



The CONNECT Architecture

Paul Grace¹, Nikolaos Georgantas², Amel Bennaceur², Gordon Blair¹, Franck Chauvel³, Valerie Issarny², Massimo Paolucci⁴, Rachid Saadi², Bertrand Souville⁴, Daniel Sykes²

¹ School of Computing and Communications, Lancaster University, UK
p.grace@lancaster.ac.uk

² INRIA, CRI Paris-Rocquencourt, France
{nikolaos.georgantas, amel.bennaceur, valerie.issarny, rachid.saadi,
daniel.sykes}@inria.fr

³ School of Electronics Engineering and Computer Science, Peking University, China
franck.chauvel@sei.pku.edu.cn

⁴ Laboratories Europe GmbH, Munich, Germany
{paolucci, souville}@docomolab-euro.com

Abstract. Current solutions to interoperability remain limited with respect to highly dynamic and heterogeneous environments, where systems encounter one another spontaneously. In this chapter, we introduce the CONNECT architecture, which puts forward a fundamentally different method to tackle the interoperability problem. The philosophy is to observe networked systems in action, learn their behaviour and then dynamically generate mediator software which will connect two heterogeneous systems. We present a high-level overview of how CONNECT operates in practice and subsequently provide a simple example to illustrate the architecture in action.

Keywords: Interoperability, emergent middleware, modelling, synthesis, middleware, protocol

1 Introduction

1.1 Motivation: The Interoperability Problem

Interoperability is a measure of the ability of systems to connect, understand and exchange data with one another. As such, it reveals one of the fundamental problems in computer science. Indeed, the world wide budget for Interoperability is estimated to be in excess of \$1 Trillion [7]). In the chapter '*Interoperability in Complex Distributed Systems*' of this book that surveys the interoperability problems and state of the art solutions [5], the important barriers to fully achieving interoperability are identified as:

- *Data heterogeneity.* Applications may use data that is represented in different ways and/or have different meanings.

- *Middleware heterogeneity.* Different protocols are used to advertise and search for services, e.g., Service Location Protocol (SLP), Jini, Universal Plug and Play (UPnP), and Lightweight Directory Access Protocol (LDAP). Further, services use different protocols to exchange and use data, e.g., Remote Method Invocation protocols such as SOAP, Java RMI and IIOP; or different messaging protocols such as Java Message Service (JMS) or Microsoft Message Queuing (MSMQ).
- *Application heterogeneity.* The application interfaces may be different in terms of the descriptions of operations, e.g., the behaviour provided by one operation in one interface may be provided by multiple operations in the other interface. Interfaces may also be heterogeneous in terms of the order in which operations must/should be called.
- *Heterogeneity of non-functional properties.* Systems may have particular non-functional properties, e.g., latency of message delivery, dependability measures and security requirements that must be resolved with respect to the connected system.

As a traditional solution to this problem, middleware-based standards (e.g. Web Services [6] or CORBA [26]) allow systems to be designed in advance in order to interoperate with each other. However, where environments are heterogeneous and dynamic (e.g. pervasive computing) such standards cannot be agreed upon in advance, nor can they deal with the heterogeneity of the networked systems in these environments. Interoperability platforms and transparent interoperability solutions offer more dynamic approaches. Interoperability platforms such as ReMMoC [16] and UIC [30], allow clients to be developed transparently from the heterogeneous middleware that may be spontaneously encountered in the future; these plug-in software at runtime that can communicate with the encountered protocol. While suitable for systems that know they will need to interoperate with heterogeneous protocol, this approach cannot solve the problem of two legacy platforms required to interoperate with one another; INDISS [8] and uMiddle [24] are examples of transparent interoperability solutions that dynamically translate through an intermediary language to achieve this requirement. However, in all of these cases, only a subset of the above four barriers are attempted to be resolved; see [5] for a detailed analysis of the state of the art which illustrates this observation.

Therefore, we advocate that new approaches are required to tackle interoperability in a fundamentally different way to achieve the objective of *universal and long-lived interoperability*. This goal is akin to the ideas of *universal translation*, a common device often appearing in science fiction; for example, the Babel Fish in “The Hitchhikers Guide to the Galaxy” [1] offers universal translation to allow native speech to be automatically and transparently translated to the language of any of the listeners, i.e., everyone speaks and hears their own language.

1.2 The CONNECT Approach

The approach of CONNECT is to produce *emergent middleware*, i.e., rather than create another middleware technology that is destined to be yet another legacy

platform that in turn adds to the interoperability problem, we propose the novel approach of *generating the required middleware at runtime*, i.e., we synthesize the necessary software to connect (translate between) two end-systems. For example, if a client application developed using SOAP [17] encounters a CORBA server then the framework generates a CONNECTOR that resolves the heterogeneity of the data exchanged, the application behaviour, and the lower level middleware and network communication protocols.

To underpin the creation of *emergent middleware*, the CONNECT architecture performs the following important phases of system behaviour.

- *Discovering the functionality* of networked systems and applications advertised by legacy discovery protocols, e.g., Service Location Protocol (SLP) and Simple Service Discovery Protocol (SSDP). Then, transforming this discovered information to a rich intermediary description (the CONNECT Networked System Model) that can then be used to syntactically and semantically match heterogeneous services.
- *Using learning algorithms* to dynamically determine the interaction behaviour of a networked system from its intermediary representation and producing a model of this behaviour in the form of a labelled transition system (LTS). A full description of how learning is enabled in CONNECT is provided in [19].
- *Dynamically synthesizing* a software mediator. Taking as input the Networked System model and the learned LTS of two networked systems, CONNECT uses a formal approach to match the application behaviour of these systems and then map them onto one another to form the application mediator (to resolve the application behaviour differences); more detailed information about this method is provided in the chapter ‘Application-layer CONNECTOR Synthesis’ [28]. Further, the differences in the middleware protocols are resolved through a similar formal method for matching and mapping of middleware protocols to produce middleware mediation methods; this is presented in the chapter ‘Middleware-layer CONNECTOR Synthesis’ [20]. The combination of the synthesized application-level and middleware-level mediators form the CONNECTOR mediator.
- *Deployment in the network environment*. The CONNECTOR mediator is made concrete by deploying it upon appropriate *listeners* and *actuators* that can communicate directly with networked systems using their legacy protocols.
- *Verification & validation* of the CONNECTOR is performed by enablers during mediator synthesis phase and also after deployment to ensure the correctness of the CONNECTOR and the running CONNECTED system with respect to the requirements (and importantly the non-functional requirements) and intents of the involved networked systems. This process ensures the long-lived nature of a CONNECT solution. The methods to perform verification and validation are provided in the chapter ‘Dependability and Performance Assessment of Dynamic CONNECTED Systems’ [3]

1.3 Structure of the Chapter

This chapter first provides a broad overview of the CONNECT architecture, identifying the key functions and principles, and then a simple example is utilised to illustrate how the overall architecture operates. Only a subset of the technical details are introduced, instead the chapter points the interested reader to other publications (including further chapters of this book) in order to discover the richer details and formal methods. The chapter is organised as follows. The overall CONNECT architecture is presented in Section 2; in particular this highlights how networked systems are first discovered and modelled, and then how the emergent CONNECTORS between them are realised. To illustrate an important feature of the architecture, a description of the technologies employed to deploy CONNECTORS is given in Section 3, here the methods to dynamically generate middleware protocol listeners and actuators are discussed. We then present a case study showing how a CORBA-based networked system achieves interoperability with a SOAP asynchronous messaging networked system using the CONNECT architecture in Section 4. Finally, in Section 5 we offer conclusions about the architecture and then pinpoint areas of interest for future research.

