ICAS 2016
30th Congress of the International Council of the Aeronautical Sciences
DCC, Daejeon, Korea ; September 25-30, 2016

GENERAL MESSAGE RACES IN DATA DISTRIBUTION SERVICE PROGRAMS FOR AIRBORNE SOFTWARE

Hyun-Ji Kim*, Ok-Kyoon Ha*, and Yong-Kee Jun*
***Gyeongsang National University, South Korea**
hecho3927@gmail.com; jassmin@gnu.ac.kr; jun@gnu.ac.kr

Keywords: *future airborne capability environment, data distribution service, message races*

Abstract

Data Distribution Service (DDS) is a dependable communication middleware for airborne software architecture to provide real-time interoperable data exchanges. It is important to efficiently detect general message races for debugging DDS programs, because it is the most serious type of software faults and the presence of general message races in the programs has not been reported. This paper present an experimental fault case to prove the presence of general message races in DDS programs for airborne software.

1 Introduction

According to the Future Airborne Capability Environment (FACE) [1] which is a technical standard for airborne software, Data Distribution Service (DDS) [2,3,4,5] is the representative middleware for the Transport Service Segment to exchange data on publish/subscribe messaging paradigm. Unintended general message races may occur during these data communication.

General message races [6,7] are one of the most serious concurrent faults which occur when two or more messages are transmitted to a waiting receive event on a communication channel and their arrival order is not guaranteed. If racing messages are received in a nondeterministic order, it may causes unintended results in the execution. Therefore, general message races must be detected.

The presence of unintended general message races in the DDS programs has not been reported. According to a prior work, Pardo-Castellote [4] mentioned that complexity may be increased because an event on a thread does not guarantee the sequence to read messages released by other threads in a same participant in DDS programs. Mutschler and Philippsen [5] say that it is possible to rearrange the order of out-of-order events which caused by DDS using EBS's event detector because DDS cannot fix the order of events. Therefore, it is necessary to prove the presence of general message races which cause unexpected results. It is difficult to locate general message races, because they do not occur in every execution of DDS programs. This unintended message races are the most difficult and serious software fault which takes at least days of sometimes months to debug them. If general message races exist in DDS programs, then it is required to develop technology and tools to detect and remove them.

To prove the presence of general message races in DDS programs, we show a fault case how it lead to general message races on the different order of events. Our experiments are carried on an Intel Quad Core system under windows 7 64 bit OS. We installed RTI Connext DDS 5.1.0 and visual studio 2010 for source compile.

The remainder of this paper is organized as follows. Section 2 provides background for understanding our work. In Section 3 and Section 4, we present our own findings on the cases of massage races on DDS. Finally, we conclude our paper and give directions for future work in section 5.

