

ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

**DESARROLLO DE UN MÓDULO QUE IMPLEMENTE LAS
FUNCIONALIDADES DEL PROTOCOLO RTPS PARA SER
UTILIZADO EN APLICACIONES DISTRIBUIDAS DE TIEMPO REAL**

**PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN
ELECTRÓNICA Y REDES DE INFORMACIÓN**

RUBIO PROAÑO ANDRÉS XAVIER
andresrubio@msn.com

TELLO GONZÁLEZ ALEJANDRA BEATRIZ
alejitat_28@hotmail.com

DIRECTOR: ING. XAVIER CALDERÓN, MSc.
xavier.calderon@epn.edu.ec

Quito, Septiembre 2015

DECLARACIÓN

Nosotros, Andrés Xavier Rubio Proaño, Alejandra Beatriz Tello González, declaramos bajo juramento que el trabajo aquí descrito es de nuestra autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que hemos consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedemos nuestros derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

Andrés Xavier Rubio Proaño

Alejandra Beatriz Tello González

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Andrés Xavier Rubio Proaño y Alejandra Beatriz Tello González, bajo mi supervisión.

Ing. Xavier Calderón. MSc.
DIRECTOR DEL PROYECTO

AGRADECIMIENTO

Doy las Gracias primeramente a Jesús, quien es digno de toda la honra, toda la adoración.

A Mercedes y Xavier, quienes son mis Padres, y a los cuales admiro demasiado, y amo con todo mi corazón.

A Jose, mi hermano, quien ha estado en cada etapa de mi vida, compartiendo momentos felices y tristes y a quien amo demasiado.

A mis abuelos Miguel y Rosa, quienes han sido parte fundamental en mi vida, y han estado pendientes de mi siempre.

A mis abuelos José y Luz, a quienes admiro por su amor hacia Dios, y quienes fueron usados por Dios para llegar a los pies de Cristo.

A Alejandra, por su amistad y entrega en el proyecto.

A la Escuela Politécnica Nacional, en especial a Xavier Calderón, Agustín Méndez, y a Ernesto Jiménez quienes nos han guiado y han estado pendiente de nosotros en el desarrollo del proyecto.

Andrés Xavier Rubio Proaño

DEDICATORIA

Dedicado al Dios eterno,a mi madre, a mi padre y a mi hermano quien siempre a estado conmigo.

Andrés Xavier Rubio Proaño

AGRADECIMIENTO

A Dios, por siempre estar presente en todo momento de mi vida.

A mis padres: Luz Beatriz González Arciniega y Luis Guilberto Tello Vicuña que me han enseñado a no darme por vencida, porque gracias a ellos pude alcanzar esta meta profesional y por el apoyo y confianza que me brindaron a lo largo de mi vida, porque si no los tuviera a mi lado nunca hubiera llegado hasta aquí.

A mi abuelita Celita por siempre darme su amor incondicional y sentirme apoyada siempre.

A mis hermanos: Luis y Jorge que siempre estuvieron conmigo apoyándome en todo momento, dándome ánimos y que con su cariño pude culminar mis estudios.

A Miguel mi novio, que gracias a su amor, apoyo y confianza he podido culminar esta etapa universitaria, le agradezco por siempre estar conmigo ayudándome siempre.

A toda mi familia porque han sido un pilar importante en el transcurso de mi carrera.

A los amigos con los cuales he llegado a formar un lazo muy importante a lo largo de mi vida, los quiero mucho.

A Andrés el coautor de este proyecto quien con su ayuda y apoyo se logró culminar con este proyecto exitosamente.

A la Escuela Politécnica y a mis profesores en especial al Ingeniero Xavier Calderón que gracias a el me pude involucrar en el proyecto de investigación, a Agustín Méndez por siempre estar dispuesto a darnos una mano con el proyecto de titulación.

Alejandra Beatriz Tello González

DEDICATORIA

Dedicado a mi Dios y a las personas que más amo: Celita, Luis, Beatriz, Jorge, Luis A. y Miguel.

Alejandra Beatriz Tello González

ÍNDICE DE CONTENIDO

DECLARACIÓN	i
CERTIFICACIÓN	ii
AGRADECIMIENTO	iii
DEDICATORIA.....	iv
AGRADECIMIENTO	v
DEDICATORIA.....	vi
ÍNDICE DE CONTENIDO	vii
ÍNDICE DE TABLAS	xiii
ÍNDICE DE FIGURAS	xix
ÍNDICE DE ESPACIOS DE CÓDIGO	xxv
RESUMEN	xxvii
PRESENTACIÓN	xxix
 1. CAPÍTULO 1	1
MARCO TEÓRICO	1
1.1. INTRODUCCIÓN	1
1.2. MIDDLEWARES.....	1
1.3. SISTEMAS DISTRIBUIDOS.....	3
1.4. MIDDLEWARES DE TIEMPO REAL.....	4
1.4.1. CORBA y RT-CORBA	5
1.4.2. The Ada Distributed Systems Annex	9
1.4.3. The Distributed Real-Time Specification for Java.....	11
1.4.4. The Data Distribution Service for Real-Time Systems.....	13
1.5. COMPARACIÓN ENTRE LAS DIFERENTES TECNOLOGÍAS DE MIDDLEWARES DE COMUNICACIÓN DE TIEMPO REAL	16
1.5.1. Gestión de los recursos del procesador	17
1.5.2. Gestión de recursos de red	19

1.5.3. Cuadro Comparativo de las diferentes tecnologías.....	21
1.6. CARACTERÍSTICAS Y FUNCIONALIDADES DEL DDS	22
1.6.1. Características	22
1.6.2. Funcionalidades	34
2. CAPÍTULO 2	40
ANÁLISIS DE REQUISITOS PARA LA IMPLEMENTACIÓN DE UN MÓDULO QUE SOPORTE EL PROTOCOLO RTPS.....	40
2.1. INTRODUCCIÓN	40
2.2. ANÁLISIS DE PAQUETES DE LOS DIFERENTES MENSAJES RTPS.....	40
2.2.1. Estructura de los mensajes RTPS.....	40
2.2.2. Estructura de los submensajes RTPS.....	41
2.2.3. AckNack Submessage	42
2.2.4. Data Submessage.....	44
2.2.5. DataFrag Submessage	48
2.2.6. Gap Submessage.....	52
2.2.7. Heartbeat Submessage.....	54
2.2.8. HeartBeatFrag Submessage.....	57
2.2.9. InfoDestination Submessage	59
2.2.10. InfoReply Submessage	60
2.2.11. InfoSource Submessage	61
2.2.12. InfoTimestamp Submessage	62
2.2.13. NackFrag Submessage	64
2.2.14. Pad Submessage	66
2.2.15. InfoReplyIp4 Submessage	67
2.3. ANÁLISIS DE REQUISITOS	68
2.4. MÓDULO DDS	69

2.4.1. Publicador	69
2.4.2. Suscriptor	69
2.4.3. Topic.....	69
2.5. MECANISMO Y TÉCNICAS PARA EL ALCANCE DE LA INFORMACIÓN.....	71
2.6. LECTURA Y ESCRITURA DE DATOS	72
2.6.1. Escritura de Datos.....	72
2.6.2. Ciclo de Vida de los Topic-Instances	72
2.6.3. Lectura de Datos.....	74
2.6.4. Datos y Metadatos	74
2.6.5. Notificaciones.....	75
2.7. MÓDULO RTPS	76
2.7.1. Módulo estructura.....	76
2.7.2. Módulo Mensajes	78
2.7.3. Módulo Comportamiento	85
2.7.4. Módulo Descubrimiento.....	97
3. CAPÍTULO 3	100
DISEÑO E IMPLEMENTACIÓN DE UN MÓDULO QUE PERMITA INTERACTUAR AL PROTOCOLO RTPS CON DDS	100
3.1. INTRODUCCIÓN	100
3.2. API-RTPS.....	100
3.2.1. Diagramas de Clase a partir del API-RTPS.....	101
3.3. DISEÑO DEL MÓDULO	104
3.3.1. Adaptación del API-RTPS para interactuar con DDS	105
3.4. IMPLEMENTACIÓN DEL MÓDULO.....	116
3.4.1. Submódulo de transporte	116
3.4.2. Submódulo de mensaje y encapsulamiento	120

3.4.3. Submódulo de descubrimiento	121
3.4.4. Submódulo de comportamiento	123
3.4.5. Submódulo configuración	128
3.5. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS	136
3.5.1. Interacción del DDS con el Protocolo RTPS con Estado	136
3.5.2. Interacción del DDS con el Protocolo RTPS sin estado	165
3.5.3. Interacción del DDS con el Protocolo RTPS híbridos (con estado y sin estado)	171
3.5.4. Protocolo Descubrimiento	175
3.5.5. Intercambio de mensajes RTPS sobre la Red	176
3.6. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS	181
3.6.1. Diagramas de secuencia de la interacción de DDS con RTPS con estado	181
3.6.2. Diagramas de secuencia de la interacción de DDS con RTPS sin estado	189
3.6.3. Diagramas de secuencia de la interacción de DDS con RTPS híbridos (con estado y sin estado)	191
3.6.4. Protocolo Descubrimiento	192
4. CAPÍTULO 4	193
PRUEBAS	193
4.1. INTRODUCCIÓN	193
4.2. PRUEBAS UNITARIAS DE LA IMPLEMENTACIÓN	193
4.2.1. Codificadores	193
4.2.2. Transporte	226
4.2.3. Utils	248
4.2.4. Serializador	252
4.3. PRUEBA APLICACIÓN RTPS	320

4.3.1. Requerimientos para el uso de la aplicación.....	320
4.3.2. Aplicación chat con tecnología DDS-RTPS.....	320
4.4. PRUEBA APLICACIÓN CORBA.....	323
4.4.1. Requerimientos para el uso de la aplicación.....	323
4.4.2. Aplicación chat con tecnología CORBA	323
4.5. MANUAL DE USUARIO DE LA APLICACIÓN Y USO DE LAS LIBRERÍAS DDS-RTPS	328
4.5.1. Manual para el uso de las librerías DDS-RTPS.	328
4.5.2. Manual de Usuario de la aplicación Chat RTPS.....	333
4.6. COMPARACIÓN DE APLICACIONES	337
4.6.1. Capturas de paquetes DDS-RTPS y CORBA-RT	337
5. CAPÍTULO 5	339
CONCLUSIONES Y RECOMENDACIONES	339
5.1. CONCLUCIONES	339
5.2. RECOMENDACIONES	342
6. REFERENCIAS.....	343
7. ANEXOS	348
ANEXO A: Glosario	348
ANEXO B: Middleware	351
ANEXO B.1: Código fuente del middleware	351
ANEXO B.2: Manual de uso de la librerías DDS-RTPS	351
ANEXO C: Aplicación de escritorio	352
ANEXO C.1: Código fuente del chat RTPS	352
ANEXO C.2: Manual de usuario del chat rtps	352
ANEXO D: Aplicación Corba	353
ANEXO D.1: Código fuente chat corba	353
ANEXO E: Estándar.....	354

ANEXO E.1: The Real-TIme Publish-Suscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification	354
ANEXO E.2: Data Distribution Service for Real-Time System Specification (DDS).....	354
ANEXO F: Manual de ejemplos	355
ANEXO F.1: Manual de Usuario para la creación de ejemplos con las librerías DDS-RTPS.	355

ÍNDICE DE TABLAS

Tabla 1-1. Ejemplos de los modelos de sistemas distribuidos	4
Tabla 1-2. Capacidades de Tiempo-Real de los Estándares de Distribución [5] . .	21
Tabla 1-3. Políticas de QoS del DDS [37].....	31
Tabla 2-1. Tipos IDL primitivos.....	70
Tabla 2-2. Tipos IDL template.....	71
Tabla 2-3. Tipos IDL compuestos.	71
Tabla 2-4. Operadores para Filtros DDS y Condiciones de Consulta.....	72
Tabla 2-5. Administración del Ciclo de Vida Automática	73
Tabla 2-6. Clases y Entidades RTPS	77
Tabla 2-7. Combinación de atributos posibles en lectores asociados con escritores [41]	88
<i>Tabla 2-8. Transiciones del comportamiento en mejor esfuerzo de un Writer sin estado con respecto a cada ReaderLocator.....</i>	89
<i>Tabla 2-9. Transiciones del comportamiento en confiable de un Writer sin estado con respecto a cada ReaderLocator</i>	90
<i>Tabla 2-10. Transiciones del comportamiento en mejor esfuerzo de un Writer con estado con respecto a cada ReaderLocator</i>	92
<i>Tabla 2-11. Transiciones del comportamiento en confiable de un Writer con estado con respecto a cada ReaderLocator</i>	93
<i>Tabla 2-12. Transiciones del comportamiento en mejor esfuerzo de un Reader sin estado</i>	95
<i>Tabla 2-13. Transiciones del comportamiento en mejor esfuerzo de un Reader con estado con respecto a cada Writer asociado.....</i>	96
<i>Tabla 2-14. Transiciones del comportamiento en confiable de un Reader con estado con respecto a su Writer asociado</i>	96
Tabla 3-1. Métodos de las clases para encapsulamiento de mensajes, cabeceras e identificadores	106
Tabla 3-2. Métodos de las clases para la serialización y deserialización	107
Tabla 3-3. Métodos de las clases para el descubrimiento.	108
<i>Tabla 3-4. Métodos de las clases para la comunicación del RTPS con el DDS. .</i>	110
<i>Tabla 3-5. Métodos de la clase FakeDiscovery.</i>	112

<i>Tabla 3-6.</i> Métodos de las diferentes clases para la implementación del sistema transporte en red	113
<i>Tabla 3-7.</i> Métodos de los mensajes de descubrimiento y mensajes RTPS	114
<i>Tabla 4-1.</i> TestLocatorIpV4CDR_BE	194
Tabla 4-2. TestLocatorIpV4CDR_LE.....	194
Tabla 4-3. TestLocatorIpV6CDR_BE	195
<i>Tabla 4-4.</i> TestLocatorIpV6CDR_LE.....	196
Tabla 4-5. TestLocatorIpV6CDR_BE2	197
Tabla 4-6. TestLocatorIpV6CDR_LE2.....	197
Tabla 4-7. TestLocatorFromSample1.....	198
<i>Tabla 4-8.</i> TestLocatorFromSample2.....	199
Tabla 4-9. TestLocatorFromSample3.....	200
Tabla 4-10. TestLocatorIpV4PL_CDR_BE	200
Tabla 4-11. TestLocatorIpV4PL_CDR_LE.....	201
Tabla 4-12. TestLocatorIpV6PL_CDR_BE	202
Tabla 4-13. TestLocatorIpV6PL_CDR_LE	203
Tabla 4-14. TestLocatorIpV6PL_CDR_BE2	204
Tabla 4-15. TestLocatorIpV6PL_CDR_LE2	204
Tabla 4-16. TestLocatorFromSample1PL_CDR_LE	205
Tabla 4-17. TestLocatorFromSample2PL_CDR_LE	206
Tabla 4-18. TestLocatorFromSample3PL_CDR_LE	207
Tabla 4-19. TestGUIDCDR_BE.....	207
Tabla 4-20. TestGUICDR_LE.....	208
Tabla 4-21. TestGUIDPL_CDR_BE	209
Tabla 4-22. TestGUIDPL_CDR_LE.....	210
Tabla 4-23. TestVendorIdCDR_BE	211
Tabla 4-24. TestVendorIdCDR_LE.....	211
Tabla 4-25. TestProtocolVersionCDR_BE.....	212
Tabla 4-26. TestProtocolVersionCDR_LE	213
Tabla 4-27. TestProtocolVersionPL_CDR_BE	214
Tabla 4-28. TestProtocolVersionPL_CDR_LE.....	214
Tabla 4-29. TestListLocatorCDR_BE	215
Tabla 4-30. TestListLocatorCDR_LE	216

Tabla 4-31. TestInfoDestination	217
Tabla 4-32. TestInfoSource	218
Tabla 4-33. TestInfoReply	218
Tabla 4-34. TestInfoReplyIp4	219
Tabla 4-35. TestInfoTimestamp.....	220
Tabla 4-36. TestGAP.....	220
<i>Tabla 4-37. TestAckNack</i>	221
Tabla 4-38. TestPad.....	222
Tabla 4-39. TestHeartbeat	223
Tabla 4-40. TestNackFrag.....	223
Tabla 4-41. TestHeartbeatFrag	224
Tabla 4-42. TestDataFrag	225
<i>Tabla 4-43. TestPublishData.....</i>	226
Tabla 4-44. TestPublishPacket2	228
Tabla 4-45. GeneralRTPSMessageTesterMethod	230
Tabla 4-46. TestOpenDDS_rtps_reliability_runttest_localPacket01	237
Tabla 4-47. TestOpenDDS_rtps_reliability_runttest_localPacket02	241
Tabla 4-48. TestOpenDDS_rtps_reliability_runttest_localPacket03	245
Tabla 4-49. TestGenerator1	248
<i>Tabla 4-50. TestGeneration2</i>	249
<i>Tabla 4-51. TestWorkerVerySlow.....</i>	249
<i>Tabla 4-52. TestWorkerSlow.....</i>	250
<i>Tabla 4-53. TestWorkerQuick</i>	251
<i>Tabla 4-54. TestTimeSeconds</i>	251
<i>Tabla 4-55. TestParticipantBuiltinTopicData</i>	252
<i>Tabla 4-56. TestPublicationBuiltinTopicData</i>	253
<i>Tabla 4-57. TestSubscriptionBuiltinTopicData.....</i>	255
<i>Tabla 4-58. TestSubscriptionBuiltinTopicData.....</i>	257
<i>Tabla 4-59. TestTopicBuiltinTopicData</i>	259
<i>Tabla 4-60. TestBoolPacketLE.....</i>	262
<i>Tabla 4-61. TestCharPacketLE</i>	262
<i>Tabla 4-62. TestU8PacketLE</i>	263
<i>Tabla 4-63. TestU16PacketLE</i>	264

<i>Tabla 4-64.</i> TestU32PacketLE	265
<i>Tabla 4-65.</i> TestU64PacketLE	265
<i>Tabla 4-66.</i> TestS8PacketLE1	266
<i>Tabla 4-67.</i> TestS8PacketLE2	267
Tabla 4-68. TestS16PacketLE1	268
<i>Tabla 4-69.</i> TestS16PacketLE2	268
<i>Tabla 4-70.</i> TestS32PacketLE1	269
<i>Tabla 4-71.</i> TestS32PacketLE2	270
<i>Tabla 4-72.</i> TestS64PacketLE1	271
<i>Tabla 4-73.</i> TestS64PacketLE2	271
<i>Tabla 4-74.</i> TestSinglePacketLE	272
<i>Tabla 4-75.</i> TestDoublePacketLE	273
<i>Tabla 4-76.</i> TestBoolSequencePacketLE	274
<i>Tabla 4-77.</i> TestShortSequencePacketLE	275
<i>Tabla 4-78.</i> TestEnumSequencePacketLE	275
<i>Tabla 4-79.</i> TestIntSequencePacketLE	276
<i>Tabla 4-80.</i> TestEmptyBoolSequencePacketLE	277
<i>Tabla 4-81.</i> TestEmptyShortSequencePacketLE	278
<i>Tabla 4-82.</i> TestEmptyEnumSequencePacketLE	279
<i>Tabla 4-83.</i> TestEmptyIntSequencePacketLE	279
<i>Tabla 4-84.</i> TestBoolPacketBE	280
<i>Tabla 4-85.</i> TestCharPacketBE	281
<i>Tabla 4-86.</i> TestU8PacketBE	282
<i>Tabla 4-87.</i> TestU16PacketBE	282
<i>Tabla 4-88.</i> TestU32PacketBE	283
<i>Tabla 4-89.</i> TestU64PacketBE	284
<i>Tabla 4-90.</i> TestS8PacketBE1	285
<i>Tabla 4-91.</i> TestS8PacketBE2	285
<i>Tabla 4-92.</i> TestS16PacketBE1	286
<i>Tabla 4-93.</i> TestS16PacketBE2	287
<i>Tabla 4-94.</i> TestS32PacketBE1	288
<i>Tabla 4-95.</i> TestS32PacketBE2	288
<i>Tabla 4-96.</i> TestS64PacketBE1	289

