A Simple Erlang API for Handling DDS Data Types and Quality of Service Parameters

Wafa Helali, Khaled Barbaria and Belhassen Zouari
LIP2 Lab, University of Tunis, El Manar, Tunis, Tunisia

Keywords: Middleware, Functional Programming, DDS, Erlang.

Abstract: The choice of the programming language impacts the efficiency of the application and the robustness of the code. The characteristics of Erlang as a functional programming language supported distributed real time computing allowed us to propose eDDS: an Erlang based middleware compliant to the Data Distribution Service (DDS) standard that providing a strong Quality of Service (QoS) support. When the performance and the compliance to the norm have been easy achieved in particular on defining and setting QoS parameters, the lack of efficient and user-friendly support for data type management has been noticed. In this paper, we will explain this type checking problem and how we solved it.

1 INTRODUCTION

Programming distributed applications is a complex activity. These applications are characterized by a great hardware (physical networks, hardware platform) and software (operating systems, programming languages, etc.) heterogeneity . The design of such applications becomes more complex when we talk about real time distributed applications. Such systems now present in many areas, especially in the aerospace, defense, power systems and industrial control.

To facilitate the development of such applications, it is recommended to choose the adequate implementation support. In fact, programming models and languages impact the ability to easily write clear and reliable programs. Like presented in (Emmanuel Chailloux, Pascal Manoury and Bruno Pagano, 2001), the simplicity of a programming language is dependent on many level of abstraction: Abstraction from the *machine* and the *operational model*; Abstract *errors*: fault tolerance and error handling; Abstract *components*: a complex application can be subdivided into autonomous subprograms or components that can be reused in other context; *Interoperability* between the languages can be also a kind of abstraction: the possibility to communicate with other programs written in other programming language.

These elements motivate the choice of the functional programming paradigm (Hudak, 1989) which brings a high level of abstraction. This paradigm use functions as base of computations. A function is defined as first class citizens: it can be stored, composed and even passed as parameters. Moreover, functional languages provide the programmer with facilities to abstract data-types and easily support pattern matching. Some of them are optimized to specific research and industrial objectives (e.g. handle concurrency, support distribution, control execution time, etc.). Functional programs rely on automatic allocations and the heavy use of recursion. Consequently, they can be written and reused, executed and redeployed more easily on central, parallel and distributed architectures. This is the reason why various functional programming languages are used to program large-scale industrial systems such as the Amazon SimpleDB documentation and the Erlang based Web server yaws, as well as the Facebook instant messaging service.

The research work presented in this paper mainly uses Erlang (Ericsson Computer Science Laboratory, 1980) as a framework. This concurrent distributed real-time language, whose story is already old, still attracts many developers and designers especially from mobile technologies. We are focused in first step on how to create a simple API generated automatically to define and set quality of service parameters in our eDDS middleware: the first Erlang support of the DDS specification. In other hand, Erlang is a dynamically typed language. This characteristic can be an advantage and an inconvenient at the same time. In this paper we show how we have benefited from

Helali W., Barbaria K. and Zouari B..
A Simple Erlang API for Handling DDS Data Types and Quality of Service Parameters.
DOI: 10.5220/0005381600190026
In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2015), pages 19-26
ISBN: 978-989-758-100-7

this feature and we present our solution to solve the type checking problem caused by the lack of a strong typing system in order to make our middleware more reliable.

The rest of the paper is structured as follows. In the next two sections we briefly review the basis of our work in order to put it into context. In the section 4, we describe our Erlang DDS API for defining and setting QoS parameters and then we present the Erlang type checking problem and how we solve it. Some closely related work are presented in section 5. Our concluding remarks are presented in section 6.

2 DATA DISTRIBUTION SERVICE

DDS (Data Distribution Service) is specified by the OMG (Object Management Group , 2007) to satisfy the requirements of real-time (large scale) distributed applications, which have direct interaction with real-world objects, such as air-crafts, trains, stock transactions, and defense engines. This standard presents an evolved technology to exchange data in a distributed context. It is based on a high-level data model where the unit of transmission is the data instead of objects reference(CORBA reference passing)(Object Management Group, 2012) or messages (CORBA by value and MOM such as JMS).

