QoS-Aware Publish-Subscribe Service for Real-Time Data Acquisition*

Xinjie Lv[1,4], Xin Li[3], Tian Yang[1,4], Zaifei Liao[1,4], Wei Liu[1], Hongan Wang[2]

1:Intelligence Engineering Lab., Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
2:State Key Lab. Of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
3: Department of Computer Science and Technology, Shandong University, Jinan 250101, China
4: Graduate School, The Chinese Academy of Sciences, Beijing 100049, China

xinjie05@ios.cn, lx@mail.sdu.cn, marsyang@vip.sina.com, liaozaifei@gmail.com,water_wei@263.net, hongan@iscas.ac.cn

Abstract: Many complex distributed real-time applications, monitoring and controlling the external environment, require sophisticated processing and sharing of an extensive amount of data under critical timing constraints. In this paper, we present a comprehensive overview of the Data Distribution Service standard (DDS) and describe its QoS features for developing real-time applications. An overview about an active real-time database (ARTDB) named Agilor are also provided. For efficient describing QoS policy in Agilor, a Real-time ECA (RECA) rule model is brought forward based on common ECA rule. And then we propose a novel QoS-aware Real-Time Publish-Subscribe (QRTPS) service compatible to DDS for distributed real-time data acquisition. Furthermore, QRTPS is implemented on Agilor by expressing QRTPS with objects and ECA rules in Agilor. To illustrate the benefits of QRTPS for real-time data acquisition, an example application is presented.

Keywords: QoS, Real-Time Publish-Subscribe, ECA Rule, Active Real-Time Database

Topic: Data capture in real-time **Category:** Industry Paper

1. Introduction

Many complex distributed real-time applications require sophisticated processing and sharing of an extensive amount of data under critical timing constraints. These applications involve acquiring data from the environment, processing acquired data in the context of data acquired in the past, and providing timely response. How to

* This work was supported in part by the National High Technology Development Program of China under Grant No. 2006AA04Z182.

transmit or disseminate data timely and exactly for distributed real-time applications is a noticeable problem as yet.

Image that we receive sensor data from a mine in a colliery periodically, these data might contain information about gas, temperature, smog, etc. of every *Observation Point*. The system is required to disseminate these data under specific timing constraints for the following abnormal scene detection. Hereinto, publish-subscribe model is very suitable for such repetitive, time-critical data distribution. A limitation of most existing architectures that support publish-subscribe is their limited support for the expression and enforcement of Quality of Service (QoS) parameters (such as required bandwidth or latency, for instance). This observation applies both to models, such as the CORBA Event Service [2], CORBA Notification Service [1], Java Message Service [3] and to systems, such as CEA (Cambridge Event Architecture) [4], Distributed Asynchronous Collections [9], SIENA (Scalable Internet Event Notification Architectures) [8] or Cayuga [27]. This is a significant drawback, since QoS features are an important component of applications, and its use and support has been widely studied in the context of direct communication [5,6,7,10,22,24,25].

Data Distribution Service *(DDS)* is a newly adopted specification from the Object Management Group (OMG). *DDS* is aimed at a diverse community of users requiring data-centric publish-subscribe communications. *DDS* departs from previous approaches in two primary aspects: (1) enumerating and providing formal definitions for the QoS (Quality of Service) settings that can be used to configure the service, and (2) the tight binding of a "topic" to a data-type, thus making it more than just a "routing" label. The coupling of "topic" with a data-type, along-with the additional QoS settings enables implementation optimizations such as pre-allocating the resources needed to send or receive a "topic" [11].

To meet the requirements of real-time and active capabilities described in QoS policies of *DDS* [15], we introduce active real-time databases (ARTDB) [16] to implement these QoS policies. The ARTDB [17] is proposed to provide both active and real-time capabilities. In the context of an ARTDB, data distribution can be implemented via ECA rules, and applications can consume data at specific rate on specific condition. So it would be desirable to develop a QoS-aware Real-Time Publish-Subscribe (QRTPS) system on ARTDB. This system infrastructure should be efficient, scalable, flexible, and cater for the architecture of active real-time database system, except for providing real-time predictability.

In this paper, we introduce high-level design of an ARTDB named *Agilor*, and its active object model to incorporate timing and active features into object-oriented data model. For expressing QoS policies, we propose a Real-time ECA rule model (*RECA*). And we have implemented the QRTPS with *Agilor* on Windows 2003, which is a commercially available, general-purpose operating system and being used more and more for real-time control applications. However, particular challenges come up against for achieving QoS policies using *Agilor*.