2 A Framework for Interoperability

2.1 CONNECT Actors

Before exploring the details of the CONNECT architecture we first introduce the key actors that are involved in the CONNECT process. These are central to the underlying architectural principles:

- *Networked systems* are systems that manifest the will to connect to other systems for fulfilling some intent identified by their users and the applications executing upon them.
- *Enablers* are networked entities in the environment of networked systems that incorporate all the intelligence and logic offered by CONNECT for enabling connection between heterogeneous networked systems. Enablers constitute the CONNECT enabling architecture.
- *CONNECTORS* are the emergent connectors produced by the action of enablers.
- *CONNECTED systems* are the outcome of the successful creation and deployment of *CONNECTORS*.

A high-level view of these actors is shown in Figure 1. It can be seen that networked systems manifest their will to connect. This will, along with information about the networked systems, is communicated in the form of some input to the enablers. One or more enablers collaborate to synthesize and deploy a *CONNECTOR* that enables networked systems to connect and fulfill their individual intents.

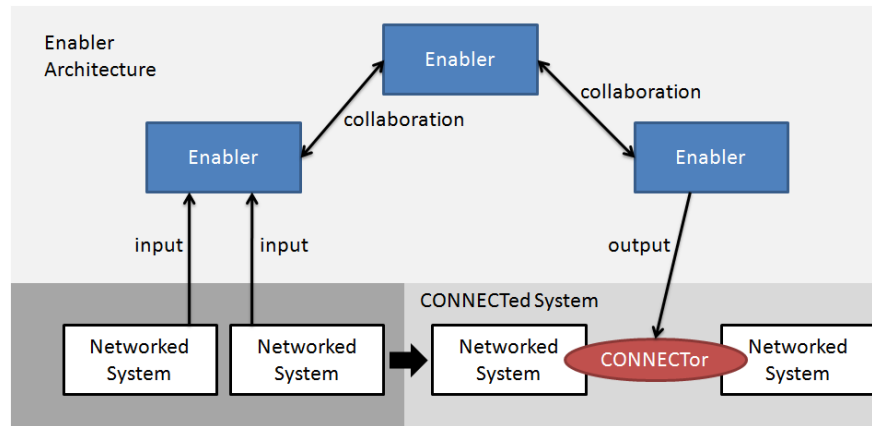


Fig. 1. Actors in the CONNECT architecture

2.2 Networked System Model

CONNECT seeks to observe, learn and model the external interaction behaviour of a networked system. This model, termed the *Networked System Model* is central to the CONNECT architecture and contains the information required by the enablers to produce the CONNECTors that ensure heterogeneous networked systems interoperate. There are two levels of interaction that must be considered by the model:

- *Middleware-layer interaction.* This includes information about the interaction protocol and the underlying network transport: what are the messages, their data content and format, and their sequence? The middleware semantics will also be covered, i.e., is this client-server, peer-to-peer, etc? Is the communication paradigm message-based or event-based, synchronous or asynchronous?
- *Application-layer interaction.* The application component describes: an intent, what external behaviour it requires, and what external behaviour it provides. The essential feature here is the interface, that is, a description of the set of functionalities of the component made accessible to (but also required from) its environment. Typically, this description comes in the form of a set of data inputs and associated outputs following a specific data type system. The application-layer will also describe its behaviour in terms of the sequence of application operations, and also the associated non-functional requirements of this behaviour.

The CONNECT Networked System model takes these abstract elements that are typically spread across different service descriptions, and the corresponding languages (e.g. Interface descriptions in WSDL [10], semantic annotations in SA-WSDL [12], and behaviour in BPEL), and integrates them into a uniform

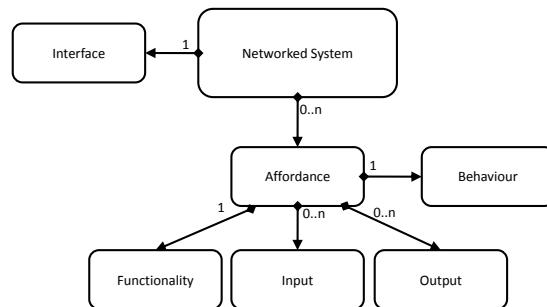


Fig. 2. Overview of the Networked System Model

model that can be shared, understood and processed by the enablers. A high-level overview of this model is shown in Figure 2 and importantly highlights the key features of the model:

- 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 action-relationships/interactions with another networked system to be matched; in short, they are complementarily providing/requesting the same thing.
- *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 non-functional requirements of the interface operations are also described.
- 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 is employed to specify this behaviour.

2.3 The CONNECT Enabler Architecture

As previously identified, it is the CONNECT enablers whose role is to co-ordinate in order to produce a CONNECTOR that will ensure two legacy applications can interact. These enablers follow an important sequence of behaviour that we now identify:

- *Discovery* enables networked systems to manifest their will to connect to other networked systems and to discover mutually interested networked systems, while at the same time allows the CONNECT enabling architecture to retrieve initial information on likely-to-be-associated networked systems.

- *Learning* is performed by enablers upon networked systems for completing the initial information about the latter provided by discovery. The outcome of combined discovery and learning should be a sufficiently good Networked System Model of a networked system.
- *Synthesis & deployment* is performed by enablers for generating and deploying an appropriate CONNECTOR that will successfully bridge the heterogeneous systems and establish a CONNECTed system.
- *Verification & validation* is performed by enablers during and after the synthesis phase for ensuring the correctness of the CONNECTOR and the running CONNECTed system with respect to the requirements and intents of the involved networked systems.

These phases of behaviour are then split into software components each responsible for a particular role; hence this software component becomes a CONNECT enabler. The Enabler architecture is then the configuration of these enabler components which are deployed in the network environment and remotely communicate with each other. Figure 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, learn their behaviour, identify a solution to ensure they interoperate, and then synthesize and deploy the required CONNECTOR. We discuss the individual enablers in turn and describe how they communicate.