<i>Tabla 4-97.</i> TestS64PacketLE2	290
<i>Tabla 4-98.</i> TestSinglePacketBE	291
<i>Tabla 4-99.</i> TestDoublePacketLE	292
<i>Tabla 4-100.</i> TestBoolSequencePacketBE	292
<i>Tabla 4-101.</i> TestShortSequencePacketBE	293
<i>Tabla 4-102.</i> TestEnumSequencePacketBE	294
<i>Tabla 4-103.</i> TestIntSequencePacketBE	295
<i>Tabla 4-104.</i> TestEmptyBoolSequencePacketBE	296
<i>Tabla 4-105.</i> TestEmptyShortSequencePacketBE	296
<i>Tabla 4-106.</i> TestEmptyEnumSequencePacketBE	297
<i>Tabla 4-107.</i> TestEmptyIntSequencePacketLE	298
<i>Tabla 4-108.</i> TestExploreMyClass1	299
<i>Tabla 4-109.</i> TestBoolPacket	300
<i>Tabla 4-110.</i> TestCharPacket	300
<i>Tabla 4-111.</i> TestU8Packet	301
<i>Tabla 4-112.</i> TestU16Packet	301
<i>Tabla 4-113.</i> TestU32Packet	302
<i>Tabla 4-114.</i> TestU64Packet	303
<i>Tabla 4-115.</i> TestS8Packet	303
<i>Tabla 4-116.</i> TestS16Packet	304
<i>Tabla 4-117.</i> TestS32Packet	304
<i>Tabla 4-118.</i> TestS64Packet	305
<i>Tabla 4-119.</i> TestSinglePacket	305
<i>Tabla 4-120.</i> TestDoublePacket	306
<i>Tabla 4-121.</i> TestPrimitivesPacket	306
<i>Tabla 4-122.</i> TestEnumPacket	307
<i>Tabla 4-123.</i> TestStruct1Packet	307
<i>Tabla 4-124.</i> TestStructMessagePacket	308
<i>Tabla 4-125.</i> TestSequenceMessagePacket	308
<i>Tabla 4-126.</i> TestSequenceMessagePacket2	309
<i>Tabla 4-127.</i> TestSequenceMessagePacket3	309
<i>Tabla 4-128.</i> TestSequenceMessagePacket3	310
<i>Tabla 4-129.</i> TestMyClassListMessagePacket1	310

<i>Tabla 4-130.</i> TestMyClassListMessagePacket2	311
<i>Tabla 4-131.</i> TestMyClassListMessagePacket3	312
<i>Tabla 4-132.</i> TestDouble.....	312
<i>Tabla 4-133.</i> TestSingle	313
<i>Tabla 4-134.</i> TestShort	313
<i>Tabla 4-135.</i> TestUShort.....	314
<i>Tabla 4-136.</i> TestInt	314
<i>Tabla 4-137.</i> TestUInt	315
<i>Tabla 4-138.</i> TestLong	315
<i>Tabla 4-139.</i> TestULong	316
<i>Tabla 4-140.</i> TestChar	316
<i>Tabla 4-141.</i> TestByte	317
<i>Tabla 4-142.</i> TestSByte	317
<i>Tabla 4-143.</i> TestBool.....	318
<i>Tabla 4-144.</i> TestDateTime.....	318
<i>Tabla 4-145.</i> TestString.....	319
<i>Tabla 4-146.</i> Comparación de las tecnologías DDS-RTPS con CORBA-RT.	338

ÍNDICE DE FIGURAS

Figura 1-1. Servicios básicos provistos por el Middleware de distribución [5]	3
Figura 1-2. Arquitectura de CORBA [5]	6
Figura 1-3. Comunicación entre entidades CORBA [5]	7
Figura 1-4. Diagrama de secuencia de una llamada remota síncrona [5]	11
Figura 1-5. Diagrama de secuencia de una llamada remota asíncrona [5]	13
Figura 1-6. Sistema Distribuido que consta de tres participantes en un solo Dominio [5]	15
Figura 1-7. Línea de tiempo en los estándares de tiempo real [5].....	22
Figura 1-8. Arquitectura del Middleware DDS [37]	24
Figura 1-9. Modelo DCPS y sus relaciones [38].....	26
Figura 1-10. Modelo DLRL [5].	28
Figura 1-11. Módulos RTPS [5].	30
Figura 1-12. Modelo Suscriptor-Solicitado y Publicador-Ofertado [37].....	32
Figura 1-13. Interoperabilidad del API [39].	33
Figura 1-14. Interoperabilidad del Protocolo de Conexión [39].....	33
Figura 1-15. Parámetros de QoS definidos por DDS [5]	35
Figura 1-16. Control del tiempo en DDS [5].	37
Figura 2-1. Estructura general mensaje RTPS [41].....	40
Figura 2-2. Cabecera del Mensaje RTPS [41].....	41
Figura 2-3. Estructura de los submensajes RTPS [41].....	41
Figura 2-4. Estructura del submensaje AckNack [41].....	42
Figura 2-5. Uso del submensaje ACKNACK.....	44
Figura 2-6. Estructura del submensaje Data [41]	44
Figura 2-7. Uso del submensaje DATA.....	47
Figura 2-8. Estructura del submensaje DataFrag [41].....	48
Figura 2-9. Uso del submensaje DATA_FRAG (parte I).	51
Figura 2-10. Uso del submensaje DATA_FRAG (parte II).	51
Figura 2-11. Estructura del submensaje GAP [41].....	52
Figura 2-12. Uso del submensaje GAP.....	54
Figura 2-13. Estructura del submensaje Heartbeat [41].....	54
Figura 2-14. Uso del submensaje HEARTBEAT.....	56
Figura 2-15. Estructura del submensaje HeartBeatFrag [41]	57

Figura 2-16. Uso del submensaje HEARTBEAT_FRAG.....	58
Figura 2-17. Estructura del submensaje InfoDestination [41].....	59
Figura 2-18. Uso del submensaje INFO_DESTINATION.	60
Figura 2-19. Estructura del submensaje InfoReply [41].....	60
Figura 2-20. Estructura del submensaje InfoSource [41]	61
Figura 2-21. Estructura del submensaje InfoTimestamp [41]	62
Figura 2-22. Uso del submensaje INFO_TS.....	64
Figura 2-23. Estructura del submensaje NackFrag [41]	64
Figura 2-24. Uso del submensaje NACK_FRAG.....	66
Figura 2-25. Estructura del submensaje Pad [41]	66
Figura 2-26. Estructura del submensaje InfoReplyIp4 [41].....	67
Figura 2-27. Objeto Topic y sus componentes. [37]	70
Figura 2-28. Módulo Estructura	77
Figura 2-29. HistoryCache.....	78
Figura 2-30. Estructura del mensaje RTPS.	79
Figura 2-31. Estructura de la cabecera del mensaje RTPS.....	80
Figura 2-32. Estructura de los submensajes RTPS.....	80
Figura 2-33. Receptor RTPS.....	82
Figura 2-34. Elementos de submensaje RTPS.	83
Figura 2-35. Submensajes RTPS.....	85
Figura 2-36. Comportamiento de un Writer sin estado con WITH_KEY Best-Effort con respecto a cada ReaderLocator [41]	89
Figura 2-37. Comportamiento de un Writer sin estado con WITH_KEY Reliable con respecto a cada ReaderLocator [41]	90
Figura 2-38. Comportamiento de un Writer con estado con WITH_KEY Best-Effort con respecto a cada ReaderLocator [41]	91
Figura 2-39. Comportamiento de un Writer con estado con WITH_KEY Reliable con respecto a cada ReaderLocator [41]	93
Figura 2-40. Comportamiento de un Reader sin estado con WITH_KEY Best-Effort [41]	95
Figura 2-41. Comportamiento de un Reader con estado con WITH_KEY Best-Effort con respecto a cada Writer asociado [41]	95

Figura 2-42. Comportamiento de un Reader con estado con WITH_KEY Reliable con respecto a cada Writer asociado [41]	96
Figura 3-1. Clases del API-RTPS para los mensaje y el encapsulamiento I	101
Figura 3-2. Clases del API-RTPS para los mensaje y el encapsulamiento II	102
Figura 3-3. Clases del API-RTPS para los mensaje y el encapsulamiento III	102
Figura 3-4. Clases del API-RTPS para el descubrimiento.....	103
Figura 3-5. Clases del API-RTPS para el comportamiento.....	104
Figura 3-6. Diagrama de clases de la encapsulación de mensajes, cabeceras e identificadores	105
Figura 3-7. Diagrama de clases para la serialización de mensajes.....	107
Figura 3-8. Diagrama de clases del submódulo descubrimiento.....	108
Figura 3-9. Diagrama de clases del submódulo comportamiento.....	110
Figura 3-10. Diagrama de clases del sistema falso de transporte.....	112
Figura 3-11. Diagrama de clases del sistema de transporte sobre la red.....	113
Figura 3-12. Diagrama de clases de los mensajes de descubrimiento y mensajes RTPS.....	114
Figura 3-13. Diseño archivo de configuración sección DDS	115
Figura 3-14. Diseño archivo de configuración sección RTPS.....	116
Figura 3-15. Intercambio de mensajes RTPS sobre la red Ejemplo 1	176
Figura 3-16. Intercambio de Mensajes RTPS Ejemplo 2	177
Figura 3-17. Intercambio de Mensajes RTPS Ejemplo 3.....	178
Figura 3-18. Intercambio de Mensajes RTPS Ejemplo 4.1	179
Figura 3-19. Intercambio de Mensajes RTPS Ejemplo 4.2.....	179
Figura 3-20. Comportamiento Best Effort Reader – Best Effort Writer en interacción con estado.....	181
Figura 3-21. Comportamiento Best Effort Reader – Best Effort Writer en interacción con estado con falla de envío de paquete.....	182
Figura 3-22. Comportamiento Reliable Reader – Reliable Writer en interacción con estado	183
Figura 3-23. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con fragmentación de datos	184
Figura 3-24. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con falla en la comunicación.....	185

Figura 3-25. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con falla de envío de paquete y con tres participantes.....	186
Figura 3-26. Comportamiento Reliable Writer – Best Effort Reader en interacción con estado.....	187
Figura 3-27. Comportamiento Reliable Writer – Best Effort Reader en interacción con estado con falla en el envío de paquetes.	188
Figura 3-28. Comportamiento Best Effort Reader – Best Effort Writer en interacción sin estado.....	189
Figura 3-29. Comportamiento Reliable Writer – Best Effort Reader en interacción sin estado.....	190
Figura 3-30. Comportamiento Reliable Stateless Writer sin estado – Reliable Stateful Reader con estado.....	191
Figura 3-31. Fases de descubrimiento de participantes.....	192
Figura 4-1. Ejecución del Chat RTPS.....	320
Figura 4-2. Conexión al servicio de Chat.	321
Figura 4-3. Intercambio de mensajes.	321
Figura 4-4. Desconexión del servicio de Chat.	322
Figura 4-5. Salida de la aplicación.	322
Figura 4-6. Paquete RTPS dentro del Chat.....	323
Figura 4-7. Chat CORBA 1	324
Figura 4-8. Chat CORBA 2	324
Figura 4-9. Chat CORBA 3	325
Figura 4-10. Chat CORBA 4	325
Figura 4-11. Chat CORBA 5	326
Figura 4-12. Chat CORBA 6	326
Figura 4-13. Captura de Paquetes CORBA 1.....	327
Figura 4-14. Captura de Paquetes CORBA 2.....	327
Figura 4-15. Captura de Paquetes CORBA 3.....	328
Figura 4-16. Captura de Paquetes CORBA 4.....	328
Figura 4-17. Creación de un proyecto en Visual Studio.	329
Figura 4-18. Creación proyecto Aplicación Consola.....	330
Figura 4-19. Agregación de referencias	330
Figura 4-20. Administrador de Referencias de Visual Studio.	331

Figura 4-21. Librerías .dll de la implementación DDS – RTPS.....	331
Figura 4-22. Librerías .dll	332
Figura 4-23. Pestaña de Referencias del proyecto creado.	332
Figura 4-24. Agregación de Referencias en una clase.....	333
Figura 4-25. Utilización de la aplicación Chat 1.....	334
Figura 4-26. Utilización de la aplicación Chat 2.....	334
Figura 4-27. Utilización de la aplicación Chat 3.....	335
Figura 4-28. Utilización de la aplicación Chat 4.....	335
Figura 4-29. Utilización de la aplicación Chat 5.....	336
Figura 4-30. Utilización de la aplicación Chat 6.....	336
Figura 4-31. Flujo de datos DDS-RTPS	337
Figura 4-32. Flujo de datos CORBA-RT	337

ÍNDICE DE ESPACIOS DE CÓDIGO

Espacio de Código 3-1. Inicio de la comunicación con UDP	117
Espacio de Código 3-2. Comunicación Multicast	117
Espacio de Código 3-3. Conexión mediante futuros.	118
Espacio de Código 3-4. Receiver Buffer.....	119
Espacio de Código 3-5. Implementación maquinaria.	119
Espacio de Código 3-6. Interfaz RTPS Engine.....	120
Espacio de Código 3-7. Implementación mensajes RTPS.	120
Espacio de Código 3-8. Implementación de la Encapsulación.	121
Espacio de Código 3-9. Implementación Serializador.	121
Espacio de Código 3-10. Implementación del Descubrimiento RTPS.....	122
Espacio de Código 3-11. Implementación SEDP.....	122
Espacio de Código 3-12. Implementación SPDP.....	123
Espacio de Código 3-13. Implementación Reader RTPS con estado.	124
Espacio de Código 3-14. Implementación del Reader RTPS sin estado.....	125
Espacio de Código 3-15. Implementación del Writer RTPS con estado.	125
Espacio de Código 3-16. Implementación del Writer RTPS sin estado.	126
Espacio de Código 3-17. Implementación del Writer DDS.	126
Espacio de Código 3-18. Implementación del Publicador.	127
Espacio de Código 3-19. Implementación del Reader DDS.	127
Espacio de Código 3-20. Implementación de Suscriptor.	128
Espacio de Código 3-21. Etiqueta DDS.	128
Espacio de Código 3-22. Etiqueta Boostrap.....	128
Espacio de Código 3-23. Etiqueta Domains.....	130
Espacio de Código 3-24. Etiqueta LogLevel.	130
Espacio de Código 3-25. Etiqueta domainParticipantFactoryQoS.	131
Espacio de Código 3-26. Etiqueta domainParticipantQoS.	131
Espacio de Código 3-27. Etiqueta topicQoS.	131
Espacio de Código 3-28. Etiqueta publisherQoS.	132
Espacio de Código 3-29. Etiqueta subscriberQoS.	132
Espacio de Código 3-30. Etiqueta dataWriterQoS.	133
Espacio de Código 3-31. Etiqueta dataReaderQoS.	133
Espacio de Código 3-32. Etiqueta RTPS.....	134

Espacio de Código 3-33. Etiqueta transport.....	134
Espacio de Código 3-34. Etiqueta Discovery.....	134
Espacio de Código 3-35. Etiqueta rtpsWriter.....	135
Espacio de Código 3-36. Etiqueta rtpsReader.....	135

RESUMEN

El presente Proyecto comprende el desarrollo de un módulo que implementa las funcionalidades del protocolo RTPS para aplicaciones distribuidas de tiempo real.

El módulo contempla una implementación básica del DDS, la implementación del RTPS y la interacción de los mismos, para el funcionamiento adecuado del middleware.

El DDS es el encargado por medio de operaciones de escritura almacenar los datos que el usuario requiera, serializar los datos y anunciar por medio de RTPS que hay una publicación de datos; además es el encargado de leer o interpretar la información que RTPS le provee, por medio de suscripciones a información publicada.

El RTPS es el encargado del transporte de los datos que DDS genere; este primeramente anuncia la presencia de entidades y por medio del protocolo de descubrimiento las demás entidades se conocen entre sí. Además se encarga de encapsular la información provista por el DDS para enviarla a los diferentes destinatarios.

La interacción de DDS con RTPS es proporcionada por un submódulo dentro de RTPS el cual se encarga del comportamiento del protocolo de comunicación RTPS ante lo que DDS requiera.

En el primer capítulo se presentan el estado actual de los middlewares de comunicaciones de tiempo real; se realiza un estudio de cada tecnología encontrada incluyendo al middleware DDS, para realizar una comparación entre ellos donde se detalla las ventajas y desventajas del uso de cada una de estas tecnologías; finalmente se analiza las características y funcionalidades más específicas definidas en el estándar publicado por la OMG sobre DDS y su interoperabilidad con el protocolo RTPS.

En el segundo capítulo se definen los requisitos necesarios para integrar el protocolo RTPS con el middleware DDS, además, se realiza un análisis previo con capturas de paquetes con la herramienta Wireshark de los diferentes mensajes RTPS y mensajes de descubrimiento RTPS.

En el tercer capítulo se diseña un módulo que permita la interacción entre RTPS y DDS, y que trabaje con el modelo Publicador/Suscriptor, el cual permite

intercambiar mensajes RTPS; además, se describe el intercambio de mensajes por medio de diagramas de secuencia; y se implementa el software en el paradigma orientado a objetos por medio del lenguaje C#.

En el cuarto capítulo se realiza tanto pruebas unitarias como una prueba entre 4 computadoras comunicándose por medio del protocolo RTPS; en las pruebas unitarias se comprueba el correcto funcionamiento de los distintos componentes del programa tales como clases y métodos; y en las pruebas con computadoras se crea un ambiente de comunicación intercambiando mensajes de descubrimiento RTPS, para luego intercambiar mensajes RTPS.

Finalmente se presentan las principales conclusiones y recomendaciones que arrojan el desarrollo y la implementación del sistema.

Adicionalmente en los anexos se incluye: un glosario de términos, todos los códigos fuentes que contienen el middleware y los códigos fuentes de la aplicación de escritorio y el del chat CORBA, además se incluye los estándares del DDS y del RTPS, y manuales de usuario de las diferentes implementaciones.

PRESENTACIÓN

El presente Proyecto de Titulación se ha realizado con el objetivo del dar a conocer la tecnología de comunicación DDS-RTPS. La comunicaciones actualmente buscan descentralizar la información por lo que DDS-RTPS provee una arquitectura capaz de cumplir con este objetivo.

Este proyecto forma parte de la investigación realizada por la Universidad Técnica Particular de Loja, la Universidad Politécnica Salesiana, la Escuela Politécnica Nacional y el Consorcio Ecuatoriano para el Desarrollo de Internet Avanzado, el cual se titula Middleware en tiempo real basado en el modelo publicación/ suscripción del cual se puede obtener mayor información en la siguiente dirección electrónica: <http://www.utpl.edu.ec/proyectomiddleware/>.

En el presente documento el lector tendrá a su disposición la información básica para comprender el funcionamiento de los diferentes middleware de comunicación incluyendo DDS-RTPS. Además, se presenta la interacción entre la tecnología DDS junto a RTPS.

Dentro del proyecto de titulación se implementa un módulo que genera las librerías de DDS y RTPS necesarias para que se desarrollen futuras aplicaciones por medio de las mismas. Además, se presenta una aplicación que utiliza las librerías mencionadas como un ejemplo del funcionamiento de esta tecnología.

Finalmente, se presenta un stack de pruebas unitarias que verifican el funcionamiento de los elementos del middleware y también se realiza una prueba que comprende a varios equipos intercambiando información a través de esta tecnología.

CAPÍTULO 1

MARCO TEÓRICO

1.1. INTRODUCCIÓN

En el presente capítulo se presenta el fundamento teórico necesario para el desarrollo de este proyecto. Inicialmente se describe el estado actual de los Middlewares de comunicaciones de tiempo real y se presenta un estudio de cada tecnología encontrada incluyendo al Middleware DDS. Posteriormente se realiza una comparación entre las tecnologías descritas anteriormente, donde se detalla las ventajas y desventajas del uso de cada una de las mismas. Finalmente se analiza características y funcionalidades más específicas definidas en el estándar publicado por la OMG sobre DDS y su interoperabilidad con el protocolo RTPS.

1.2. MIDDLEWARES

El Middleware es una capa intermedia de software, el cual se encarga de simplificar el manejo y la programación de aplicaciones, tratando de mantener la complejidad de redes y sistemas heterogéneos transparentes al usuario, por lo que se ha convertido en una herramienta esencial para el desarrollo de sistemas distribuidos.

El concepto de Middlewares es muy amplio y cuenta con varias funcionalidades:

- Middleware de Comunicaciones, el cual es una abstracción de los detalles de bajo nivel relacionados con la distribución y la comunicación.
- Middleware de Componentes [1], el cual se basa en un modelo formal que permite el desarrollo de sistemas mediante el ensamblaje de módulos de software reutilizables (componentes), los cuales han sido desarrollados previamente por otros, independientemente de la aplicación que será utilizada.
- Middleware basado en modelos o Middleware MD [2], el cual se centra principalmente en la consecución de un proceso de desarrollo sostenible en términos de costos, tiempos de desarrollo, y la calidad,

combinando un Middleware de componentes con el desarrollo de software basado en modelos.

- Middleware Adaptativo [3], el cual permite la reconfiguración de aplicaciones distribuidas para modificar funcionalidades, el uso de recursos, configuraciones de seguridad, etc.
- Middleware Sensible al Contexto [4], el cual es capaz de interactuar con el entorno en el que las aplicaciones distribuidas se ejecutan y toman acción para hacer cambios en el tiempo de ejecución.