The remainder of this paper is organized as follows. The next section introduces high-level design and active object model of *Agilor* as background, for further incorporating QRTPS into *Agilor*. In section 3, we propose Real-time ECA (*RECA*) rule model for expressing QRTPS. And an implementing QRTPS service over *Agilor* is presented in section 4. Section 5 discusses a simplified example application to

illustrate some of the previously mentioned features of QRTPS. We conclude this paper and present future work in section 6.

2. Background

2.1 Overview of *Agilor*

The *Agilor* architecture [26] as in Fig.1 consists of some kernel modules and critical services.

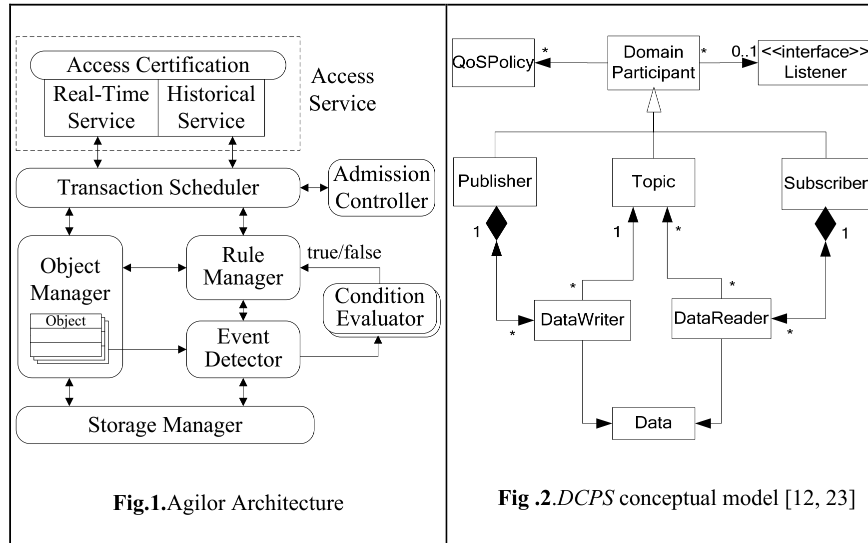

Fig.1. Agilor Architecture

Fig .2.*DCPS* conceptual model [12, 23]

The *storage manager* takes charge of persistent objects and rules storage on disks and support read/write interfaces. The *object manager* and *rule manager*, resident in main-memory, are responsible to add/delete/update objects and ECA rules separately. The *transaction scheduler* deals with all transactions and access objects through interface in the *object manager*.

The *rule manager* not only stores rules into the rule-base, but also performs rule processing by the *event detector* and *condition evaluator*. The *rule manager* also submits actions and necessary parameters (e.g. deadline, worst-case execution time) to the *transaction scheduler*. The *event detector* monitors events occurred in database and system. The *condition evaluator* checks whether specific conditions are satisfied on receiving events from *event detector*. When conditions are satisfied, relevant actions will be submitted by *rule manager* to *transaction scheduler* to execute.

The *admission controller* ensures that admitted transactions do not overburden the system. This module inspects whether to accept or reject a new transaction based on a feedback mechanism considering resources and workload in system and the importance of the transaction.

The *data access services* consist of *real-time service*, *historical service* and *access authenticate*. They support retrieval of historical data as well as real-time data (synchronous mode or subscription mode) under time constraints. In these services, real-time publish-subscription service provides push mechanism based on ECA rules. The *access certification service* ensures that access is provided only to entitled applications. An important building block of the *Agilor* is ECA rule and we will discuss its extension edition Real-time ECA in section 3.

2.2 Conceptual Model of *DDS*

DDS describes two levels of interfaces:

- A lower *DCPS (Data-Centric Publish-Subscribe)* level that is targeted towards the efficient delivery of the proper information to the proper recipients.
- An optional higher *DLRL (Data Local Reconstruction Layer)* level, which allows for a simple integration of the Service into the application layer.

We restrict our discussion of *DDS* to the *DCPS* layer. Fig.2 illustrates the overall *DCPS* model, which consists of the following entities: *DataWriter, DataReader, Publisher, Subscriber,* and *Topic*. All these classes extend *DomainParticipant*, representing their ability to be configured through QoS policies, be notified of events via listener objects, and support conditions that can be waited upon by the application. Each specialization of the *DomainParticipant* base class has a corresponding specialized listener and a set of QoS Policy values that are suitable to it.