2 Background

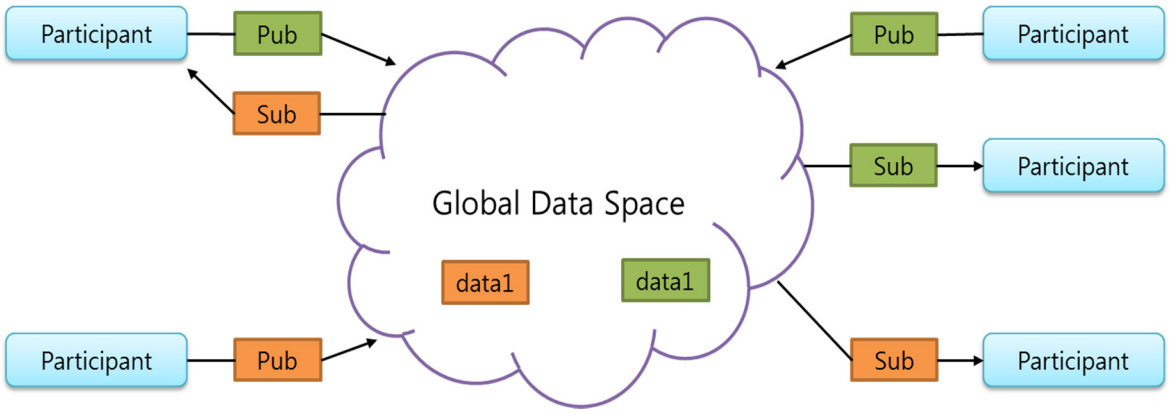


Fig. 1. DDS communication model

Future Airborne Capability Environment (FACE) which is a technical standard for airborne software, define an open avionics environment for all military airborne platform types. According to the FACE, DDS is the representative middleware for the Transport Service Segment to exchange data on publish/subscribe messaging paradigm. During exchange of data in DDS, intended general message races may occur by programmers. However, unintended general message races also may occur by various causes in the execution environment such as network latencies and programmer's faults. Therefore, they should be detected and removed because it is the most difficult and serious software fault which causes unexpected serious results. In this section, we explain what is DDS and general message races, and how general message races may occur in DDS programs.

2.1 Data Distribution Service (DDS)

DDS is a publish-subscribe data distribution middleware released by Object Management Group (OMG), and provide data-centric publish-subscribe standard. It is essential to construct information exchange platform between modules in the publish-subscribe (PS) system. The PS model connects publisher and subscriber anonymously. The system consists of multiple processes, which are executed in the separated address space of different computers. Each process is called participant, and a participant publishers and subscribes data at the same time. Publisher and subscriber exchange data in GDS where they read and write data.

Fig.1 illustrates a communication model using GDS. Exchanging data in DDS programs employs following entities:

- ***DomainParticipant*** allows to connect applications into DDS Domain, such as GDS.
- ***Topic*** is a string to control the object group in GDS. A topic consists of name and data type, and connects data related by QoS.
- ***Publisher*** is an object to release data considering data types.
- ***Subscriber*** is an object to receive data released by publishers. They make to use the data for participants.
- ***DataWriter*** defines a value of data considering a data type.
- ***DataReader*** reads a required data considering a data type.
- ***QoS*** Policy is a rule to configure a DDS system.
- ***Listener*** is an object to identify the events of applications, such as new publishers and data.

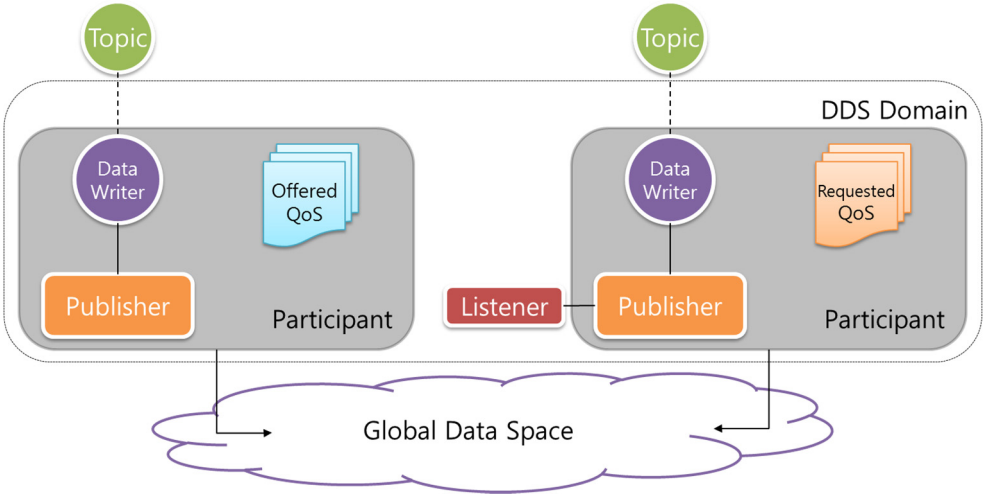


Fig. 2. Object model for DDS

Fig. 2 illustrates a situation that two participants publish and subscribe data based on GDS using a topic in a domain. A participant with publisher uses datawriter to publish data in GDS by writing it in topic, and then a participant with subscriber uses datareader to read data-instance stored in topic and subscribes the message it wants. The topic refers to atomically swapping information unit between publisher and subscriber, and consists of instances that are several specific data. Publisher or subscriber can use listener, which reports information when a new publisher/subscriber appears or new data are received. DDS uses Quality of Service (QoS) to meet the requirements of applications, and QoS makes service to do what it is given to do. The participants that share the same topic must use the same QoS, through which they can control the process of exchanging messages. When different participants access GDS to exchange data in DDS programs, they publish and subscribe the data they want without chronological order, which may result in message races.

