INMIDIO Software Developer's Guide

Table of Contents

1 Overview

Introduction

The role of the INteroperable MIddleware for service Discovery and service InteractiOn (INMIDIO) is to identify the discovery and interaction middleware protocols that execute on the network and to translate the incoming/outgoing messages of one protocol into messages of another, target protocol. The system parses the incoming/outgoing message and, after having interpreted the semantics of the message, it generates a list of semantic events and uses this list to reconstruct a message for the target protocol, matching the semantics of the original message. The INMIDIO middleware acts in a transparent way with regard to discovery and interaction middleware protocols and with regard to services running on top of them. The supported service discovery protocols are UPnP, SLP and WS-Discovery, while the supported service interaction protocols are SOAP and RMI.

Intended audience

System developers that seek to integrate heterogeneous middleware platforms and their supported service-oriented architectures inside dynamic environments.

License

INMIDIO is available under the LGPL license terms.

Language

C

Environment (set-up) info needed if you want to run this sw (service)

INMIDIO requires running a web server on the machine.

Platform

Linux

2 Deployment

2.1 System requirements

Operating System: Linux

2.2 Download

Source code files and executable file are currently available either on the AMIGO GForge site[1] under the *mdwcore/sdi_sii* structure (for Subversion users) or at the following web page: http://www-rocq.inria.fr/arles/download/inmidio/index.html.

2.3 Install

Unzip the downloaded file in a directory `$DIR`.

Running the middleware requires a web server installed on the machine. If it is not already installed, your can download and install Jakarta Tomcat[2].

Copy `$DIR/ib/libnanohttp.so` to `/usr/lib`.

2.4 Compile

To compile the middleware, from directory $DIR, execute the following commands:

./configure

make

The executable will be created in `$DIR/amigo_monitor`

2.5 Run

To execute the middleware, run

```
$DIR/amigo_monitor $webserver_dir
```

where `$webserver_dir` is the directory where the web server used by the middleware is installed.

[1] http://gforge.inria.fr/projects/amigo/

[2] http://tomcat.apache.org/

3 Component Architecture

3.1 Component interface

The INMIDIO middleware provides discovery and interaction functionalities relying/through the same interface provided/exported/offered to applications and services by the discovery and interaction protocols specifications. More details about the specifications and the functionalities the middleware supports for each protocol are provided in Section 3.4.

3.2 Mechanisms of interaction

The INMIDIO middleware provides a protocol translation process that converts messages from one protocol to another in a transparent way for client and service applications and that consists of: (i) a first translation from one *Service Discovery Protocol* (SDP) to another SDP, (ii) the creation of some files that are required by the specification of the protocol the client is based on and finally, (iii) the translation of the method calls from one *Service Interaction Protocol* (SIP) to another SIP.

For each protocol supported, the INMIDIO middleware provides an entity that we call *unit* that implements and executes the specification of a corresponding protocol in order to realize a correct protocol translation. These units are internally connected and coordinated by a component called *Monitor*. The communication among the different SD and SI protocol units in execution inside the system is through the use of internal *events*.

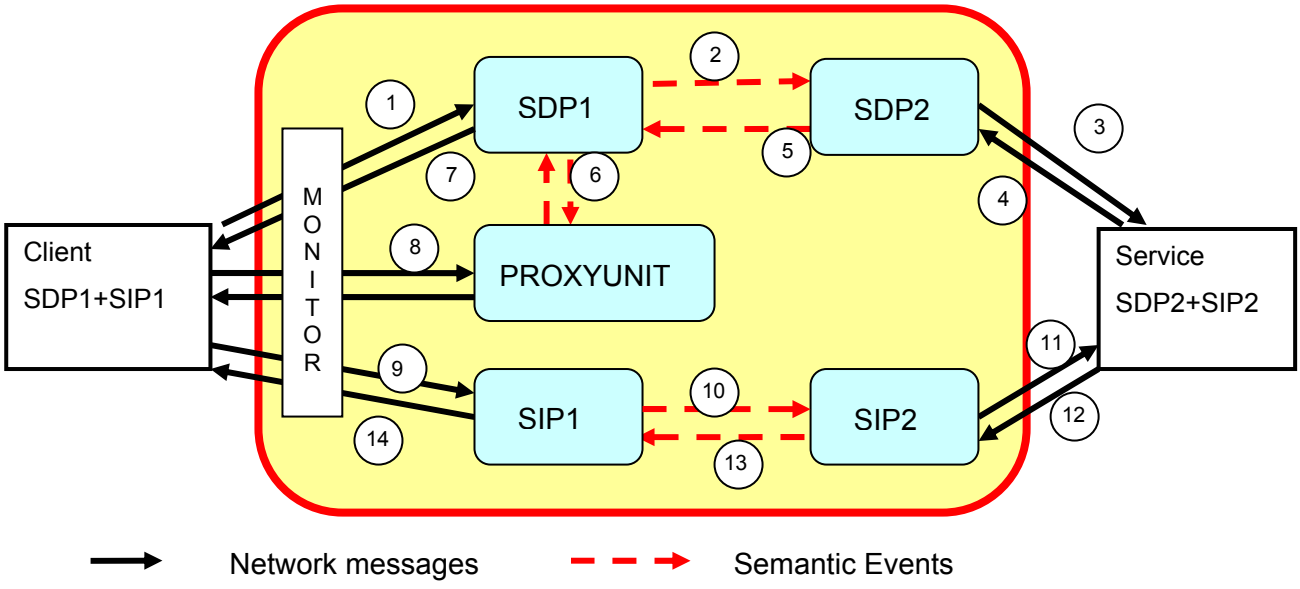

Figure 3-1: INMIDIO middleware – protocol translation process

Figure 3-1 represents the complete translation process performed by the middleware and starting with a discovery message sent by the client to discover a service and concluding with a reply to a method call from the service to the client. The example shows a client and a service that are respectively based on the discovery protocols SDP1 and SDP2 and the interaction protocols SIP1 and SIP2:

- ***Discovery: steps 1-5***. The SDP1 discovery message is received by the Monitor component of the Middleware and the SDP1 and SDP2 units translate the message into an SDP2 discovery message. The SDP2 response message from the service is translated by SDP2 unit into a series of events for SDP1.
- ***Proxy Generation: steps 6-8***. The service's description and reference are obtained from the service discovery steps 3 and 4 and are used in step 6 by a *PROXY unit* in order to generate some intermediary information that will be used by the client to invoke the service.

  The information generated consists in instantiating the appropriate files and component, in a compatible format with client's required interface with regard to the protocols SDP1 and SIP1 it is based on. In practical terms it could be the stub used by the client to serialize and deserialize the method calls for the service or an XML/WSDL file describing the interface and the reference of the service.

  In step 8 the discovery response is generated by SDP1 using the data received in the form of events from PROXY unit and from SDP2 unit. The response message is finally returned to the client.

  In step 9 the client gets access to the files about the discovered service and generated by the PROXY unit. These files are used by the client to prepare the following phase, that is, the interaction with the service.
- ***Interaction: steps 9-14***. The client may therefore invoke service operations. Service method call SIP1 is received by the Monitor component of the Middleware and the SIP1 and SIP2 units translate the message into an SIP2 service method call. The SIP2 response message from the service is translated by SIP2 unit into a series of events for unit SIP1 that generates a response message for the client.

3.3 Overview and reference to internals

In the following subsections, we provide an overview (3.3.1) and a detailed description of the source code 3.3.2) of the internal mechanisms that implement the interoperability process and the interactions between the different units and components in order to support the functionality offered by the middleware.

3.3.1 Overview

Each protocol supported by the middleware is associated to a specific address and port defined by protocol specification in the case of service discovery protocols and assigned at startup by the middleware for service interaction protocols. The monitor component is able to determine the current SDP(s) that is (are) used in the environment upon the arrival of the data at the monitored ports without doing any computation, data interpretation or data transformation. The detection is not based on the data content but on the data arrival at the specified UDP/TCP ports inside the corresponding multicast or unicast address.

In order to implement the complex distributed process required to support the function related to each specific protocol, we introduce the concept of *unit.* A unit is a self-configurable container of all various components (*parser*, *composer* and *socket*) that runs the coordination process required to implement the tasks associated to each SDP or SIP. The behavior of a unit is specified using a finite state machine. All the messages received by the monitor are delivered to the *unit* that corresponds to the couple address and port on which the message has been received. Then, the unit is in charge of the delivery to the appropriate *parser*. The parser successfully transforms the raw data flow into a series of events that represent the semantic concepts associated to the syntactic details of the SDP received message. Then, the

generated events are delivered to the local components' *composers*. The communication between the parser and the composer does not depend on any syntactic detail of any protocol. They communicate at semantic level through the use of events. Parsers and composers are dedicated to a specific protocol and the middleware embeds several parsers and composers to support different protocols. Parsers and composers are further decoupled from the transport protocol used for the receipt/sending of messages by enabling various types of *socket* components, which may further be changed at runtime. The unit is in charge of dispatching event notifications to its registered listeners through event connectors. Message-oriented connectors enable the interaction among components that are not event-oriented. Parsers are endowed with both event- and message-oriented connectors. Thus, inside the units, parsers' input ports are bound to message-oriented connectors, whereas parsers' output ports are bound to an event connector controlled through the unit's state machine. Conversely, composers' output ports are bound to message-oriented connectors, whereas composers' input ports are bound to the unit's event bus (see Figure 3-2).

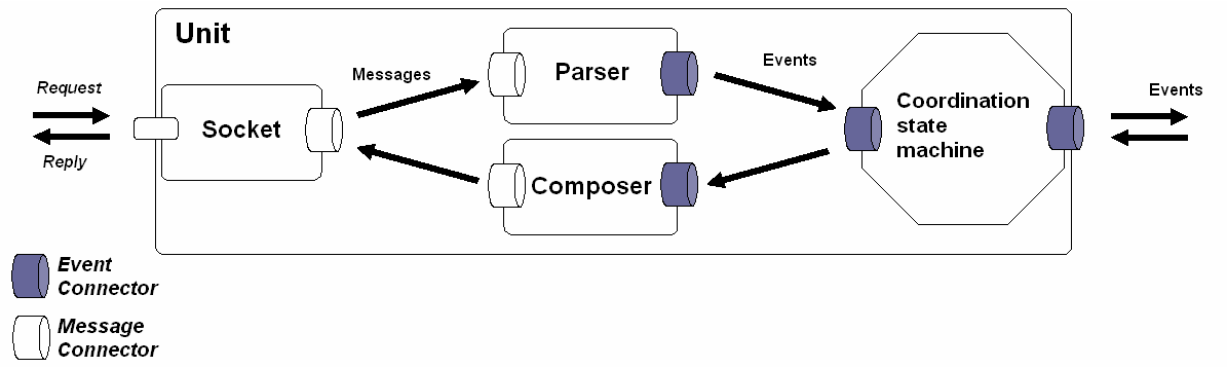

Figure 3-2: Unit configuration

Protocol interoperability is the result of the correct composition of a number of units. In the example presented in *Figure 3-3* and relative to SDPs, the translation from SLP to UPnP discovery corresponds to the composition of an SLP unit with a UPnP unit. At this level, units are only considered as computational elements that transform messages into events and vice versa. The units' internal mechanisms are totally hidden. A similar schema for interoperability is applied to service communication protocols.