2.3 Supported QoS of *DDS*

The *DCPS* entities in *DDS* include *Topics*, which describe the type of data to be written or read; *Data Readers*, which subscribe to the values or instances of particular topics; and *Data Writers*, which publish values or instances for particular topics. Various properties of these entities can be configured using combinations of the 22 QoS policies. Moreover, *Publishers* manage groups of *data writers* and *Subscribers* manage groups of *data readers*. Table 1 summarizes all the *DDS* QoS policies. Each QoS policy has several attributes with the majority of the attributes having a large number of possible values, e.g., an attribute of type long or character string. Moreover, not all QoS policies are applicable to all *DCPS* entities, nor are all combinations of policy values semantically compatible [19, 23].

Table 1. *DDS* QoS Policies [19]

***DDS* QoS Policy**	**Meanings**
User_Data	Attaches application data to *DDS* entities
Topic_Data	Attaches application data to topics

Group_Data	Attaches application data to *Publishers*, *Subscribers*
Durability	Determines if data outlives the time when written or read
Durability_Service	Details how durable data is stored
Presentation	Delivers data as group and/or in order
Deadline	Determines rate at which periodic data is refreshed
Latency_Budget	Sets guidelines for acceptable end-to-end delays
Ownership	Controls *writer(s)* of data
Ownership_Strength	Sets ownership of data
Liveliness	Sets liveness properties of *topics*, *data readers*, *data writers*
Time_Based_Filter	Mediates exchanges between slow consumers and fast producers
Partition	Controls logical partition of data dissemination
Reliability	Controls reliability of data transmission
Transport_Priority	Sets priority of data transport
Lifespan	Sets time bound for “stale” data
Destination_Order	Sets whether data sender or receiver determines order
History	Sets how much data is kept to be read
Resource_Limits	Controls resources used to meet requirements
Entity_Factory	Sets enabling of *DDS* entities when created
Writer_Data_Lifecycle	Controls data and *data writer* lifecycles
Reader_Data_Lifecycle	Controls data and *data reader* lifecycles

3. Real-Time ECA

ECA Rules are used to specify constraints that define correct states of objects as well as actions to be taken on certain events. To efficiently describe QoS policy in *Agilor*, a Real-time ECA (*RECA*) rule model is proposed, which extends common ECA [26] in complicated temporal events and composite conditions. The *RECA* rule model is divided into an event part, a condition part and an action part.

Formally, a rule is modeled by <*RN, RD, RV, E, C, A, CMEC, CMCA, CMEA, CMRR*>, in which *RN, RD, RV* is the name, deadline and value of the rule separately. The deadline reflects urgency of the rule (including the action) while the value reflects importance of the rule. The value decides the order of condition evaluations when multiple rules are triggered at the same time. *E* is a set of events that can invoke rules, *C* is a set of conditions and *A* is a set of ordered actions. Actions should be taken when specific conditions are satisfied. *CMEC* and *CMCA* are the coupling modes between event and condition evaluation and between condition and action execution separately, i.e. when condition evaluation and action execution can take place relative to the time of triggering events. *CMEA* and *CMRR* are the coupling modes between event and action execution and between two rules [20,21].

3.1 Events

Events are occurrences of interests which are predefined in the system such as data update events and clock events, Events can be divided into primitive events, which refer to simple and atomic events, and composite events, which consist of primitive events combined with event operators.

Events can be described formally as follow:

$$E ::= p \mid (\neg E) \mid (O_{[t1,t2]}E)(E_1 \wedge E_2) \mid (E_1 \vee E_2) \mid (E_1 O_{[t1,t2]} E_2) \mid (E_1 O_{\leq t2} E) \quad \textbf{(1)}$$
$$\mid (O_{[t1,t2]}(E_1 \rightarrow E_2)) \mid (O_{[t1,t2]}(E_1 \wedge E_2)) \mid (O_{[t1,t2]}(E_1 \vee E_2))$$
$$\mid (O_{[t1,t2]}((E_1 \rightarrow E) \wedge (E \rightarrow E_2)))$$

The predication p is a primitive event. The symbol '¬',' ∧ 'and' ∨ ' stand for negation(not), conjunction(and), disjunction(or) operator separately. $E_1 \mathrm{O}_{[t1,t2]} E_2$ is a sequence of events to occur over a time interval [t1,t2], in which the latter event E_2 must occur after the former event E_1 between [t1,t2]. The composite event arises when the last event in the sequence has occurred.

Three kinds of primitive events are realized in *Agilor* and they are

(1) *system events*: some particular events of operating system or database system, e.g. OnTimer, OnIOError,

(2) *method events*: data manipulation events of an object/class, e.g. OnUpdate and OnDelete, and

(3) *custom events*: events predefined by user's application for specified purpose, such as sensor failure event. Custom events are always triggered explicitly by application calling RaiseCustomEvent() function.