Profundizando en el ámbito de comunicaciones, se toma el primer grupo anteriormente descrito, que corresponde a los Middleware de Comunicaciones, el cual proporciona las bases para el desarrollo de Middlewares de alto nivel. Este maneja internamente los detalles del proceso de interconexión entre nodos que por lo general incluye las siguientes características básicas:

- Direccionamiento o asignación de identificadores a entidades con la finalidad de indicar su ubicación.
- Marshalling¹ o transformación de los datos en una representación adecuada para la transmisión sobre la red.
- Envío o la asignación de cada solicitud a un recurso de ejecución para su procesamiento.
- Transporte o establecimiento de un enlace de comunicaciones para el intercambio de mensajes entre redes vía unicast o multicast.

En la Figura 1-1. Se puede apreciar los servicios básicos que provee un Middleware.

¹ Marshalling, es un mecanismo ampliamente usado para transportar objetos a través de una red.

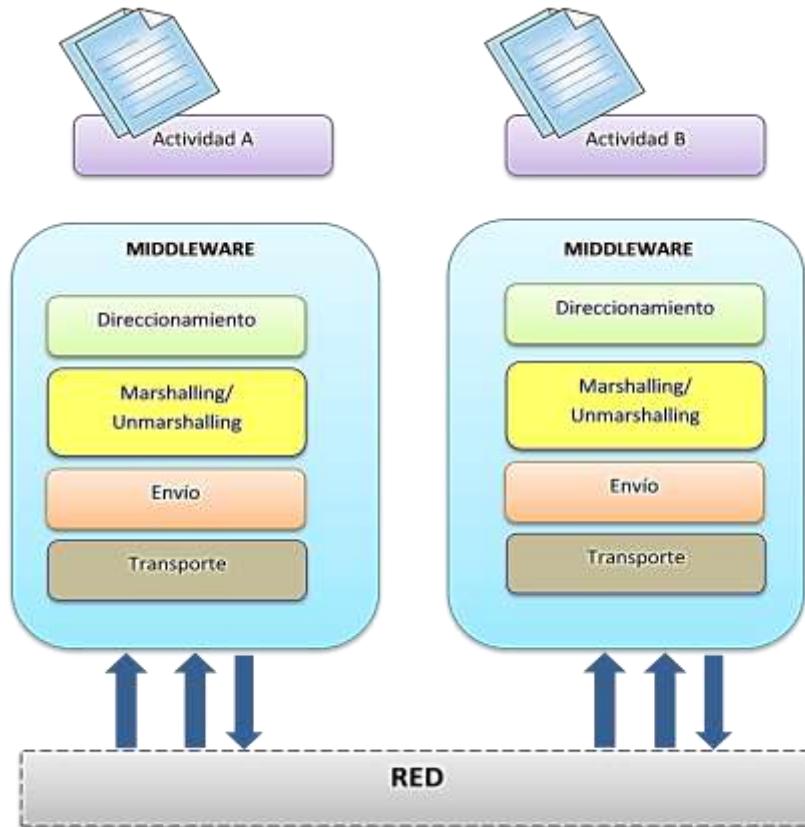


Figura 1-1. Servicios básicos provistos por el Middleware de distribución [5]

1.3. SISTEMAS DISTRIBUIDOS

Debido a la estabilidad y a su impacto en la industria, los Middlewares de comunicación están estandarizados. Existen varios modelos de sistemas distribuidos donde la comunicación ha sido abstraída dentro de Middlewares, entre estos modelos tenemos a:

- RPC
- DOM
- MOM
- Data-Centric Model
- Tuplespaces paradigm

Dentro de estos modelos de sistemas distribuidos se tiene como ejemplos más representativos las siguientes tecnologías detalladas en la Tabla 1-1.

Tabla 1-1. Ejemplos de los modelos de sistemas distribuidos

Modelos de Sistemas Distribuidos	Ejemplos
RPC	<ul style="list-style-type: none"> • Open Software Foundation/Distributed Computing Environment (OSF/DCE) • Distributed Systems Annex of Ada (DSA) [6]
DOM [7]	<ul style="list-style-type: none"> • Common Object Request Broker Architecture (CORBA) [8] • Java Remote Method Invocation (RMI) [9] • Distributed Systems Annex of Ada (DSA)
MOM	<ul style="list-style-type: none"> • Java Message Service (JMS) [10] • Data Distribution Service for Real-Time Systems (DDS) [11]
Data-Centric Model	<ul style="list-style-type: none"> • Data Distribution Service for Real-Time Systems (DDS)
Tuplespaces paradigm	<ul style="list-style-type: none"> • JavaSpaces [12] • Simple Scalable Streaming System (S4) [13] • S-NET [14]

1.4. MIDDLEWARES DE TIEMPO REAL

A diferencia de los sistemas de propósito general, un sistema de tiempo real se define como un tipo especial de sistema cuya corrección lógica se basa tanto en la exactitud como también en la disminución de retardos en la información. Sin embargo, no es suficiente que el software haya sido programado con una lógica correcta; las aplicaciones también deben satisfacer determinadas restricciones temporales. Para este fin, las aplicaciones en tiempo real se basan en un esquema de planificación para especificar un criterio que ordene el uso de recursos del sistema.

El problema de obtener predicciones temporales computacionalmente factibles, fiables y precisas, puede ser resuelto mediante la aplicación de diferentes técnicas analíticas para un solo procesador, multiprocesador o sistemas de tiempo real distribuido [6].

En el caso de los sistemas distribuidos, este proceso es un reto incluso para los sistemas distribuidos aparentemente simples en donde las dependencias complejas entre los datos o subprocesos asignados en diferentes procesadores podrían estar presentes, y por lo tanto, las redes y los procesadores se deben programar juntos [7] [8] [9].

En los sistemas de propósito general, el uso de la tecnología de Middlewares tiene como objetivo facilitar la programación de aplicaciones distribuidas. Con este fin, el Middleware proporciona una abstracción de alto nivel de los servicios ofrecidos por los sistemas operativos, sobre todo los relacionados con la

comunicación. Por lo tanto, los desarrolladores solo son responsables de definir que parte de la aplicación puede ser accesible de forma remota, mientras el Middleware establece y gestiona transparentemente la comunicación entre los nodos del sistema distribuido. Además, los sistemas en tiempo real también se benefician de estas abstracciones de alto nivel.

Sin embargo, los Middlewares de propósito general no se pueden aplicar directamente a sistemas de tiempo real. En general, el proceso de distribución presenta varias posibles fuentes de indeterminismo, incluyendo los algoritmos de marshalling/unmarshalling, transmisión y recepción de colas para mensajes de red, retrasos en el servicio de transporte, o en el envío de solicitudes.

El Middleware de tiempo real apunta a resolver estos problemas mediante la implementación de mecanismos previsibles, tales como el uso de redes de comunicación en tiempo real de propósito especial o la gestión de parámetros de planificación². En consecuencia, este tipo de Middleware se dirige no solo a los problemas de distribución, sino también debe proporcionar a los desarrolladores los mecanismos que permitan que el comportamiento temporal de la aplicación distribuida sea determinístico.

1.4.1. CORBA Y RT-CORBA³

CORBA [10] es un Middleware basado en el modelo de sistema distribuido DOM, el cual utiliza el paradigma Cliente-Servidor y cuya característica principal es facilitar la interoperabilidad entre aplicaciones heterogéneas⁴. CORBA fue desarrollado por un consorcio industrial llamado OMG, una visión general de la arquitectura CORBA se muestra en la Figura 1-2.

² Planificación, se refiere al término scheduling.

³ RT-CORBA, CORBA de tiempo real

⁴ Aplicaciones Heterogéneas, Se refiere a las aplicaciones codificadas en diferentes lenguajes de programación, ejecución en diferentes plataformas y/o las implementaciones de middlewares desarrolladas por diferentes empresas.

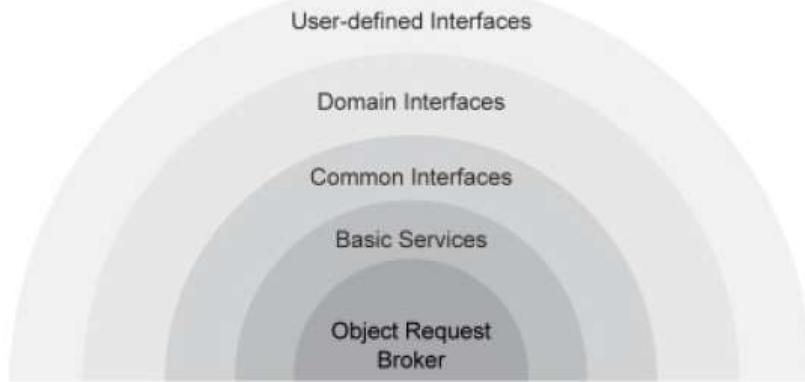


Figura 1-2. Arquitectura de CORBA [5]

Esta arquitectura está integrada por los siguientes componentes:

- *Object Request Broker (ORB)*, representa el núcleo del Middleware y es responsable de coordinar la comunicación entre los nodos cliente y servidor.
- *Interfaces del Sistema*, estas consisten en un conjunto de interfaces agrupadas en función de su ámbito de aplicación, que incluyen:
 - Una colección de servicios básicos, como la concurrencia , la persistencia y la ubicación de los objetos, los mismos que dan soporte al ORB.
 - Un conjunto de interfaces comunes a través de una amplia gama de dominios de aplicación, por ejemplo, la administración de bases de datos, la compresión de la información y la autenticación.
 - Un conjunto de interfaces para un dominio particular de la aplicación, por ejemplo, las telecomunicaciones, la banca y las finanzas.
 - Interfaces definidas por Usuario, las cuales no se encuentran estandarizadas.

Puesto que no hay software, sistema operativo, o lenguaje de programación que reúna todos los requisitos industriales, el objetivo principal de CORBA es proporcionar soluciones para dar soporte a los sistemas heterogéneos, basándose en dos aspectos básicos:

- *Middleware con independencia de lenguaje o Multilenguaje*, los objetos CORBA están definidos por el uso de un lenguaje de descripción llamado Interface Definition Language o IDL. Actualmente, dentro del estándar CORBA existen las normas para la asignación de tipos de datos para varios lenguajes de programación tales como Ada, java o C.
- *Middleware con independencia de plataforma o Interoperabilidad*, CORBA define un protocolo de transporte genérico denominado General Inter-ORB Protocol o GIOP. Este protocolo garantiza la interoperabilidad entre objetos CORBA independientemente de si se asignan a los ORB de diferentes fabricantes o para diferentes plataformas. El protocolo Internet Inter-ORB o IIOP es la especificación de asignación del protocolo GIOP sobre redes TCP/IP, que son consideradas el transporte base para las implementaciones en CORBA.

La comunicación entre los nodos es llevada a cabo mediante el uso de varias entidades CORBA como se ilustra en la Figura 1-3 y se describe a continuación.

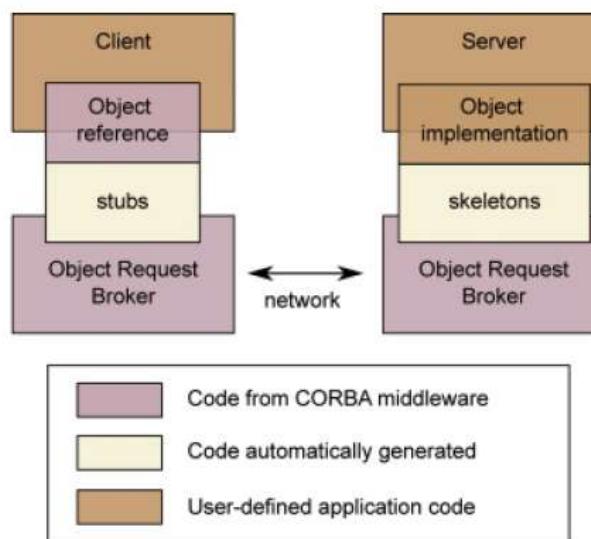


Figura 1-3. Comunicación entre entidades CORBA [5]

- *Object Request Broker*, el ORB proporciona mecanismos para invocar de forma transparente un método remoto como si se tratara de un método local. Por lo tanto el ORB abstracta la ubicación de los objetos remotos y el método de comunicación con ellos.

- *Cliente Stubs y Servidor Skeletons*, estos representan las partes del código, que por lo general son generados automáticamente, a cargo de redirigir la llamada remota a través del ORB, así como la realización de las operaciones de marshalling y unmarshalling.
- *Object Reference*, esta no es una referencia clara que únicamente determina la localización de un objeto remoto y se la conoce como Interoperable Object Reference o IOR. El IOR incluye detalles de todos los protocolos de red y puertos receptores que el ORB puede utilizar para procesar las solicitudes entrantes. Esta referencia es generada y gestionada por el Portable Object Adapter o POA.
- *Redes de Comunicación*, los nodos , el cliente y el servidor se comunican a través del ORB usando el protocolo GIOP. Este protocolo está en la parte superior de la capa transporte del modelo OSI y puede ser implementado en la parte superior de varios protocolos de red; sin embargo, el estándar CORBA solo incluye directrices para implementarlo en redes basadas en IP.

Aunque CORBA proporciona soporte completo para objetos distribuidos, este estándar no incluye soporte para aplicaciones en tiempo real. Por lo tanto, esta falencia fue abordada por la OMG a través de un conjunto opcional de extensiones para CORBA que fueron llamadas RT-CORBA [11]. Estas extensiones se describen a continuación:

- *RT-ORB*, una extensión ORB, que añade funciones para la creación y la destrucción de entidades específicas de tiempo real, por ejemplo, los Mutex, los threadpools, o las políticas de planificación y permite la asignación de prioridades para su uso por hilos internos ORB.
- *RT-POA*, representa una extensión del POA [10] y provee soporte para la configuración de las políticas de tiempo real definidas por RT-CORBA. Estas políticas manejan los modelos de prioridad de propagación de extremo a extremo, la gestión de llamadas remotas, la prioridad en conexiones agrupadas, o la selección y configuración de los protocolos de red disponibles.
- *Prioridad y Asignación de Prioridad*, estas representan un interfaz que definen un tipo de datos de prioridad genérica, es decir

independientemente del sistema operativo, y proporcionan operaciones para asignar prioridades nativas dentro de las prioridades RT-CORBA y viceversa.

- *Mutex*, es una interfaz portable para acceder a los Mutex proporcionados por la RT-ORB. Esta proporciona mecanismos de sincronización para controlar el acceso a los recursos compartidos.
- *RTCurrent*, es una interfaz para determinar la prioridad de la invocación actual, es decir, permite a la prioridad de los hilos de aplicaciones que sean manejados.
- *ThreadPool*, es un mecanismo para controlar el grado de concurrencia durante la ejecución de las llamadas remotas en el lado del servidor.
- *Servicio de Planificación*, es un servicio que simplifica la configuración de aspectos de sincronización del sistema. Por medio de este servicio, RT-CORBA permite que la aplicación especifique sus requerimientos en función de diversos parámetros tales como las prioridades, lazos, o plazos de ejecución previstos, mientras que el Middleware será responsable de la creación de los recursos necesarios para cumplir con ellos.

El uso de estas entidades RT-CORBA permite el desarrollo de sistemas de tiempo real críticos como sistemas de control en tiempo real, así como sistemas no críticos como el de las agencias de viajes o sistemas de compra en línea. Actualmente RT-CORBA se emplea en una amplia gama de escenarios como por ejemplo: en robótica industrial [12] la cual puede considerarse una tecnología madura.

1.4.2. THE ADA DISTRIBUTED SYSTEMS ANNEX

El lenguaje de programación Ada [13] es un estándar internacional que incluye un anexo dedicado al desarrollo de aplicaciones distribuidas, el cual corresponde al anexo E o Ada DSA. La principal fortaleza de DSA es que el código fuente está escrito sin tener en cuenta de si va a ser ejecutado en una plataforma distribuida o en un solo procesador.

En el diseño de sistemas distribuidos, una aplicación diseñada para un solo procesador puede ser dividida en diferentes funcionalidades tales que, cuando

actúen juntos puedan proporcionar una servicio particular para usuarios finales. La ejecución de cada una de estas funcionalidades pueden ser distribuidas a través de varios nodos interconectados, mientras que en los usuarios finales se invoca de forma transparente el servicio. En el lenguaje de programación Ada, cada parte de la aplicación se asigna de forma independiente a cada nodo la cual es llamada partición. Formalmente, de acuerdo al manual de referencia de Ada, “una partición es un programa o parte de un programa que puede ser invocado desde fuera de la aplicación Ada.” [13]

El particionamiento de una aplicación de la DSA no está definido en el estándar, pero su implementación está definida. Las particiones se comunican entre sí mediante el intercambio de datos a través de las RPC y de los objetos distribuidos. La DSA define dos tipos de particiones: activos, los cuales pueden ser ejecutados en paralelo uno con otro, posiblemente en direcciones separadas de memoria y en computadores independientes; y pasivos, los cuales son particiones sin una tarea o hilo de control, por ejemplo, los nodos de almacenamiento.

Las particiones activas se comunican a través del subsistema de comunicación de particiones o PCS, un interfaz definido por lenguaje es responsable del subprograma de enrutamiento de llamadas de una partición a otra. El acceso a la PCS no debe hacerse directamente desde la capa aplicación, sino este debe hacerse por medio de los Stubs de llamadas y recepción. El PCS soporta compiladores usados para generar Stubs de una interfaz estándar sin preocuparse de la implementación. A pesar de este esfuerzo de normalización , una reciente revisión del lenguaje de programación [14] permite el uso de interfaces alternativas para PCS con el fin de facilitar la interoperabilidad con otro Middleware.

Los componentes de alto nivel del modelo de distribución propuesto por la DSA están ilustrados en la Figura 1-4. Esta figura representa el diagrama de secuencia de una llamada remota síncrona entre dos particiones: una partición que requiere servicios remotos y una que proporciona estos servicios a través de un interfaz de llamada remota.

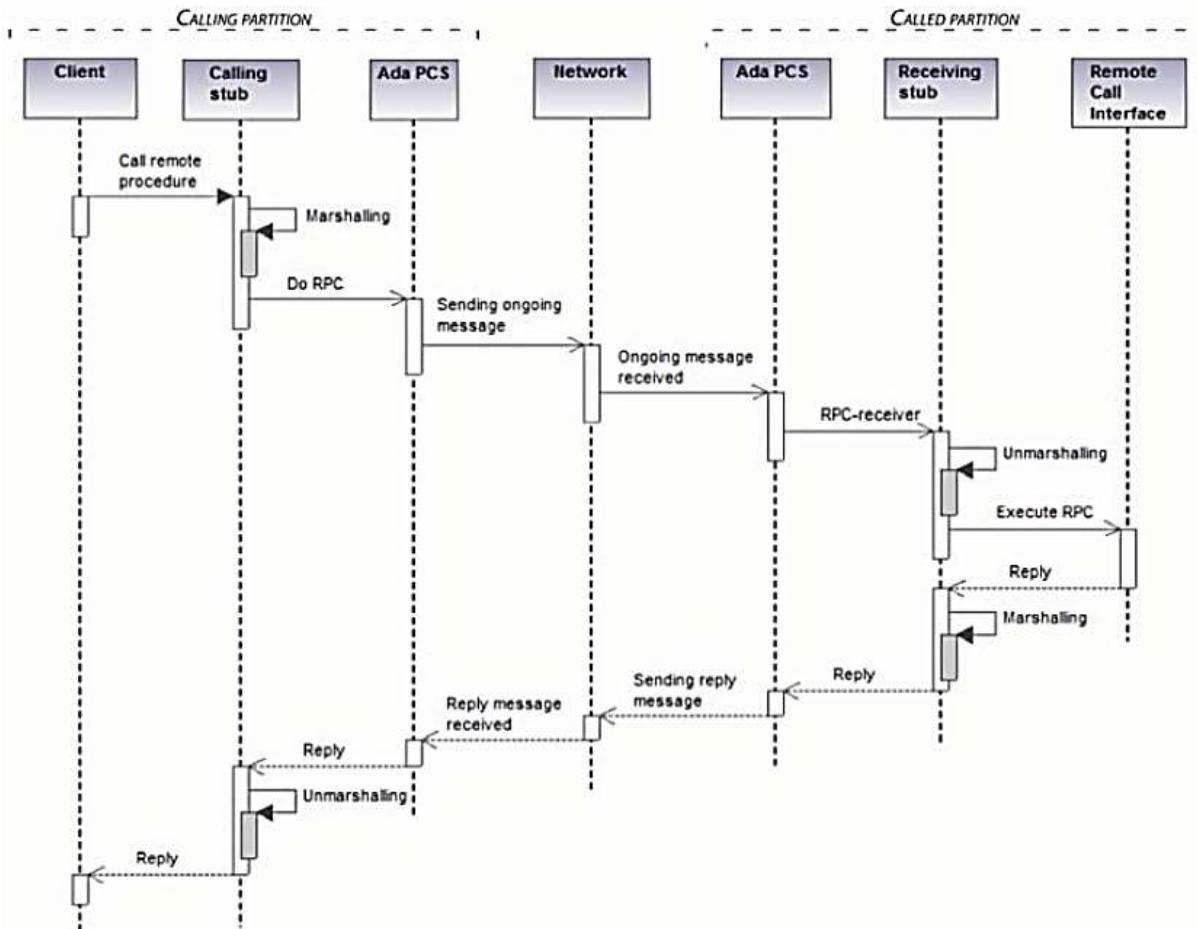


Figura 1-4. Diagrama de secuencia de una llamada remota síncrona [5]

Aunque la DSA permite que los sistemas distribuidos sean construidos de manera simple, no están específicamente diseñados para soportar aplicaciones previsibles, y la mayoría de los problemas que afectan el determinismo se han quedado en la implementación. Sin embargo, existen algunas investigaciones previas en este campo, y hay implementaciones que muestran que pueden ser usadas para aplicaciones de tiempo real [15]. Aunque este anexo no ha tenido un impacto comercial muy significativo [16], Ada ha sido tradicionalmente usado y aun se usa para construir los sistemas de un solo procesador en tiempo real, así que vale la pena considerar el análisis de esta norma y su desarrollo futuro.