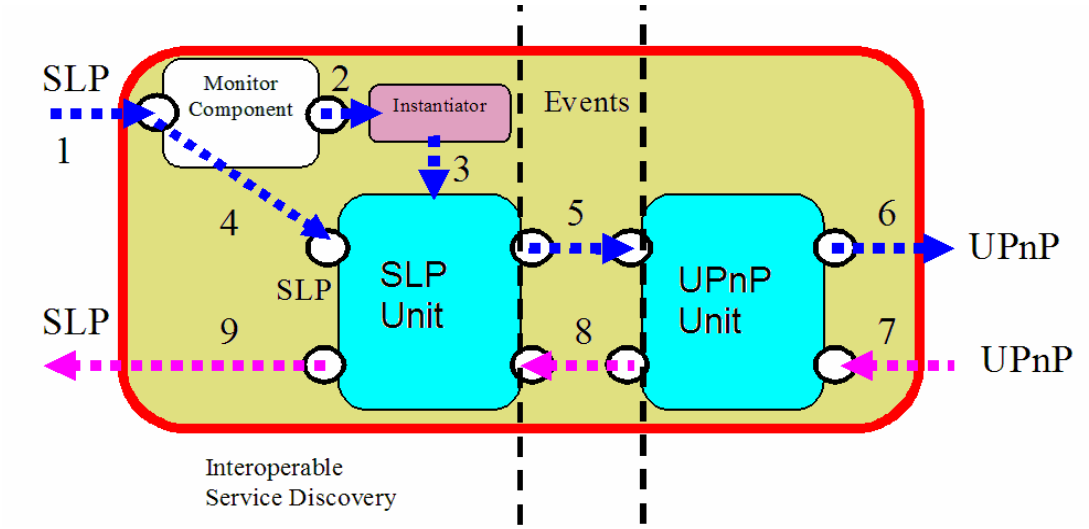

Figure 3-3: SDP interoperability mechanisms

At start-up, the monitor initializes the units supported by the INMIDIO middleware in order to listen to the network for incoming messages and sets up the connections among the different units supported.

The SDP units will be connected together in order to translate discovery messages and will be associated to the corresponding proxy unit.

The SIP are set up by the monitor at startup at the assigned ports and the corresponding sockets are enabled in listening mode in order to receive service call messages from clients on the network.

3.3.2 Reference to internals

Figure 3-4 shows the structure of the source code of the INMIDIO middleware. Each type of component of the system (e.g., unit, parser, composer, socket) is defined by a subdirectory of the `include` directory and implemented in a subdirectory of the `src` directory.

The `samples` directory contains some examples of clients and services based on the protocols supported by the middleware (for more details, see the tutorial chapter in the user guide).

Below, for each component, we describe the details of the data structures used to define it and the main functions used for its implementation.

```
src
src/unit
src/socket
src/protocol
src/parser
src/msg
src/monitor
src/io
src/event
src/engine
src/composer
src/ser
src/sha
src/vm
src/kazlib

include
include/composer
include/engine
include/event
include/io
include/monitor
include/msg
include/parser
include/protocol
include/socket
include/unit
include/ser
include/sha
include/vm
include/kazlib
include/nanohttp

samples
samples/clients
samples/lib
samples/res
samples/services
```

Figure 3-4: directory structure

3.3.2.1 Monitor

When the system is initialized, it is the responsibility of the *Monitor* to set up the correct configuration of unit composition to achieve service discovery and interaction interoperability.

For the discovery, this consists in setting the connection between the right sockets (listening for incoming network messages at the specified multicast addresses and ports) to the associated SD protocol units. For the interaction, this consists in setting the connection between the right sockets (listening for incoming network messages at the local network unicast address and port assigned by the monitor) to the associated SI protocol units.

The initialization of monitor is implemented in `src/monitor/monitor.c` using the data structures defined in `include/monitor/monitor.h`.

3.3.2.2 Event

All the events treated inside the system are represented by the structure `EVENT_PTR` that is used for all kind of events independently of the type of information transmitted. The structure `EVENT_PTR` is defined in `include/event/event.h` and contains the type, the associated value, a stream identifier (`stream_id`) and the source of the event. The functions to handle the creation and the properties of the event are implemented in `src/event/event.c`.

This definition of `EVENT_PTR` allows to publish and deliver events between the different components transparently, without being aware of the content of the event and to move the handling of the event exclusively inside the component that receives the event.

All the possible values supported for the type of an event are defined in `include/event/event.h` and are listed *Figure 3-5*. An event can carry different types of values: numbers, strings, void pointer. The type of this value depends on the definition of event type. For example if the event type is *`SDP_REQ_SERVICE_TYPE`*, the content of the value will be a string containing the service type identifier.

The identifier `stream_id` defines a flow of events related to the same discovery message translation process and all the related network messages (including the response messages form services) will generate events with the same `stream_id`. The engine will handle all the events received by the unit taking into account this identifier as a filter to identify the different flow of events related to the different running discovery process running (each one associated to a different status of the state machine. Ffor more details, see the subsection about the state machine engine §3.3.2.8).

The source of the event contains two fields: a reference to the source unit (required to avoid loops between units in event delivery when and event is published) and the reference to the thread of the parser that generated the event (`src_thread_parser`). The `src_thread_parser` is used in case of multiple replies from different services matching the service type in the request message. `src_thread_parser` will be used as a filter between the different flows of events generated by the different parsers (each one parsing a different reply): as the events generated by these parsers will have the same `stream_id` (as just explained above), the `src_thread_parser` will be used to separate the events from different service response messages.

As the internal communication inside each unit and between different units is based on events, each component that wants to communicate through events must provide some event connector functions. If the component needs to publish events, it will implement a function to create the list of its event listener components and a function to publish an event on all its defined listeners. On the other hand, if the component needs to receive events because it is an event listener it will implement a function to receive events. For each component (parser, composer, unit and socket) the functions implemented for event handling are defined in the specific subsections.

```
SDP_REQ_LANGAGE
SDP_REQ_SCOPE
SDP_REQ_PREDICAT
SDP_REQ_REQUESTER
SDP_REQ_ID
SDP_REQ_DELAY
SDP_REQ_MULTICAST
SDP_REQ_UNICAST
SDP_REQ_SERVICE_TYPE


SDP_UPNP
SDP_JINI
SDP_SLP
SDP_WSD
SCP_SOAP
SCP_RMI

SDP_START
SDP_STOP

SDP_VERSION
SDP_ERROR
SDP_SOURCE_ADDR
SDP_SOURCE_PORT

SDP_SERVICE_ID

SDP_SERVICE_REQUEST
SDP_SERVICE_RESPONSE
SDP_SERVICE_ALIVE
SDP_SERVICE_BYEBYE

SDP_SERVICE_LIFETIME
SDP_RES_OK
SDP_RES_ERR
SDP_RES_ATTR
SDP_RES_SOURCE_ADR
SDP_RES_SERVICE_TYPE
SDP_SEND_SERVICE_REQUEST
SDP_SEND_SERVICE_REPLY
```

```
SDP_SERVICE_DESCRIPTION
SDP_SERVICE_METHOD_DESCRIPTION
SDP_SERVICE_DESCRIPTION_SERVICETYPE
SDP_SERVICE_DESCRIPTION_INTERFACETYPE
SDP_SERVICE_DESCRIPTION_INTERFACE_NAMESPACE
SDP_SERVICE_VAR_DESCRIPTION

GENERATE_PROXY

SDP_SERVICEPROXY_URL_CTRL

SDP_SERVICE_DESCR_URL_NOPATH
SDP_SERVICE_DESCR_URL_PATH
SDP_INTERFACE_DESCR_URL_PATH
SDP_SERVICE_CTRL_URL_NOPATH
SDP_INTERFACE_CTRL_URL_PATH

SCP_RMIOBJ_NUM
SCP_RMIOBJ_UNIQUE
SCP_RMIOBJ_TIME
SCP_RMIOBJ_COUNT
SCP_RMIOBJ_IP
SCP_RMIOBJ_PORT
SCP_RMI_STUB_OBJ


SCP_REQUEST
SCP_RESPONSE
SCP_SERVICE_NAME
SCP_METHOD_NAME
SCP_METHOD_NAMESPACE
SCP_ARGUMENT_IN
SCP_ARGUMENT_OUT

SCP_SEND_SERVICE_REQUEST
SCP_SEND_SERVICE_REPLY
SCP_SOCKET_ID
```

Figure 3-5: List of event types supported

3.3.2.3 Message

All network messages are represented by the common structure `MSG_PTR` that is used for all types of messages independently of the network protocol (e.g., UDP, TCP, HTTP) they belong to. The structure `MSG_PTR` is defined in `include/msg/msg.h` and contains the source and destination address and port together with the length of the content and the data carried by the message. The functions to handle the creation and the properties of the message are implemented in `src/msg/msg.c`.

This definition of `MSG_PTR` allows parsers and composers to exchange messages with sockets transparently, without being aware of the concrete network protocol used for transmission and to move the specific network protocol management exclusively inside the socket component that delivers or receives the message.

3.3.2.4 Socket

Sockets are in charge of sending and receiving messages using a specific transport protocol. As we currently assume all-IP networks, we define the corresponding types of socket

components: multicast sockets and unicast sockets, where the latter may be either connection-oriented or connection-less. Socket components offer flexibility enabling the implementation of system components in a way that is independent of the underlying transport.

All the common functions required for the implementation of the different types of sockets supported by INMIDIO middleware are implemented in `src/socket/socket.c` and the data structures used are defined in `include/socket/socket.h`.