The method events are significant events in ARTDB and any data manipulation event of an object/class can be a potential method event. Method events will be triggered automatically when the corresponding method is invoked.

A method event can be defined by 6-tuple <*EN, T, OM, CMME, EP, SL*>, where *EN* is the name of event, *T* is the time of occurrence, *OM* is the name of object method which should be one of the existing methods in object base, and the coupling mode *CMME* is an indication of whether the event should be generated before or after the execution of the method. *EP* is the parameters of the event corresponding to the parameters of the method, i.e. *MP*, which will be passed to condition evaluator for check. *SL* is a subscribers list made up of the rules and composite events which subscribe the event. The subscribers of an event are notified when the event occurs.

It is important to associate real-time objects with events for triggering mechanism. The object-oriented data model is enhanced by incorporating primitive event detection as part of object management. In order to express more complicated temporal events, we extend common ECA rules as in Table 2. For complicated temporal events scan, a useful approach has been to adopt Nondeterministic Finite Automata (NFA) to represent the structure of an event sequence [28, 29]. Furthermore, the NFA-based approach can be extended to handle sequence construction, as proposed in YFilter [28] in the context of XML message filtering.

Table 2. Extension of Complicated Temporal Events

Primitive temporal events		
Event type	**Definition**	**Semantics**
Durative event	a specific event E occurring in a specified interval limited by two time instants X, Y	$O_{[X,Y]}E$
Composite temporal events		
Event type	**Definition**	**Semantics**
Time constrained sequence	The time constrained sequence of event E1 and E2, E1 Seq-within[X minutes] E2, occurs when both E1 and E2 have occurred in that order within X minutes. Other events may occur between E1 and E2.	$E_1 \rightarrow O_{\leq X} E_2$
Durative sequence	The durative sequence of events E1 and E2, E1 Seq-During[X,Y] E2, occurs when both E1 and E2 have occurred in that order in a specific interval from time instant X to Y. Other events may occur between E1 and E2.	$O_{[X,Y]}(E_1 \rightarrow \Diamond E_2)$
Durative conjunction	The durative conjunction of event E1 and E2, E1 AND-During[X, Y] E2, occurs when both E1 and E2 have occurred in any order during a specific interval from time instant X to Y with possible other events in between.	$O_{[X,Y]}(E_1 \wedge E_2)$
Durative disconjunction	The durative disconjunction of events E1 and E2, E1 OR-During[X,Y] E2, occurs when either E1 or E2 occurs or when both E1 and E2 occur during a specific interval from time instant X to Y	$O_{[X,Y]}(E_1 \vee E_2)$
Durative between	The durative between of Event E1 and E2, denoted as Between-During (E1, E2)[X,Y], occurs when there are events occur between the initiating event E1 and terminating event E2 in a given interval limited by starting time X and ending time Y, ignoring the relative order of their occurrences.	$O_{[X,Y]}((E_1 \rightarrow \Diamond E) \wedge (E \rightarrow \Diamond E_2))$

3.2 Conditions

The event indicates the need to check; whereas the condition determines what to check. The condition set C describes the situations that are used to check whether all prerequisites are satisfied for actions.

Conditions can be described formally as follow:

$$C::=p|(\neg C)|\ (C_1 \wedge C_2)|(C_1 \vee C_2) \qquad \textbf{(2)}$$

The predication p is a primitive condition and in *Agilor* it can be

(1) *Selection condition*: evaluation of a single attribute value of one object (e.g. OP.Gas>20),

(2) *Aggregation condition*: comparison of a single attribute aggregated over multiple instances (e.g. Max(OP. smog)>100),

(3) *Join condition*: comparison of a single common attribute of multiple homogeneous objects (e.g. OP1 Pressure= OP2.Pressure),

(4) *Transition condition*: comparison of a single attribute over multiple instances (e.g. OP1. Gas> OP1.GetLast (Gas)), and

(5) *Application-specific condition*: evaluation of functions predefined by applications.

Applications also can define composite conditions by combining a set of primitive conditions with logical operators such as disjunction and conjunction.

3.3 Actions

The action set A defines a set of ordered actions, which are similar to the definition of methods in object model. Actions could be database operations including deletion and update, as well as external actions such as procedure call (e.g. publish data or signal an alarm to the applications). Deadline as an additional parameter should be assigned to the action. It is relative to the occurrence time of the triggering event. For example, a triggered action must be finished in 10 milliseconds after a temporal attribute X is updated.

3.4 Coupling Modes

The *CMEC, CMCA, CMEA* and *CMRR* identify the time semantics when condition evaluation and action execution can take place relative to the triggering event, with the constraint that condition evaluation must be performed before action execution.