The Discovery Enabler The discovery enabler leverages existing service discovery protocols such as SLP [18], UPnP [15], and WS-Discovery [25] in order to initially find out what networked systems are operating in the environment, what their intent and requirements for connection are, and whether other networked systems match these requirements. The discovery enabler receives both the advertisement messages and lookup request messages that are sent within the network environment by listening on known multicast addresses (used by legacy discovery protocols). These messages are then processed and their information from the legacy messages is extracted to form a *partial networked system model* for each of the networked systems, where the partial model consists of the affordance, and the application interface as shown in Figure 2. Further the discovery enabler can also extract information about the middleware protocols employed to provide initial input to the model of behaviour in Figure 2; for example, this information could be extracted from the WSDL binding element [10] in the case of WS-Discovery, or by parsing the protocol part of the URL returned by a discovery protocol (as in the case of SLP [18] and Bonjour⁵).

Initial matching is performed between discovered systems to determine whether two networked systems are candidates to have a CONNECTOR generated between. The matching method examines the affordances of the two systems and employs ontology-based matching to identify if the two are a good match. On a match, the CONNECT process is initiated; first the current partial Networked System

⁵ <http://developer.apple.com/networking/bonjour/specs.html>

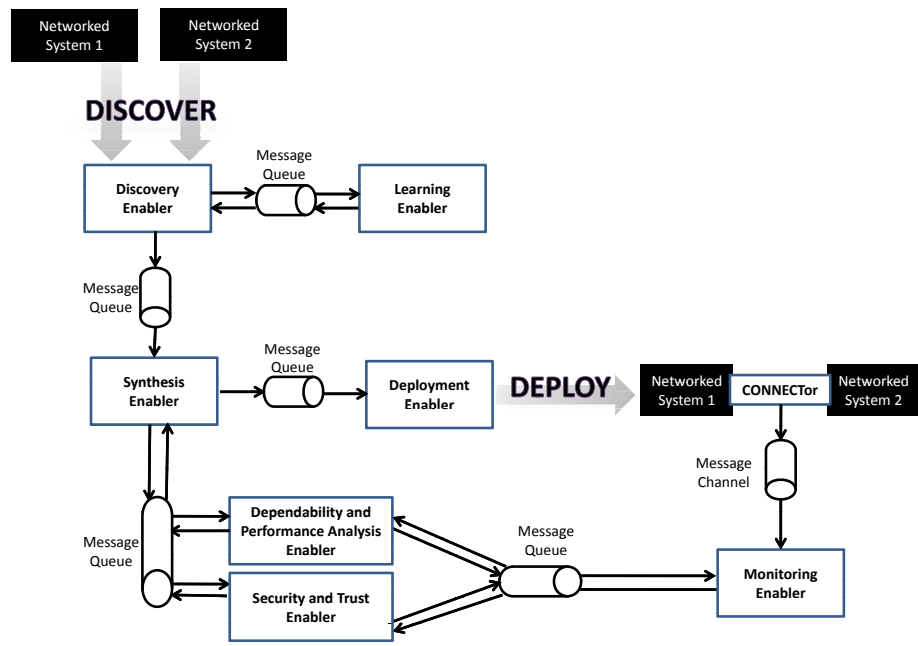


Fig. 3. The CONNECT Enabler architecture

Model of each system is sent to the *learning enabler*, which then adds to the behaviour description to the model to complete a richer view of the system's behaviour. On the completion of the Networked System Model, the discovery enabler sends this model to the *synthesis enabler*.

The Learning 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 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 [29]. The learning method utilises two inputs: i) the interface description in the Networked System Model, and ii) the semantic annotations (that annotate the interface) which provide richer meanings to the tool. The learning enabler produces an LTS describing the interaction behaviour; this 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.

Synthesis Enabler The role of the synthesis enabler is to take the Networked System Models of two systems and then synthesize the mediator component that is employed by the CONNECTOR to co-ordinate the interaction between the two. Here, the synthesis enabler creates a mediator to resolve: i) application-level interoperability, and ii) middleware level interoperability. The LTS received from the discovery and learning phase is middleware specific, i.e., the transitions are strongly correlated to the behaviour of the middleware protocol. The first step is to abstract the behaviour of the system in a middleware-agnostic way to capture the application behaviour. This mapping is underpinned by a set of middleware rules and domain ontologies that describe how middleware behaviour can be abstracted towards a common representation of application behaviour—the *middleware agnostic LTS*). The methods to create middleware agnostic LTS are described in [20].

The next step is to create a common (application-level) abstraction of the networked systems. This method takes into account the ontology-based specification of each networked system and the common ontology specification for the application domain to produce corresponding abstract LTS for the middleware-agnostic LTS. Once complete, the two LTS can be matched and mapped to create the mediator. First, the existence of common traces in the LTS that lead the two systems to achieve a common goal is automatically checked; if there is a match and at least one common trace is found (which leads to achieve the specified common goal), the mapping between the two LTSs, over the common traces, is automatically performed and producing an abstract LTS that models the interaction behavior of the mediator. This is only a brief overview of this method and further information can be found in [28].;

Finally, the abstract LTS is made concrete by reapplying the middleware-specific information that was abstracted upon earlier in the method; this produces the concrete CONNECTOR LTS that can be synthesized to create the software that can be directly deployed in the CONNECTORS between the two legacy networked systems. From this the software The synthesis enabler can then output two alternative software types (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 *'executable' LTS model*. The concrete 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.

Deployment Enabler The Deployment Enabler receives as input the mediator code (or the LTS model) and the original Networked System Models; its objective is to finalise and then deploy the CONNECTOR in each 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. We discuss how the listeners and actuators are realised in Section 3.
- It deploys and manages the executable code (or the LTS model) of the CONNECTORS in the network. For this, the enabler utilises OSGi⁶ techniques; that is, the components that form the CONNECTORS are bundled into OSGi components this allows them to be automatically deployed and executed upon network hosts running an OSGi platform (after being downloaded to the appropriate location); that is, the components that form the CONNECTORS are bundled into OSGi components this allows them to be automatically deployed and executed upon network hosts running an OSGi platform (after being downloaded to the appropriate location).

Dependability and Performance Analysis/Security and Trust (SXT)

Enabler 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 enhances the initial LTS in order that it better meets the requirements of the connection; these enhanced models are returned to the synthesis enabler. The dependability enabler also continuously determines if the CONNECTOR maintains its non-functional requirements (as identified in the networked system's interface). 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.