Sockets listen on the network at a fixed address (multicast or unicast) and port and this action is enabled on the socket using the function:

```
void socket_listen(SOCKET_PTR socket, char* addr, unsigned short port )
```

When a message is received from the network, the socket creates an instance of a structure representing a message (`MSG_PTR`) and this structure is published on all listeners of the socket using the function:

```
void socket_publish_msg(MSG_PTR msg, SOCKET_PTR socket)
```

Some functions are provided to add a component as a message listener of the socket:

```
int socket_add_msglistener(SOCKET_PTR socket, struct listener* evt_list)
int socket_add_msglistener_unit(SOCKET_PTR socket, struct unit* u)
int socket_add_msglistener_parser(SOCKET_PTR socket, struct parser* p)
int socket_add_msglistener_monitor(SOCKET_PTR socket, struct monitor* m)
```

On the other hand, sockets also have the role of sending messages on the network to a multicast or unicast address and port. This action is performed by the socket when the following function is called:

```
void socket_msg_received(MSG_PTR msg, LISTENER_PTR node)
```

Some other functions are provided to create and initialize the socket structure and to set the values of the socket:

```
SOCKET_PTR socket_create_without_unit()
SOCKET_PTR socket_create(UNIT_PTR u)
int socket_get_localport(SOCKET_PTR socket)
int socket_set_localport(SOCKET_PTR socket, int local_port)
```

As seen above, these are the common functions implemented to support all the sockets, but for each specific implementation of a protocol, the socket must provide a function that listens to the network to receive messages and a function that sends a message on the network to the specified address and port. The HTTP socket is implemented in `src/socket/HTTPsocket.c`, the UDP socket is implemented in `src/socket/UDPsocket.c`, the UDP multicast socket is implemented in `src/socket/UDPMCsocket.c` and the TCP socket is implemented in `src/socket/TCPsocket.c`.

3.3.2.5 Parser

The role of a parser component is to wait for messages, parse their content and generate a sequence of semantic events in conformance with the implemented protocol specification. Parsers are decoupled from the transport protocol by means of socket components, which may be changed at runtime.

All the common functions required for the implementation of the different types of parsers supported by the INMIDIO middleware are implemented in `src/parser/parser.c` and the data structures used are defined in `include/parser/parser.h`.

Network messages will be delivered to the parser through the function:

```
void parser_msg_received(MSG_PTR msg, LISTENER_PTR node)
```

that will execute the `parse` function defined for the specific parser that receives the message. The `void *parse(void* ptr)` function is the core of the specific parser and will generate events and publish them: they will be automatically delivered to the parser's event listeners using the function:

```
void parser_publish_event(EVENT_PTR evt, PARSER_PTR parser)
```

The parser's event listeners may be defined using the functions:

```
int parser_add_evtlistener(PARSER_PTR parser, struct listener* evt_list)
int parser_add_evtlistener_unit(PARSER_PTR parser, struct unit* u)
```

As the parsers are also event listeners, they can receive events and a function is defined for this purpose:

```
void parser_event_received(EVENT_PTR evt, LISTENER_PTR node)
```

A function is provided to create and initialize the parser structure:

```
PARSER_PTR parser_create(UNIT_PTR u)
```

These are the common functions required to support all the parsers, but for each specific protocol parser, a series of function must implement the specification of the protocol. For example, the SLP parser is implemented in `src/parser/SLPParser.c` and the SOAP parser is implemented in `src/parser/SOAPParser.c`.

3.3.2.6 Composer

The role of a composer is to generate well-formed messages in conformance with the specific protocol implemented and at the same time coherent with the semantic events received from the event notification mechanism.

All the common functions required for the implementation of the different types of composers supported by the INMIDIO middleware are implemented in `src/composer/ composer.c` and the data structures used are defined in `include/composer/composer.h`.

Events are received on the function

```
void composer_event_received(EVENT_PTR evt, LISTENER_PTR node)
```

and treated in the main thread that runs the specific composer algorithm to handle received events and implemented taking into account the specifications of the composer's protocol. In this main thread, the message to be sent on the network is prepared in conformance with the events received by the composer. When the message is ready to be sent, the function

```
void composer_publish_msg(MSG_PTR msg, COMPOSER_PTR composer)
```

is used to publish the message on the composer's message listener, that is, a socket that will take care of the action of sending the message on the network to the address and port obtained by events. The following functions are used to specify the relation of message publish/subscriber between the composer and the socket:

```
int composer_add_msglistener(COMPOSER_PTR compo,struct listener* evt_list)
int composer_add_msglistener_socket(COMPOSER_PTR composer, struct socket* s)
int composer_add_msglistener_unit(COMPOSER_PTR composer, struct unit* u)
```

As the composer is also a publisher of events, it implements the functions to publish an event and to add a listener to its list of event listeners:

```
void composer_publish_event(EVENT_PTR evt, COMPOSER_PTR composer)
int composer_add_evtlistener(COMPOSER_PTR composer, struct listener*
int composer_add_evtlistener_unit(COMPOSER_PTR composer, struct unit* u)
```

Finally, a function is provided to create and initialize the composer structure:

```
COMPOSER_PTR composer_create(UNIT_PTR u)
```

These are the common functions required to support all the composers, but for each specific protocol composer, a series of function must implement the specification of the protocol. For example, the SLP composer is implemented in `src/composer/SLPcomposer.c` and the SOAP composer is implemented in `src/composer/SOAPcomposer.c`.

3.3.2.7 Unit

A unit implements event-based interoperability for a specific protocol by translating messages of the specific protocol to and from semantic events associated with service discovery and interaction and by implementing coordination processes over the events according to the behaviour prescribed by the protocol specification. Units are composed and communicate through their event connectors, whereas they use their socket components to interact with components that are outside the protocol interoperability system. Within a unit, coordination and composition rules among embedded protocol components are specialized with respect to a given protocol according to the unit state.

All the common functions required for the implementation of the different types of units supported by the INMIDIO middleware are implemented in `src/unit/unit.c` and the data structures used are defined in `include/unit/unit.h`.

Each unit defines the list of composers, parsers and sockets supported in order to realize the specification of the SD or SI protocol and the functions implemented to to add one of these components are the following:

```
int unit_addsocket(UNIT_PTR unit, char* key, SOCKET_PTR s)
int unit_addparser(UNIT_PTR unit, char* key, PARSER_PTR p)
int unit_addcomposer(UNIT_PTR unit, char* key, COMPOSER_PTR c)
```

As the unit is an event publisher, it implements the functions to publish an event and to add a listener to its list of event listeners:

```
void unit_dispatch_evt_to_listeners(EVENT_PTR evt, void* u)
void unit_dispatch_evt_to_composers(EVENT_PTR evt, void* u)
void unit_dispatch_evt_to_units(EVENT_PTR evt, void* u)
void unit_dispatch_evt_to_unit_proxy(EVENT_PTR evt, void* u)

int unit_add_evtlistener(UNIT_PTR unit, LISTENER_PTR evt_list)
int unit_add_evtlistener_parser(UNIT_PTR unit, PARSER_PTR p)
int unit_add_evtlistener_unit(UNIT_PTR unit, UNIT_PTR u)
int unit_add_evtlistener_unit_proxy(UNIT_PTR unit, UNIT_PTR u)
```

and events are received through the function:

```
void unit_event_received(EVENT_PTR evt, LISTENER_PTR node)
```

as the unit is also a publisher of messages, it implements the functions to publish a message and to add a listener to its list of message listeners:

```
void unit_publish_msg(MSG_PTR msg, UNIT_PTR unit)

int unit_add_msglistener(UNIT_PTR unit, struct listener* evt_list)
int unit_add_msglistener_unit(UNIT_PTR unit, struct unit* u)
int unit_add_msglistener_parser(UNIT_PTR unit, struct parser* p)
int unit_add_msglistener_socket(UNIT_PTR unit, struct socket* s)
```

the function to publish a message is:

```
void unit_msg_received(MSG_PTR msg, LISTENER_PTR node)
```

Some other functions are provided to create and initialize the unit structure and to manage the components supported by the unit:

```
UNIT_PTR unit_create(char* sm_filename, ACTION_PTR extra_actions)

SOCKET_PTR unit_getsocket(UNIT_PTR unit, char* key)
PARSER_PTR unit_getparser(UNIT_PTR unit, char* key)
COMPOSER_PTR unit_getcomposer(UNIT_PTR unit, char* key)
```

Each SD, SI and Proxy unit defines a finite state machine that describes its behavior in a `.sm` file that implements the specification of the protocol supported by the unit in order to control all the components (composers, the parsers and the sockets) and coordinate the internal interaction between them.

The functions available for the `.sm` unit state machine are implemented in `src/unit/unit.c` and consist of utility functions to print a string and the received event:

```
void print_str(char* string)
void print_evt(EVENT_PTR evt)
```

some functions to set the current socket, parser and composer to be activated:

```
void set_socket_and_composer(void* u, char* socket_name, char* composer_name, EVENT_PTR evt)
void set_socket_and_parser(void* u, char* socket_name, char* parser_name, EVENT_PTR evt)
void set_socket_composer_and_parser(void* u,char* sock,char* compos,char* parser,EVENT_PTR evt)
void set_composerparser(void* u, char* composerparser_name, EVENT_PTR evt)
```

some functions to dispatch events to event listeners connected to the unit:

```
void unit_dispatch_evt_to_listeners(EVENT_PTR evt, void* u)
void unit_dispatch_evt_to_composers(EVENT_PTR evt, void* u)
void unit_dispatch_evt_to_units(EVENT_PTR evt, void* u)
void unit_dispatch_evt_to_unit_proxy(EVENT_PTR evt, void* u)
```

some functions to order to the unit's component to send a discovery or communication message to the client or to the service:

```
void unit_send_disc_request(EVENT_PTR evt, void* u)
void unit_send_disc_reply(EVENT_PTR evt, void* u)
void unit_send_comm_request(EVENT_PTR evt, void* u)
void unit_send_comm_reply(EVENT_PTR evt, void* u)
```

and finally a function to order to the unit's component to generate proxy files for the client:

```
void unit_create_proxy(EVENT_PTR evt, void* u)
```

3.3.2.8 State machine engine

The `src/engine` and `include/engine` directories contain the implementation of the engine that implements the unit state machine execution. For more details about the implementation of the state machine engine, see Section 5.1.1).

3.4 Detailed documentation

After the general description of the behavior of units, we see now the details of the different protocols supported by the INMIDIO middleware and the functionalities implemented by the respective units and their internal components. For each protocol supported by the middleware (SD protocol units: SLP, UPnP and WS-Discovery, SI protocol units: RMI and SOAP, Proxy units: UPnPProxy, RMIProxy and WSDProxy) we specify the functionalities supported for the protocol and the imposed constraints.

3.4.1 SLP

The *Service Location Protocol* (SLP) specifications (RFC2608)[3] provide a framework to allow networking applications to discover the existence, location, and configuration of networked services in enterprise networks. SLP specifications define the format for all the different types of discovery messages exchanged among clients and service and the official couple of multicast address and port reserved for SLP protocol: the UDP multicast address is 239.255.255.253 and the port is 427.

SLP requires an SLP daemon to run on the machine in *Service Agent* (SA) mode (see RFC2608) to handle the registration messages from services and the discovery messages from the clients. When the daemon receives a discovery message it will send back a response to the SLP client with the address/endpoint of registered service that matches the identifier.

SLP standard does neither define a language for the description of the service (service type and service interface) nor an interaction protocol or a specification for the communication between client and service: they are free to communicate with any network communication protocol. The INMIDIO Middleware supports only the RMI interaction protocol for SLP based clients and services (see INMIDIO Middleware User Guide for more details).