1.4.3. THE DISTRIBUTED REAL-TIME SPECIFICATION FOR JAVA

Además de las normas de distribución, hay otras soluciones no estandarizadas que han despertado gran interés entre los desarrolladores. Este es el caso del lenguaje de programación Java y sus extensiones para sistemas distribuidos en tiempo real, el cual es un estándar de facto.

Java fue diseñado inicialmente como un lenguaje de programación para sistemas de propósito general y, por lo tanto, tiene varios inconvenientes para el desarrollo de aplicaciones previsibles, especialmente aquellos aspectos relacionados con la gestión de los recursos internos como la memoria o la programación del procesador [17]. Para los sistemas distribuidos de tiempo real, uno de los trabajos de investigación más notable es la Distributed Real-Time Specification for Java o DRTSJ [18], que integra dos tecnologías existentes de Java:

- *Real-Time Specification for Java o RTSJ* [19], el cual define una especificación de Java para abordar las limitaciones del lenguaje cuando se usa en sistemas de tiempo real. Como una de las principales directrices fue evitar hacer extensiones sintácticas del lenguaje, se logró el soporte en tiempo real a través de nuevas bibliotecas, un mejor mecanismo de Java, y una Máquina Virtual de Java en tiempo real o JVM con soporte tanto para de propósito general y aplicaciones en tiempo real. Sin embargo, esta especificación fue concebida solo para sistemas de un solo procesador.
- *Remote Method Invocation o RMI* [20], el cual define un modelo DOM basado en objetos Java que definen una nueva interfaz, llamada remota, lo que permite la diferenciación de los objetos distribuidos de los locales. Un visión general del alto nivel de los componentes implicados en la arquitectura RMI, la cual se muestra en la Figura 1-5, que representa el diagrama de secuencia de una llamada remota asíncrona entre un cliente y un servidor. Esta arquitectura tiene los siguientes componentes.

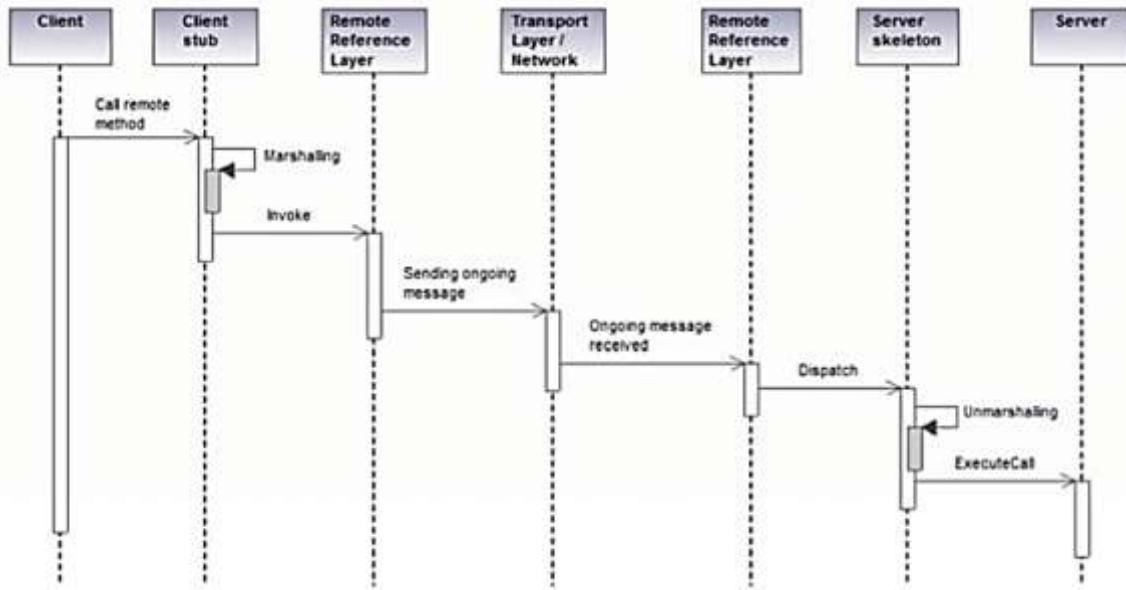


Figura 1-5. Diagrama de secuencia de una llamada remota asíncrona [5]

- *Cliente Stubs o Proxy* y *Servidor Skeleton*, los cuales representan la interfaz entre la capa aplicación y el resto del sistema RMI, son los encargados de proporcionar servicios de distribución transparentes.
- *Remote reference Layer*, el cual se encarga de manipular la semántica de las invocaciones remotas, tanto en la parte del cliente como en el servidor.
- *Capa Transporte*, el cual se utiliza para establecer las conexiones y gestionar los detalles de la comunicación de bajo nivel. La especificación define la conexión del protocolo RMI, que se basa en dos protocolos: Serialización de Objetos Java y HTTP.

Aunque Java es uno de los lenguajes de programación más populares, no define la especificación DRTSJ, aún no se ha difundido, aunque existe un primer borrador [21] y en el sitio Web del grupo de trabajo [18], la cual describe las características importantes de la futura especificación. Sin embargo, varias líneas de la investigación pretenden adaptar el lenguaje de programación a un modelo determinístico no solo para ambientes con un solo procesador, sino también para los distribuidos [22] [17].

1.4.4. THE DATA DISTRIBUTION SERVICE FOR REAL-TIME SYSTEMS

La difusión asíncrona de la información ha sido un requisito común para varias aplicaciones distribuidas, tales como sistemas de control, sensores de redes

y los sistemas de automatización industrial. El DDS [23] tiene como objetivo facilitar el intercambio de datos en este tipo de sistemas a través del paradigma publicador-suscriptor. A diferencia de otros Middlewares las especificaciones que siguen este paradigma, la comunicación está centrada en los datos propuesto por el modelo DDS, es decir, la atención se centra en los datos en sí. En una arquitectura data-centric se debe formalmente definir el tipo de datos que se comparte en la periferia del sistema, y luego la información se intercambia de forma anónima simplemente escribiendo y leyendo este tipo de datos. Con un enfoque centrado en los datos, el Middleware es consciente del contenido de la información intercambiada y de lo que puede manejar directamente él, por ejemplo, la filtración de datos.

Al igual que con la mayoría de los estándares que se define dentro de la OMG, el DDS soporta multilenguaje y multiplataforma mediante el uso del lenguaje IDL [10] para definir tipos de datos compartidos y el DDSI [24] para interoperar entre diferentes implementaciones. Más allá de esto, la OMG tiene publicado el Extensible and Dynamic Topic Types specification [25], que proporciona apoyo a los sistemas distribuidos extensibles y evolutivos utilizando DDS. Esta especificación permite a los tipos de datos ser definidos dinámicamente, es decir pueden ser utilizados sin compilación, o modificados, es decir, los campos de los datos se pueden agregar o quitar. Para este fin, el DDS proporciona un sistema de datos estructurales, nuevas representaciones de tipos de datos, diferentes formatos de serialización o de codificación, y una nueva API para la gestión de los tipos de datos en tiempo de ejecución.

El modelo conceptual DDS se basa en la abstracción de un tipo de datos globales, donde el Publicador y el Suscriptor escriben (producir) y leen (consumir) datos, respectivamente; lo que lleva al Middleware centralizado a la obtención de datos de forma independiente de su origen.

Para manejar mejor el intercambio de datos, el estándar define un conjunto de entidades que participan en el proceso de comunicación. Las aplicaciones que desean compartir información con otras, pueden utilizar este espacio de datos globales y declarar su intención de publicar los datos a través de la entidad DW. Del mismo modo, las aplicaciones que necesiten recibir información pueden utilizar la entidad DR para solicitar datos particulares. Las entidades Publicadoras y Suscriptoras son contenedoras de los DW y DR, respectivamente, los cuales

comparten los parámetros de QoS comunes. Del mismo modo, estas entidades se agrupan en un Dominio. Solo las entidades que pertenecen al mismo dominio pueden comunicarse. En capas superiores, todos las entidades participantes contienen todos los DW y DR, Publicador y Suscriptor que comparten un QoS común en el Dominio correspondiente.

Para intercambiar información entre las entidades, el Publicador solo necesita saber acerca de un tema específico, como por ejemplo, el tipo de dato para compartir y el Suscriptor requiere el registro de su interés en recibir determinados temas, mientras que el Middleware puede establecer y gestionar la comunicación de forma transparente. Dentro de la definición de un tema, uno o más elementos pueden ser designados como una *lave*. Esta entidad permite la existencia de múltiples instancias del mismo tema, permitiendo así que los DR diferencien la fuente de origen de los datos, por ejemplo, un grupo de sensores de temperatura que proporcionan información de diferentes áreas. La Figura 1-6 muestra un sistema distribuido que consta de tres participantes en un solo Dominio y dos tópicos.

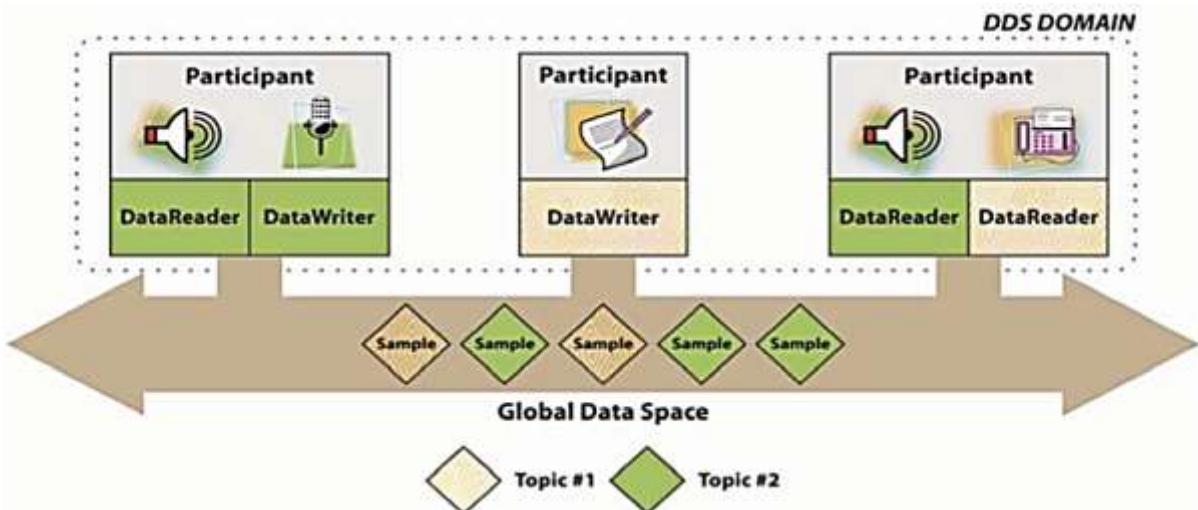


Figura 1-6. Sistema Distribuido que consta de tres participantes en un solo Dominio [5]

Ambos tópicos tienen un solo DW, el cual es el encargado de generar nuevo grupos de datos. Sin embargo, las actualizaciones sucesivas del Tópico #1 solo serán recibidos por un solo DR, mientras que las nuevas muestras del Tópico #2 serán recibidos por dos DR.

Los Publicadores y Suscriptores no están obligados a comunicarse directamente entre sí, pero están relacionados de forma flexible en función de lo siguiente:

- *Time*, los datos pueden ser almacenados y recuperados después, por ejemplo, cuando un nuevo Suscriptor se une al sistema distribuido y requiere información sobre el anterior estado del sistema.
- *Space*, porque para los Publicadores de datos no es necesario saber acerca de cada receptor individual, mientras que los Suscriptores no necesitan conocer la fuente de origen de los datos, los Publicadores y Suscriptores no se conocen entre sí.

Como se mencionó anteriormente, el desarrollo de sistemas distribuidos con DDS está unido a otra especificación que establece las principales directrices para la realización de la comunicación entre las entidades: el DDSI. Este protocolo tiene como objetivo garantizar la interoperabilidad entre diferentes implementaciones, utilizando el estándar del protocolo RTP [24] en tiempo real del Publicador-Suscriptor, junto con la Representación de Datos Común o CDR, el cual se define en CORBA [10]. Aunque esta especificación se centra en las redes IP, ningún otro protocolo de red en tiempo real puede ser usado. Por último, aunque DDS ha sido diseñado para ser escalable, eficiente y predecible, pocos investigadores han evaluado sus capacidades en tiempo real [26]. Sin embargo se considera una tecnología madura y ya se ha desplegado en varios escenarios en tiempo real, tales como sistemas de Defensa Nacional [27], Automatización [28], o Espacio [29].

1.5. COMPARACIÓN ENTRE LAS DIFERENTES TECNOLOGÍAS DE MIDDLEWARES DE COMUNICACIÓN DE TIEMPO REAL.

Después de analizar los diferentes estándares de distribución orientadas al desarrollo de aplicaciones en tiempo real, se halla las semejanzas y diferencias entre estas tecnologías, así como evaluar su capacidad para ser utilizados en los sistemas de tiempo real. Para una mejor comparación, primeramente se explica una serie de requisitos que un estándar de distribución para sistemas de tiempo real deben considerar [5]:

- *El soporte para la planificación en procesadores y redes*, los sistemas de tiempo real requieren un estricto control de la ejecución de hilos y de la transmisión de mensajes. Esto incluye el soporte para las políticas de planificación encargadas de ordenar el acceso concurrente de los hilos y de los mensajes a los procesadores y redes de comunicación, respectivamente.
- *Control de parámetros de planificación*, el Middleware debe proveer mecanismos para configurar los parámetros de planificación de cada hilo que pueden ser ejecutados en el procesador. Así como soportar, la asignación de parámetros de planificación a mensajes de red.
- *Gestión de Hilos o Patrones de concurrencia*, representa el patrón de diseño que trata sobre el paradigma de multi-hilos, que controla y procesa la difusión de la información. Esta característica es especialmente importante en el lado del receptor.
- *El acceso controlado a recursos compartidos*, esto se puede lograr a través de la aplicación de protocolos de sincronización, como locks, mutex y semáforos.

Hay dos tipos de entidades de planificación que pueden ser identificadas en sistemas de tiempo real: Hilos para procesadores y Mensajes para redes de comunicaciones. Además, esas entidades también pueden producir algún tipo de contención⁵ que debe ser tenido en cuenta en el diseño en tiempo real.

1.5.1. GESTIÓN DE LOS RECURSOS DEL PROCESADOR

El comportamiento temporal de Middleware de distribución está fuertemente determinado por las políticas de planificación y los patrones de concurrencia [30]. En el caso de las políticas de planificación, es necesario identificar cuales mecanismos están provistos dentro del Middleware para seleccionar una política específica de planificación para las entidades planificables responsables de atender a los servicios remotos. En el caso de los patrones de concurrencia se trata con las opciones disponibles para determinar cuál hilo es responsable para el envío o recepción de peticiones/ datos remotos.

⁵ Contención, se refiere a la ocupación del canal de comunicaciones.

Primeramente es posible que los planificadores estén soportados directamente en el sistema operativo. Sin embargo mientras estos estándares de distribución tienen por objetivo el desarrollo de sistemas de tiempo real, sería conveniente incluir las operaciones en sus APIs para establecer políticas de planificación específicas y sus correspondientes parámetros de planificación de los hilos del Middleware.

Ambas especificaciones Ada y RT-CORBA dan soporte a diferentes políticas de planificación, incluyendo las políticas FPS. Igualmente, el marco de planificación definido por DRTSJ puede ser ampliado con el fin de soportar diferentes políticas de planificación. Sin embargo, el modelo propuesto en DDS no incluye planificación en los procesadores lo cual no está definido en el estándar. Aunque el estándar DDS define varios parámetros temporales ninguno es apropiado para construir hilos en el procesador: parámetro *DeadLine*⁶ que puede ser utilizado en algunos casos, como por ejemplo, sistemas EDF [31], pero en el estándar no se considera su uso. Una situación similar se da con el parámetro *Latency_Budget* que propone una dosificación de datos, como por ejemplo, reunir un conjunto de muestras de datos a ser enviado en un solo paquete de red de gran tamaño.

RT-CORBA es la única especificación que proporciona mecanismos para especificar los parámetros de planificación que se utilizarán durante la ejecución de las operaciones solicitadas en el nodo remoto. La especificación para sistemas estáticos define dos políticas, *Server_Declared* y *Client_Propagated*, que impone restricciones en la asignación de prioridades y luego reduce la planificabilidad del sistema [32]. Además, aunque el modelo de transformación de prioridades permite la modificación de las políticas, esta se lleva a cabo dentro del código de aplicación suministrado, por lo que cualquier cambio en los parámetros de planificación debe ser revisado.

El procesamiento de llamadas remotas o datos entrantes representan un proceso de dos etapas que incluye: la escucha de eventos de E/S en las redes de comunicación y el procesamiento de mensajes de red y la ejecución del código de la aplicación asociado a llamadas remotas o datos entrantes, y una posible respuesta. La escucha de eventos de E/S es internamente realizado y controlada

⁶ Deadline, fija la separación máxima de tiempo entre dos actualizaciones.

por el Middleware, mientras que la ejecución del código de aplicación se refiere a la interacción entre el Middleware y la aplicación. Los estándares de distribución no definen que patrón de concurrencia debe ser usado en la parte de escucha de eventos pero especifica que implementación debe atender las solicitudes remotas concurrentes, por ejemplo, Ada DSA indica explícitamente este aspecto, mientras que RT-CORBA indica implícitamente por medio de la definición de los *Threadpools*. En cuanto a la ejecución del código de aplicación, RT-CORBA y Ada DSA dirigen la ejecución del código de la aplicación en el contexto de un hilo interno del Middleware, mientras DDS permite el uso de hilos internos del Middleware, por medio del mecanismo de escucha, o la aplicación de hilos, a través de estructuras de espera y establecimiento o por pedido de la disponibilidad de los datos en un DR. De todos modos, independientemente del hilo o hilos responsables del procesamiento de cada etapa, es importante que el Middleware proporcione los mecanismos necesarios para controlar sus parámetros de planificación.

Finalmente, el acceso determinístico de los recursos compartidos evita el problema de inversión de prioridades [9]. RT-CORBA, Ada y RTSJ incluyen el uso de protocolos de sincronización para el acceso a secciones críticas, aunque solo los dos últimos especifican que implementaciones necesitan soportar protocolos específicos, por ejemplo, el priority ceiling protocol⁷. [33]

1.5.2. GESTIÓN DE RECURSOS DE RED

En relación a las redes de comunicaciones, ni RT-CORBA ni Ada DSA incluyen la posibilidad de asignar parámetros de planificación; por lo tanto, las implementaciones son responsables de proveer el soporte necesario. En cuanto DRTSJ, su última publicación solo establece que tanto en tiempo real como en redes de propósito general la gestión de recursos de red debe ser soportada. En el caso de DDS, la especificación solo considera redes basadas en políticas de planificación con prioridad fija y excluye cualquier tipo de redes predecibles utilizadas en la industria, por ejemplo, time-triggered networks⁸. Esto puede ser tratado mediante la modificación de la definición del parámetro *Transport_Priority* [26]

⁷ Priority ceiling protocol, es un protocolo que se utiliza en tiempo real para gestionar el acceso a recursos compartidos, evitando la inversión de prioridad y la exclusión mutua.

⁸ Time-triggered networks, es una red basada en un protocolo abierto para sistemas controlados, diseñada para aplicaciones industriales y redes de vehículos.

Aunque la mayoría de las normas analizadas se centran en las redes basadas en Ethernet, como por ejemplo, RT-CORBA con TCP/IP y DDS con UDP/IP, esta red de comunicación no es por sí apta para proporcionar tiempos de respuesta determinísticos.

