380, 2006.

[26] R. Ramdhany, P. Grace, G. Coulson, and D. Hutchison. Manetkit: Supporting the dynamic deployment and reconfiguration of ad-hoc routing protocols. In J. Bacon and B. F. Cooper, editors, *Middleware*, volume 5896 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2009.

[27] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. Technical Report TR74-218, Department of Computer Science, Cornell University, Ithaca, New York, 1974.

[28] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051, 6222.

[29] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), Apr. 1998.

[30] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[31] R. Spalazzese, P. Inverardi, and V. Issarny. Towards a formalization of mediating connectors for on the fly interoperability. In *The IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA)*, 2010.

[32] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 1–8. ACM, 2008.

[33] D. Sykes, W. Heaven, J. Magee, and J. Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 431–438. ACM, 2010.

[34] D. Sykes, J. Magee, and J. Kramer. Flashmob: distributed adaptive self-assembly. In *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems*, pages 100–109. ACM, 2011.

[35] W. Van Lancker, D. Van Deursen, E. Mannens, and R. Van de Walle. Implementation strategies for efficient media fragment retrieval. *Multimedia Tools and Applications*, Mar. 2011.