For the implementation of the SLP protocol, the middleware provides an SLP Unit that requires an `SLPparser` and an `SLPcomposer` respectively used to parse and to generate the SLP messages. As SLP is based on UDP, the socket components required are the `UDPSocket` and the `UDPMulticastSocket`. Figure 3-6 represents the state machine diagram for the SLP unit that coordinates these components. The SLP unit state machine does not support replies from multiple services: the SLP reply message received by `SLPparser` must contain only one URL entry.

[3] http://www.openslp.org/doc/rfc/rfc2608.txt

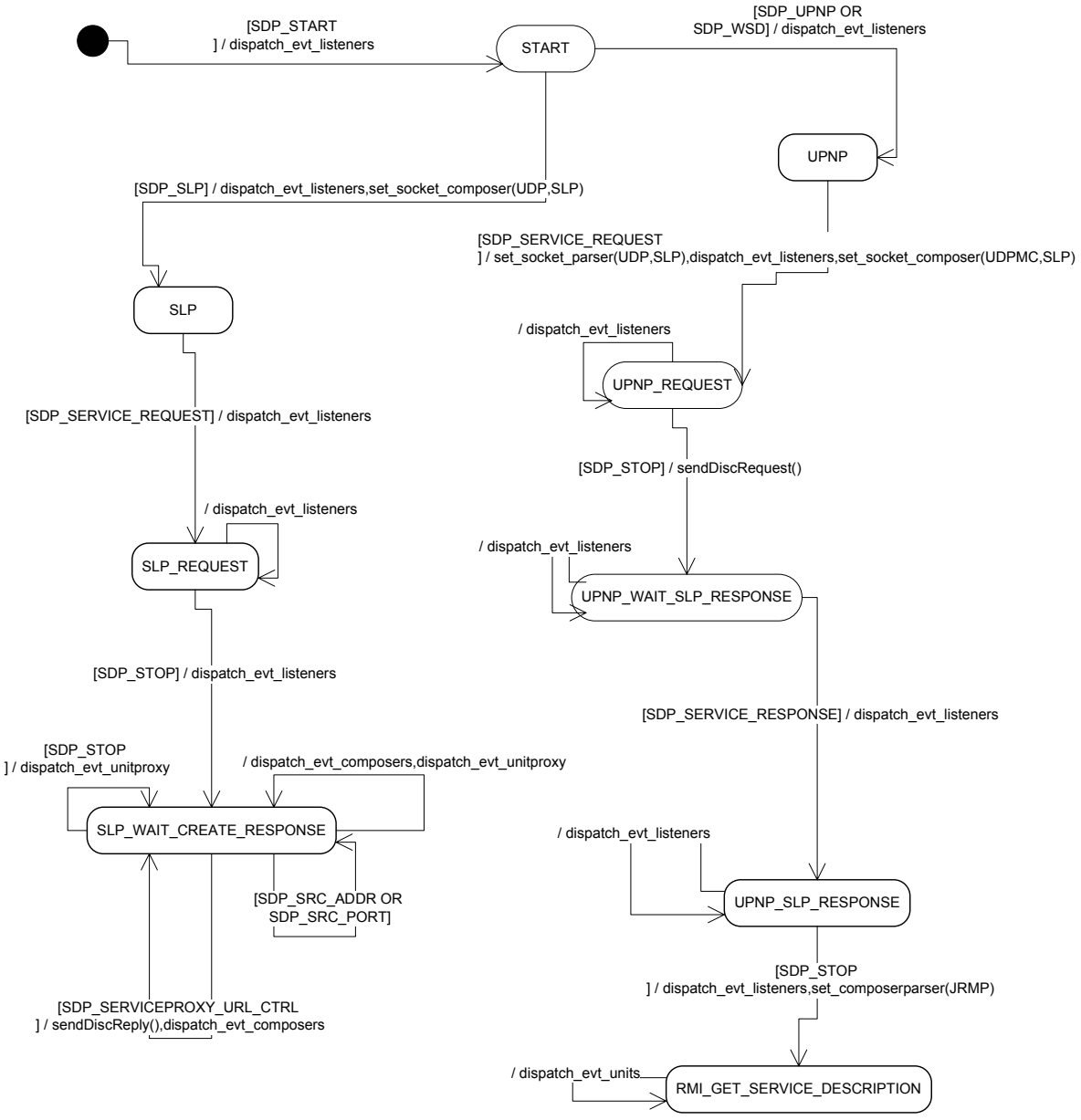

Figure 3-6: SLP unit state diagram

The SLP specifications (RFC2608) define the following types of messages and for each type, they define the format and content of the message:

Service Request	SrvRqst	1
Service Reply	SrvRply	2
Service Registration	SrvReg	3
Service Deregister	SrvDeReg	4
Service Acknowledge	SrvAck	5
Attribute Request	AttrRqst	6

```
Attribute Reply          AttrRply            7
DA Advertisement         DAAdvert            8
Service Type Request     SrvTypeRqst         9
Service Type Reply       SrvTypeRply         10
SA Advertisement         SAAdvert            11
```

The INMIDIO middleware does not support the complete list of messages but It supports only those required for the discovery of services and so, for the communication between the different SLP entities (SLP service, SLP clients, SLP daemon):

- `Service Request` `SrvRqst` 1

  The `SCOPE` used by default for all the SLP messages is fixed at the value `DEFAULT`.

- `Service Reply` `SrvRply` 2

  The `SCOPE` used by default for all the SLP messages is fixed at the value `DEFAULT`. The middleware currently supports only one URL entry in the `SrvRply` message (see RFC2608 for the format of URL Entries in the reply message).

- All the other types of messages are recognized by the system but the `SLPparser` does not generate the corresponding events.

3.4.2 UPnP

The *Universal Plug and Play* (UPnP) specifications[4] define an architecture for pervasive peer-to-peer network connectivity of intelligent appliances, wireless devices, and PCs of all form factors. It is designed to bring easy-to-use, flexible, standards-based connectivity to ad-hoc or unmanaged networks whether in the home, in a small business, public spaces, or attached to the Internet. UPnP is a distributed, open networking architecture that leverages TCP/IP and the Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices in the home, office, and public spaces. The UPnP architecture defines the protocols for discovery, description, control, eventing, and presentation between clients (*UPnP control points*) and services (*UPnP devices*):

- Step1 - Discovery: when a device is added to the network, the UPnP discovery protocol allows that device to advertise its services to control points on the network. Similarly, when a control point is added to the network, the UPnP discovery protocol allows that control point to search for devices of interest on the network. The fundamental exchange in both cases is a discovery message containing a few, essential specifics about the device or one of its services, e.g., its type, identifier, and a pointer to more detailed information. The UPnP discovery protocol is based on the *Simple Service Discovery Protocol* (SSDP) and the multicast address and port reserved for SSDP are respectively 239.255.255.250 and 1900.
- Step2 - Description: the UPnP description for a device is expressed in XML and includes vendor-specific, manufacturer information like the model name and number, serial number, manufacturer name, URLs to vendor-specific Web sites, etc. The description also includes a list of any embedded devices or services, as well as URLs for control, eventing, and presentation. For each service, the description includes a list of the commands, or *actions*, the service responds to, and parameters, or *arguments*, for each action; the description for a service also includes a list of variables; these variables model the state of the service at run time, and are described in terms of their data type, range, and event characteristics.
- Step3 - Control: UPnP specifications define how a UPnP client can send a suitable control message (message call) to the control URL for the service (provided in the device

[4] http://www.upnp.org/download/UPnPDA10_20000613.htm

description). Control messages are expressed in XML using the Simple Object Access Protocol (SOAP). Like function calls, in response to the control message, the service returns any action-specific values.

  - Query for variable: in addition to invoking actions, UPnP clients can also poll the service for the value of a state variable by sending a query message

- Step4 – Eventing: A UPnP description for a service includes a list of actions the service responds to and a list of variables that model the state of the service at run time. The service publishes updates when these variables change, and a control point may subscribe to receive this information. The service publishes updates by sending event messages. Event messages contain the names of one of more state variables and the current value of those variables. These messages are also expressed in XML and formatted using the General Event Notification Architecture (GENA).
- Step5 – Presentation: If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a browser, and depending on the capabilities of the page, allow a user to control the device and/or view device status.

For the implementation of the UPnP protocol, the middleware provides a UPnP Unit that requires an `SSDPparser` and an `SSDPcomposer` respectively used to parse and to generate the SSDP messages in the discovery step. As SSDP is based on UDP, the socket components required are the `UDPSocket` and the `UDPMulticastSocket`.
After the discovery step, the description of the device and service is implemented by the `DeviceDescription` and `ServiceDescription` parsers and composers that are based on HTTP protocol to get access to the content of the XML files that describe the device and the service. As a result, the UPnP unit also requires the `HTTPSocket` component. *Figure 3-7* represents the state machine diagram for the UPnP unit that coordinates these components. The UPnP unit state machine supports replies from multiple services: each SSDP reply message received will be parsed by a different `SSDPparser`.

The INMIDIO middleware does not support all the steps defined by the UPnP standard but it supports only the messages required for the discovery and description of services. As the control is essentially based on SOAP, the support for this step will be examined in the appropriate section (§3.4.5):

- Step1 - Discovery is supported. The standards define four types of messages for the discovery step:
  - Advertisement `ssdp:alive (NOTIFY)` generated when a device is started up.
    - This type of message is recognized by the system but the `SSDPparser` does not generate the corresponding events.
  - Advertisement `ssdp:byebye (NOTIFY)` generated when the device shuts down.
    - This type of message is recognized by the system but the `SSDPparser` does not generate the corresponding events.
  - Discovery `ssdp:discover (M-SEARCH)` is supported for both `urn:schemas-upnp-org:device:deviceType:v` and `urn:schemas-upnp-org:service:serviceType:v`.
  - Response `HTTP 200 OK` is supported.
- Step2 - Description is supported

  - The XML specification language for description of devices and service is completely supported.
  - The middleware limits the number of supported services for each device to only one service: all the method supported for the device must be included in the only service provided by the device.
  - As the UPnP description specification for services does not define arrays and complex types, the middleware supports only a limited set of simple data types: string, integer and boolean.

- Step3 - Control is supported:
  - Requests and responses are supported by the SOAP unit (§3.4.5)
    - `UPnPError` fault not supported.
  - Query for variable not supported.
- Step4 - Eventing is not supported.
- Step5 - Presentation is not supported.