This underlying model, focused on data, allows to identify and control the access to all data which circulate in the system through a Global Data Space (GDS). Therefore, the integration of applications becomes simple and easy: the participants using DDS can read an write data efficiently and naturally with a typed interface that allows the developers to specify only the data that will be transferred or read and the associated quality requirements , regardless of the other low level considerations related to the effective dissemination of this data. The communication between the distributed nodes is accomplished with the aid of the following entities: Domain, DomainParticipant, Publisher, DataWriter, Subscriber, DataReader and Topic. These entities, depicted in Figure 1 are described below. **Domain** is a construct that binds all the applications entities able to interact with each other. An application can participate in several domains at the same time. Domain is consequently a means to easily define independent partitions in the distributed system. Partitions can be defined to isolate applications running on the same physical computer from each other.

The **DomainParticipant** acts as container for all other DCPS (Data Centric Publish Subscribe) entities. It represents the local membership of the application

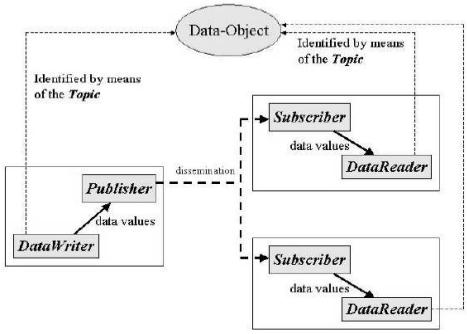

Figure 1: DDS Entities (Object Management Group , 2007).

in a Domain. It is responsible for creating Publishers, Subscribers and Topics.

Topics provide the basic connection point between publishers and subscribers. A topic is identified by a unique name within a Domain. It is associated to a specific data type that can be communicated when publishing or subscribing on this Topic.

The **Publisher** is the object responsible of data dissemination. A publisher can send data of different types. It includes several **DataWriters**, each DataWriter is associated with a specific data type. It is the object which allows applications to update the data values that will be published under a given Topic.

The **Subscriber** receives the published data that corresponds to its subscriptions. It makes it available to the application through many **DataReaders**, DDS associates a DataReader to each specific data type.

3 ERLANG: DYNAMICALLY TYPED FUNCTIONAL DISTRIBUTED LANGUAGE

Erlang is a concurrent functional programming langage developed at Ericsson in 1980 and intended for the implementation of its telecommunication systems. It is successfully used in major industrial projects and large scale applications such as SimpleDB document, databases of Amazon and the yaws HTTP server, as well as Facebook. Erlang is both a language and runtime environment. It provides many libraries grouped under the name of OTP (Open Telecom Platform). It possesses qualities that simplify program design and ensures efficient and comfortable design and development of applications: (Cesarini and Thompson, 2009):

- **High-level Constructs.** Erlang is a declarative language, dynamically typed. There are no assignments nor mutable data structures. An Erlang program can contain several data types including

atoms, numbers, and process identifiers as well as compound data types like lists, tuples and records. Functions are *first-class data*: a function can be stored in a list, returned by a function, or communicated between processes. Erlang functions can be defined as a set of clauses or equations. The selection of a clause is done by the *pattern matching mechanism*.

- **Memory Management.** The management of memory in Erlang (allocation and deallocation) is done automatically. Each process has its own garbage collector. This makes development program easier and faster by eliminating programming errors such as memory deallocation and buffer overflows.
- **Concurrency.** Unlike other languages like C or Java that directly use the threads of operating system, Erlang manages the created processes by the virtual machine (VM). Each process handles its own memory space and runs independently of the others. The processes communicate with each other by using a message exchange mechanism. Response time of systems based on this mechanism is about some microseconds regardless of the system load .
- **Distribution.** Erlang processes are created by the virtual machine. Multiple virtual machines (Erlang nodes) can be connected to each other and the processes running on different nodes communicate with each other by using the same primitive message sending that used by processes in the same node.
- **Robustness.** Erlang provides mechanisms for handling errors and exceptions: Erlang process can be connected together, if one crashed, the other will be informed. An other mechanism provided by Erlang/OTP is the supervision that can be used to monitor and handle process termination. Using these techniques makes Erlang programs shorter and easier to understand by separating the specific code of the application from the code that manages the dysfunctions.