Table 3. Coupling Modes Definition

Coupling Mode Name	Optional Value	Meanings
CMEC	immediate	When events occur, the transaction is suspended, and condition evaluation is performed immediately
	detached	Condition evaluation is done in another transaction.
CMCA	immediate	The triggered action is executed immediately after condition evaluation.
	detached	The triggered action is treated as a new separate transaction.
CMEA	immediate	The triggered action is executed immediately after event occurs.
	detached	The triggered action is treated as a new separate transaction.
CMRR	immediate	Execution of one rule immediately triggers another rule.

	concurrent	Many rules may be triggered at the same time.

To avoid unpredictable increase of the execution time of the triggering transaction, in a real-time environment, the combinations of *CMEC* and *CMCA* had better be immediate-detached or detached-immediate. Similarly, the limit is put on the depth of triggered rules to avoid uncontrolled cascade triggering.

3.5 Semantic for *RECA* Rules

In *Agilor*, deadline and value are added into every *RECA* rule and the semantic for rules is defined as follow:

Rule::=BEGIN RULE <*RuleName*>
VALUE <*Value***>**
WHEN <*Event*>
IF <*Condition*> **CMEC** [*immediate*|*detached*] **CMEA** [*immediate*|*detached*] **THEN**
<*Action*> **WITHIN** <*Deadline*>
CMCA [*immediate*|*detached*]
CMRR [*immediate*|*concurrent*]
END RULE

The basic structures of the rules in *Agilor* represent triggering events, conditions and actions, as well as the timing constraints and coupling modes. *RECA* examples would be described in section 5.

4. Expressing QRTPS

4.1 *Agilor-DDS* Equivalents

A map of key *Agilor* concepts and terminology and the *DDS* equivalents is summarized below.

Table 4. *Agilor-DDS* Equivalents

Agilor	DDS
Client	Application
Provider = client library + server	Middleware, Service
Plant	Domain represents a global data space, comprised of a set of communicating user applications
Connection Connect()	DomainParticipant enable()
Thread	Publisher, Subscriber

Topic(Issue) A named physical resource that gathers and disseminates messages addressed to it	Topic An abstraction with a unique name, data-type, and QoS, used to connect matching DataWriters and DataReaders
TagValue	DataSample
IssueProducer	DataWriter
IssueConsumer	DataReader

4.2 Framework and Sequence of QRTPS

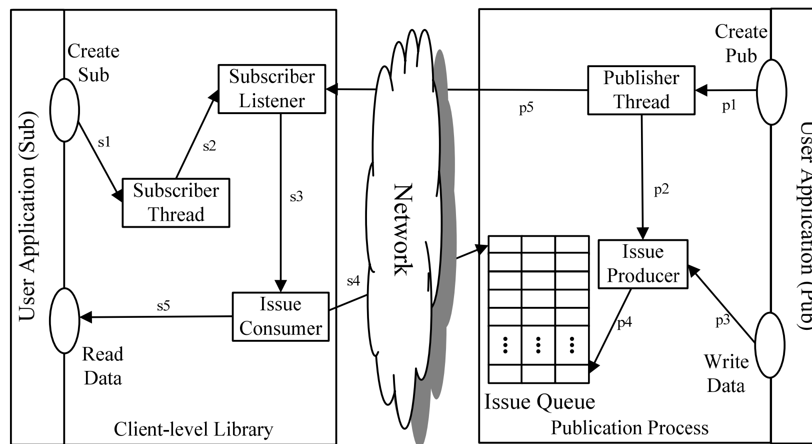

Fig.3. Framework and Sequence of QRTPS Service

In Fig.3, p1 to p5 denote the steps in a publication and s1 to s5 express the processes in a subscription.

The p1, p2 of Fig.3 shows the creation of the *Publisher* and *IssueProducer* respectively. The p3, p4 shows the process of user application writes data to *IssueProducer* and the process of *IssueProducer* writes data to *Issue Queue*. The p5 step shows that the corresponding notifications are propagated according to the current *Publisher*'s policy regarding sending.

The s1 of Fig.3 shows the *Subscriber*'s creation. The s2 shows the use of a *SubscriberListener*: It must first be created and attached to the *Subscriber*. Then when notification arrives, it is made available to each related *IssueConsumer*. Then the *SubscriberListener* is triggered (s3). The application must then get the list of affected *IssueConsumer* objects; then it can read (s4) the data directly from *Issue Queue*. The s5 shows the process of user application read data from *IssueConsumer*.