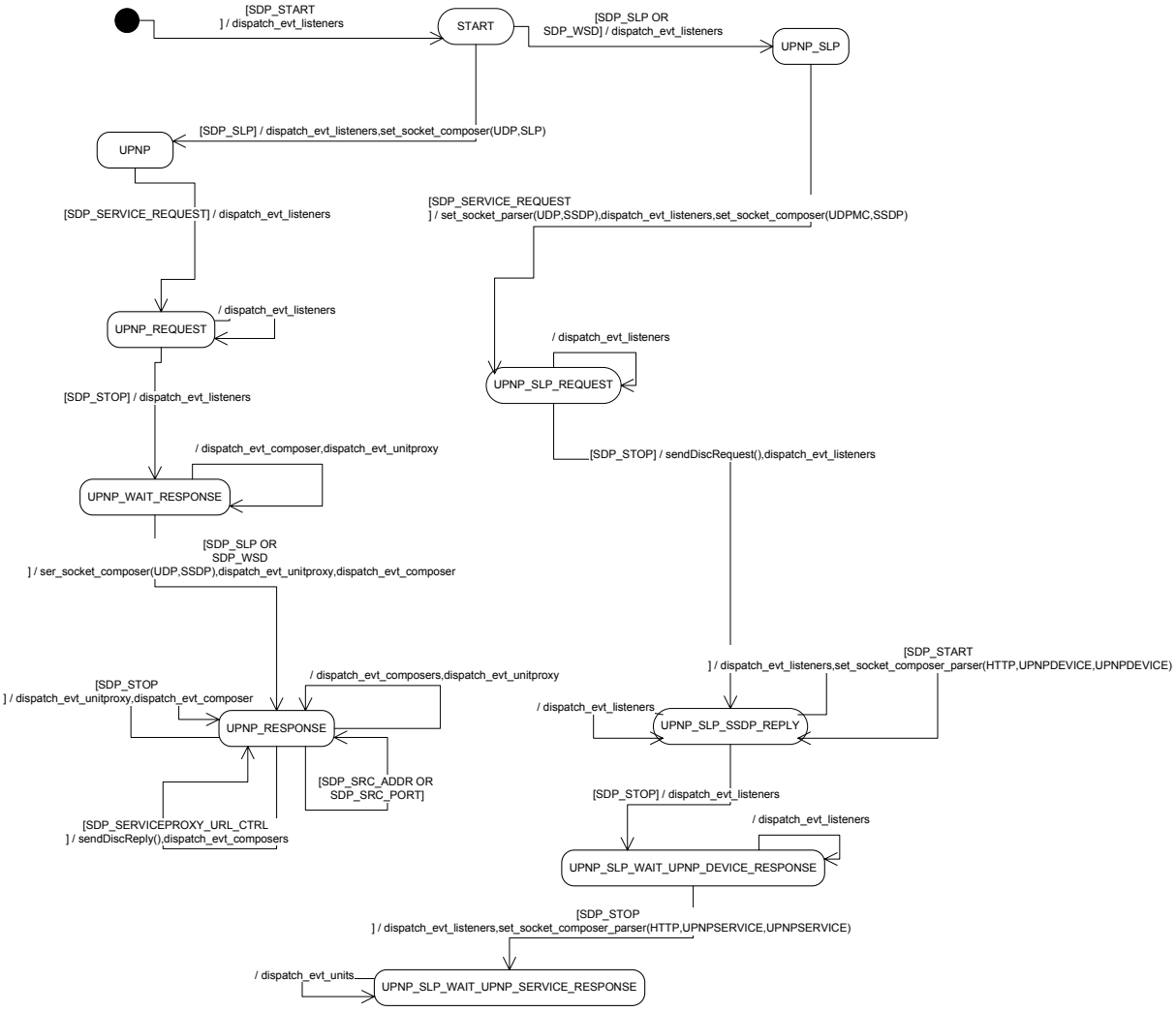

Figure 3-7: UPnP unit state diagram

3.4.3 WS-Discovery

The *Web Services Dynamic Discovery (WS-Discovery)* specification[5] defines a multicast discovery protocol to locate services. The primary mode of discovery is a client searching for one or more target services. To find a target service by the type of the target service, a scope in which the target service resides, or both, a client sends a probe message to a multicast group: the multicast address and port respectively reserved for WS-Discovery are 239.255.255.250 and 3702. Target services that match the probe send a response directly to the client. To locate a target service by name, a client sends a resolution request message to the same multicast group, and again, the target service that matches sends a response directly to the client.

[5] http://schemas.xmlsoap.org/ws/2005/04/discovery/

The WS-Discovery relies on the definition of WSDL[6] for the description of service interface. The protocol uses SOAP over HTTP for interaction between client and service.

The WS-Discovery specification defines the types of messages, their format and content of each message. As WS-Discovery is a Web Services oriented protocol, the content of all its defined messages is expressed in XML. The types of messages supported by the standard are the following:

`Hello`
A message sent by a Target Service when it joins a network; this message contains key information for the Target Service.

`Bye`
A message sent by a Target Service when it leaves a network.

`Probe`
A message sent by a Client searching for a Target Service by Type and/Or Scope

`Probe`
A message sent by a Client searching for a Target Service by Type and/Or Scope

`Resolve`
A message sent by a Client searching for a Target Service by name.

`Resolve`
A message sent by a Client searching for a Target Service by name.

In WS-Discovery there can be three steps in discovery of services but depending upon the `Target-Service` these steps can be short-circuited into 2 or even a single step by providing additional information at an earlier stage. For example, the normal message exchange pattern would require a client to send `Resolve` Message to get the transport level address and then it requires the client to use DNS services to acquire the IP Address. But some `Target Services` can short-circuit this step and provide the IP Address within the `ProbeMatch` in the first and only step of the discovery.

For the implementation of the WS-Discovery protocol, the middleware provides a WS-Discovery Unit that requires a `WSDparser` and a `WSDcomposer` respectively used to parse and to generate the WS-Discovery messages. As WS-Discovery is based on UDP, the socket components required are the `UDPSocket` and the `UDPMulticastSocket`. Figure 3-8 represents the state machine diagram for the WS-Discovery unit that coordinates these components. As WS-Discovery is based on WSDL for the description of services, we have developed a `WSDLParser` component for the recognition and composition of WSDL. This component has the ability to parse a complete WSDL description and generate corresponding events which are later used in the creation of a proxy. The reverse component `WSDLComposer` has the ability to take service description as input and generate a complete WSDL from it which can be seen as a shadow of the desired service. The WS-Discovery unit state machine does not support replies from multiple services: the only the first WS-Discovery reply message received from a service will be treated by the `WSDparser`.

The INMIDIO middleware does not support the complete list of WS-Discovery messages but it supports only the messages required to implement the short-circuited discovery described above. Below, we provide more details about the support for the different types of messages in the middleware:

`Hello`

[6] http://schemas.xmlsoap.org/ws/2005/04/discovery/ws-discovery.wsdl

This type of message is recognized by the system but the `WSDparser` does not generate the corresponding events.

`Bye`
This type of message is recognized by the system but the `WSDparser` does not generate the corresponding events.

`Probe`
Supported.

`ProbeMatch`
Supported.

`Resolve`
This type of message is recognized by the system but the `WSDparser` does not generate the corresponding events.

`ResolveMatch`
This type of message is recognized by the system but the `WSDparser` does not generate the corresponding events.

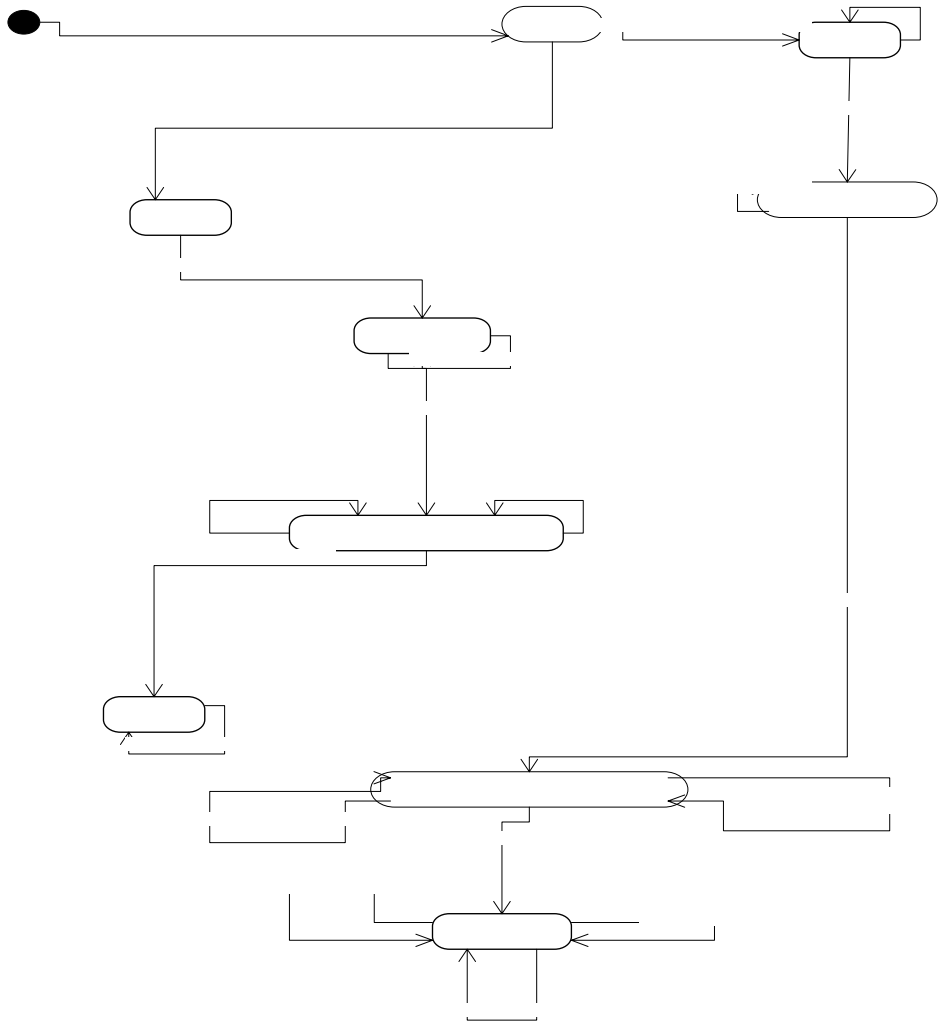

Figure 3-8: WS-Discovery unit state diagram

3.4.4 RMI

Remote Method Invocation (RMI) is a Java-specific protocol for communication between Java remote objects. The RMI specifications[7] define the protocol to execute the invocation of a method of a remote interface on a remote object. A method invocation on a remote object has the same syntax as a method invocation on a local object. The RMI specifications are based on the *Java Remote Method Protocol* (JRMP) and on the Java Object Serialization. The JRMP is used as a transport protocol to transfer data across the network while the Object Serialization protocol is used to marshal call and return data.