4 BUILDING SAFE, EFFICIENT DISTRIBUTED SYSTEM WITH EDDS

The main role of middleware is to simplify the development of distributed applications by providing an API for an easy and comfortable programming. eDDS provides a simple API to create DDS entities, define and set QoS policies. In other hand, middleware should help the developers of the distributed systems during the development of these applications. In this section we present how we respect these requirements.

4.1 Presentation of eDDS

eDDS is the first Erlang implementation of the DDS specification. eDDS is built upon CORBA architecture. DDS entities are implemented as CORBA objects deployed on various communicating nodes. Figure 2 shows the architecture of eDDS: an intermediate server (Repsitory) allows publishers and subscribers to discover each other. It acts as an intermediary that matches compatible publishers and subscribers (wrt Topic and QoS). When a client requests a subscription for a given Topic, the server locates this Topic and informs all existing DataWriter for the location of the new DataReader. The communication between nodes is based on the Erlang message passing mechanism: the data will be published as an Erlang message from DataWriters to DataReaders.

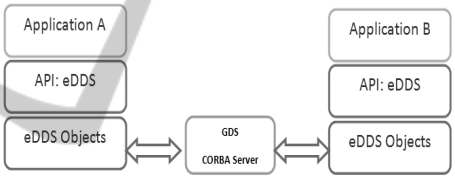

Figure 2: The Architecture of eDDS.

4.2 Defining and Setting QoS Parameters on eDDS

The most important advantage of DDS is the possibility to tailor and control a wide range of quality of service parameters (Object Management Group , 2007). These QoS policies are multiple and diverse, they address different areas of middleware behavior and several aspects of data like: *data availability* (Durability, Lifespan, History); *data delivery* (Reliability, Destination Order, Ownership); *data timeliness* (Latency_Budget, Deadline) and *maximum resources* used in the system (Resource_Limits, Time_Based_Filter).

QoS management is one of the key distinguishing features of the DDS when compared to other pub/sub standards. These QoS parameters allow to specify exactly how the information should flow between publisher and subscriber sides: application developers only indicate 'what' they want to publish and specify the QoS parameters that should be respected, but they are not responsible for 'how' this QoS should be

achieved (Corsaro et al., 2006). These QoS policies are represented in the specification like IDL structures that will be transformed as Erlang records after compilation. A record is an Erlang data type structure used to store a fixed number of elements. The next example represents the transformation of the presentation QoS parameter in Erlang record after compilation.

The manipulation of Erlang records can be difficult especially for the developers that are not familiar with this language. This data structure requires the use of numerous symbols like ”#”, ”(”, ”{”. Thus, it is useful for any Erlang program which uses records to define the methods of treating, reading and writing the record element. Our idea consists in using an API that simplify the use of records and make it more reliable (Trung, 2009).

```
% name of the QoS policy  passed as parameter
type(Record) when is_records (Record,
'DDS_PresentationQosPolicy')->
{ok,'DDS_PresentationQosPolicy'}.

% fields of the given QoS parameter
fields('DDS_PresentationQosPolicy')->
[access_scope,coherent_access,ordered_access].

% set/get access\_scope value of the PreQos
set(Record,access_scope,Value)
get(Record,access_scope)
```

This API is generated automatically from the records by applying the ”parse” function to the DDS.hrl file that contains definitions of all DDS QoS policies. The syntactic analysis is made in two stages: the first step serves to extract all attributes (file, elements and subfields) which compose the header file. Each element of this list will be also analyzed using the ”parseRecord” function. The result of applying this function to the PresentationQos policy is presented in Figure 3.