5. Case Study

This section aims to illustrate some of the previously mentioned features of QRTPS with a simplified example application. It was developed using the commercially available QRTPS on *Agilor*. The example uses the QRTPS for the implementation of a sensor-based active monitoring system. A simplified overview of the proposed system is shown in figure 4.The system consists of five *Observation Points* (*OP*) in a coal mine and there are many sensors in each *OP* to measure different indicators. For the sake of simplicity, we consider three sensors (e.g. gas, temperature, smog) on each observation point. It is further assumed that these sensors are able to convert the signals coming from the sensors into meaningful information which is then published via QRTPS.

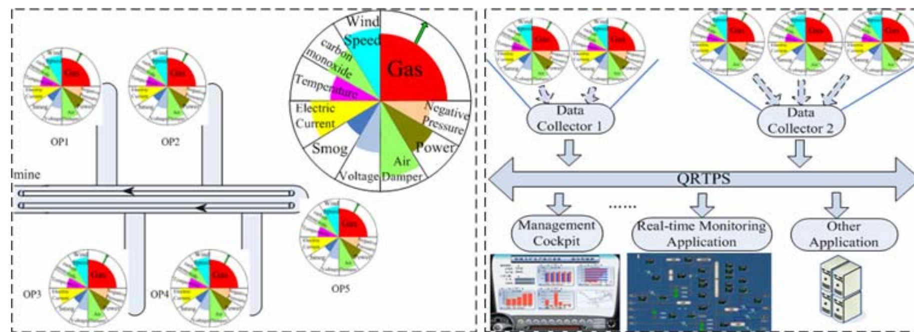

Fig.4. Overview on the example application

In order to demonstrate some features of QRTPS, we define the following requirements for the overall system:

1. At the initial start-up each data collector publishes meta information about its sensors in order to allow subscribing applications to interpret the sensor data. Late-joining applications shall receive this information automatically.
2. Real-time Monitoring Application shall receive gas sensor readings every 500ms and temperature readings every 1000ms, regardless how fast the data collector publish these information.
3. The data collector and connected applications only consider the temperature readings of *OP1* at a time. If no data from this most-trusted sensor source is received within 4000ms, temperature data shall automatically be received from another *OP*, allowing a seamless failover.

We concentrate on the data model and the definition of the QoS-settings to meet the requirements based on QRTPS.

5.1 Related Data Structures

The development of a data-centric application starts with the definition of the data structures that shall be exchanged between the applications. In our case, we create two data structures for each sensor type. The first data structure contains the actual sensor data to be transferred during the assembly process. It contains fields to uniquely identify the sensor and a field for the current sensor reading. In order to interpret the sensor values correctly, subscribers need additional meta information. These details are modeled in an additional structure and only need to be published when the application starts or if sensors are exchanged. This way, sensor data and its interpretation are separated and network load can be reduced significantly. Subscribing applications can be dynamically reconfigured to accurately interpret incoming data from different sensors. As an example, the data structures for gas sensors are shown below according to the Interface Definition Language (IDL). The structure *Gas* is used to transmit the sensor readings, whereas *GasSensorInfo* contains meta-information to interpret the sensor data correctly. For the other sensor types the data structures are defined accordingly.

```
Class Gas{
  private:
  long datacollector_id;
  long observationpoint_id;
  double value;
};
```

```
Class GasSensorInfo{
  private:
  long datacollector_id;
  long observationpoint_id;
  MeasuringUnit unit;
  double maxGas;
  double minGas;
  double sampleRate;
};
```

These classes have to be used by the application programs that implement the *data collector* and any other participating system like Real-time monitoring application.

5.2 Settings of Quality-of-Service Parameters

In order to tailor the data transfer according to the requirements, the QoS parameters for each *IssueProducer* and *IssueConsumer* need to be configured in the *RECA* rules. Table 5 shows for each requirement the QoS settings for the *data readers* on the subscribing side and *data writers* for publishing applications. When a *data collector* executes its initialization, it retrieves the meta-data for each sensor from a configuration file and creates the corresponding data structures. The *IssueProducer* for these data types are configured according to table 5 and each *data collector* publishes this information only once. After this publication, it can initialize the publishers for the actual sensor data transmission and start publishing the sensor reading. Subscribing applications like management cockpit or the real-time monitoring application first initialize the subscribers for the meta information and hence retrieve these data before they start subscribing to the actual sensor data. Since the QoS parameters are set to HISTORY.depth=1 and DURABILITY.kind= TRANSIENTLOCAL, it is assured by the middleware that even late-joining applications will get the necessary information to interpret all sensor readings. This

approach significantly saves network resources, since meta information is only transferred when it is necessary. QRTPS allows the implementation of this feature with only minor programming effort, whereas in traditional distributed systems providing late-joining applications with historical data is an error prone and complex task. The same argument applies for the second requirement. Monitoring applications can limit the number of sample readings simply by setting the time based filter QoS policy on their data readers. In this way, they can be protected from being flooded with too much data. In traditional client-server based applications, this "impedance mismatch" is a major problem. QRTPS overcomes this problem by the simple setting QoS parameters.