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

The Connect Message Bus The enablers and CONNECTORS use a simple message-based communication model to exchange information with one another. A Java Messaging Service (JMS) implementation⁷ is used to implement the Message Bus. The reason for this choice of communication model is that two styles of communication are important in the CONNECT enabler architecture and are both provided by the technology:

⁶ <http://www.osgi.org>

⁷ <http://www.oracle.com/technetwork/java/index-jsp-142945.html>

- Point-to-Point exchange between enablers. As described earlier, the enablers send content (e.g., models and code) to be processed by a specific party, e.g., the discovery and learning enabler communicating to build the Networked System Model. JMS allows the behaviour to be achieved using a message queue as illustrated in Figure 3.
- Publish-Subscribe communication regarding CONNECTOR behaviour. The CONNECTORS produce events in order for them to be monitored; enablers can subscribe to the channels that the CONNECTORS publish these events to. For example, in Figure 3 the monitoring enabler subscribes to this channel in order to monitor CONNECTOR events.

2.4 CONNECTORS

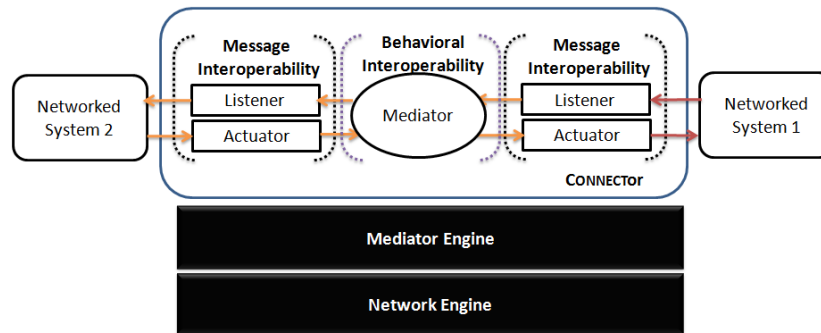


Fig. 4. The CONNECTOR Architecture

We now introduce the software elements that make up an individual CONNECTOR and also how they interact in order to achieve interoperability. This CONNECTOR architecture is illustrated in Figure 4. The software elements 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., the SOAP listener parses SOAP messages. A listener produces an *Abstract Message* (see Section 3 for more information about abstract messages) 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. The API of the listener in Java is shown in Figure 5, the packet in a byte array is passed to the `MessageParse` method and a Java Object (`AbstractMessage`) representing the Abstract Message is produced.

- 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 API of the actuator in Java is shown in Figure 5, a byte array is produced when the `AbstractMessage` object is passed to the `MessageCompose` method.
- The *Mediator* forms the central co-ordination element of a generated CONNECTOR. Its role is to translate the content received from one protocol (using `Abstract Message`) into the content required to send to the corresponding protocol. The mediator therefore addresses the challenges of mapping between: 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 uniform interface provided by the network engine is similar to network programming libraries provided.
- The *Mediator engine* in the figure is an optional element of the architecture depending upon the implementation approach taken for mediators. The behaviour of the mediator is determined by a high-level model determining the operations to take. In the case where this model is turned 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. This flexibility in the intermediary architecture allows us to investigate the benefits of the two approaches, i.e., to investigate the performance gains of direct code generation, versus the ability to easily adapt the behaviour of the CONNECTOR at runtime when it is a model executed on the mediation engine..

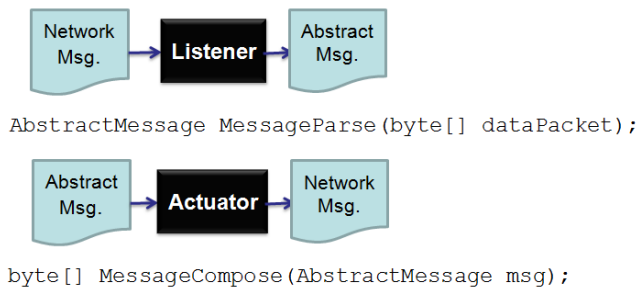


Fig. 5. Listeners and Actuators API

2.5 Summary

This section has introduced the overall CONNECT architecture that puts the philosophy of discovery, learning and synthesis of CONNECTORS into practice. Further information about the behaviour of CONNECT enablers can be found in chapters of this book [3] [20] [28]. We will now look more closely at the problem of communicating with networked systems, i.e., how the software mediators can send and receive messages in the protocols that are utilised. For example, if the networked systems use SOAP and IIOP how can the mediator send and receive SOAP and IIOP messages.

3 Communicating with Legacy Protocols

CONNECTORS work by taking the concrete messages of legacy protocols and then creating an abstract representation of this data (the abstract message) such that it can be used to translate to one or more messages of a different legacy protocol. The translated abstract message then being composed into the concrete message format of the destination protocol. To illustrate this, consider Figure 6 which shows two protocol messages broken down into their field content; the message on the left is an SLP lookup message, whereas the message on the right is an SSDP lookup message. Both are performing the same function searching for a service of a given service type (this is the data contained in the **SrvType string** field of SLP and the **Service Type** field of SSDP). To achieve interoperability between them we need to extract data from the original concrete message, translate this, and then compose new concrete messages. This is a key underlying principle of the CONNECT architecture and in this section we discuss techniques to manipulate network messages. We first introduce the concept of abstract message, and then present solutions to marshal and unmarshal legacy protocol messages to/from this representation.