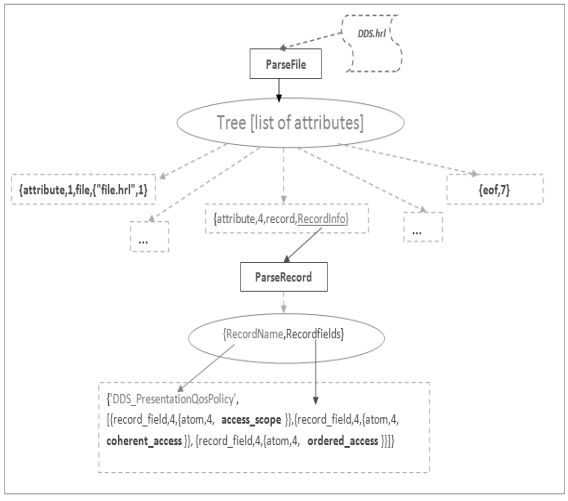

Figure 3: Generate setget function (step1).

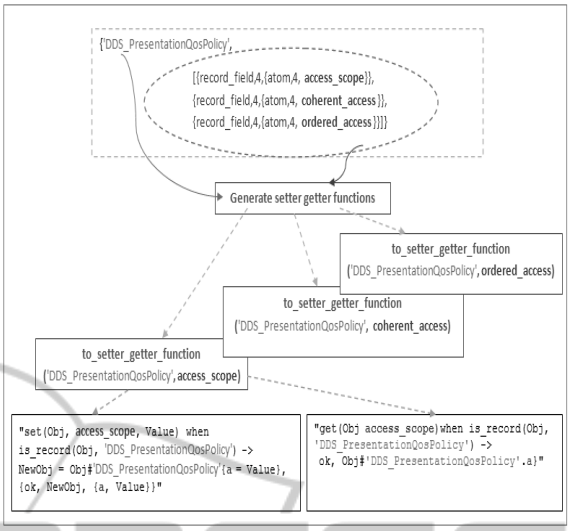

Figure 4: Generate setget function (step2).

In the next step, the result variables (RecordName, RecordFields) will be used as parameters for the generative functions which are responsible for creating all the necessary functions to treat the QoS parameters. Then, it should collect all the creating functions by using the function ”listflatten” and then write them on the API file by using the function ”write” of the ”file” module. Figure 4 shows how to generate the functions related to the Presentation QoS parameter. To ameliorate this API we generate an other function that permit to setting all fields of a given QoS parameter at the same time like presented in the next listing.

```
set_PresentationQosPolicy(Record, access_scope,
V_as,coherent_access,V_ca,ordered_access,V_oa).
```

The next example shows the difference of using this API before and after amelioration. We purpose that initial PresentationQoS parameter is ”PreQoS” and we want to change their three fields (access_scope, coherent_access, ordered_access). By using the new setting function we can change the QoS parameter in one line of code. The quality of service parameters in eDDS are usually represented by complex records and contain multiple fields. That is why, it is important to make our API handling this kind of records. Like presented below, we generate two functions that make

changing the PresentationQoS Parameter using the generated API before Amelioration	changing the PresentationQoS Parameter using the generated API after Amelioration
PreQoS1=set(PreQoS, access_scope,Val2), PreQoS2=set(PreQoS1, coherent_access,false), PreQoS3=set(PreQoS2, ordered_access,3)	PreQoS1=set_DDS_PresentationQosPolicy(PreQoS, access_scope,Val2,coherent_access,false,order ed_access,true)

Figure 5: Setting the Presentation QoS parameter.

setting and getting subfields for complex records.

```
getSF( Record, Field,sub_field).
setSF(Record, Field,sub_field,V).
```

The following example present how we use these functions to set the presentation parameter in the QoSPublisher policy (we want to change the value of ordered_access to false). This PublisherQoS policy is a complex record: the presentation parameter is also a record. Without our API the setting operation would make on two steps: we should create the new presentation value before make it into the new PubQos value. This will be reduced at one line of code by calling the ”setSubField” function (setSF).