Table 5. QoS Settings for the Application

Requirement	Data Entity	QoS Policy of IssueConsumer	QoS Policy of IssueProducer
1	*GasSensorInfo, TemperatureSensorInfo, SmogSensorInfo*	HISTORY.depth = 1 RELIABILITY.kind=RELIABLE DURABILITY.kind =TRANSIENT_LOCAL	HISTORY.depth = 1 DURABILITY.kind =TRANSIENT_LOCAL
2	*Gas* *Temperature*	TIME_BASED_FILTER =500ms TIME_BASED_FILTER =1000ms	
3	*Temperature*	OWNERSHIP.kind=EXCLUSIVE OWNERSHIP_STRENGTH.value = datacollectorid of the data collector DEADLINE.period.sec = 5	OWNERSHIP.kind =EXCLUSIVE DEADLINE.period.sec = 5

The third requirement allows subscribing applications to get temperature readings only from one most-trusted sensor. If this sensor stops working because of damage or other reasons, the applications shall automatically use the readings from a temperature sensor of another data collector. This automatic and dynamic failover to a backup sensor would require enormous programming effort if implemented manually. With QRTPS we have to set the OWNERSHIP.kind parameter to "exclusive" to ensure that readers will only receive data from a single sensor. Additionally, each temperature data writer is configured with the identifier of its corresponding data collector, resulting in a hierarchical order. The data readers in each subscribing application will only receive temperature readings from the data collector with the highest identifier. In this context, the DEADLINE QoS specifies that the subscribers will automatically failover to the sensor of the data collector with second-highest ID if it does not receive data within the specified time period. This way, fault-tolerant distributed applications can easily be developed with the ability to dynamically react to failures in the system.

5.3 Translating QoS Settings to *RECA*

Each requirement's QoS parameters can be translated into a *RECA* rule or a set of *RECA* rules coupled by *RECA*'s coupling modes (section3.4). Due to space limit, we translate QoS parameters only mentioned on *IssueConsumer* in Table 5 into *RECA* as following:

```
BEGIN RULE Rule_1        // Requirement 1
  VALUE   1
  WHEN OnStartUp("DataCollectorID")
  IF True   THEN
      global GasMetaInfo ggasmetainfo; //Using global variable
      ggasmetainfo. datacollector_id=1;
      ggasmetainfo .observationpoint_id=1;
      ……
      ggasmetainfo. sampleRate=5;
       WITHIN 1
END RULE
```

```
BEGIN RULE Rule_2        // Requirement 2
  VALUE   1
  WHEN OnTimer("Timer1 ",1)
  IF True   THEN
      variant currentvalue;
      currentvalue =TagValue("OP1.TempSensor1");
      ……
     WITHIN 1
END RULE
```

```
BEGIN RULE Rule_3        //Requirement 3
  VALUE   1
  WHEN OnTimer("Timer2",10)
  IF True   THEN
      variant currentvalue;
      long currenttime;
      long begintime;
      begintime = CurrentTime();
      while(TagState("TempSensor2")) {
          currentvalue =TagValue("OP1.TempSensor1");
          currenttime = CurrentTime();
         if((currenttime- begintime)>4) {
            currentvalue =TagValue("OP2.TempSensor2");
            ……
            break;
         }
      }
     WITHIN 1
END RULE
```

6. Conclusions and Future Work

In this paper, a new *DDS* compatible real-time service for data-centric publish-subscribe communication has been presented. The service is particularly targeting real-time applications which need to manage resource consumption and timeliness of the data transfer. QRTPS allows many-to-many communication and alleviates a number of common problems which are of particular interest for the development of distributed assembly systems. For example, with its sophisticated Quality-of-Service support, communication can be tailored according to the system requirements and typical challenges such as the delivery of historical data to late-joining applications are achieved automatically and in an efficient manner. Future work can look into adaptive adjusting QoS parameters during the course of execution in order to provide better performance with limited resources.

References

[1] Object Management Group, OMG Headquarters, 250 First Avenue, Suite 201, Needham, MA 02494, USA. Notification Service Specification, June 2000.

[2] Object Management Group, OMG Headquarters, 250 First Avenue, Suite 201, Needham, MA 02494, USA. Event Service Specification, March 2001.

[3] Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303, USA. Java Message Service, November 1999.

[4] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. IEEE Computer, March 2000.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differenciated services, December 1998. RFC 2475.