For the JRMP transport protocol, the RMI specification defines two different types of streams: *Out* and *In* reflecting a client perspective and for each stream, different types and format of messages are defined:

- Out stream:
    - `Call`: encodes a method invocation.
    - `Ping`: a transport-level message for testing liveness of a remote virtual machine.
    - `Dgcack`: an acknowledgment directed to a server's distributed garbage collector that indicates that remote objects in a return value from a server have been received by the client.
- In stream:
    - `ReturnData`: the result of a "normal" RMI `call`.
    - `HttpReturn`: a return result from an invocation embedded in the HTTP protocol.
    - `PingAck`: the acknowledgment for a `Ping` message.

The RMI specification defines how the `call` and `return` data in RMI calls are formatted using the Java Object Serialization protocol. Each method invocation's `CallData` is written to a Java object output stream that contains the `ObjectIdentifier` (the target object of the call), an `Operation` (a number representing the method to be invoked), a `Hash` (a number that verifies that client stub and remote object skeleton use the same stub protocol), followed by a list of zero or more `Arguments` for the call.

For the implementation of the RMI protocol, the middleware provides an RMI Unit that requires a `JRMPparser` and a `JRMPcomposer` respectively used to parse and to generate the JRMP messages in the interaction step. Figure 3-9 represents the state machine diagram for the RMI unit that coordinates these components.

The Monitor component of the INMIDIO middleware supports two different functions of the RMI implementation: the RMI Registry service lookup operation and the RMI service method call. The Monitor is waiting at the assigned port to receive RMI incoming calls. The `JRMPparser` is listening for messages received at that port. When a call is received, it is interpreted and the decoding algorithm identifies the type of call:

- If it is an RMI Registry lookup call, the request will concern a proxy generated by RMIProxyUnit (§3.4.7) and the data required to generate the lookup reply for the client in an RMI-compatible format are available in a table of service descriptions. This table stores the information about the discovered services, retrieved during the discovery interoperability phase and necessary in order to implement the communication interoperability phase.

[7] http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf

- If it is a service method call, the information embedded into the JRMP call stream (`objID`, `objtime`, …) together with the service description table, will be used to identify the service to be called and the definition of the method (and its arguments: names and data types).

The INMIDIO middleware does not support all the specifications defined by the RMI standard but it supports only the mandatory features required to support the interaction between a client and a service:

- JRMP protocol
    - The `Call` is the only type of message supported because it is the only message mandatory to implement a service method call. All the other types of messages are recognized by `JRMPParser` by not interpreted and no events are generated.
- Object serialization protocol:
    - Both JDK1.1 serialization protocol and Java 2 serialization protocol are supported.
    - Return value: not supported. As multiple return values are not supported in RMI specification, we define `RMIHolders` classes (one `Holder` class is defined for each basic type `String`, `Integer`, …). To implement the support for the return value:
        - add the support for return value in method `composerJRMP_Call` in file `composer/JRMPComposer.c` and in file `vm/jclass_parser.c`.
    - Argument types supported: `int`, `Boolean` and `string`. Complex types and arrays are not supported. To implement the support for complex types:
        - Add event types to support the description of arrays and complex data type arguments.
        - `JRMPParser`: it currently implements the RMI lookup protocol to get the stub and parse its definition to generate the related service description events. Supporting the complex types requires implementing the protocol to get all the classes that implement the complex types the stub (and the RMI service) depends on: they will be parsed and the related events for the description of complex data type arguments will be generated.
        - `ser/objectparser.c` and `ser/objectcomposer.c`: must respectively implement the deserialization and the serialization of Java objects that represent complex data types and Java arrays.
        - The RMIProxy unit (§3.4.7) will receive the event that describe complex data type and array arguments and must generate the corresponding Java classes. These Java classes will be provided to the client together with the Java stub class.
    - Exceptions in return value are not supported.

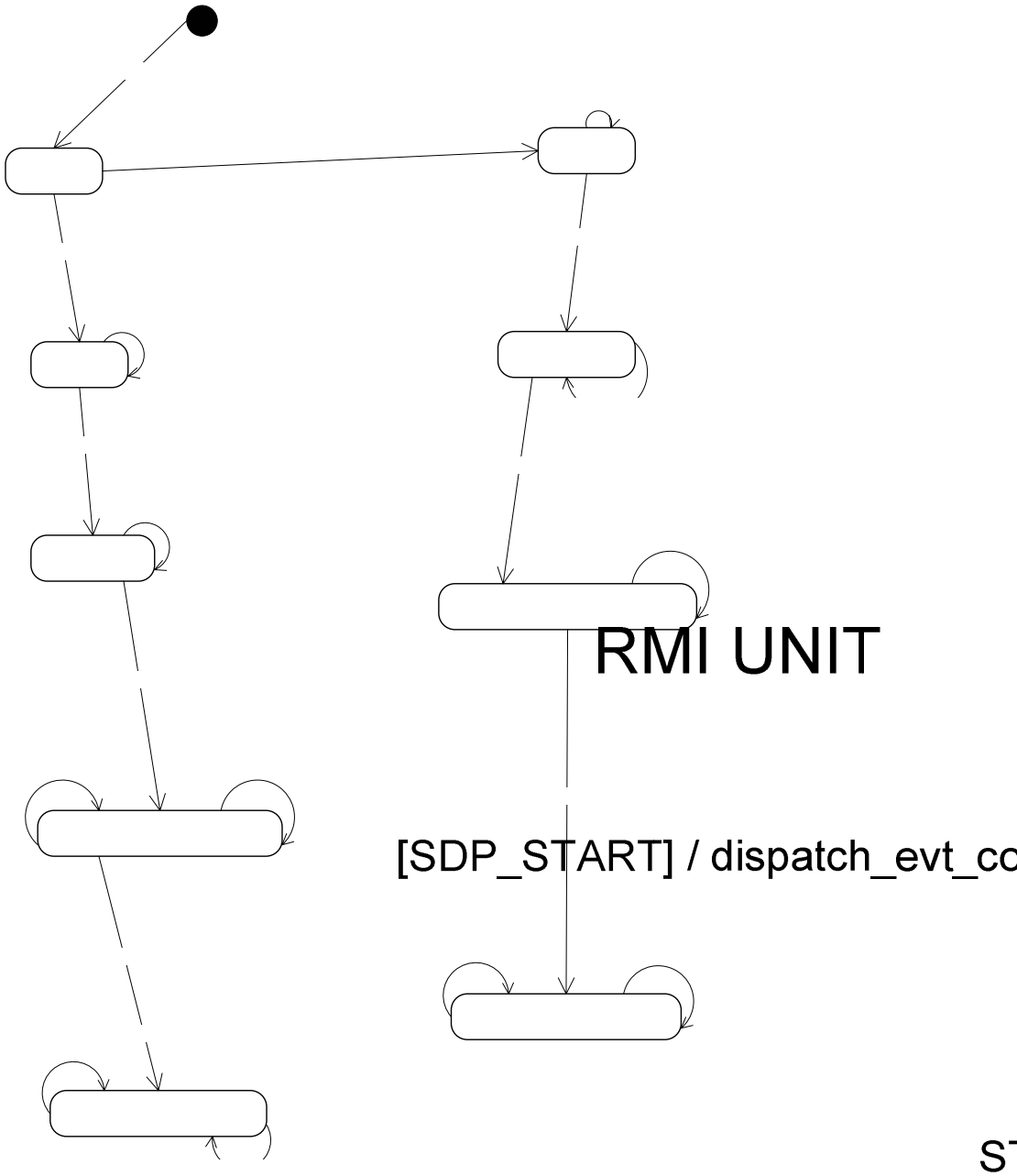

Figure 3-9: RMI unit state diagram

3.4.5 SOAP

The *Simple Object Access Protocol* (SOAP) is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. The SOAP specifications[8]

[8] http://www.w3.org/2000/xp/Group/

uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics. The SOAP distributed processing model supports one-way messages and request/response interactions.

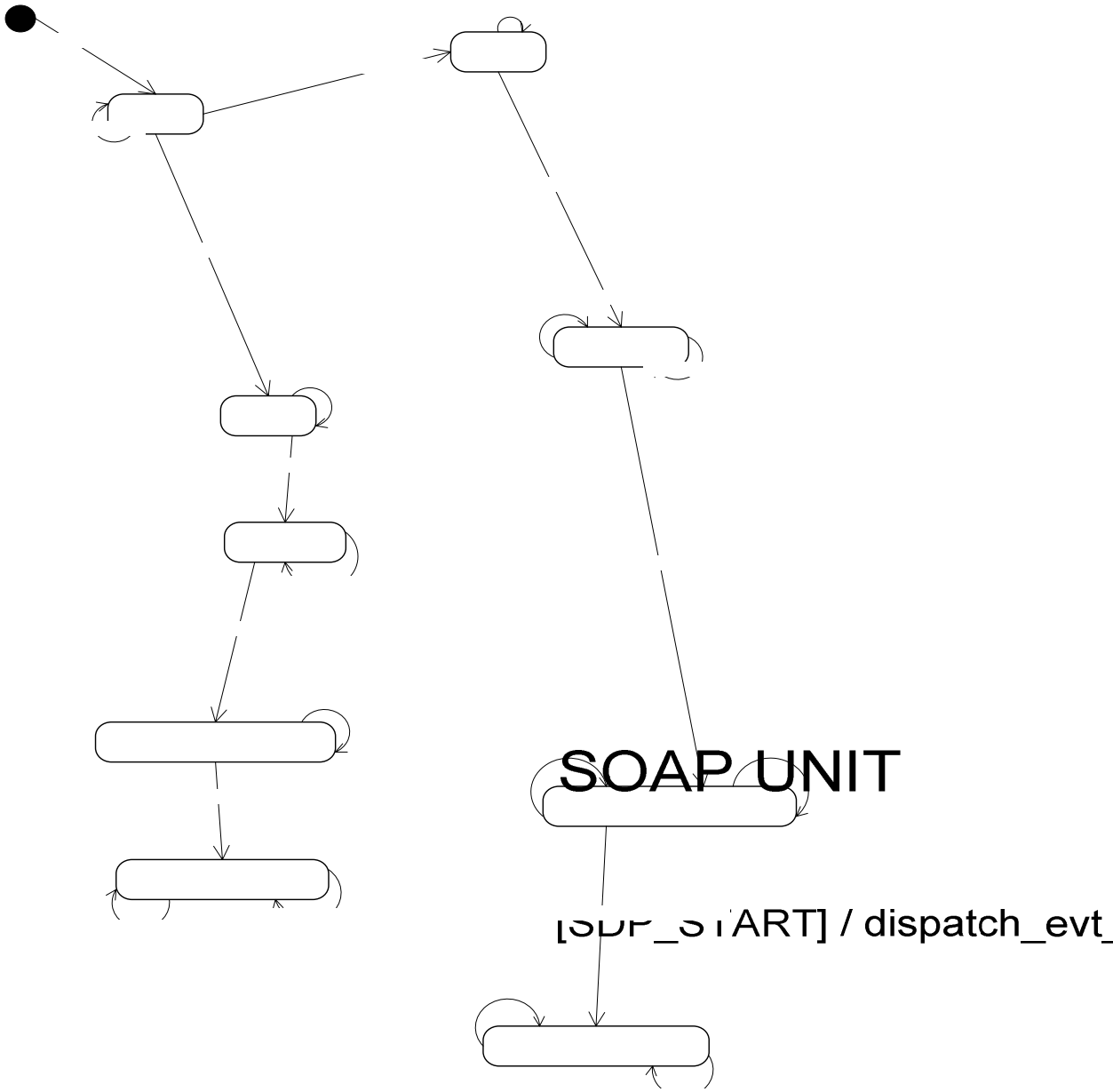

Figure 3-10: SOAP unit state diagram

For the implementation of the SOAP protocol, the middleware provides a SOAP Unit that requires a `SOAPparser` and a `SOAPcomposer` respectively used to parse and to generate the SOAP messages in the interaction step. As the middleware supports only HTTP as underlying protocol for SOAP, the unit also requires the `HTTPSocket` socket component. Figure 3-10 represents the state machine diagram for the SOAP unit that coordinates these components.