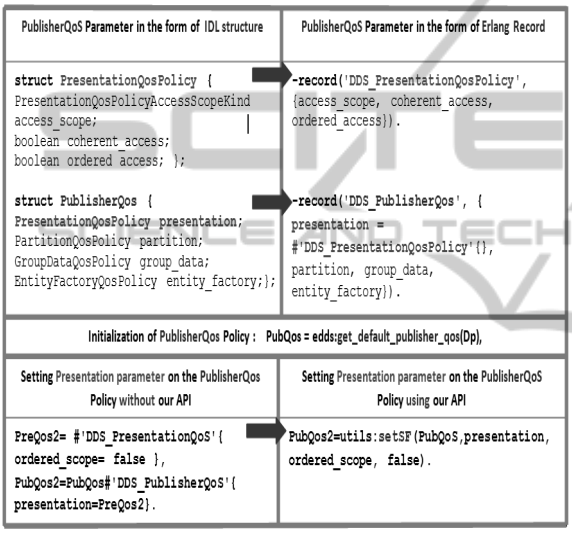

Figure 6: Setting the Presentation parameter on the QoSPublisher policy.

4.3 Benefit of Dynamic Typing: Generic DataWriter

In computer science, a data type, defines the possible values that can be given to the data and the operators that can be applied on. Dynamic typing is a characteristic for many programming languages such as Erlang. In this paradigm, the verification of types does not happen at the compilation step but it is left at run time. This simplest level provides lower development costs and flexible programming. It is important to note that this is different than typeless: both dynamically and statically typed programming language are typed (Tratt, 2009).

In contrast to other implementation of DDS (Java, c, c++) that require to create a specific DataWriter (resp DataReader) for each new data type. On eDDS, we took advantage of the possibility to abstract data types given by Erlang to develop only one generic

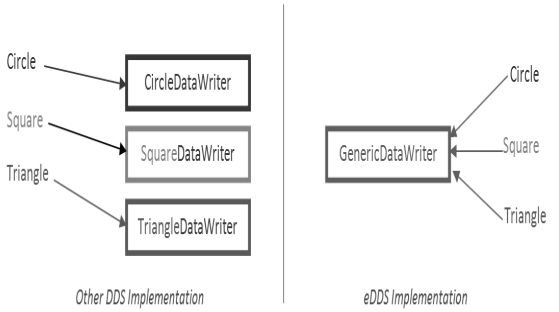

Figure 7: Generic DataWriter.

DataWriter (resp one generic DataReader) for all data types. More precisely, this DataWriter (resp this DataReader) has a generic ”write” (resp a generic ”read) function by with an application can write (resp read) any data for any data type. The participant application should specify the desired data type as an argument when creating the DataWriter or the DataReader. In the next example, we present how to create DataWriter for the cricle data type respectively in eDDS and in OpenDDS (a C DDS implementation) (Object Computing Inc (OCI), OpenDDS,).

- Generic DataWriter in eDDS:

```
Datawriter=edds:create_datawriter(Pub,
DP,CircleTopic,Qos,Listener,Mask),
edds:write(DataWriter,circlevalue).
```

- Specific DataWriter in OpenDDS:

```
DDS::DataWriter_var writer = pub->
create_datawriter(circleTopic,QOS,Lis,Mask);
Circle::CircleDataWriter_var circle_writer
= Circle::CircleDataWriter::_narrow(writer)
circle_writer->
     write(circlevalue, DDS::HANDLE_NIL);
```

4.4 Inconvenient of Dynamic Typing: Typechecking Problem

4.4.1 Presentation of the Problem

As presented on the previous section, eDDS defines a generic DataWriter (resp DataReader) for all data types. However, having a single generic DataWriter not avoid the need to check the data types. Before sending any data (resp receiving any data) we must verify its type. That’s why, type checking of sent and received data is required. The next example present this type checking problem: ”write” function will accept any data type given as parameter. This is due to that the transmission of data is doing in the form of bit sequences.

```
Create a DataWriter that will send a Circle Value
Datawriter=edds:create_datawriter(Pub,DP,CircleTopic,Qos,Listener,Mask)
edds:write(DataWriter,Circlevalue).%% true
edds:write(DataWriter,Trianglevalue).%% true
edds:write(DataWriter,Squarevalue).%% true
```

Figure 8: Type checking problem.