[6] Ed.R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP)—version 1 functional specification, September 1997. RFC 2205.

[7] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview, June 1994. RFC 1633.

[8] Antonio Carzaniga. Architectures for an Event Notification Service Scalable to Wide-area Networks. PhD thesis, Politecnico di Milano, December 1998.

[9] P. Th. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In In 14th European Conference on Object Oriented Programming (ECOOP 2000), pages 252–276, June 2000.

[10] J. Wroclawski. The use of RSVP with IETF integrated services, September 1997. RFC 2210.

[11] Rajive Joshi , Gerardo-Pardo Castellote, A Comparison and Mapping of Data Distribution Service and High-Level Architecture,2006 www.rti.com/docs/Comparison-Mapping-DDS-HLA.pdf

[12] Data Distribution Service for Real-time Systems Version 1.2, 2007,http://www.omg.org/cgi-bin/doc?formal/07-01-01

[13] Berndtsson M. and Hansson J. Workshop Report: The First International Workshop on Active and Real-Time Database Systems. ACM SIGMOD Record, 25(1), 1996, pp.64-66.

[14] B. Adelberg, B. Kao, et al. Overview of the Stanford Real-time Information Processor STRIP. ACM SIGMOD Record, 25(1), 1996, pp.34-37.

[15] K. Ramamritham, C. Shen, et al. Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations. Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium. Denver, Colombia, 1998, pp.102~111.
[16] J. Huang, J. Stankovic, D. Towesly and K. Ramamritham. Experimental Evaluation of Real-Time Transaction Processing. Proceedings of the 10th IEEE Real-Time Systems Symposium, 1989, pp.144-153.
[17] Shen C., Gonzalez O., and Mizunuma I. User Level Scheduling of Communicating Real-Time Tasks. Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium. Vancouver, Canada, 1999, pp.164-175.
[18] LIU Wei,WANG,Qiang,WANG Hongan,DAI Guozhong, Adaptive Real-Time Publish-Subscribe Messaging for Distributed Monitoring Systems, Chinese of Journal Electronics, 2005
[19] Joe Hoffert, Douglas Schmidt, and Aniruddha Gokhale, ,A QoS Policy Configuration Modeling Language for Publish/Subscribe Middleware Platforms, DEBS '07, June 20-22, 2007, Toronto, Ontario, Canada
[20] Jan-Hinrich Hauer, Vlado Handziski, Andreas KÄopke, Andreas Willig, Adam Wolisz, A Component Framework for Content-based Publish/Subscribe in Sensor Networks, In Proc. of 5th European Workshop on Wireless Sensor Networks (EWSN), Bologna, Italy, January 2008 Springer
[21] Wei Liu, Ying Qiao, A Visual Specification Tool for Event-Condition-Action Rules Supporting Web-based Environment, International Conference on Enterprise Information Systems 2008(To Appear).
[22] Filipe ARA´UJO, Lu´ıs RODRIGUES, On QoS-Aware Publish-Subscribe, DEBS2002, Proceedings of the International Workshop on Distributed Event-Based Systems,Vienna, Austria, July, 2002
[23] Angelo CORSARO, Leonardo QUERZONI, Sirio SCIPIONI,Sara TUCCI PIERGIOVANNI and Antonino VIRGILLITO, Quality of Service in Publish/Subscribe Middleware, Global Data Management , IOS Press,2006
[24] PATRICK TH. EUGSTER , PASCAL A. FELBER , RACHID GUERRAOUI , ANNE-MARIE KERMARREC, The Many Faces of Publish/Subscribe, ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 114–131.
[25] Mohsen Sharifi, Majid Alkaee Taleghan, and Amirhosein Taherkordi,A Publish-Subscribe Middleware for Real-Time Wireless Sensor Networks, ICCS 2006, Part I, LNCS 3991, pp. 981 – 984, 2006.
[26] LI Xin, WANG Qiang, WANG Kun , WANG Yong-yan, WANG Hong-An ,An Active Real-Time Database for Intelligent Monitoring Systems, Chinese of Journal Electronics, 2008(To Appear).
[27] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White, Towards Expressive Publish/Subscribe Systems, EDBT 2006, LNCS 3896, pp. 627–644, 2006.
[28] Diao, Y., Altinel, M., Zhang, H., Franklin, M.J., and Fischer, P.M. Path sharing and predicate evaluation for high-performance XML filtering. TODS, 28(4), 467-516, Dec. 2003.
[29] Gehani, N.H., Jagadish, H.V., and Shmueli, O. Composite event specification in active databases: Model and implementation. In VLDB, 327-338, 1992.