The Monitor component of the INMIDIO middleware waits at the port assigned to SOAP to receive incoming calls. The `HTTPSocket` socket is connected to that port and the SOAP message received is published by the socket on the `SOAPparser`. SOAP is the reference interaction protocol for both UPnP and WS-Discovery, thus the SOAP call received on the assigned port can be either from a UPnP or a WS-Discovery client. The destination service of the call can be identified by the Path value of the POST line in the HTTP header. This Path will identify a service in a service definition table. This table stores the information about the discovered services, retrieved during the discovery interoperability phase and necessary in order to implement the communication interoperability phase.

The INMIDIO middleware does not support all the specifications defined by the SOAP standard but it supports only the mandatory features required to support the interaction between a client and a service:

- SOAP RPC requests and response: supported.
    - Only the HTTP binding for SOAP is supported: only the HTTP Header POST command is supported, while the HTTP Header GET command is not supported for SOAP calls.
    - Argument types supported: `int`, `boolean` and `string`. Complex types and arrays are not supported. To implement the support for arrays and complex types:
        - Add event types to support the description of arrays and complex data type arguments.
        - `parser/SOAPparser.c` and `composer/SOAPcomposer.c`: must respectively implement the deserialization and the serialization of SOAP method calls containing arguments that represent arrays and complex data types.
- SOAP Fault specification is not supported

3.4.6 UPnPProxy

As described in the above section about UPnP, UPnP defines an XML definition language for the description of services. This XML language provides tags to describe the location of the description of the service, of the endpoint where service call must be sent, the description of the interface (with methods and their parameters' names and types).

The essential role of UPnPProxy is to provide the XML files describing the service to a UPnP client. And these files must be consistent with the description step of the UPnP specification. These files allow the client to discover the service location and its description and definition and finally invoke the actions provided by the service.

The XML files are created by UPnPProxy using the information contained in the events received by the SD protocol unit during the service discovery interoperability process. The XML files are created on the middleware and provided to the UPnP client via an HTTP server (hosted on the middleware machine) that is in charge of replying to the HTTP GET requests from the client. Figure 3-11 represents the state machine diagram for the UPnPProxy unit that coordinates these components.

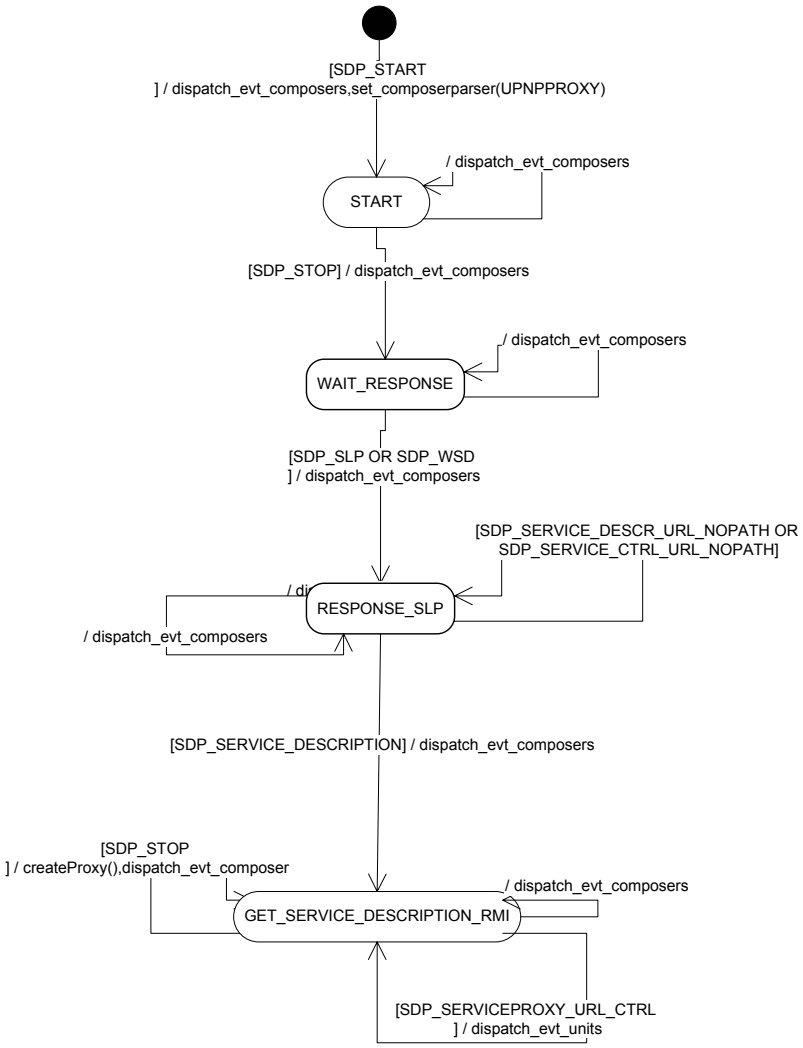

Figure 3-11: UPnPProxy unit state diagram

3.4.7 RMIProxy

RMI uses a standard mechanism for communicating with remote objects, also employed in many Remote Procedure Call (RPC) systems: *stubs* (also called *proxy*) and *skeletons*. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements. The skeleton represents the service in the remote method call invocation.

When a stub's method is invoked, the following actions are executed:

- The stub initiates a connection with the remote Java Virtual Machine (JVM) containing the remote object.
- The stub marshals (writes and transmits) the parameters to the remote JVM.
- The stub waits for the result of the method invocation.

- The stub unmarshals (reads) the return value or exception returned.
- The stub returns the value to the caller.

The stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

The RMI service that provides the remote object registers on the RMI Registry providing a stub to access its methods. The mechanism used by the client to download the stub is the Dynamic Class Loading defined in RMI specifications and that is based on Java Object serialization specification[9].

The RMIProxy unit generates the stub for the RMI client: this action consists in creating a Java class compatible with the RMI specification, the Java Virtual Machine specification[10] and the Java Object serialization specification. This stub will be downloaded by the client using the standard RMI protocol used for lookup operations on the RMI registry. The client will use the RMI protocol as if it was communicating with a real RMI registry as provided by the Java platform while it is actually communicating with the RMI Registry implementation running on the middleware (3.4.4).

The information about the service description is received by RMIProxy unit in the form of events. The `RMIProxyComposer` will be used in order to create the stub and its address will be notified to the SD protocol unit to generate a valid discovery response message. Figure 3-12 represents the state machine diagram for the RMIProxy unit that coordinates these components.

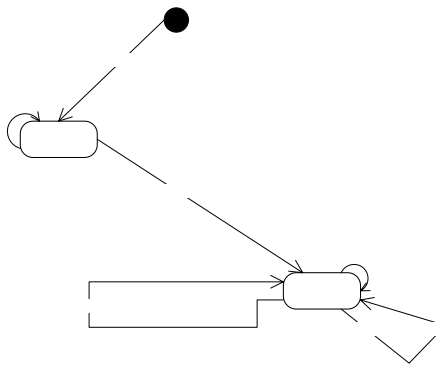

Figure 3-12: RMIProxy unit state diagram

3.4.8 WS-DiscoveryProxy

As the WS-Discovery based client does not require any special stubs or service description files to access the service, the essential role of WS-DiscoveryProxy unit is to provide the address of the SOAP entrypoint on the middleware where SOAP method calls must be sent from the client: this address will be provided to the WS-Discovery unit to create a discovery message response to the client.

[9] http://java.sun.com/j2se/1.5/pdf/serial-1.5.0.pdf

[10] http://java.sun.com/docs/books/vmspec/index.html

The `WSDProxyComposer` will be used in order to generate this address and to notify it to the SD protocol unit to generate a valid discovery response message. Figure 3-13 represents the state machine diagram for the WS-DiscoveryProxy unit that coordinates these components.

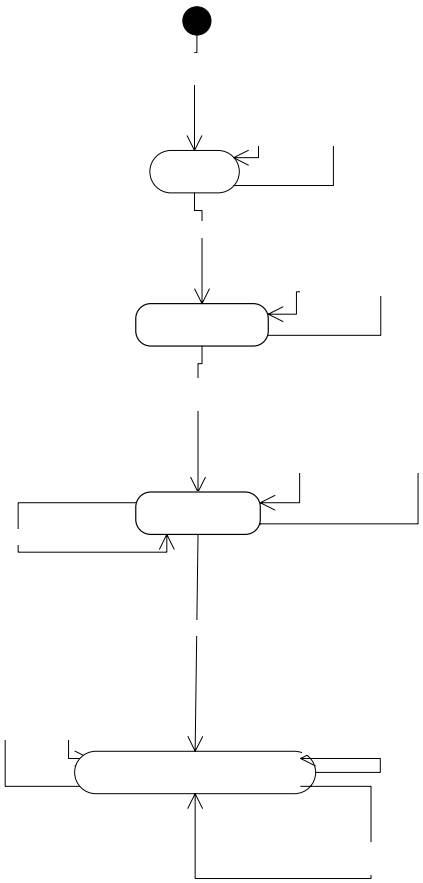

Figure 3-13: WS-DiscoveryProxy unit state diagram

4 Tutorial

4.1 Component development

The component oriented framework implemented by the INMIDIO middleware allows to easily develop and deploy new components into the middleware. In this section we describe, for each type of component (parser, composer, socket and unit), the essentials steps to be implemented in order to add a component for the support of a new SD or SI protocol and make it available to the existing middleware components.