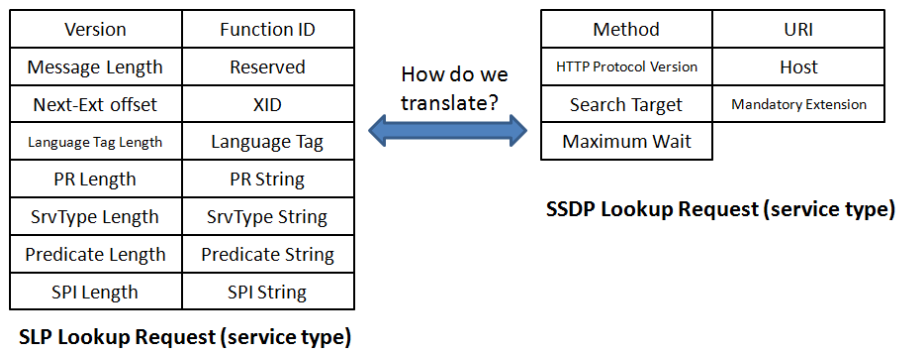


Fig. 6. Message formats of heterogeneous protocols—SLP and SSDP

3.1 Abstract Messages

A network message (as employed by a legacy communication protocol) is typically organized as a sequence of text lines for text-based protocols, or of bits, for a binary protocol. Messages are composed of fields. A CONNECTOR must extract relevant fields from the received message and use them to create one or more messages according to the target protocols. Similarly, it must extract relevant fields from the received responses and ultimately create a response according to the source protocol. Hence, the design of CONNECTORS is based upon these message-based events; and the key design principle is to derive information from network messages and then describe them in a protocol independent manner. We term this protocol independent description of a message: **the Abstract Message**. Received network messages are converted to an Abstract Message, correspondingly the Abstract Message is used to build the network message that must be sent.

```

<xsd:schema>
  <xsd:element name="Field">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="label" type="xsd:string"/>
        <xsd:element name="length" type="xsd:integer"/>
        <xsd:element name="type" type="xsd:string"/>
        <xsd:element name="mandatory" type="xsd:boolean"/>
        <xsd:element name="value" type="xsd:any"/>
        <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="AbstractMessage">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

Fig. 7. The Abstract Message Schema

The schema for the Abstract Message content is illustrated in Figure 7. This shows that an Abstract Message consists of a set of fields; a field can be either primitive or structured. A *primitive field* is composed of a label naming the field, a type describing the type of the data content, a length defining the length in bits of the field, a boolean stating if this is a mandatory or optional field, and the value of the field, i.e., the data content. A *structured field* is composed of multiple primitive fields. For example, a URL field is composed of four primitive fields: the protocol, the address, the port, and the resource location.

Abstract Messages then represent the interface between the Listeners, Actuators and the Mediator, and the underlying network messages themselves. In

order to achieve interoperability dynamically, the `CONNECTor` receives network messages from a networked system (in the format of the protocol employed by this legacy system). This event will trigger the execution of the Mediator, whose behaviour will determine the sequence of actions that manipulate the listeners and actuators. For example, it may receive one or more messages in the Abstract Message format and it may send one or more messages by composing a new Abstract Message and sending this to an Actuator to be delivered to the target networked system.

3.2 From Abstract Message to Concrete Message

To form a `CONNECTor` the mediator must be able to communicate with the networked systems using their legacy protocol. Hence, the mediator is composed with Listeners and Actuators as described earlier in the vision of the `CONNECT` architecture (see Section 2.4). Within `CONNECT`, the general philosophy employed for the deployment of Listeners and Actuators is to utilise DSLs to describe protocol messages. These high-level descriptions are then used to create the software components that will be deployed in the `CONNECTors`. A Message Description Language (MDL) is the language used to describe a message format; the MDL specification for a particular protocol then describes its set of messages only. Message composers and parsers are implemented as general interpreters that execute the message description language specifications that are loaded. For example, a parser that interprets an SLP MDL instance will only parse SLP messages into the abstract message representation, i.e., it interprets the incoming message based upon the specification. Hence parsers are specialised to a particular protocol by associating the protocol specification to produce the Listener. Actuators are created using the same process to specialise generic message composers for text and binary protocols. An overview of this specialisation process is illustrated in Figure 8.

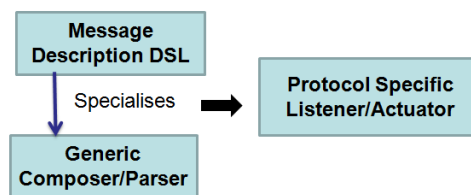


Fig. 8. The approach for generating Listeners and Actuators

There are a number of languages that can be used to parse network messages or parse data files. We investigated each of these as potential languages to be used in `CONNECT`; the results of this are seen in Table 1. It can be seen that a number of the tools focus solely on generating software to parse only data and messages, i.e., `BinPac`, `Datascript` and `PacketTypes`; therefore, these are unsuitable as it is

equally important to be able to generate the composer part of the CONNECTOR. Similarly, a number of the languages only consider binary data (i.e., all except PADS and ASN1.0); however, CONNECT requires the parsing of heterogeneous protocols which may use text or XML. In the example in Figure 6 SLP is a binary message, whereas SSDP is a text message. Hence, the only potential solutions are: i) PADS which offers the additional benefit of being able to infer data descriptions from received data [14], or ii) ASN 1.0. The drawback of these two are that they are not specifically designed for network packets, and we found when creating descriptions of example packet formats for SLP and GIOP that we were unable to successfully create the correct parsers and composers. Given the results of this investigation, CONNECT proposes new Message Description Languages along with their corresponding tools in order to first provide a simple mechanism to parse and compose network packets.

Tool	Language	Generate Parser	Generate Composer	Domain
ASN 1.0 [31]	Java/C	x	x	Many encodings: binary, text, xml
BinPAC [27]	C++	x		Binary data and network packets
Datascript [2]	Java	x		Binary data
PADS [13]	C/ML	x	x	Binary or Text data
PacketTypes [23]	ML	x		Binary network packets
Melange [22]	ML	x	x	Binary network packets

Table 1. Comparison of Data and Message Description Languages

CONNECT is flexible to allow different types of language to be used to specify message formats; each language is termed an MDL. This flexibility better supports the parsing and composing of a wide range of protocols. For example, specialised languages for binary messages, text messages and XML messages can be utilised. To illustrate the approach we present a language for binary messages, and then a language for text messages. It is important to identify here that the role of these languages is to extract the information into a representation that is usable within CONNECT; the languages themselves do not seek to understand the content of the message, nor are they concerned with the application semantics of the message. Take for example an RPC request message invoking an operation Foo, these languages can extract the value 'Foo' for the label 'operation' but cannot determine its purpose.

Binary MDL For conciseness we consider one protocol, the Internet Inter-ORB Protocol (IIOP). This example also serves to illustrate in general how communication with any protocol can be achieved from a high-level specification of the message format. Figure 9 shows the specified message format of the IIOP protocol, which is a General Inter-ORB Protocol (GIOP) message as identified

by ⁸ transported over a TCP connection. In this specification there are three important constructs that are employed to describe the general outline of the messages for one protocol:

- **<Types>** list the types of each individual field type, e.g., the **VersionMajor** field type is an integer value. Types are separated from the message specification in order for field types to be reusable across multiple messages.
- **<Header>** includes the message format of the header for the binary protocol messages. If a header specification is present this is common to every message in the protocol (only one Header can be defined). In this GIOP message both messages: **GIOPRequest** and **GIOPReply** have the defined header **GIOP**.
- **<Message>** describes the packet format for the body of a particular message. Each protocol will typically contain multiple message bodies, for example the IIOP protocol here contains message bodies for two GIOP messages: a GIOP request message, and a GIOP reply message.