2.2 General Message Race

In message-passing parallel programs widely used in parallel computers, message race is one of the serious concurrency bugs. Message race may occur when two or more participants send messages over a same communication channel without guaranteeing the order of their arrivals. Message races must be detected for debugging DDS programs because their nondeterministic arrival leads to unintended nondeterministic results of the program.

Fig. 3 is an example of a general message race. In the figure, P1, P2, and P3 refer to parallel processes, circles on the processes mean send/receive events, and arrows refer to

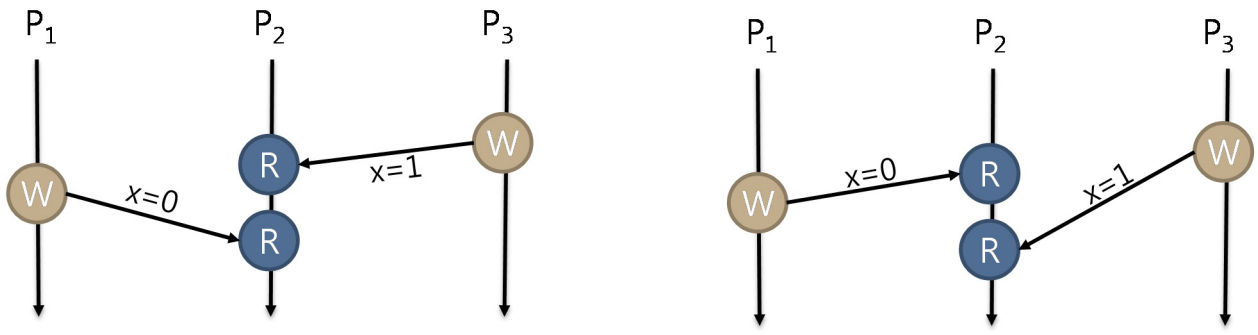


Fig. 3. An example of general message races

transferring messages. P1 sends the message ‘x=0’ to P2, and P3 sends the message ‘x=1’ to P2 while P2 receives these messages from P1 and P3 in the order they arrive. In the receive event *R* of P2, a message race may occur, because the order relation between messages of P1 and P3 is not guaranteed.

Mutschler and Philippsen [5] say that DDS causes out-of-order events and cannot fix the order of events, but it is possible to rearrange their order through EBS’s event detector. However, there has been no report yet on the existence and its confirmation of unintended message races in DDS programs.

3 Design of experiments

To prove the presence of unintended message races in DDS programs, we experiment on every case where message races may occur depending on the arrival order of w to find out all cases lead to nondeterministic results. There may be two types of unintended message races depending on which w occurs earlier. In this section, we check and compare the read values of r based on the order of w for the two types.

For this experiment, we suppose that there are two publishers and a subscriber. The publishers include their own datawriter to send a message defined by w and the subscriber includes its own datareader to receive a message denoted by *r*. In our experiments, we redefine *w* events as w_{p1} for publisher1 and w_{p2} for publisher2, and *r* event as r_s for subscriber. w_{p1} and w_{p2} send messages to r_s, and their arrival order is not guaranteed, which may lead to unintended message races. Fig. 4 depicts a situation of general message races in DDS. In the figure, w_{p1} sends the message 'hello' to r_s and w_{p2} sends the message 'world' to r_s. In this case, r_s reads either ‘hello’ or ‘world’ depending on the order they arrive because the arrival order is not guaranteed. Therefore, the experiment shows unintended results, indicating that message races may occur.