4.4.2 The Solution: Check Function

Data types are represented as Erlang records. All data types used in the global data space are registered on a header file located on the server. However, each participated application has its local data type file that contains data types which will be used by this application to publish or receive data. We have created a validator which use the same parsing technique presented previously and generates automatically **check** functions that take on parameters the data and the type that will be checked(check (Data, Type)). These functions can be called by the function write or read to check data before make it available to the user: data type must be the same defined on the DataWriter (or the DataReader). The two Figures "7" and "8" present the principals steps to generate this function. We take for example the TempSensor data type.

```
-record(sensor,{
        id:: integer(),
        temp :: integer(),
        press :: integer()}).
```

In fact, the usefulness of this parsing technique can not end here. There are an other type of checking function based on this principe:

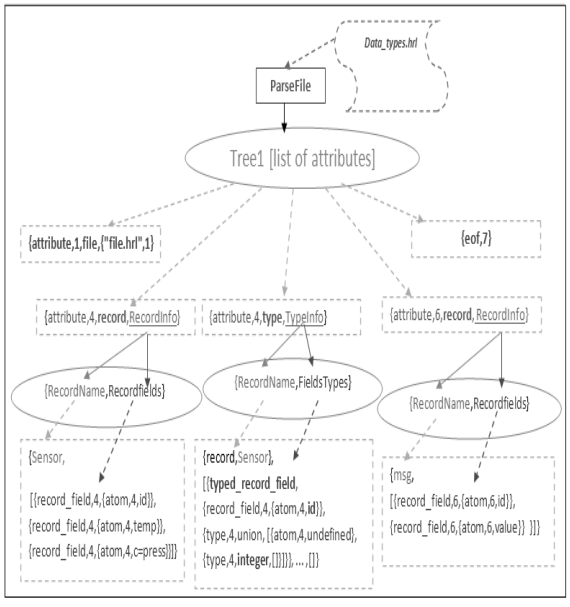

Figure 9: Generate check function (step1).

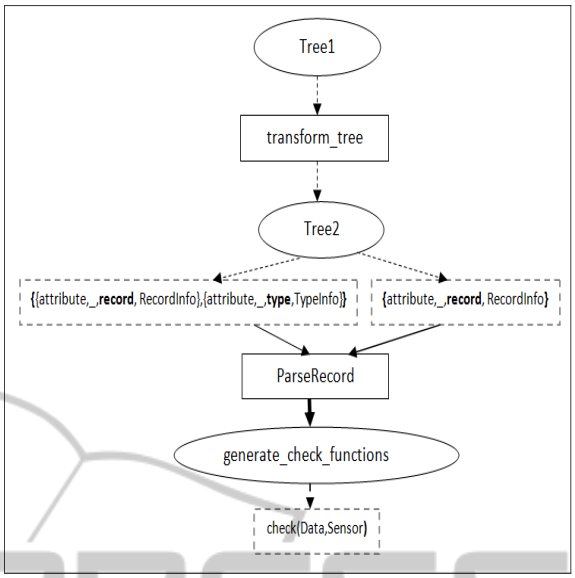

Figure 10: Generate check function (step2).

check_type_exists(name, HeaderFile) function to check whether the type name given in parameter already exists or not. Before creation of any new data type, the system must verify if there is already data type have the same name. This function is used also when creating a new topic and we want to verify if data type name used is registered in the local DataType_file.

```
check_type_exists(Name, HeaderFile) ->
L=get_recordnames(HeaderFile),
find_element(Name,L).
```

4.4.3 Evaluation

In this section we will show the importance of using the check function in our eDDS API to make Erlang based DDS applications more reliable. We present below the differences between the two cases: an application d'ont use an explicit type checking and an other that will use an explicit type checking.

- **Without Explicit Type Checking.** In the next example we want to show the risks that can meet developers of eDDS applications without checking the type of data that will be sent or received. In Publication Side, We create a CircleTopic that uses the Circle data type and we associate to it a DataWriter. We did the same thing in the subscription side and we associate a DataReader to the CircleTopic. As shown in this example, we did not sent a circle as expected but a square sample. We notice that no error is marked in the development stage or in the phase execution: The square date is received by the DataReader.