```

<Types>
<Protocol:String [GIOP]><VersionMajor:Integer [1]>
<VersionMinor:Integer [2]><Reserved:null>
<Frag:Boolean><Endian:Boolean [f-Endian]>
<MessageType:Integer [f-MsgType]><RequestID:Integer [f-UniqueID]>
<MessageLength:Integer [f-MsgLength]><Response:Boolean [true]>
<ObjectKeyLength:Integer><ObjectKey:Octets>
<ParameterArray:CORBAParameters>
\textbf{<EndTypes>}

<Header:GIOP>
<Protocol:32><VersionMajor:8><VersionMinor:8>
<Reserved:8><MessageType:8><MessageLength:32>
<End:Header>

<Message:GIOPRequest>
<Rule:MessageType=0>
<RequestID:32><Response:8><Reserved:24>
<TargetAddress:32><ObjectKeyLength:32>
<ObjectKey:ObjectKeyLength><align:32>
<OperationLength:32><Operation:OperationLength>
<align:32><ContextListLength:32>
<ServiceContext:ContextListLength><align:64>
<ParameterArray:eof>
<End:Message>

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

```

Fig. 9. Partial view of the GIOP message description

Hence, **<Header>** and **<Message>** specify the content of the message headers and bodies. The information specified within these then describes the fine-

⁸ <http://www.omg.org/spec/CORBAe/20080201/GIOP.idl>

grained field content. To do this, both headers and bodies are composed of `<label:size>` entries for each field in the message. The *size* is the length of the field content in bits. There is one special label: `<rule:field=value>`; this is used to relate the correct message body with the header. For example, the GIOP `GIOPRequest` message applies when the value of the `MessageType` field in the header equals zero.

Other interesting features of the `<Types>` specifications are functions and constant values. Functions can be defined on types using the `[f-method()]` construct, e.g., `[f-MsgLength]` in Figure 9 is a built in function to return the length of the composed message. They are generally useful for calculating values that must be composed when creating a message (rather than parsing), i.e., the named f-method is executed by the marshaller to get the value that must be written. Similarly, constants are values that can be composed directly by the marshaller with the given value, e.g., `<Protocol:String[GIOP]>` states the the `Protocol` field is always the value 'GIOP'.

Text MDL Text based protocols are different from binary protocols and therefore, a new MDL is required to generate the Listeners and Actuators. We again use one example specification to highlight the features of the Text MDL; a subset of the messages within SSDP is specified in Figure 10. Like the binary approach there is a list of field labels with their corresponding types in the `<Types>` section and again `<Header>` and `<Body>` are used to describe the individual messages. The key difference in this language is that we utilise field delimiters rather than bit lengths to distinguish the length of the fields. For example in the `<Header>`, `<Method:32>` means that the field is terminated by the '32' ASCII character, i.e., a space. In the case where multiple characters are used to delimit we employ commas to list the character values e.g. `<Version:13,10>` is a backslash r followed by a backslash n.

Another important feature of text protocols is that they are typically self-describing, i.e., the field label as well as the value will form the content of the message. For example, a HTTP message may contain "Host:www.lancs.ac.uk"; this defines a field with a label Host and a value www.lancs.ac.uk. Hence, text protocols are not rigidly defined in terms of the fields and their order. To support this property we employ the `<Fields: >` construct; this will parse/compose a list of free form self-describing fields into their label, size and values. For example, `<Fields:13,10:58>` splits fields using the 13,10 delimiter, then it uses the 58 value (a colon) to split the field into its label and value. The label must relate to a type specified in the `<Types>` section.

4 CONNECT in Action

To demonstrate the potential of the CONNECT architecture we consider a single case within a distributed marketplace scenario. Consider a stadium where vendors are selling products such as popcorn, hot dogs, beer and memorabilia, and consumers can search for products and place an order with a vendor. Both

```

<Types>
<Method:String>
  <URI:String>
  <HTTP_Version:String>
  <MX:Integer>
  <MAN:String>
  ...
<EndTypes>

<Header:SSDP>
  <Method:32>
  <URI:32>
  <Version:13,10>
  <Fields:13,10:58>
<End:Header>