Sin embargo, la evolución de la tecnología Ethernet en los últimos años, con la definición de nuevos estándares, como 802.1p [34], el cual prioriza los diferentes mensajes, junto con su bajo costo, ha dado lugar a un creciente interés en la industria en el uso de este enfoque en el desarrollo futuro de los sistemas en tiempo real.

Cuando el Middleware de distribución se lleva a cabo en los sistemas operativos y en los protocolos de red con la planificación basada en prioridades, es fácil de transmitir la prioridad en la que un servicio remoto debe ser ejecutado dentro de los mensajes enviados a través de la red, por ejemplo este esquema es utilizado por la política *Client_Propagated* en RT-CORBA. Sin embargo, esta solución no funciona si las políticas de planificación son complejas, tales como el marco flexible de planificación basado en los contratos utilizados [35] [36]. El envío de los parámetros del contrato a través de la red es ineficiente porque son de gran tamaño. De la misma manera, el cambio dinámico de los parámetros del contrato en el nodo remoto es también ineficiente, por lo tanto, otros esquemas de configuración son requeridos para este tipo de sistema.

Otro factor importante a considerar es el tamaño de los mensajes de red, el cual debe ser delimitada y conocida antes del análisis de temporización. Este punto es particularmente crítico en el diseño de aplicaciones previsibles con DDS, mientras que a un mensaje DDSI puede comprender un número indefinido de mensajes, incluyendo no solo el metatráfico, sino también los datos de usuario.

Además este mecanismo es poco eficiente para minimizar el tiempo de respuesta promedio, no suele ser adecuado para sistemas de tiempo real que tienen por objetivo garantizar los límites de la latencia en cada flujo de red. Por lo tanto, depende de la implementación proporcionar los medios para definir el tamaño máximo de un mensaje DDSI.

Finalmente, la presencia de los mensajes y operaciones que pertenecen al Middleware puede provocar un incremento en los tiempos de respuesta de aplicaciones críticas. Aunque, esta sobrecarga depende casi exclusivamente de

cada aplicación, este efecto es más significativo en estándares tal como DDS, el cual define un conjunto de entidades que pueden consumir recursos del procesador y de la red.

1.5.3. CUADRO COMPARATIVO DE LAS DIFERENTES TECNOLOGÍAS

En la Tabla 1-2 se resume el análisis de acuerdo al grado de soporte de los requerimientos propuestos en la sección anterior, como las políticas de planificación, los parámetros de configuración de la planificación, los patrones de concurrencia, y el acceso controlado a los recursos compartidos.

La mayoría de estas características, las cuales son requeridas en el peor caso del comportamiento temporal, permanecerá abierto a las implementaciones, como las mostradas en la Tabla 1-2. En consecuencia, la elección de un Middleware particular, determina no solo el rendimiento de las aplicaciones, sino también su previsibilidad, y por lo tanto la capacidad de cumplir con la separación máxima de tiempo entre dos actualizaciones. La elección del patrón de concurrencia para el procesamiento de llamadas remotas o datos entrantes es particularmente relevante, aunque esta característica dependa de las implementaciones. Para lo cual el Middleware escogido de acuerdo a la priorización de transporte, es DDS.

Finalmente, en la Figura 1-7 se muestra la evolución de estos estándares de distribución de tiempo real.

Tabla 1-2. Capacidades de Tiempo-Real de los Estándares de Distribución [5].

Middleware	Recursos del Procesador				Recursos de Red	
	Políticas de Planificación	Configuración de los Parámetros de Planificación	Patrones de Concurrencia	Protocolos de Sincronismo	Políticas de Planificación	Configuración de los Parámetros de Planificación
RT-CORBA	FPS	Client_Propagated	Threadpool	Requerido	Implementación definida	Implementación definida
	EDF	Server Declared				
	LLF	Priority_Transfer				
	MAU					
	FPS					
Ada DSA	No apropiable	Implementación definida	Implementación definida	Priority Ceiling	Implementación definida	Implementación definida
	Round-robin					
	EDF					
DDS	Implementación definida	Implementación definida	Implementación definida	Implementación definida	FPS	Prioridad de Transporte
	FPS	Implementación definida	Implementación definida	Priority inheritance	Implementación definida	Implementación definida
Java DRTSJ	Usuario definido			Priority Ceiling		

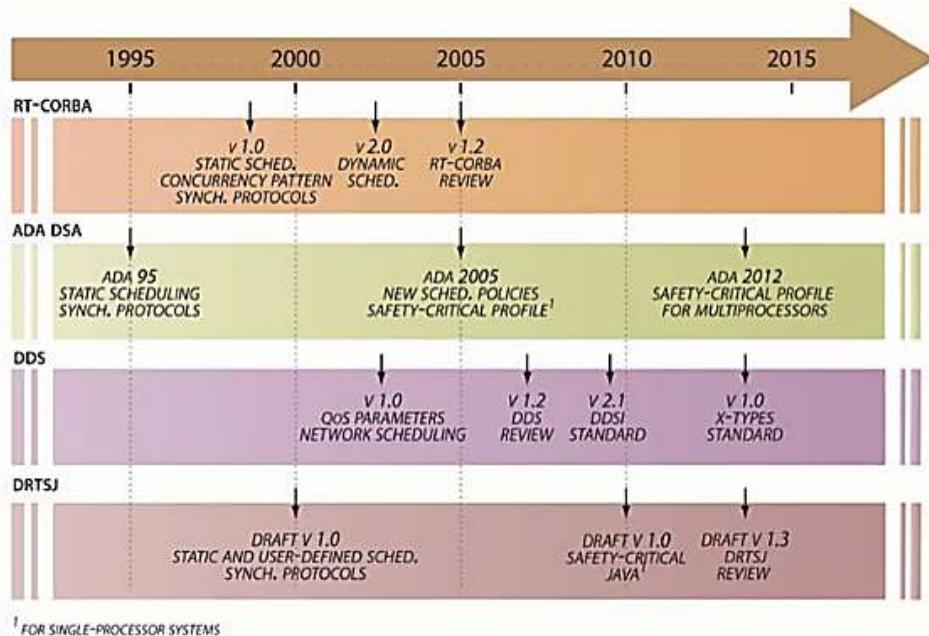


Figura 1-7. Línea de tiempo en los estándares de tiempo real [5]

1.6. CARACTERÍSTICAS Y FUNCIONALIDADES DEL DDS

1.6.1. CARACTERÍSTICAS

1.6.1.1. Arquitectura

Una arquitectura Publicador-Suscriptor promueve un bajo acoplamiento en la arquitectura de datos y es flexible y dinámica; esta es fácil de ser adaptada y extendida a sistemas basados en DDS. El modelo Publicador-Suscriptor conecta a los generadores de información anónima o el Publicador con los consumidores de información o Suscriptor. La mayoría de aplicaciones distribuidas que utilizan el modelo Publicador-Suscriptor están compuestos de procesos, cada uno corriendo por espacios de memoria separados y comúnmente en diferentes computadoras. Se denominará a cada uno de estos procesos como participantes. Un participante puede simultáneamente publicar y suscribir información. El aspecto definido en el modelo Publicador-Suscriptor se refiere al desacoplamiento tanto en espacio, tiempo y flujo entre publicador y suscriptor.

La información transferida por comunicaciones de tipo *data-centric* pueden ser clasificadas en: señales, flujos, y estados.

Las señales, representan los datos que están en continuo cambio, por ejemplo la lectura de un sensor. Las señales pueden trabajar análogamente como IP , es decir haciendo su mejor esfuerzo.

Los flujos, representan capturas de valores de un *data-object* que puede ser interpretada en el contexto de una captura previa. Los flujos necesitan tener confiabilidad.

Los estados, representan el estado de un conjunto de objetos o sistemas codificados como el valor más actual de un conjunto de atributos de datos o de estructuras de datos. El estado de un objeto no cambia necesariamente en cualquier periodo. Los rápidos cambios pueden ser seguidos por intervalos largos sin cambios en el estado. Los participantes que requieren información del estado, se encuentran típicamente interesados en el valor más actual. Sin embargo, como el estado puede no cambiar a lo largo del tiempo, el Middleware puede necesitar asegurar que el estado más actual entregue confiabilidad. En otras palabras, si el valor se ha perdido, entonces no es siempre aceptable esperar hasta que el valor vuelva a cambiar.

El objetivo del estándar DDS, es facilitar una distribución eficiente de la información en un sistema distribuido. Los participantes usando DDS pueden leer y escribir datos eficientemente y naturalmente⁹ con un tipo de interfaz. Por debajo, el Middleware DDS distribuye los datos, por lo tanto cada lector puede acceder a los valores más actuales. Por tanto, el servicio crea un espacio de datos global donde cualquier participante puede leer como escribir. Este también crea un nombre de espacio que permite a los participantes encontrar y compartir objetos.

El objetivo de DDS son los sistemas de tiempo real; el API y QoS han sido escogidos para balancear el comportamiento predecible y la eficiencia/ rendimiento de la implementación. Una visión general de la Arquitectura se muestra en la Figura 1-8.

⁹ Naturalmente, se refiere a que la interfaz puede ser similar a una ya usada por variables locales lectura/ escritura.

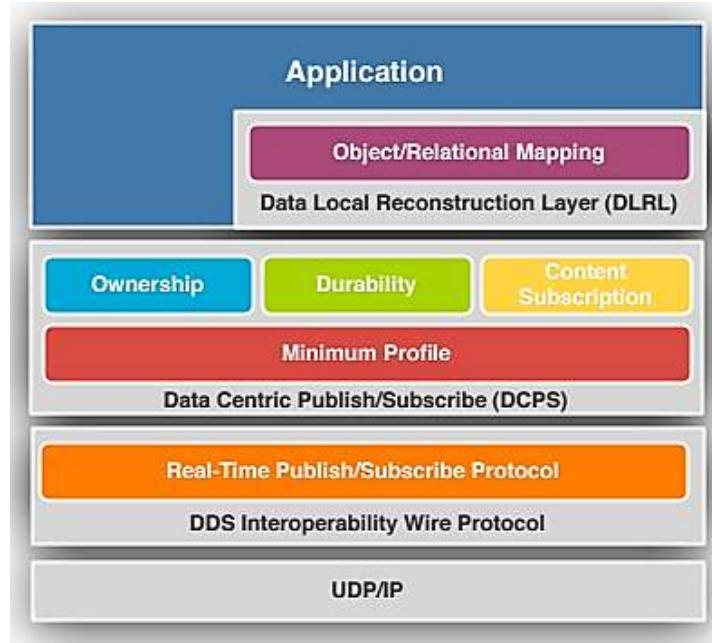


Figura 1-8. Arquitectura del Middleware DDS [37]

El estándar DDS describe dos niveles de interfaces y un nivel de comunicaciones:

- Un nivel bajo denominado *data-centric Publish-Subscribe o DCPS*, el cual está orientado a la entrega eficiente de información adecuada a los destinatarios correctos.
- Un nivel opcional alto denominado *data-local reconstruction layer o DLRL*, el cual permite una integración simple a la capa de aplicación.
- El nivel de comunicaciones se denomina *DDS Interoperability Wire Protocol*, el cual opera con el protocolo RTPS.

1.6.1.1.1. DCPS o Data-Centric Publish-Subscribe

La comunicación es llevada a cabo con la ayuda de las siguientes entidades: *DomainParticipant*, *DataWriter*, *DataReader*, *Publisher*, *Subscriber*, y *Topic*. Todas estas entidades son representadas por clases que extienden a la *DCPSEntity*, la cual muestra su capacidad de ser configurada a través de las políticas de QoS, ser notificado de eventos vía, y soportar condiciones que pueden ser esperadas por la aplicación. Cada especialización de una clase base *DCPSEntity* tiene un

correspondiente *Listener* especializado y un conjunto de valores de *QoS Policy* que son deseables.

El Publicador representa a los objetos responsables para la emisión de datos. Un Publicador debe poner a disposición datos de diferente tipo.

Un *DataWriter* es la cara al Publicador; los participantes usan *DataWriter* para comunicar el valor y los cambios en los datos de un determinado tipo. Una vez que la nueva información ha sido comunicada al Publicador, es la responsabilidad del Publicador determinar cuándo es apropiado emitir el correspondiente mensaje y llevar a cabo realmente la emisión, es decir el Publicador hará esto de acuerdo a su calidad de servicio o a la QoS asociada al correspondiente *DataWriter*, y/ o a su estado interno.

El Suscriptor recibe los datos publicados y los hace disponibles al participante. Un Suscriptor debe recibir y despachar datos de diferentes tipos especificados. Para acceder a los datos recibidos, el participante debe utilizar un tipo *DataReader* asociado al suscriptor.

La asociación de un objeto *DataWriter*, el cual representa a una publicación, con el objeto *DataReader*, que representa la suscripción, es hecha por la entidad *Topic*.

La entidad *Topic*, asocia un nombre que es único en el sistema, un tipo de dato, y QoS relacionado a su propia información. La definición de Tipo entrega suficiente información para que el servicio manipule los datos, por ejemplo, serializa los datos dentro de un formato de red para ser transmitido. La definición de Tipo puede ser hecha por medio de un lenguaje textual, por ejemplo algo como “*float x; float y;*” o por medio de un “*plugin*” operacional, el cual provee los métodos necesarios.

DCPS puede también soportar suscripciones con filtrado del tipo *content-based*¹⁰ por medio de un filtro el cual corresponde a las políticas de QoS. Esta es una característica opcional ya que el filtrado del tipo *content-based* ocupa muchos recursos e introduce retardos que son difíciles de predecir.

La capa DCPS se encuentra compuesta de cinco módulos: infraestructura, tópico, publicación, suscripción, y dominio.

¹⁰ *Content-based filtering*, es una técnica de filtro de información, utilizando distintos ítems de información

- El módulo Infraestructura contiene las clases *DCPSEntity*, *QoS Policy*, *Listener*, *Condition*, y *WaitSet*. Esta abstracción de clases soporta dos estilos de interacción: *notification-based* y *wait-based*. Estos implementan interfaces que son mejoradas en otros módulos.
- El módulo *Topic-Definition* contiene a las clases *Topic* y al *TopicListener* y generalmente todo lo que es necesario para que la aplicación defina sus tipos de datos, cree tópicos, y asocie QoS a estos.
- El módulo publicación contiene las clases Publicador, el *DataWriter* y el *PublisherListener*, y generalmente todo lo que es necesario en el lado de la publicación.
- El módulo suscripción contiene las clases Suscriptor, el *DataReader*, y el *SubscriberListener*, y generalmente todo lo que es necesario en el lado de la suscripción.
- El módulo de dominio contiene la clase *DomainParticipantFactory*, que actúa como una entrada al servicio, así también como la clase *DomainParticipant*, la cual es un contenedor para otros objetos.

A continuación en la Figura 1-9 se muestra el modelo DCPS y sus relaciones.

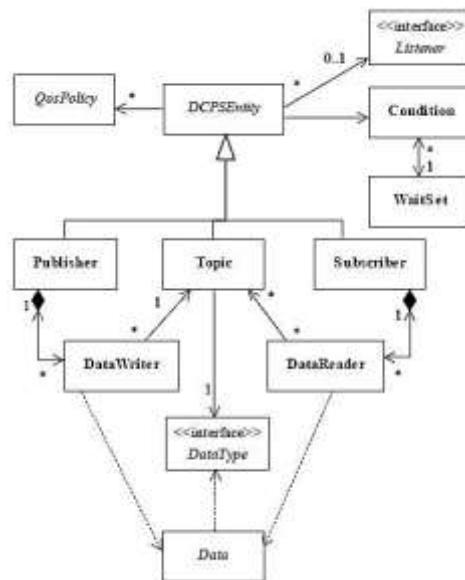


Figura 1-9. Modelo DCPS y sus relaciones [38]

1.6.1.1.2. DLRL o Data Local Reconstruction Layer

La capa DLRL es opcional, el propósito de esta es proveer un acceso más directo para el intercambio de datos, perfectamente integrado con constructores y lenguaje nativo. La orientación a objetos ha sido seleccionada por todos los beneficios de ingeniería de software.

DLRL está diseñada para permitir al desarrollador de aplicaciones usar características subyacentes de DCPS. Sin embargo, este puede tener conflicto con el propósito principal de esta misma capa, ya que es de fácil uso y permite una integración completa dentro de la aplicación. Por lo tanto, algunas características de DCPS pueden ser solo utilizadas a través de DCPS y no ser accesibles del DLRL.

DLRL permite que una aplicación describa objetos por medio de: métodos y atributos; los atributos pueden ser locales, es decir que no participan en la distribución de datos; o pueden ser compartidos, es decir que participan en la distribución de datos y estos se encuentran asociados a las entidades DCPS. DLRL gestionará los objetos DLRL en una caché, por ejemplo dos diferentes referencias a un mismo objeto o un objeto con la misma identidad no apuntará a la misma dirección de memoria.

DLRL define dos tipos de relaciones entre objetos DLRL:

- *Herencia*, la cual organiza las clases DLRL
- *Asociaciones*, las cuales organizan las instancias DLRL.

Una herencia simple es permitida entre objetos DLRL. Cualquier objeto que herede de un objeto DLRL se convierte en un objeto DLRL. Los objetos DLRL pueden, además, heredar de cualquier lenguaje de objetos nativos.

El extremo de la asociación se lo denomina relación.

Las asociaciones soportadas son:

- *Una relación a uno*, es llevada a cabo por un atributo de un solo valor, es decir, hace referencia a un objeto.
- *Una relación a varias*, es llevada a cabo por atributos de varios valores, es decir, una colección de referencias a varios objetos.

Las relaciones soportadas son:

- *Relaciones de uso plano*, las cuales no tienen impacto en el ciclo de vida del objeto.
- *Composiciones*, constituyen el ciclo de vida del objeto.

La Figura 1-10 representa el modelo del DLRL.

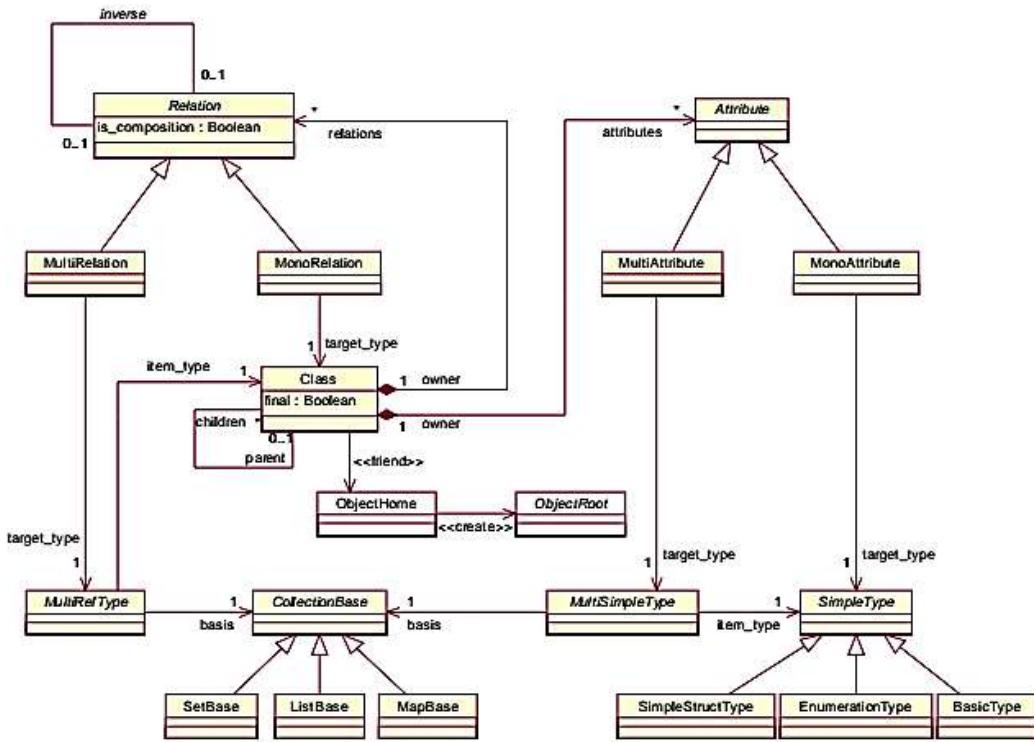


Figura 1-10. Modelo DLRL [5].

1.6.1.1.3. DDS Interoperability Wire Protocol

RTPS fue específicamente desarrollado para soportar los requerimientos únicos de los sistemas de datos distribuidos. Como uno de los dominios de aplicación a los que apunta DDS, es la comunidad de automatización industrial que define requerimientos para un protocolo de Publicación-Suscripción, el cual es compatible con DDS.

El protocolo RTPS está diseñado para soportar transporte multicast y transporte no orientado a la conexión tal como UDP.

Las principales características del protocolo RTPS son:

- Propiedades para el rendimiento y calidad de servicio, los que permiten tener comunicación segura entre el Publicador-Suscriptor y que haga el mejor esfuerzo para aplicaciones de tiempo real sobre redes IP.
- Tolerancia a fallos para permitir la creación de redes sin puntos de fallos.