4.1.1 Parser development

Add a `.c` file in `src/parser/Xparser.c` that implements at least the following functions:

- A function `X_parser_create` to create the parser: data structures memory allocation and initialization of the fields and parser's callback functions for parsing, publishing events and receiving messages:

```
PARSER_PTR X_parser_create(UNIT_PTR u)
{
        fprintf(stdout, "Creating X parser\n");
        PARSER_PTR p = parser_create(u);
        p->func_msg_received = parser_msg_received;
        p->func_parse = X_threaded_msg_parsing;
        p->func_publish_event = parser_publish_event;
        p->evt_listeners = NULL;
        p->factory = X_parser_create;
        return p;
}
```

- A parsing function `X_threaded_msg_parsing` as defined in `X_parser_create` in the parser's field value `p->func_parse`. This is a callback function that will be invoked by the function `void parser_msg_received(MSG_PTR msg, LISTENER_PTR node)` when a message a `MSG_PTR` is received:

```
void *X_threaded_msg_parsing(void* ptr)
{
        fprintf(stdout, "X_threaded_msg_parsing. \n");
        PARSINGTHREADDATA_PTR data = (PARSINGTHREADDATA_PTR)ptr;

        MSG_PTR msg = (MSG_PTR)data->msg;
        PARSER_PTR parser = (PARSER_PTR)data->parser;

        EVENTQUEUE_TABLE_ENTRY_PTR entry_tmp =
          eventqueue_table_get_entry_by_current_socket_reply_port(parser->unit->event_table,
                                              msg->dest_port);

        if (entry_tmp == NULL && msg->dest_port == X_MULTICAST_PORT)
        {
        /*
         * In this case, the message is a discovery message received from an X client
         */

          pthread_t thread_event_src = pthread_self();
          EVENTSRC_PTR evt_src = event_create_src((void*)thread_event_src, data->parser->unit);

                ……
          int evt_stream_id = (int)(evt_src->src_thread_parser);
                ……
                ……
           parser_publish_event(event_create(SDP_START, evt_stream_id,(void*)evt_src),parser);
                ……
                ……
```

```
            ……
            parser_publish_event(event_create(SDP_STOP, evt_stream_id,(void*)evt_src),parser);
        }
        else if (entry_tmp != NULL)
        {
        /*
         * In this case the message is a reply to a discovery message received from an X service
         */
          EVENTQUEUE_TABLE_ENTRY_PTR entry =
                eventqueue_table_copy_entry(parser->unit->event_table, entry_tmp);

          pthread_t thread_event_src = pthread_self();
          entry->sub_stream_id = (int)thread_event_src;

          EVENTSRC_PTR evt_src = event_create_src((void*)entry->sub_stream_id, parser->unit);

          setCurrentState(X_STATE_RESPONSE, entry->sub_stream_id, parser->unit->engine);

          int evt_stream_id = (int)(evt_src->src_thread_parser);
                ……
                ……
           parser_publish_event(event_create(SDP_START, evt_stream_id,(void*)evt_src),parser);
                ……
                ……
                ……
            parser_publish_event(event_create(SDP_STOP, evt_stream_id,(void*)evt_src),parser);

        }

}
```

4.1.2 Composer development

Add a `.c` file in `src/composer/Xcomposer.c` that implements at least the following functions:

- A function `X_composer_create` to create the composer: data structures memory allocation and initialization of the fields and composer's callback functions for composing and receiving events:

```
COMPOSER_PTR X_composer_create(UNIT_PTR u)
{
        fprintf(stdout, "Creating X composer\n");
        COMPOSER_PTR c = composer_create(u);
        c->run_composer = X_run_composer;
        c->factory = X_composer_create;
        c->func_event_received = X_composer_event_received;
        return c;
}
```

- A function `X_run_composer` that implements the main thread of the composer as defined in `X_composer_create` in the composers's field value `p->run_composer`. This is a callback function that will be invoked when the composers is activated by the unit:

```
void *X_run_composer(void* ptr)
{
        COMPOSETHREADDATA_PTR data = (COMPOSETHREADDATA_PTR)ptr;
        EVENTQUEUE_TABLE_ENTRY_PTR entry = (EVENTQUEUE_TABLE_ENTRY_PTR)data->entry_event_queue;
        COMPOSER_PTR composer = (COMPOSER_PTR)data->composer;
        pthread_mutex_lock(&(entry->mutex));
        pthread_t thread_composer = pthread_self();

        while (1)
        {

           EVENT_PTR evt = eventqueue_table_get_event(entry);
           if (evt != NULL)
           {
                switch (evt->type)
                {
                   case SDP_SOURCE_ADDR:
                        ……
```

```
                    ……
                    ……
                    break;
                case SDP_SOURCE_PORT:
                    ……
                    ……
                    ……
                    break;

                ……
                ……
                ……
                ……
                ……
                ……
                 case SDP_SEND_SERVICE_REPLY:
                 {
                    MSG_PTR msg = X_composer_X_reply(X_msg);
                    composer_publish_msg(msg, composer);
                  }
                    break;

                  case SDP_SEND_SERVICE_REQUEST:
                  {
                    MSG_PTR msg = X_composer_X_request(X_msg);

                    COMPOSERTHREAD_LIST_ENTRY_PTR first_composer_thread  =
                            eventqueue_table_remove_first_composer_thread(entry);
                    entry->current_socket_reply_port =
                            first_composer_thread->socket_reply_port;

                    composer_publish_msg(msg, composer);
                    pthread_cond_signal(&(first_composer_thread->cond));
                    pthread_mutex_unlock(&(entry->mutex));
                   }
                    break;
            }
          }
        }
}
```

4.1.3 Socket development

Add a `.c` file in `src/socket/Xsocket.c` that implements at least the following functions:

- A function `X_socket_create` to create the socket: data structures memory allocation and initialization of the fields and socket 's callback functions for publishing and receiving messages:

```
SOCKET_PTR X_socket_create(UNIT_PTR u)
{
        fprintf(stdout, "Creating X socket\n");

        SOCKET_PTR s = (SOCKET_PTR)malloc(sizeof(SOCKET));
        s->func_msg_received = socket_msg_received;
        s->func_publish_msg = socket_publish_msg;
        s->msg_listeners = NULL;
        s->unit = u;
        s->func_send = X_socket_send;
        s->factory = X_socket_create;
        s->func_listen = X_socket_listen;
        return s;
}
```

- A function `X_socket_listen` as defined in `X_socket_create` in the sockets's field value `s->func_listen`. This is a callback function that will be invoked by the unit when the socket is activated to receive network messages:

```
 void* X_socket_listen(void* ptr) {
```

```
        SOCKETLISTENTHREADDATA_PTR data = (SOCKETLISTENTHREADDATA_PTR)ptr;
        SOCKET_PTR this_socket = data->socket;
        char* addr_str = data->addr;
        unsigned short port = data->port;
        ……
        ……
        /* bind to addr_str and port
            and created the related sock */
        ……
        ……
        recv_len = recvfrom(sock, recv_str,MAX_LEN, 0,(struct sockaddr*)&from_addr,&from_len));
        ……
        ……
        MSG_PTR msg = msg_create(recv_str, recv_len, inet_ntoa(from_addr.sin_addr),
                                ntohs(from_addr.sin_port), mc_addr_str, mc_port, this_socket);

        socket_publish_msg(msg, this_socket);
}
```

- A function `X_socket_send` as defined in `X_socket_create` in the sockets's field value `s->func_send`. This is a callback function that will be invoked by the function `void socket_msg_received(MSG_PTR msg, LISTENER_PTR node)` when a message a `MSG_PTR` is published in order to be sent on the network:

```
void* udp_socket_send(void* ptr)
{
        SOCKETTHREADDATA_PTR data = (SOCKETTHREADDATA_PTR)ptr;
        MSG_PTR msg = data->msg;
        SOCKET_PTR this_socket = data->socket;
        ……
        ……
    /* create addr destination using values : msg->dest_address and msg->dest_port*/
        ……
        ……
    sendto(fd, msg->data, msg->length, 0, (struct sockaddr *) &addr, sizeof(addr)) ;
}
```

4.1.4 Unit development

Add a `.c` file in `src/unit/Xunit.c` that implements at least the following functions:

- A function `X_unit_create` to create the unit: data structures memory allocation and initialization of the fields and definition of the components (parsers, composers and sockets) required for the implementation of the unit. The event publish/subscribe relations with other units (SD protocol units and Proxy unit) are created using the `Yunit_create2` (see below for description of this function) and `unit_add_evtlistener_unit`.

```
UNIT_PTR X_unit_create(char* sm_filename)
{

        UNIT_PTR u = unit_create(sm_filename, NULL);
        fprintf(stdout, "Creating X unit: %p\n", u);

        PARSER_PTR p = X_parser_create(u);
        char key_p[255];
        strcpy(key_p, "X");
        unit_addparser(u, key_p, p);
        unit_add_msglistener_parser(u, p);
        parser_add_evtlistener_unit(p, u);

        COMPOSER_PTR c = X_composer_create(u);
        char key_c[255];
        strcpy(key_c, "X");
        unit_addcomposer(u, key_c, c);

        SOCKET_PTR s = udpmcsocket_create(u);
        char key_s[255];
```

```
        strcpy(key_s, "UDPMC");
        unit_addsocket(u, key2, s);

        UNIT_PTR Y_unit = Yunit_create2("Yunit.sm");
        unit_add_evtlistener_unit(u, Y_unit);
        unit_add_evtlistener_unit(Y_unit, u);

        UNIT_PTR Xproxy_unit = Xproxy_unit_create2("Xproxy_unit.sm");
        unit_add_evtlistener_unit_proxy(u, Xproxy_unit );
        unit_add_evtlistener_unit(Xproxy_unit , u);

        return u;
}
```

- A function `X_unit_create2` to create the unit: data structures memory allocation and initialization of the fields and definition of the components without setting the event publish/subscribe relations with other units.

```
UNIT_PTR X_unit_create2(char* sm_filename)
{

        UNIT_PTR u = unit_create(sm_filename, NULL);
        fprintf(stdout, "Creating X unit: %p\n", u);

        PARSER_PTR p = X_parser_create(u);
        char key_p[255];
        strcpy(key_p, "X");
        unit_addparser(u, key_p, p);
        unit_add_msglistener_parser(u, p);
        parser_add_evtlistener_unit(p, u);

        COMPOSER_PTR c = X_composer_create(u);
        char key_c[255];
        strcpy(key_c, "X");
        unit_addcomposer(u, key_c, c);

        SOCKET_PTR s = udpmcsocket_create(u);
        char key_s[255];
        strcpy(key_s, "UDPMC");
        unit_addsocket(u, key2, s);

        return u;
}
```

Create a `.sm` file that implements the unit's state machine and that will be used as an argument for `X_unit_create` to instantiate the unit (for the specification language of `.sm` files, see the Appendix 5.1).

5 Appendix

5.1 Description of tools/languages provided by the component

5.1.1 Description of language for Unit's state machine

For the description of the syntax of language for the definition of the units state machines, see the related document[11].

5.2 FAQ

[11] http://www-rocq.inria.fr/arles/download/inmidio/EBEMI.pdf