<Message:SSDP_Search>
  <Rule:Method=M-SEARCH>
<End:Message>

<Message:SSDP_Response>
  <Rule:Method=HTTP/1.1>
<End:Message>

<Message:SSDP_Notify>
  <Rule:Method=NOTIFY>
<End:Message>

```

Fig. 10. Partial SSDP Message Description

merchants and consumers use mobile devices with wireless networks deployed in the stadium. Merchants publish product info which the consumers can browse through. When a consumer requests a product, the merchant gets a notification of the amount ordered and the location of the consumer, to which he can respond with a yes/no. Given the scale of the scenario there are many potential interoperability issues (e.g. due to the unpredictable application and middleware technologies employed by both vendors consumers), hence we look at just one particular vendor and consumer case:

The client consumer application uses UPnP to perform lookup requests for nearby vendors, and then a message-based communication protocol (in this case SOAP) to interact with the found vendor, while the service merchant advertises their services using SLP and then employs an RPC-based protocol for communication with client (in this case CORBA, more specifically the IIOP protocol).

We apply the CONNECT architecture to build a CONNECTOR that allows the consumer to interact with the vendor in the face of this heterogeneity.

4.1 Phase 1: Discovery

The discovery enabler first monitors the running systems, and receives the UPnP lookup requests that describe the consumer application's requirements. It also receives the notification messages from the vendor in SLP that advertise the provided interface. The two plug-ins for the discovery enabler (SLP and UPnP plug-ins) listen on the appropriate multicast addresses: 239.255.255.253 port 427

for SLP, and 239.255.255.250 port 1900 for UPnP. These plug-ins then transform the content of the messages into both the affordance and interface descriptions (WSDL specifications) of the two networked systems as per the requirements of the Networked System Model. From this, the initial matchmaking is performed and given the similarity of application and operations provided— the two systems are determined to match, and it is now the objective to build a CONNECTOR that will allow the two to interact. A partial view of the two WSDL descriptions is shown in Figure 11. It is important to observe here that the two share the same data schema and thus we don't investigate here how CONNECT resolves data heterogeneity problem.

<pre> <portType name="MarketPlace"> <operation name="getInfo"> <output message="getInfoRequest"/> <input message=" getInfoResponse "/> </operation> <operation name="buyProduct"> <output message=" buyProductRequest"/> <input message=" buyProductRepsonse"/> </operation> </portType> </pre>	<pre> <portType name="MarketPlace"> <operation name="getPrice"> <input message=" getPriceRequest"/> <output message=" getPriceResponse "/> </operation> <operation name="getUnsold"> <input message=" getUnsoldRequest"/> <output message=" getUnsoldResponse "/> </operation> <operation name="buyProduct"> <input message=" buyProductRequest"/> <output message=" buyProductRepsonse"/> </operation> </portType> </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fig. 11. WSDL interfaces for consumer (left) and vendor (right) marketplace applications

The discovery process also determines how these abstract operations are bound to concrete middleware protocols. In the consumer case they are bound to the SOAP asynchronous message protocol; here each of the messages that form an operation are sent asynchronously, point to point between the peers, e.g., the `buyProductRequest` of the `buyProduct` is sent as a SOAP message to the vendor. In the vendor case, the abstract operations are bound to the IIOP synchronous RPC protocol; here, the two messages that form the input and output of the operation are synchronously sent on the same transport connection, e.g., the `getPriceRequest` is received by the vendor who responds synchronously with the `getPriceResponse`.

4.2 Phase 2: Learning

The WSDL of the client and vendor in Figure 11 illustrate the heterogeneity of the two interfaces; they offer the same functionality, but do so with different behaviour. The next step in the CONNECT architecture is to learn the behaviours of these two systems. The learning enabler receives the WSDL documents from the discovery enabler and then interacts with deployed instance of the CORBA

vendor application in order to create the behaviour models for both the consumer and the vendor in this case. These are produced as LTS models and are illustrated for the SOAP consumer in Figure 12 and for the IIOP vendor in Figure 13. Here we can see that a vendor and consumer behaviour differs due to the heterogeneity of operations available from the interfaces. At this point we now have a completed Networked System Model (a description of the interface and behaviour of the system) for each of the two networked systems and can proceed to enable their interoperation.

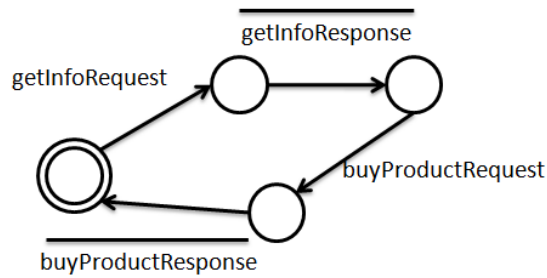


Fig. 12. LTS describing the behaviour of the SOAP consumer application

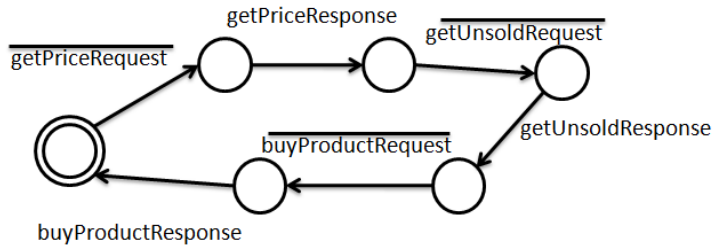


Fig. 13. LTS describing the behaviour of the IIOP vendor application

4.3 Phase 3: Synthesis of a mediator

The final step in the CONNECT process is to create the CONNECTOR that will mediate between the consumer’s request and the merchant’s response. To complete this the two LTS models are passed to the synthesis enabler. This performs two tasks:

- *Behaviour matching.* An ontology is provided for the domain that states where sequences of operations are equivalent, e.g., that the `getInfo` operation of the consumer and the `getPrice` combined with `getUnsold` of the

- vendor are equivalent. Further information about how the ontology-based behavioural matching is given in [4] [28].
- *Model synthesis.* The enabler produces an LTS that will mediate between the two systems; this LTS is shown in 14. Here you can see how the interoperation is co-ordinated; the application differences and middleware differences are resolved as the mediator executes through each of the states of the LTS. Note, the transitions correspond to a message sent via a particular middleware protocol, i.e., either SOAP or IIOP as indicated by the dashed line here.

A CONNECTOR is then realised by using a model to code transformation to generate an executable mediator that can be deployed in the network between the two networked systems. The mediator for the SOAP and IIOP applications is seen in 14. Here, the protocol messages are sent or received as per the protocol specification (the dotted line indicates that these are SOAP message, while the complete line indicates it is the IIOP protocol). Hence, the use of appropriate listeners and actuators (specific to the protocol) as described in Section 2.4 overcomes the problem of middleware heterogeneity, whereas the mediated sequence of application messages overcomes the application heterogeneity.

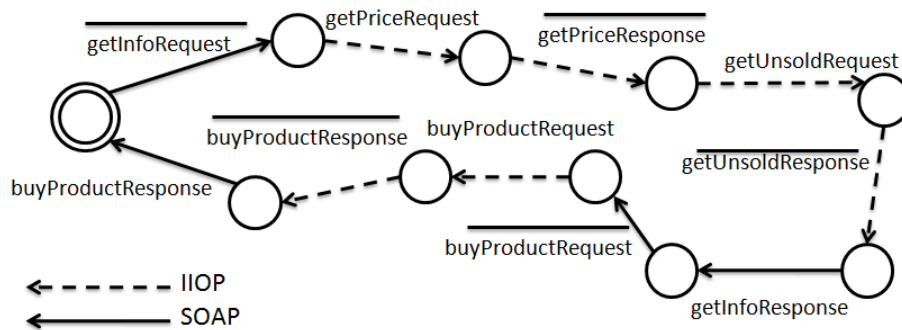


Fig. 14. LTS describing the mediated interaction between the two systems

5 Conclusions and Future Perspectives

5.1 Concluding Remarks

The overall aim of CONNECT is to bridge the interoperability gap that results from the use of different data and protocols by the different entities involved in the software stack such as applications, middleware, platforms, etc. This aim is particularly targeted at heterogeneous, dynamic environments where systems must interact spontaneously, i.e., they only discover each other at runtime. This

chapter has presented the CONNECT architecture to meet this particular objective; here we have seen how software enablers co-ordinate to create CONNECTORS that translate between the heterogeneous legacy protocols.

This chapter has examined the problem of communicating with legacy protocols in further detail, and has shown how domain specific languages that describe message formats (MDLs) can be used to generate the required middleware dynamically. These listeners and actuators can receive and send messages that correspond to the protocol specification and therefore are able to address the heterogeneity of middleware protocols. Subsequently the generated software mediators that co-ordinate the operation of listeners and actuators are able to handle the variations in application operations (as shown in Section 4). For deeper insight into how these mediators are specified and created, the interested reader is pointed to following:

- [19] offers a comprehensive description of how CONNECT leverages active learning to determine the behaviour of a networked system.
- [4] examines the generation of CONNECTORS in greater detail, illustrating the matching and mapping of networked system models and also describing the code generation techniques utilised.

We now discuss interesting directions for future research. Some of these are being actively pursued within the CONNECT project, whereas some are more general areas of research that can add to the understanding of interoperability solutions.

5.2 Future Research Direction: Advanced Learning of Middleware Protocols