- Extensibilidad para permitir que el protocolo sea extendido y mejorado con nuevos servicios con compatibilidad hacia atrás e interoperabilidad.
- Conectividad plug-and-play para que las nuevas aplicaciones y servicios estén automáticamente descubiertos y las aplicaciones puedan unirse y dejar la red en cualquier momento sin necesidad de reconfiguración.
- Configurabilidad para permitir el balanceo de requerimientos para la confiabilidad y la puntualidad de cada entrega de datos.
- Capacidad para permitir que los dispositivos implementen un subconjunto del protocolo y que aun así participen en la red.
- Escalabilidad para sistemas que potencialmente escalen en redes extensas.
- Seguridad de tipo de datos para prevenir errores en la programación de aplicaciones que puedan comprometer las operaciones en los nodos remotos.

El protocolo RTPS está descrito en términos de un modelo de plataforma independiente o PIM, o un conjunto de PCMS o Modelo específico de plataforma. El PIM RTPS contiene cuatro módulos los cuales son:

- *Estructura*, el cual define los actores del protocolo.
- *Mensajes*, define un conjunto de mensajes que cada extremo puede intercambiar.
- *Comportamiento*, define un conjunto de interacciones legales en el intercambio de mensajes y como estos afectan el estado de la comunicación en los extremos.
- *Descubrimiento*, define como las entidades son automáticamente descubiertas y configuradas.

En la Figura 1-11 se puede observar la interacción del protocolo RTPS con DDS.

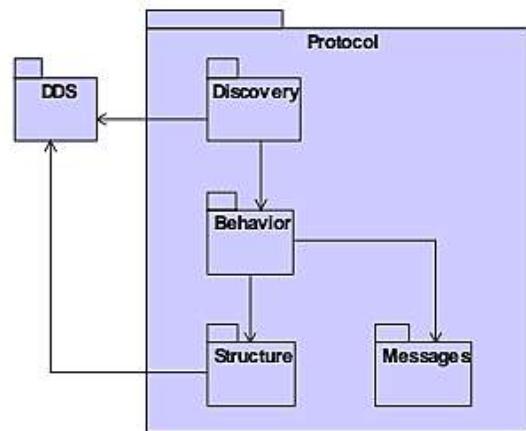


Figura 1-11. Módulos RTPS [5].

1.6.1.2. Descubrimiento

DDS provee descubrimiento dinámico en los Publicadores y Suscriptores. Este descubrimiento dinámico hace que las aplicaciones de DDS sean extensibles. Esto significa que la aplicación no tiene que conocer o configurar a los extremos para que se comuniquen porque estos son descubiertos por DDS. DDS descubrirá que en un extremo está publicando datos, suscribiéndose a datos, o ambos. Este descubrirá el tipo de datos que está siendo publicado o suscrito. También se descubrirá las características de comunicación ofrecida por los publicadores y las características solicitadas por los suscriptores. Todos estos atributos son tomados en consideración durante el descubrimiento dinámico y la asociación de participantes DDS.

Los participantes DDS pueden estar en la misma máquina o a través de la red: la aplicación usa el mismo API DDS para la comunicación. Porque no hay necesidad de conocer o configurar direcciones IP o tomar en cuenta las diferentes arquitecturas de computadores.

1.6.1.3. Políticas de Calidad de Servicio

El servicio de Distribución de Datos confía en el uso de Calidad de Servicio a medida de los requerimientos de una aplicación para el servicio. La Calidad de Servicio actualmente tiene un conjunto de características que maneja un comportamiento dado del servicio.

La descripción de todas las políticas de QoS soportadas por el servicio DCPS se encuentran definidas en el estándar.

Una política de QoS puede ser establecida en todos los objetos *DCPSEntity*. En varios casos para que la comunicación funcione apropiadamente, una política de QoS en el lado del Publicador debe ser compatible con la política correspondiente en el lado del Suscriptor. Por ejemplo, si un suscriptor pide confiabilidad en la información recibida, mientras que el correspondiente publicador define una política de mejor esfuerzo, la comunicación no se establecerá tal como fue requerida. Para abordar este problema y mantener el desacople deseable de la publicación y la suscripción en medida de lo posible, la especificación para políticas de QoS sigue el modelo Suscriptor-Solicitado, Publicador- Ofertado. En la Tabla 1-3 se mostrará las Políticas de QoS.

Tabla 1-3. Políticas de QoS del DDS [37].

QoS Policy	Applicability	RxO	Modifiable	
DURABILITY	T, DR, DW	Y	N	Data Availability
DURABILITY SERVICE	T, DW	N	N	
LIFESPAN	T, DW	-	Y	
HISTORY	T, DR, DW	N	N	
PRESENTATION	P, S	Y	N	Data Delivery
RELIABILITY	T, DR, DW	Y	N	
PARTITION	P, S	N	Y	
DESTINATION ORDER	T, DR, DW	Y	N	
OWNERSHIP	T, DR, DW	Y	N	Data Timeliness
OWNERSHIP STRENGTH	DW	-	Y	
DEADLINE	T, DR, DW	Y	Y	
LATENCY BUDGET	T, DR, DW	Y	Y	
TRANSPORT PRIORITY	T, DW	-	Y	Resources
TIME BASE FILTER	DR	-	Y	
RESOURCE LIMITS	T, DR, DW	N	N	
USER_DATA	DP, DR, DW	N	Y	Configuration
TOPIC_DATA	T	N	Y	
GROUP_DATA	P, S	N	Y	

En este modelo en el lado del Suscriptor se puede especificar una lista ordenada de las peticiones para una política particular de QoS en un orden decreciente. En el lado del Publicador se especifica un conjunto de valores ofertados para esta política de QoS. El Middleware escogerá el valor que concuerde lo más posible a lo solicitado en el lado del Suscriptor, que está ofertado en el lado del Publicador; o puede rechazar el establecimiento de la comunicación entre los dos objetos *DCPSEntity*, sin la QoS solicitada y ofertada no pueden llegar

a un acuerdo. En la Figura 1-12 se muestra el Modelo Suscriptor-Solicitado y el Publicador-Ofertado.

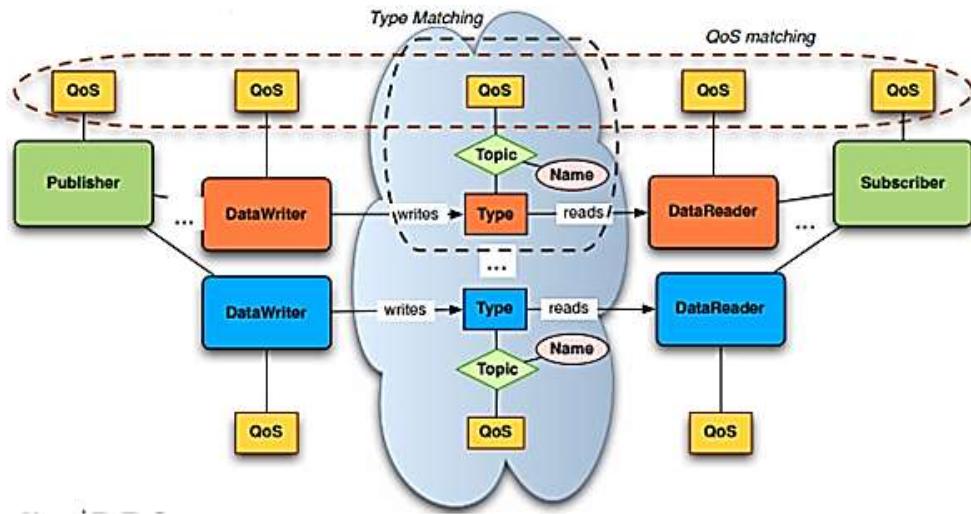


Figura 1-12. Modelo Suscriptor-Solicitado y Publicador-Ofertado [37].

1.6.1.4. Interoperabilidad

Existen múltiples aspectos de interoperabilidad en el Middleware. Estos incluyen al API, el *Wire Protocol* o Protocolo de conexión y la cobertura de QoS. Todos estos elementos tienen un rol importante en la interoperabilidad dentro de las tecnologías de Middleware. En el caso del DDS, hay estándares que especifican el API, el protocolo de conexión y la cobertura de QoS, que debe ser adherida a todos los participantes en las implementaciones DDS.

1.6.1.4.1. API y su interoperabilidad

El API es la interfaz entre el DDS y la aplicación. Este comprende los tipos de datos específicos y llamadas a funciones para que la aplicación interactúe con el Middleware, ya que el API está estandarizado, los clientes del estándar DDS pueden reemplazar implementaciones DDS con una recompilación simple, para no cambiar el código. Un API estandarizado permite portabilidad del Middleware DDS y elimina el bloqueo de un propietario.

La Figura 1-13 muestra la interoperabilidad del API.

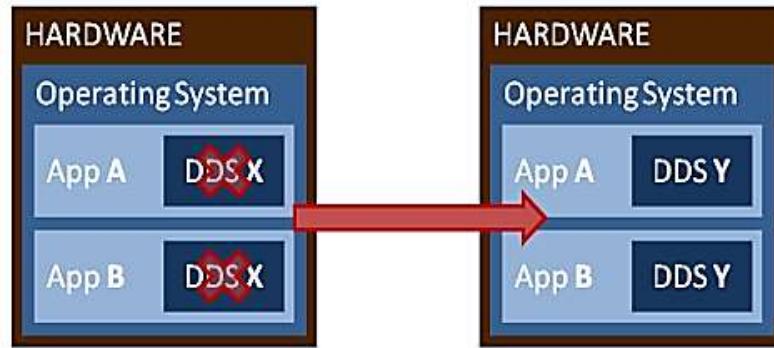


Figura 1-13. Interoperabilidad del API [39].

1.6.1.4.2. Protocolo de Conexión y su Interoperabilidad

El protocolo RTPS es responsable de la interoperabilidad del DDS sobre la conexión. La OMG también gestiona el estándar RTPS y adhiere a este múltiples proveedores del DDS. RTPS es usado como el protocolo de transporte de datos subyacente a la comunicación dentro del DDS. Este provee soporte para todas las tecnologías DDS, como el descubrimiento dinámico, las comunicaciones seguras, la independencia de plataforma, y las asociaciones de QoS. Este aspecto de interoperabilidad permite a un usuario del DDS fácilmente extender su sistema distribuido añadiendo diferentes implementaciones DDS. DDS utiliza estratégicamente comunicación de datos basada en multicast y unicast para las necesidades de las aplicaciones. DDS provee una implementación nativa de RTPS, es decir no existen gateways RTPS, ni demonios, ni aplicaciones en otro plano, ya que se busca el mejor rendimiento posible. En la Figura 1-14 se puede ver la interoperabilidad del protocolo de conexión.

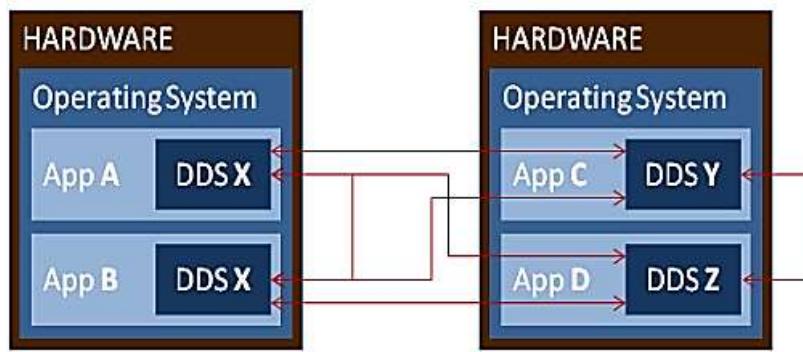


Figura 1-14. Interoperabilidad del Protocolo de Conexión [39].

1.6.1.4.3. Cobertura de la QoS y su interoperabilidad

Las políticas de QoS permiten a la aplicación medir el comportamiento específico de la comunicación. Esta medición incluye diferentes aspectos de la

comunicación. Ejemplos de las políticas de QoS incluyen: la confiabilidad es decir, ¿qué requerimientos de confiabilidad necesitan esos datos?; la durabilidad es decir, ¿cuán grande son los datos guardados para posibles publicaciones futuras?; historial y límites de recursos es decir, ¿cuáles son los requerimientos de almacenamiento?; filtrado y presentación es decir, ¿qué información deben ser presentados al suscriptor, y cómo?; y la propiedad es decir, ¿existen requisitos para la redundancia o para la desconexión?. Estas son solamente una pequeña parte de las 22 distintas políticas de QoS definidas por el estándar DDS. Estas políticas de QoS proveen un conjunto completo de opciones de configuración, permitiendo a las aplicaciones tomar fácilmente ventaja de estrategias de comunicación muy complejas y poderosas. Las características de QoS tienen un rol a través de todos los aspectos de interoperabilidad. Por supuesto, el API para la configuración de QoS y los protocolos de conexión para la interacción del QoS deben estar estandarizados para proveer implementaciones portables y conexiones interoperables dentro del DDS. Sin embargo, la cobertura de estas políticas de QoS depende del proveedor.

El estándar DDS además de especificar el API DDS, también categoriza las características de QoS dentro de perfiles que definen diferentes niveles de conformidad. El perfil mínimo tiene más de 22 políticas de QoS, y define un conjunto mínimo de políticas de QoS que deben estar cubiertas para una implementación DDS para ser compatible con el estándar, es decir, interoperable.

Todos estos aspectos de interoperabilidad puestos juntos permiten una gran flexibilidad a los clientes del Middleware.

1.6.2. FUNCIONALIDADES

El estándar DDS fue diseñado explícitamente para construir sistemas distribuidos en tiempo real. Para este fin, la especificación añade un conjunto de parámetros de calidad de servicio para configurar propiedades no funcionales. En este caso, el DDS provee una alta flexibilidad en la configuración de sistemas por medio de la asociación del conjunto de parámetros de calidad de servicio a cada entidad.

Además, el DDS permite la modificación de algunos parámetros en tiempo de ejecución, mientras se realiza una reconfiguración dinámica del sistema. Este conjunto de parámetros de calidad de servicio, permite varios aspectos de los datos,

referidos a los recursos de red y recursos informáticos a ser configurados y pueden ser clasificados en las siguientes categorías, como se muestra en la Figura 1-15:

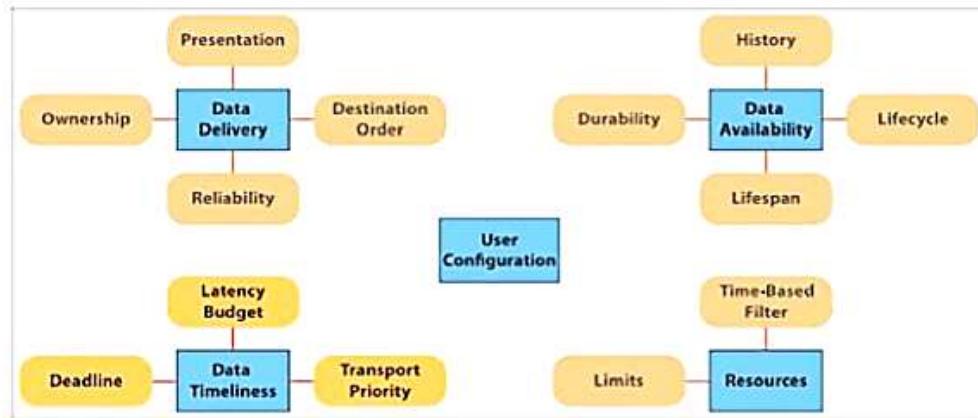


Figura 1-15. Parámetros de QoS definidos por DDS [5].

- *Data Availability* o disponibilidad de los datos, esto comprende parámetros para el control de políticas de consulta y almacenamiento de la información. Los parámetros que pertenecen a esta categoría son: *Durability*, *Lifespan*, *History* y *Lifecycle*.
- *Data Delivery* o entrega de datos, especifica como la información debe ser transmitida y presentada a la aplicación. Los parámetros que pertenecen a esta categoría son: *Presentation*, *Reliability*, *Partition*, *Destination_Order*, y *Ownership*.
- *Data Timeliness* o Puntualidad de la información, esta controla la latencia en la distribución de los datos. Los parámetros que pertenecen a esta categoría son: *Deadline*, *Latency_Budget*, y *Transport_Priority*.
- *Maximum Resources* o Máximos de Recursos, esta limita la cantidad de recursos que pueden ser usados en el sistema a través de parámetros tales como: *Resource_Limit* o *Time-Based_Filter*.
- *User Configuration* o Configuración de Usuario, estos parámetros permiten que la información extra sea añadida a cada entidad en la capa aplicación.

Finalmente, esta especificación sigue el modelo Suscriptor-Solicitado, y el Publicador-ofertado para establecer los parámetros de QoS. Mediante el uso de este modelo, ambos el Publicador y el Suscriptor deben especificar parámetros compatibles de QoS para establecer la comunicación. De otra manera, el Middleware debe indicar a la aplicación que la comunicación no es posible.

1.6.2.1. Gestión de Recursos del Procesador

El estándar DDS no aborda explícitamente la planificación de hilos en los procesadores, ya que esto es un aspecto de implementación definida. Sin embargo, un subconjunto de parámetros de QoS, definidos por el estándar están enfocados al control del comportamiento temporal y el mejoramiento de la previsibilidad de la aplicación. Los tres parámetros de la puntualidad de datos, que están resaltados en la Figura 1-15, son importantes en la gestión de recursos de sistemas de tiempo real. En particular, el estándar ha definido los siguientes parámetros para la gestión de recursos de procesador:

- *Deadline*, este parámetro indica la cantidad máxima de tiempo disponible para enviar/ recibir muestras de datos pertenecientes a un tópico particular. Sin embargo, este no define ningún mecanismo asociado para asegurar estos requerimientos temporales; por lo tanto, este parámetro de QoS solo representa un servicio de notificación en el cual el Middleware informa a la aplicación que el tiempo límite se ha perdido.
- *Latency_Budget*, este parámetro es definido como el retardo máximo aceptable en la entrega de mensajes. Sin embargo, el estándar hace énfasis en que este parámetro no puede ser llevado a cabo o controlado por el Middleware; por lo tanto, este puede ser usado para optimizar el comportamiento interno del Middleware.

Estos dos parámetros de QoS, incluso si ambos comparten objetivos similares, se aplican a diferentes niveles, como se ilustra en la Figura 1-16. Esta figura muestra cómo el parámetro *deadline* es monitorizado dentro de la capa DDS, mientras que el parámetro *Latency_Budget* se aplica dentro de la capa DDSI.

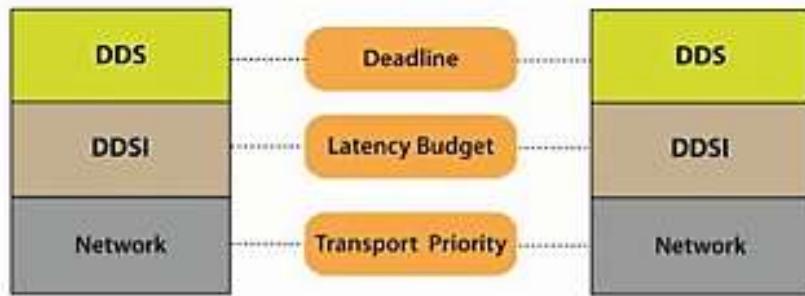


Figura 1-16. Control del tiempo en DDS [5].

El DDS define diferentes mecanismos para permitir la comunicación entre entidades. En el lado del Publicador, el mecanismo de comunicación es sencillo: cuando los nuevos datos están disponibles, el DW realiza una llamada de escritura simple, como por ejemplo, escribir o eliminar, para publicar datos dentro del dominio del DDS. Entonces, la muestra de datos es transmitida usando modos de comunicación de los tipos asincrónicos, uno a uno o uno a varios. Sin embargo, el DDS también da soporte a hilos de llamadas bloqueadas hasta que la muestra de datos haya sido entregada y confirmada por los DR asociados.

En el lado del Suscriptor, la recepción de los datos puede ser realizada con *polling*¹¹ [40], modo sincrónico, y asincrónico. Estos modelos no sólo son válidos para la recepción de datos sino también para la notificación de cualquier cambio en el estado de la comunicación, por ejemplo, para el no cumplimiento de las peticiones de calidad de servicio. En particular, la aplicación podría ser notificada a través de los siguientes modelos:

- *Polling*, como los hilos de una aplicación pueden invocar operaciones no bloqueantes para obtener datos o cambios en el estado de la comunicación.
- *Listeners*, adjunta una función de devolución de llamada para las modificaciones de acceso asincrónico en el estado de la comunicación mientras que la aplicación se sigue ejecutando, es decir que los hilos del Middleware son responsables de la gestión de cualquier cambio en el estado de la comunicación.
- *Conditions* y *Wait-Sets*, los cuales permiten que los hilos de la aplicación sean bloqueados hasta conocer una o varias condiciones.