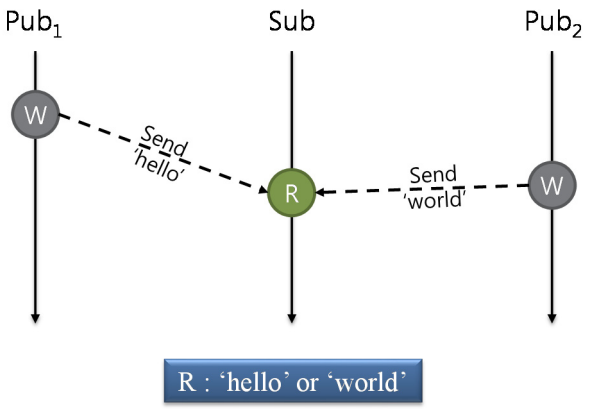


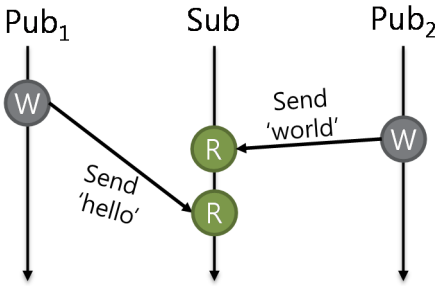
Fig. 4. An execution of the publisher/ subscriber program in DDS

***Program* 1** Publisher 1( )

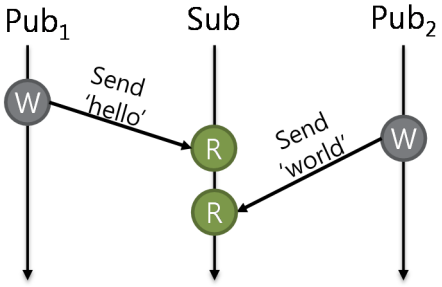
```
01: …
02: char A[5] = {'H','e','l','l','o'};
03: …
04: int main( ) {
05: …
06:     for(i=0;i<1;i++){
07:         retcode = DDS_StringDataWriter_write
             (string_writer, A, &DDS_HANDLE_NIL);
08:         printf(“w in publisher1 : ”);
09:         puts(A);
10:     }
11:  …
12: }
```

***Program* 2** Subscriber( )

```
01: …
02: int main ( ) {
03: …
04:    puts(“Ready to read data.”);
05: …
06: }
07: static void on_data_available_callback(…) {
08: …
09: for ( ; ; ) {
10:       retcode = DDS_StringDataReader_read_next
                    (string_reader, Sample, &info);
11: …
12:     if (info.valid_data) {
13:             printf(“r in subscriber: ”);
14:             puts(sample);
15: …
16: }
```



(a) $w_{p2} \rightarrow r_s$ after $w_{p1} \rightarrow r_s$



(b) $w_{p1} \rightarrow r_s$ after $w_{p2} \rightarrow r_s$

Fig. 5. A general message race case for Fig. 4

Program 3 Publisher 2( )

```
01: …
02: char B[5] = {'W','o','r','l','d'};
03: …
04: int main( ) {
05: …
06:     for(i=0;i<1;i++){
07:         retcode = DDS_StringDataWriter_write
(string_writer,
B, &DDS_HANDLE_NIL);
08:         printf("w in publishers : ");
09:         puts(B);
10:     }
11:  …
12: }
```

3.1 $w_{p2} \rightarrow r_s$ after $w_{p1} \rightarrow r_s$

In Fig 5(a), r_s reads the message 'hello' that w_{p1} sends and then the other message 'world' that w_{p2} sends. This is because when w_{p1} and w_{p2} send the messages, 'hello' is stored earlier than 'world in the queue.

Program 1 and Program 3 display codes of publisher 1 and publisher 2 respectively, which transmit messages to a subscriber at the same time, while Program 2 shows codes of the subscriber. In Program 1 and Program 3, the string data 'hello' and 'world' to be sent are stored in the line 2 respectively, and each message to be sent to the subscriber is published in the line 10. In Program 2, the message that the subscriber receives first is subscribed in the line 7 by calling the callback function. The received message is printed in the lines 12 to 14. In this experiment, the subscriber receives the message 'hello' of publisher 1 in advance.

3.2 $w_{p1} \rightarrow r_s$ after $w_{p2} \rightarrow r_s$

In Fig. 5 (b), r_s reads the message 'world' that w_{p2} sends and then the other message 'hello' w_{p1} sends. This is because when w_{p1} and w_{p2} send the messages, 'world' is stored earlier than 'hello' in the queue. In Figure 6 and Figure 8, the string data 'hello' and 'world′ sent by publisher 1 and publisher 2 are stored in the line 2 respectively, and each message to be sent to the subscriber is published in the line 10 of Figure 7. In this experiment, the subscriber receives the message 'hello' of publisher 2 in advance.