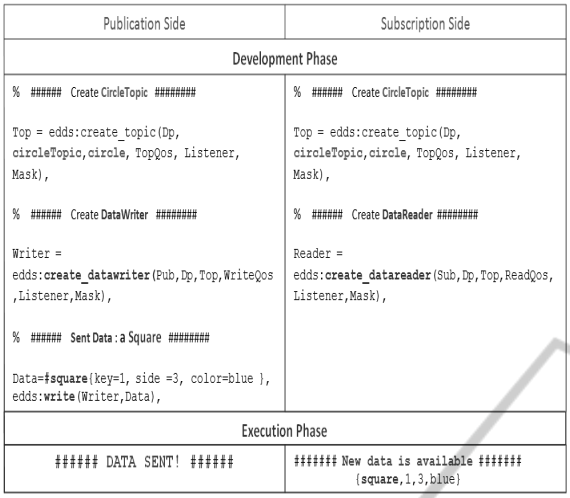

Figure 11: Sending data without explicit type checking.

- **With Explicit Type Checking.** In this case we will verify the data before publication. We modified the write function like presented below. It must recover the data type used by the concerned datawriter and then we call the check function and passing as parameters the type name and the data. If check function returned true, the data will be send to the datareader. If not, an error message will be displayed.

```
edds: write (Writer,Data)->
%...
if (check(Data,Type) == true) ->
'DDS_DataWriter':write(Writer,Data);
true-> io:format("\n ### Invalide Data ###")
end.
```

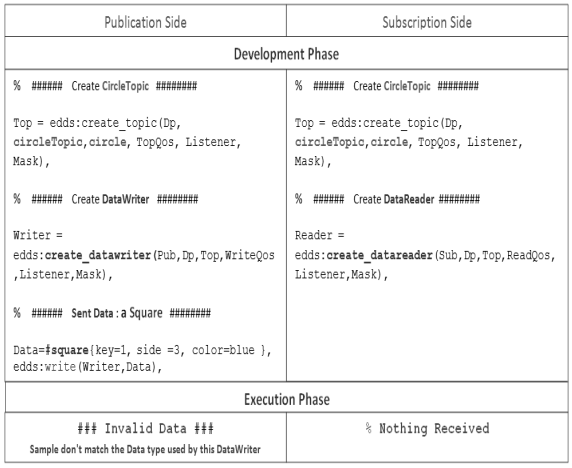

Figure 12: Sending data with explicit type checking.

Like the previous example, We create a DataWriter and a DataReader associated to the CircleTopic and we write a square data instead of a circle data. As noticed below, this data has not published and nothing received on the subscription side. For more reliability, it is recommend to make the same modification of the write function in the read function on the subscription side to check the received data before making it available to the application: the data should respect the type of the concerned Datareader.

4.4.4 Discussion

While DDS data types are represented in Erlang as records, it is logical to think of the BIF is_record to verify these types. A simple example presented below shows the difference between check and is_record functions. This BIF handles only the simple case of records. It takes as parameters the data and a record name and returns a boolean value to indicate if this data is an instance of this record or not. But this BIF suffers same drawbacks: it does not work correctly if in the record declaration we specified also the types of record fields. Our check function is more accurate. It will solve this problem. We can take for example the Sensor record defined previously.

```
Data1 =#Sensor{i d =1 , temp =30 , press =10}
Data2 =#Sensor{i d =2,temp = temperature,press=10}
is_record(Data1,Sensor)     %% true
is_record(Data2,Sensor)     %% true
check(Data1,Sensor) %% true
check(Data2,Sensor) %% false
```

In an other hand, this aspect of dynamic typing and the fact that not having the verification type during the compilation step is also recommended by DDS. In 2012, the OMG bring out an extensible specification of DDS just to adapt this principe (Object Management Group , 2012). They motioned that is preferable to have a dynamic API that allows type definition, as well as publication and subscription data types without compile-time knowledge of the schema.

5 RELATED WORK

- **Erlang Type Checking Systems**
  There are many attempts and approaches to enhance Erlang with type checking system, some of them are well known on the Erlang community. (Frank Huch, 2001) presented an approach for the formal verification of Erlang programs using abstract implementation and model checking; (Chanchal Kumar Roy, Thomas Noll and Banani Roy, 2006) provide a contribution to the formal modeling and verification of programs written in

Erlang: mapping to the lambda calculus; (Thomas Noll, Lars ake Fredlund and Dilan Gurov, 2002) are developed the Erlang Verification Tool; *Dialyzer (Erlang, 2007)*: A Discrepancy Analyzer for Erlang program which is used in various telecommunication projects. It is a defect detection tool that use a static analysis to detect anomalies+ in the Erlang code. It's part of the Erlang/OTP distribution since 2007; *Typer (Tobias Lindahl Konstantinos Sagonas, 2005)*: An automatic type annotator for Erlang program based on type inference of success typing. It check specified type against the inferred type. It is part of Erlang/OTP since 2008.