¹¹ Polling, hace referencia a una operación de consulta constante

Ambas representan el mecanismo de sincronización para gestionar cualquier cambio en el estado de la comunicación.

1.6.2.2. Gestión de Recursos de Red

En materia de redes, esta especificación define un conjunto de características enfocadas a garantizar el determinismo en las comunicaciones, tal como el uso de parámetros de planificación en redes y la definición del formato para el intercambio de mensajes.

El paso de parámetros de planificación para las redes de comunicación es llevada a cabo a través de otro parámetro de QoS incluido en la categoría de *data timeless*, mostrado en la Figura 1-15:

- *Transport-Priority*, a diferencia de *Latency-Budget*, que intenta optimizar el comportamiento interno del Middleware, este parámetro prioriza el acceso a la red de comunicación, como se muestra en la Figura 1-16. Además, mientras que las comunicaciones son unidireccionales, este solo está asociado con entidades DW.

Por otra parte, la especificación DDSI define el conjunto de normas y características requeridas para habilitar la comunicación entre entidades DDS. Aunque esta especificación no está particularmente orientada al uso de redes en tiempo real, esta no opone su uso y solo muestra un conjunto de requisitos para las redes subyacentes. El punto más importante tratado por la especificación se encuentra descrito en el protocolo RTPS, el cual es responsable específicamente de cómo se difunden los datos entre los nodos. Esto requiere la definición de los protocolos de intercambio de mensajes y formatos de mensajes. En particular, la estructura de un mensaje RTPS consiste de una cabecera de tamaño fijo seguida por un número variable de submensajes. Al procesar cada submensaje independientemente, el sistema puede descartar mensajes desconocidos o erróneos lo cual facilita futuras extensiones del protocolo.

Otra característica clave de DDS es la sobrecarga introducida por las operaciones internas del Middleware. En este caso, el estándar define una serie de operaciones a ser llevadas a cabo por las implementaciones que puedan consumir recursos tanto del procesador como de la red. En particular, DDS proporciona un servicio para la gestión de entidades remotas llamadas *Discovery* o Descubrimiento. Este servicio describe como se obtiene información sobre la presencia y

características de cualquier otra entidad inmersa en el sistema distribuido. Aunque el estándar describe un protocolo específico para el descubrimiento con el propósito de la interoperabilidad, este permite que otros protocolos de descubrimiento puedan ser aplicados. Bajo el protocolo de descubrimiento requerido, las implementaciones deben crear un conjunto de entidades DDS por defecto. Estas entidades presentes son responsables del establecimiento transparente de la comunicación con el usuario y del descubrimiento de la presencia o ausencia de entidades remotas, por ejemplo, un sistema de *plug-and-play*¹². Este tipo de tráfico en la red, el cual es interno en el Middleware, es llamado metatráfico y puede ser considerado en los análisis de tiempo.

¹² Plug-and-Play, se refiere a un conjunto de protocolos de comunicación que permiten descubrir de manera transparente la presencia de otros dispositivos en la red.

CAPÍTULO 2

ANÁLISIS DE REQUISITOS PARA LA IMPLEMENTACIÓN DE UN MÓDULO QUE SOPORTE EL PROTOCOLO RTPS

2.1. INTRODUCCIÓN

En el presente capítulo se definen los requisitos necesarios para integrar el protocolo RTPS con el Middleware DDS que se describió en el capítulo anterior. Primeramente, se realizan capturas de paquetes con la herramienta Wireshark de los diferentes mensajes RTPS y mensajes de descubrimiento RTPS. Finalmente, se obtiene un análisis detallado de los mismos y se definen los requisitos necesarios para la implementación.

2.2. ANÁLISIS DE PAQUETES DE LOS DIFERENTES MENSAJES RTPS

2.2.1. ESTRUCTURA DE LOS MENSAJES RTPS

2.2.1.1. Estructura general

En la Figura 2-1 se muestra la estructura general del mensaje RTPS incluyendo el tamaño en bytes de cada campo.

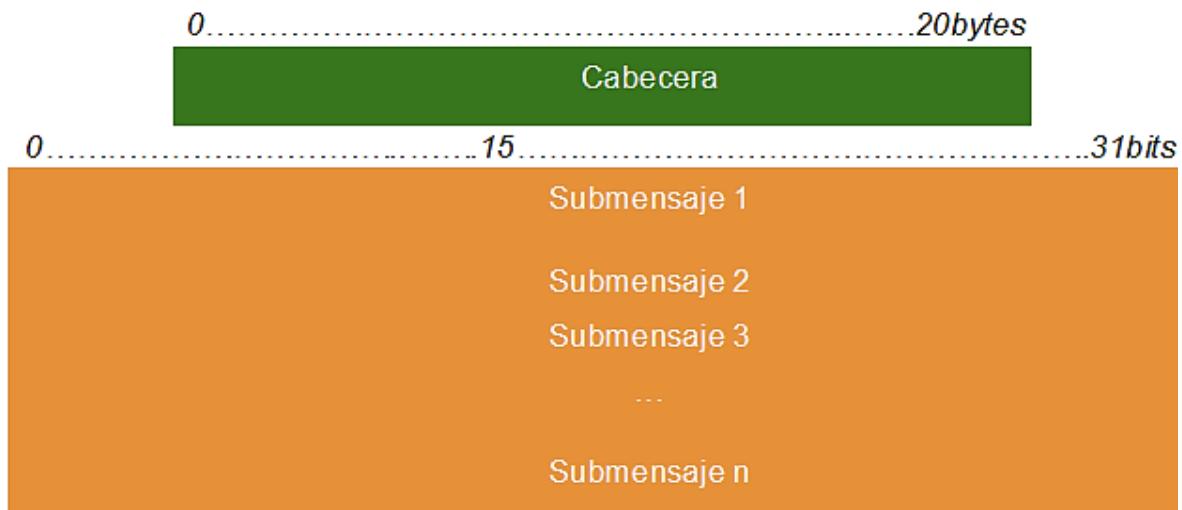


Figura 2-1. Estructura general mensaje RTPS [41]

Los mensajes RTPS explícitamente no envían el tamaño, pero utilizan el transporte con el cual se envían los mensajes, para el caso del tamaño, se envía en la carga UDP/IP.

2.2.1.2. Cabecera

La cabecera del mensaje RTPS está formada por los campos mostrados en la Figura 2-3, los cuales están encargados de proporcionar información de la versión del protocolo y de identificar al mensaje.



Figura 2-2. Cabecera del Mensaje RTPS [41]

2.2.2. ESTRUCTURA DE LOS SUBMENSAJES RTPS

En la Figura 2-3 se muestra la estructura del submensaje, la cual está compuesta por una cabecera submensaje y el contenido de submensaje. El submensaje tal como se muestra esta alineado a los 4 bytes.

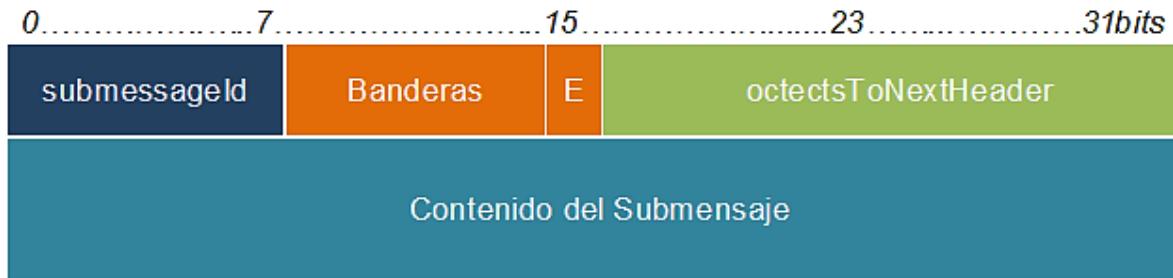


Figura 2-3. Estructura de los submensajes RTPS [41]

2.2.2.1. Lista de submensajes

A continuación se muestra el listado de submensajes posibles dentro de un mensajes RTPS.

- Pad
- AckNack
- Heartbeat
- GAP
- InfoTimeStamp
- InfoSource
- InfoReply
- InfoDestination
- InfoReplyIp4
- NackFrag
- HeartbeatFrag

- Data
- DataFrag

2.2.3. ACKNACK SUBMESSAGE

En la Figura 2-4 se muestra la estructura del submensaje *AckNack*.

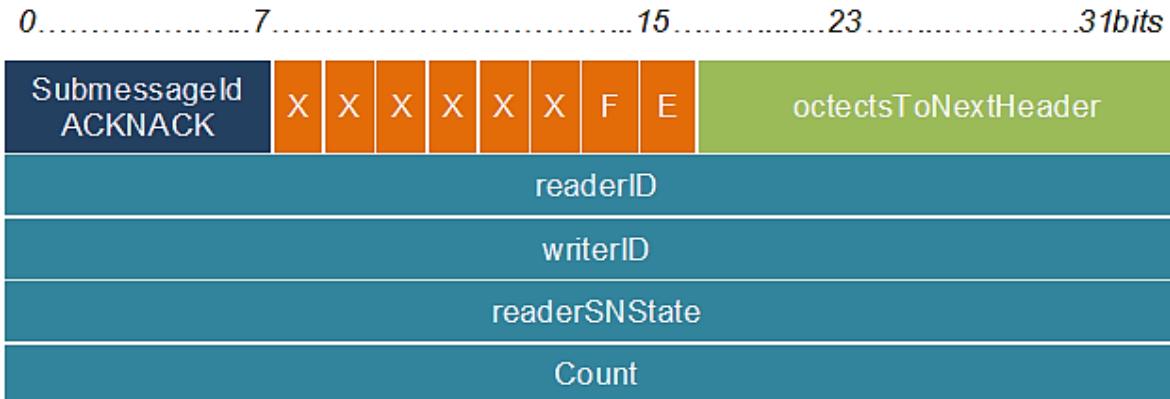


Figura 2-4. Estructura del submensaje *AckNack* [41]

Este submensaje se utiliza para comunicar el estado de un *lector* a un *escritor*. El submensaje permite al lector dar a conocer los números de secuencia que ha recibido y los que le faltan todavía al escritor. Este submensaje puede utilizarse para hacer acuses de recibo tanto positivos como negativos.

2.2.3.1. Banderas en el encabezado de submensaje

En el **FinalFlag**, cuando este campo tiene el valor de 1 significa que el lector no requiere una respuesta del escritor, sin embargo, si este se establece en 0 significa que el escritor debe responder al mensaje de *AckNack*.

En el **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de bits.

2.2.3.2. Otros elementos en el encabezado del submensaje

El **readerId**, identifica la entidad lectora que realiza el acuse recibo de cierto número de secuencia o las solicitudes para recibir ciertos números de secuencia.

El **writerId**, identifica la entidad escritora a la cual tiene como objetivo el submensaje *AckNack*. Es la entidad del escritor que pide reenviar algunos números de secuencia o está siendo informado de la recepción de ciertos números de secuencia.

El **readerSNState**, comunica el estado del lector al escritor, enviando números de secuencia que confirman la llegada de los datos en el lector.

El **Contador**, se incrementa cada vez que se envía un mensaje *AckNack*. Proporciona los medios para que un escritor detecte los mensajes duplicados de *AckNack* que pueden derivarse de los diferentes canales de comunicación.

2.2.3.3. Validez

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el *submessageLength* en el encabezado de submensaje es demasiado pequeño.
- Cuando el *readerSNState* no es válido.

2.2.3.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor.

2.2.3.5. Interpretación lógica

El lector envía el submensaje *AckNack* al escritor para comunicar su estado con respecto a los números de secuencia utilizados por el escritor.

El escritor se identifica únicamente por su GUID, el cual es un atributo de todas las Entidades RTPS, e identifica de manera única a las entidades RTPS en el Dominio DDS. El GUID del escritor se obtiene utilizando el estado del receptor.

El lector se identifica únicamente por su GUID. El GUID del lector se obtiene utilizando el estado del receptor.

El mensaje tiene dos propósitos simultáneamente:

- Reconocer en el submensaje todos los números de secuencia dentro del *SequenceNumberSet*.
- Reconocer los acuses de recibo negativos que aparecen explícitamente en el *readerSNState*.

2.2.3.5.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje *AckNack*.

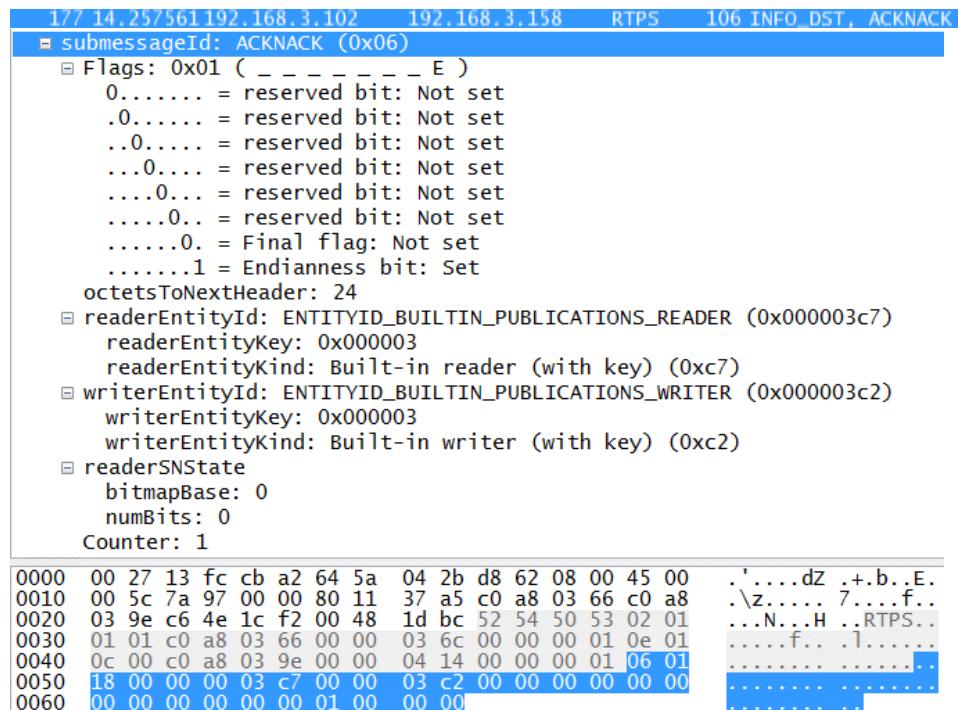


Figura 2-5. Uso del submensaje ACKNACK.

Una explicación del uso del AckNack ha sido descrita en el capítulo 3 en el ejemplo 2 de la sección 3.5.5

2.2.4. DATA SUBMESSAGE

En la Figura 2-6 se muestra la estructura del submensaje Data.

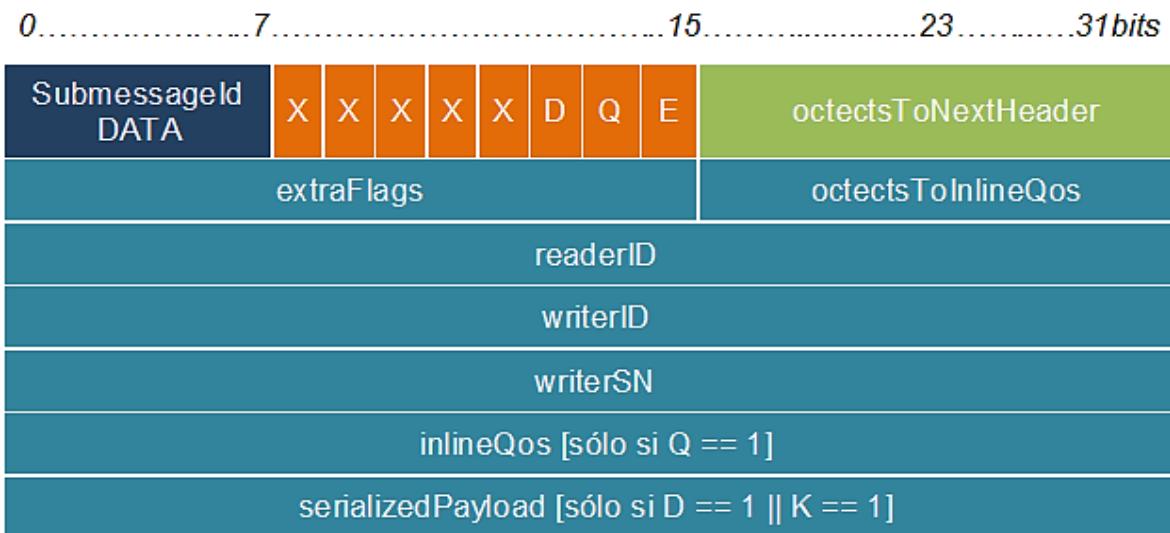


Figura 2-6. Estructura del submensaje Data [41]

El submensaje notifica al lector RTPS los cambios en un objeto de datos pertenecientes al escritor RTPS. Los posibles cambios incluyen el valor y el ciclo de vida del objeto de datos.

2.2.4.1. Banderas en el encabezado de submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de bits.

La **InlineQosFlag**, indica al lector la presencia de un *ParameterList* que contiene los parámetros de calidad de servicio que debe utilizarse para interpretar el mensaje. Cuando Q = 1 significa que el submensaje data contiene información dentro del inlineQos.

El **DataFlag**, indica al lector que el elemento de submensaje *dataPayload* contiene el valor serializado del objeto de datos. Se representa con el literal 'D'.

La **KeyFlag**, indica al lector que el elemento de submensaje *dataPayload* contiene el valor de la clave del objeto de datos serializado. Se representa con el literal 'K' .

El *DataFlag* se interpreta en combinación con el *KeyFlag* de la siguiente manera:

- D = 0 y K = 0 significa que no hay ningún dato dentro del *payload*.
- D = 1 y K = 0 significan que el *payload* contiene datos serializados.
- D = 0 y K = 1 significan que el *payload* contiene la clave serializada.
- D = 1 y K = 1 es una combinación no válida en esta versión del Protocolo.

2.2.4.2. Otros elementos en el encabezado del submensaje

El **readerId**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El **writerId** , identifica la entidad RTPS escritora que realizó el cambio.

El **writerSN**, identifica el cambio y el orden relativo de todos los cambios realizados por el **escritor** RTPS identificados por el writerGuid. Cada cambio obtiene un número de secuencia consecutiva. Cada escritor RTPS mantiene su propio número de secuencia.

El **inlineQos**, presenta información de QoS solamente si el *InlineQosFlag* está situado en la cabecera.

El **serializedPayload**, presenta informacion solamente si el *DataFlag* o el *KeyFlag* cumplen las siguientes condiciones.

- Si el *DataFlag* tiene el valor de 1, entonces contiene el valor encapsulado del cambio.
- Si el *KeyFlag* tiene el valor de 1 , entonces contiene la encapsulación de la clave del submensaje.

El campo **extraFlags**, ofrece espacio para 16 bits adicionales para banderas a parte de los 8 bits ya proporcionados. Estos bits adicionales apoyarán la evolución del protocolo sin comprometer la compatibilidad hacia atrás. Esta versión del protocolo debe establecer todos los bits en el *extraFlags* en cero.

El campo **octetsToInlineQos**, es representado por un *shortUnsignedCDR*. contiene el número de octetos que representan al *inlineQos*. Si el *inlineQos* no está presente (es decir, el *InlineQosFlag* no está establecido), entonces el *octetsToInlineQos* contiene el desplazamiento al siguiente campo después de la *inlineQos*.

Las implementaciones del protocolo que están procesando un submensaje recibido siempre deben usar el *octetsToInlineQos* para omitir cualquier elemento de encabezado submensaje.

2.2.4.3. Validez

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando en el encabezado del submensaje el *submessageLength* es demasiado pequeño.
- Cuando el *writerSN.value* no es estrictamente positivo o es un numero de secuencia desconocido.
- Cuando el *inlineQos* no es válido.

2.2.4.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.4.5. Interpretación lógica

El escritor envía el submensaje data al lector para comunicar los cambios realizados. Estos incluyen los cambios en el valor, así como los cambios en el ciclo de vida del objeto de datos. Éstos se comunican por la presencia del *serializedPayload*. Cuando están presentes, el *serializedPayload* se interpreta

como el valor del objeto de datos o como la clave que identifica el objeto de datos del conjunto de objetos registrados.

- Si el DataFlag es igual a 1 y el KeyFlag es igual a 0, el elemento *serializedPayload* se interpreta como el valor del objeto *dataobject*.
- Si el *DataFlag* es igual a 0 y el *KeyFlag* es igual a 1, el elemento *serializedPayload* se interpreta como el valor de la clave que identifica la instancia del objeto de datos registrada.
- Si el *InlineQosFlag* es igual a 1, el elemento *inlineQos* contiene los valores de QoS que reemplazan a los del escritor RTPS.

2.2.4.5.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje DATA.