4 Analysis

We found the presence of message races due to nondeterministic execution of the program. As we intended, the subscriber received a message 'hello' because $w_{p2} \rightarrow r_s$ after $w_{p1} \rightarrow r_s$ in almost of all executions of the program. However, the subscriber received a message 'world' first because $w_{p1} \rightarrow r_s$ after $w_{p2} \rightarrow r_s$ in some cases. Therefore, the subscriber can receive first either a message 'hello' or 'world'.

Table 1 shows the data values r reads in the order the two w events occur in cases of message races. Each w of different publishers has no order relations, so the data value r reads is different depending on which w occurs earlier. As seen in

Table 1, when the intended value is ‘world’ and w_{p1} is executed earlier, r actually reads ‘hello’. Also, when the intended value is ‘hello’ and w_{p2} is executed earlier, *r* actually reads ‘world’. This indicates that the intended values and the actual values are different, which may lead to nondeterministic results in DDS programs. Therefore, the experimentations prove that message races exist.

Table 1. The results of experimentation about general message race

Order of Event Execution	Data value *R* reads	
	Intended value	Actual value
$w_{p2} \rightarrow r_s$ after $w_{p1} \rightarrow r_s$	‘world’	‘hello’
$w_{p1} \rightarrow r_s$ after $w_{p2} \rightarrow r_s$	‘hello’	‘world’

5 Conclusion

Unintended message races must be detected for debugging DDS programs, because it is the most serious type of software fault. In DDS programs, unintended message races may occur between participants that exchange messages through publish-subscribe programming model. They may occur due to the programmer’s mistake or network delay. It is difficult to identify the location of message races and it takes a few days or sometimes months for debugging because they do not occur in every execution of the program. There is no report to the presence of message races in DDS programs, which has to be proved in the first place. This paper proved by experiment that message races exist when two or more write events send messages to a read event without the order guaranteed, showing the intended values are different from the actual values. This paper demonstrated that message races may occur in DDS programs. In the future study, sophisticated techniques which detect and remove message races should be developed.

References

[1] The Open Group, *Technical Standard for Future Airborne Capability Environment*, Edition 2.1, May 2014.

[2] Héctor Pérez, J. Javier Gutiérrez. A survey on standards for real-time distribution middleware. *ACM Computing Surveys*, 46(4), April 2014, Article No. 49.

[3] Jinsong Yang, Kristian Sandström, Thomas Nolte, Moris Behnam. Data Distribution Service for Industrial Automation. *Emerging Technologies & Factory Automation, IEEE 17th Conference on*, pp. 1-8, Sept 17-21, 2012.

[4] Pardo-Castellote G. OMG Data-Distribution Service: Architectural Overview. *Distributed Computing Systems Workshops 23rd International Conference on*, pp. 200-206, May 19-22, 2003.

[5] Christopher Mutschler, Michael Philippsen. Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares. *7th ACM international conference on Distributed Event-Based Systems*, pp. 147-158, June 29, 2013.

[6] Mi-Young Park, and Yong-Kee Jun. Detecting Unaffected Race Conditions in Message-Passing Programs. *11th European PVM/MPI User's Group Meeting, Lecture Notes in Computer Science*, 3241: 268-276, Springer-Verlag, Sept. 2004.

[7] Netzer, R. H. B., T. W. Brennan, and S. K. Damodaran-Kamal. Debugging Race Conditions in Message-Passing Programs. *Sigmetrics Symp. on Parallel and Distributed Tools*, pp. 31-40, ACM, May 1996.

Acknowledgements

This work was supported by Research fund, Gyeongsang National University, 2015 and the BK21 Plus Program (Research Team for Software Platform on Unmanned Aerial Vehicle, 21A20131600012) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education.

Contact Author Email Address

mailto: jassmin@gnu.ac.kr

Copyright Statement

The authors confirm that they, and/or their company or organization, hold copyright on all of the original material included in this paper. The authors also confirm that they have obtained permission, from the copyright holder of any third party material included in this paper, to publish it as part of their paper. The authors confirm that they give permission, or have obtained permission from the copyright holder of this paper, for the publication and distribution of this paper as part of the ICAS proceedings or as individual off-prints from the proceedings.