- **The Scala API for DDS**
  Scala stands for scalable langage, it is new language (2003) that blend Object oriented and functional constructs into a statically strongly typed language with sophisticated type inference. (Corsaro, 2012) created a scala API for DDS. In contrast of our eDDS that completely based on Erlang, Escalier is based on the Open Splice Middleware. Also This API is very simple to use specially when creating DDS entities and setting Qos parameters. Like the principe of eDDS, this approach used the notion of Type parameterized of Scala (Martin Odersky, Lex Spoon, and Bill Venners, 2008) to create a generic DataWriter (resp DataReader). This principe alows to define many specifics types with one generally written class. If we take the same example of TempSensor: the user can write any data not only those its type is TempSensor. So there is a problem of verification of data that will be transmitted.

6 CONCLUSIONS

To facilitate the development of distributed real time systems that require various quality of service aspects, such as predictable performance, secure communications, availability and fault tolerance. An Application Programming Interface is required to abstract the platform specific details of the underlying QoS implementation. This paper shows how to create a simple DDS API by taking advantage of a functional programming language such as Erlang to define and set QoS policy. In fact, we used a record parsing technique to generate automatically the desired API. This technique is adopted in many area in eDDS specially to solve Erlang problem type checking caused by its dynamically typed characteristic in order to respect the nature of DDS that provides a strong typed distributed system. As a future work and in order to improve our API, we can make the possibility to add or remove fields in a given data type like presented in the Extensible and Dynamic Topic Types for DDS specification.

REFERENCES

Ericsson Computer Science Laboratory (1980). ERLANG programming language. http://www.erlang.org. Accessed: 24-07-2014.

Cesarini, F. and Thompson, S. (2009). *Erlang programming*. O'Reilly.

Chanchal Kumar Roy, Thomas Noll and Banani Roy (2006). Towards Automatic Verification of Erlang Programs by Lambda Calculus Translation.

Corsaro, A. (2012). High performance distributed computing with DDS and Scala. In *PrismTech Corp*.

Corsaro, A., Querzoni, L., Scipioni, S., Piergiovanni, S. T., and Virgillito, A. (2006). Quality of Service in Publish/Subscribe Middleware. *Chapter in Global Data Management*.

Emmanuel Chailloux, Pascal Manoury and Bruno Pagano (2001). *Developing Applications With Objective Caml*. Editions O'REILLY.

Erlang (2007). The DIscrepancy AnaLYZer for ERlang applications. http://www.erlang.org/doc/man/dialyzer.html.

Frank Huch (2001). Verification of Erlang Programs using Abstract Interpretation and Model Checking. In *Proceeding ICFP '99 Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 261 – 272.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411.

Martin Odersky, Lex Spoon, and Bill Venners (2008). Programming in Scala. http://www.artima.com/pins1ed/.

Object Computing Inc (OCI), OpenDDS. http://www.opendds.org.

Object Management Group (2007). Data Distribution Service for Real-time Systems Specification. version 1.2.

Object Management Group (2012). Extensible and Dynamic Topic Types for DDS. version 1.0.

Object Management Group (2012). Common Object Request Broker Architecture (CORBA). http://www.omg.org/spec/CORBA/3.3. version 3.3.

Thomas Noll, Lars ake Fredlund and Dilan Gurov (Springer-Verlag 2002). Erlang verification tool.

Tobias Lindahl Konstantinos Sagonas (2005). TYPER: A Type Annotator of Erlang Code.

Tratt, L. (2009). Dynamically typed languages. *Advances in Computers*, 77:149–184.

Trung (2009). Record introspection at compile time. http://erlangexamples.com/tag/record/. Accessed: 28-11-2014.