In terms of *advanced learning*, we envisage further investigation of the role learning occupies within the architecture. At present, learning is focused solely on the behaviour model from the networked system model; that is, it aims to identify the application behaviour of a system. While this is important to the automation of CONNECTORS, it only focuses on part of the behaviour. At present, the middleware protocol behaviour and their corresponding message formats must be defined (and be known by CONNECT) in advance. If a new system employs a novel protocol then CONNECT is unable to resolve the interoperability, hence the approach is not future proof. Rather it is required that we equally apply learning approaches at the middleware level; this would not be executed as frequently (e.g. within the flow of the CONNECT process) because a new protocol need only be learned once. There has been interesting work in the learning of message formats and communication protocol sequences for the purpose of network security, examples include: Polyglot [9], Tupni [11], and Autoformat [21] which employ binary analysis techniques to extract information about the protocols by observing the binary executables in action; these have the potential to form the basis of the learning the MDL specifications automatically. However, they remain limited to understanding the content of a message, what it does

and what the purpose of the individual fields are—they can only deduce the field boundaries; hence, further research could look at the automated understanding of protocol content (which is potentially very important to understand if two protocols are compatible for interoperation).

5.3 Future Research Direction: The Role of Ontologies in Interoperability Frameworks

While only briefly discussed here, ontologies have an important role in the CONNECT architecture. Ontologies have been successfully employed within Web 2.0 applications, however these have only really considered the top level concerns such as discovering semantically similar systems. CONNECT is pushing the role of ontologies further, and is investigating going deep with the use of ontologies, i.e., using them at both the middleware and application level. Hence, to achieve better interoperability solutions ontologies cross-cut all of the CONNECT functions and enablers. This work is in the initial stages: this chapter has introduced the role of ontologies in the discovery, matching, and synthesis of CONNECTORS rather than explain the methods; here, ontologies feature in the networked model and are employed in discovery and matching of affordances and descriptions, while matching of systems (including alignment based upon ontologies) leads to the synthesis of CONNECTORS. In this domain an exciting area of future work is the application of ontologies to the lowest level of the CONNECT architecture, i.e. the interoperation between middleware protocols; ontologies can be applied to classify (discover the behaviour of) new network protocols and then use this to determine the low-level interoperability bridges (i.e., the matching and mapping of data field content between protocol messages, for example the matching of the 'methodName' field in XML-RPC with the operation field in IIOP and the subsequent translation of the data between the two—one is a null terminated string, the other isn't).

5.4 Future Research Direction: Interoperability considering Non-functional requirements

Networked systems also have *non-functional properties* and requirements which must be considered in order to ensure correct interoperation between networked systems. Future work should place equal importance on these requirements. To underpin this, this will first involve extracting the non-functional requirements from networked systems and adding these to the interface description in the Networked System Model. This will involve extending the discovery process to discover non-functional descriptions of the systems which are also published using discovery protocols. Finally, the CONNECTORS must maintain particular non-functional requirements, e.g., dependability, security and trust are important and diverse properties within networked systems that must be maintained by an interoperability solution (and are particularly important in pervasive environments). Future research in this direction must consider solutions to correctly

ensure that the interoperability solutions meets any of these domain requirements.

References

1. Douglas Adams. *The Hitchhiker's Guide To The Galaxy*. Pan Books, 1979.
2. Godmar Back. Datascript - a specification and scripting language for binary data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 66–77, London, UK, 2002. Springer-Verlag.
3. Antonia Bertolino, Antonello Calabro, Felicita Di Giandomenico, and Nicola Nostro. *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM-11:CONNECT)*, chapter Dependability and Performance Assessment of Dynamic CONNECTed Systems. LNCS series, 2011.
4. Antonia Bertolino, Paola Inverardi, Valérie Issarny, Antonino Sabetta, and Romina Spalazzese. On-the-fly interoperability through automated mediator synthesis and monitoring. In *ISoLA (2)*, pages 251–262, 2010.
5. Gordon Blair, Massimo Paolucci, Paul Grace, and Nikolaos Georgantas. *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM-11:CONNECT)*, chapter Interoperability in Complex Distributed Systems. LNCS series, 2011.
6. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. In <http://www.w3.org/TR/sawSDL/>. W3C, February 2004.
7. Michael Brodie. The long and winding road to industrial strength semantic web services. In *Proceedings of the 2nd International Semantic Web Conference (ISWC2003)*, October 2003.
8. Yérom-David Bromberg and Valérie Issarny. INDISS: Interoperable discovery system for networked services. In *Middleware*, pages 164–183, 2005.
9. Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.
10. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. In <http://www.w3.org/TR/wsdl>, March 2001.
11. Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 391–402, New York, NY, USA, 2008. ACM.
12. Joel Farrell and Holger Lausen. Semantic annotations for wsdl and xml schema. In <http://www.w3.org/TR/sawSDL/>, August 2007.
13. Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 2–15, New York, NY, USA, 2006. ACM.
14. Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *Proceedings of the*

- 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '08, pages 421–434, New York, NY, USA, 2008. ACM.
15. UPnP Forum. Upnp device architecture version 1.0. <http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf>, October 2008.
 16. P. Grace, G. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(1):2–14, January 2005.
 17. M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. Frystyk Nielsen, A. Kar-markar, and Y. Lafon. Soap version 1.2 part 1: Messaging framework. In <http://www.w3.org/TR/soap12-part1>, April 2001.
 18. E. Guttman, C. Perkins, and J. Veizades. Service location protocol version 2, IETF RFC 2608. <http://www.ietf.org/rfc/rfc2608.txt>, June 1999.
 19. Falk Howar, Bengt Jonsson, Maik Merten, Bernhard Steffen, and Sofia Cassel. On handling data in automata learning - considerations from the connect perspective. In *ISoLA (2)*, pages 221–235, 2010.
 20. Valeerie Issarny, Amel Bennaceur, and Yerom-David Bromberg. *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM-11:CONNECT)*, chapter Middleware-layer CONNECTOR Synthesis. LNCS series, 2011.
 21. Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through connect-aware monitored execution. In *In 15th Symposium on Network and Distributed System Security (NDSS)*, 2008.
 22. Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. Melange: creating a "functional" internet. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 101–114, New York, NY, USA, 2007. ACM.
 23. Peter J. McCann and Satish Chandra. Packet types: abstract specification of network protocol messages. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA, 2000. ACM.
 24. Jin Nakazawa, Hideyuki Tokuda, W. Keith Edwards, and Umakishore Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, Washington, DC, USA, 2006. IEEE Computer Society.
 25. OASIS. Web services dynamic discovery (wsdiscovery) version 1.1. <http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf>, July 2009.
 26. OMG. The common object request broker: Architecture and specification version 2.0. Technical report, Object Management Group, 1995.
 27. Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 289–300, New York, NY, USA, 2006. ACM.
 28. Romina Spalazzese Paola Inverardi and and Massimo Tivoli. *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Connectors for Eternal Networked Software Systems (SFM-11:CONNECT)*, chapter Application-layer CONNECTOR Synthesis. LNCS series, 2011.

29. H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In *FASE*, pages 377–380, 2006.
30. Manuel Roman, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2:–, May 2001.
31. Duke Tantiprasut, John Neil, and Craig Farrell. Asn.1 protocol specification for use with arbitrary encoding schemes. *IEEE/ACM Trans. Netw.*, 5:502–513, August 1997.