No.	Time	Source	Destination	Protocol	Length	Info
131	13.064990	192.168.3.102	239.255.0.1	RTPS	806	INFO_TS, DATA(p)
■ submessageId: DATA (0x15)						
■ Flags: 0x05 (_ _ _ _ D _ E)						
0..... = reserved bit: Not set						
.0..... = reserved bit: Not set						
..0..... = reserved bit: Not set						
...0.... = reserved bit: Not set						
....0... = Serialized Key: Not set						
.....1.. = Data present: Set						
.....0. = Inline QoS: Not set						
.....1 = Endianness bit: Set						
octetsToNextHeader: 728						
0000 0000 0000 0000 = Extra flags: 0x0000						
Octets to inline QoS: 16						
■ readerEntityId: ENTITYID_UNKNOWN (0x00000000)						
readerEntityKey: 0x000000						
readerEntityKind: Application-defined unknown kind (0x00)						
■ writerEntityId: ENTITYID_BUILTIN_SDP_PARTICIPANT_WRITER (0x000100c2)						
writerEntityKey: 0x000100						
writerEntityKind: Built-in writer (with key) (0xc2)						
writerSeqNumber: 1						
■ serializedData						
encapsulation kind: PL_CDR_LE (0x0003)						
encapsulation options: 0x0000						
■ serializedData:						

Figura 2-7. Uso del submensaje DATA.

Una explicación del uso del Data ha sido descrita dentro en el capítulo 3 en el ejemplo 2 de la sección 3.5.5

2.2.5. DATAFRAG SUBMESSAGE

En la Figura 2-8 se muestra la estructura del submensaje *DataFrag*.

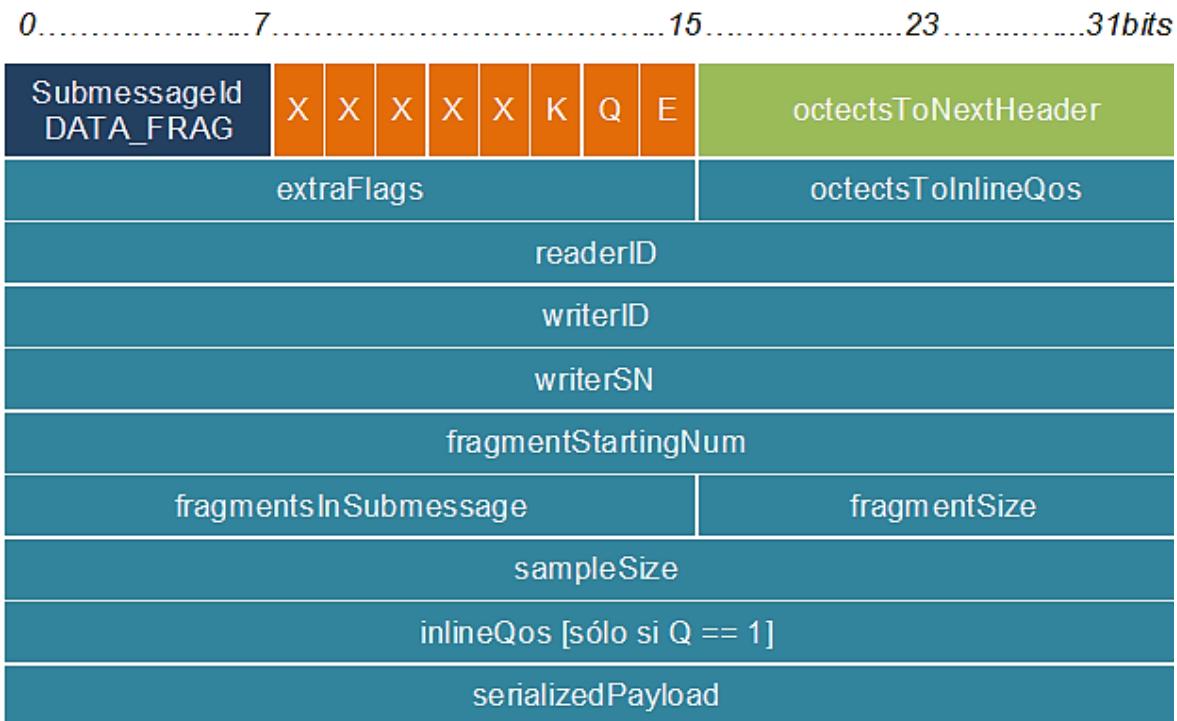


Figura 2-8. Estructura del submensaje *DataFrag* [41]

El submensaje *DataFrag* es un submensaje de datos que es fragmentado y enviado como múltiples submensajes. Los fragmentos contenidos en los submensajes *DataFrag* son reensamblados por el lector RTPS posteriormente.

Un submensaje *DataFrag*, ofrece las siguientes ventajas:

- Mantiene las variaciones en el contenido y estructura de cada submensaje al mínimo.
- Evita tener que agregar información de fragmentación como parámetros del *inlineQoS* en el submensaje *data*.

2.2.5.1. Banderas en el encabezado de submensaje

El **InlineQosFlag**, indica que la entidad lectora está siendo informada del cambio: cuando se establece en 1 significa que el submensaje DataFrag contiene el inlineQos.

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **KeyFlag**, cuando es igual a 0, significa que el *serializedPayload* contiene los datos serializados, y cuando es igual 1 significa que el *serializedPayload* contiene la clave serializada.

2.2.5.2. Otros elementos en el encabezado del submensaje

El **readerId**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El **writerId**, identifica la entidad RTPS escritora que realizó el cambio.

El **writerSN**, identifica el cambio y el orden relativo de todos los cambios realizados por el escritor RTPS identificados por el writerGuid. Cada cambio obtiene un número de secuencia consecutiva. Cada escritor RTPS mantiene su propio número de secuencia.

El **fragmentStartingNum**, indica el fragmento inicial de la serie de fragmentos de *serializedData*. La numeración de los fragmentos comienza con el número 1.

El **fragmentInSubmessage**, indica el número de fragmentos consecutivos contenidos en este submensaje, a partir de las *fragmentStartingNum*.

El **dataSize**, indica el tamaño total en bytes de los datos originales antes de la fragmentación.

El **fragmentSize**, indica el tamaño de un fragmento individual en bytes. El tamaño del fragmento máximo equivale a 64K.

El **inlineQos**, presenta información de QoS solamente si el *InlineQosFlag* está situado en la cabecera.

El **serializedPayload**, presenta información sólo si el *DataFlag* es igual a 1.

El *DataFrag* representa parte del valor del objeto de datos-nuevo después del cambio.

- Si el *DataFlag* es igual a 1, entonces contiene un conjunto de fragmentos de la encapsulación del nuevo valor del objeto *dataobject* después del cambio consecutivo.
- Si el *KeyFlag* es igual a 1, contiene un conjunto de fragmentos de la encapsulación de la clave del mensaje.

En cualquier caso el conjunto consecutivo de fragmentos contiene partes de *fragmentsInSubmessage* y comienza con el fragmento identificado por *fragmentStartingNum*.

2.2.5.3. Validez

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el *writerSN.value* no es estrictamente positivo o es un numero de secuencia desconocido.
- Cuando *fragmentStartingNum* no es estrictamente positivo o excede el número total de fragmentos.
- Cuando el *fragmentSize* excede al *dataSize*.
- Cuando el tamaño del *serializedData* es superior al producto *fragmentsInSubmessage * fragmentSize*.
- Cuando el *inlineQos* no es válido.

2.2.5.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.5.5. Interpretación lógica

El submensaje *DataFrag* extiende el submensaje data usando el *serializedData* a ser fragmentado y enviado como múltiples submensajes. Una vez que el *serializedData* llega al lector RTPS, la interpretación de los submensajes DataFrag es idéntica a la del submensaje Data.

A continuación se describe cómo volver a unir la información dentro de los submensaje DataFrag.

El tamaño total de los datos que se reensambla está dada por el *dataSize*. Cada submensaje DataFrag contiene un segmento contiguo de estos datos en su elemento *serializedData*. El tamaño del segmento se determina por la longitud del elemento *serializedData*. Durante la desfragmentacion, el desplazamiento de cada segmento se determina por:

$$(fragmentStartingNum - 1) * fragmentSize$$

Los datos son reensamblados completamente cuando se han recibido todos los fragmentos. El número total de fragmentos esperado equivale a:

$$(dataSize / fragmentSize) + ((dataSize \% fragmentSize))$$

Tenga en cuenta que cada submensaje DataFrag puede contener múltiples fragmentos. Un escritor RTPS seleccionará al *fragmentSize* basándose en el

tamaño más pequeño de mensaje soportado a través de todos los transportes. Si algunos lectores RTPS pueden ser alcanzados a través de un transporte que soporte mensajes más grandes, el escritor RTPS puede empaquetar los múltiples fragmentos en un solo submensaje DataFrag o incluso puede enviar un submensaje regular de datos si la fragmentación ya no es necesaria.

2.2.5.1. Ejemplo

En las siguientes figuras se muestra dos mensajes RTPS fragmentados.

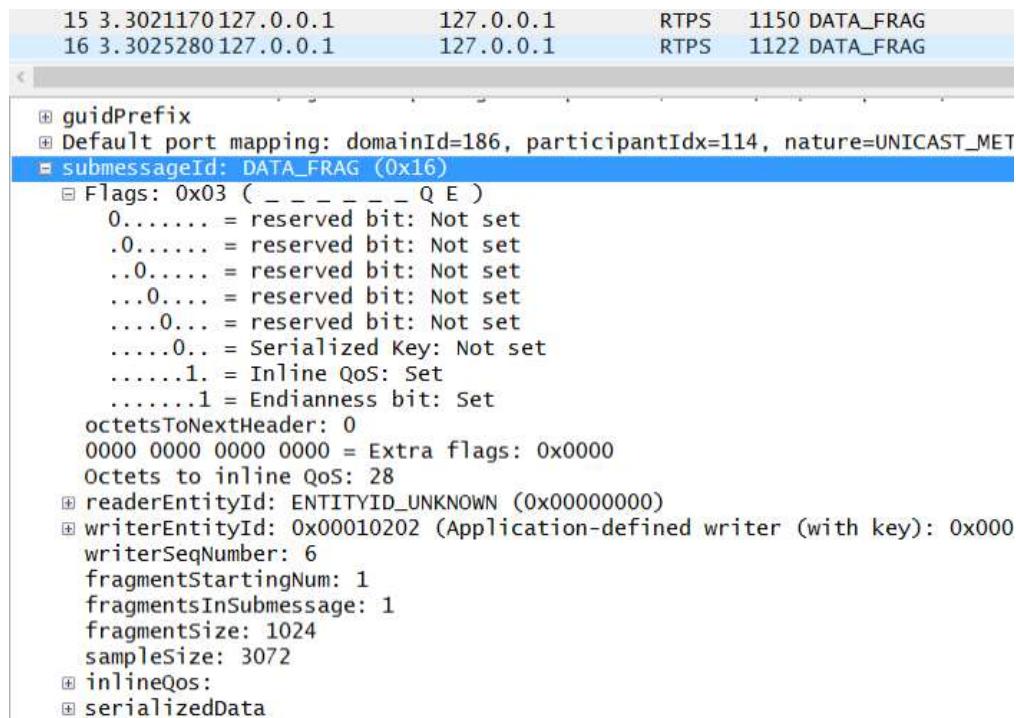


Figura 2-9. Uso del submensaje DATA_FRAG (parte I).

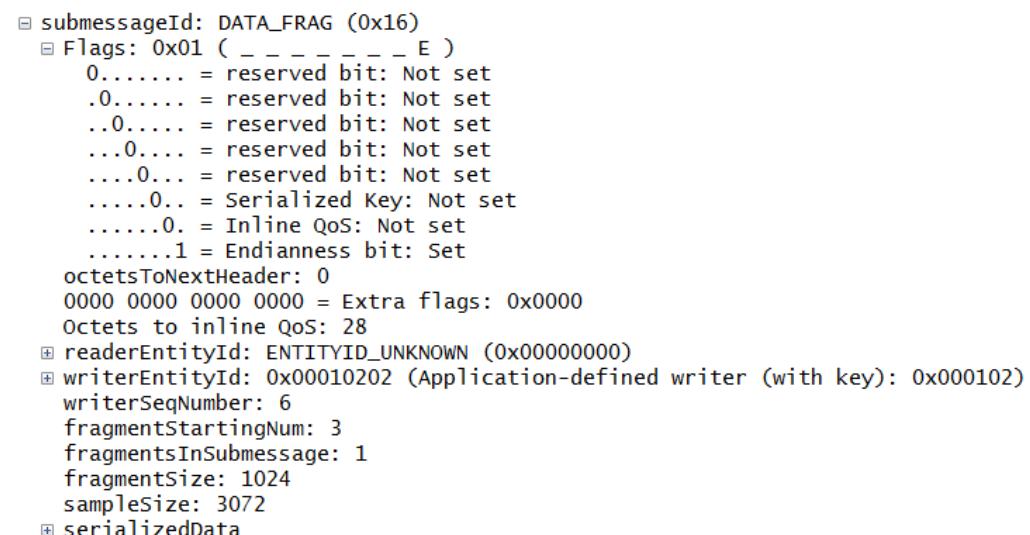


Figura 2-10. Uso del submensaje DATA_FRAG (parte II).

2.2.6. GAP SUBMESSAGE

En la Figura 2-11 se muestra la estructura del submensaje *GAP*.

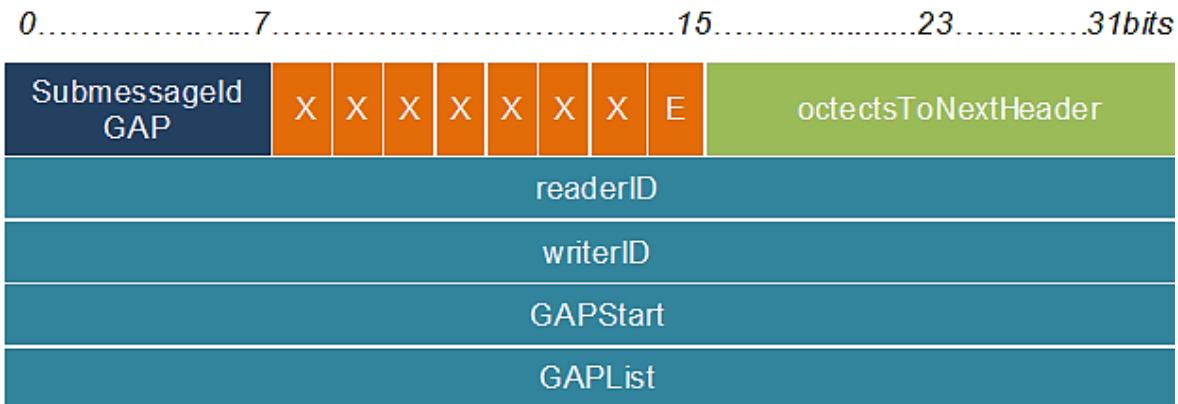


Figura 2-11. Estructura del submensaje *GAP* [41]

Este submensaje es enviado de un escritor RTPS a un lector RTPS e indica al lector que un rango de números de secuencia ya no es relevante. El conjunto puede ser un rango contiguo de números de secuencia o un conjunto específico de números de secuencia.

2.2.6.1. Banderas en el encabezado de submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.6.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El **writerId**, identifica la entidad RTPS escritora que realizó el cambio.

El **GAPStart**, identifica el primer número de secuencia.

El **GAPList**, tiene dos propósitos:

- Identifica el último número de la secuencia en el intervalo.
- Identifica una lista adicional de números de secuencia que son irrelevantes.

2.2.6.3. Validez

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el *GAPStart* es cero o negativo.
- Cuando el *GAPList* no es válido.

2.2.6.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.6.5. Interpretación lógica

El escritor RTPS envía el submensaje *GAP* al lector RTPS para comunicar que ciertos números de secuencia ya no son relevantes. Esto es causado típicamente en el lado del escritor. En este escenario, nuevos valores de los datos podrán sustituir a los viejos valores de los objetos de datos que fueron representados por los números de secuencia que aparecen como irrelevantes en el submensaje.

Los números de secuencia irrelevantes comunicados por el submensaje *GAP* se componen de dos grupos:

- Toda la secuencia de números en el rango *GAPStart* \leq *sequence_number* \leq *GAPList.base* -1
- Todos los números de secuencia que aparecen explícitamente enumerados en el *GAPList*.

2.2.6.5.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje *GAP*.

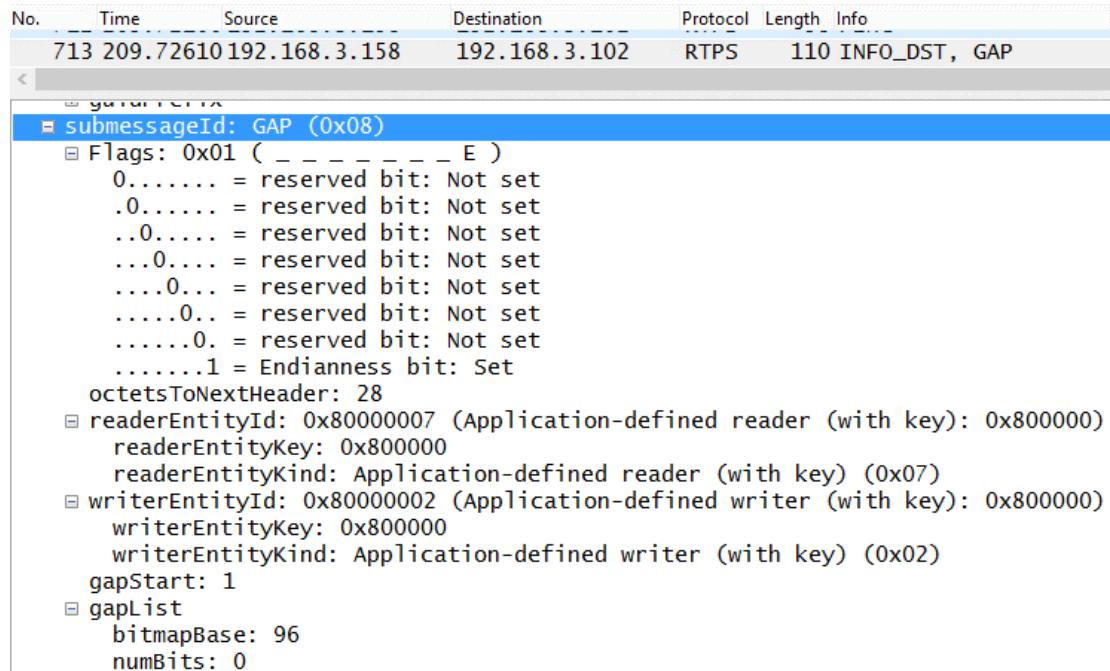


Figura 2-12. Uso del submensaje GAP.

2.2.7. HEARTBEAT SUBMESSAGE

En la Figura 2-13 se muestra la estructura del submensaje *Heartbeat*.

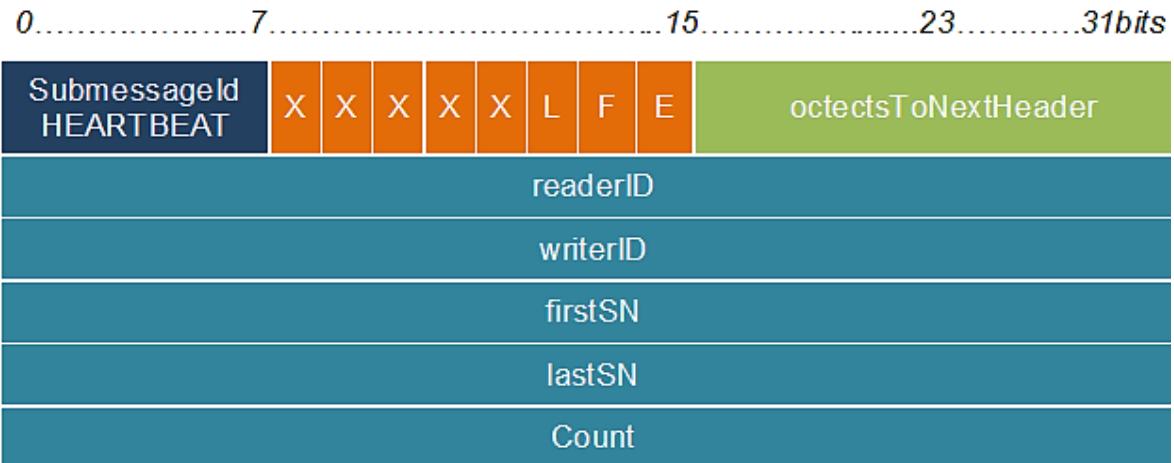


Figura 2-13. Estructura del submensaje Heartbeat [41]

Este mensaje se envía desde un escritor RTPS a un lector RTPS para comunicar la secuencia de cambios que el escritor tiene disponibles.

2.2.7.1. Banderas en el encabezado del submensaje

El **FinalFlag**, indica si el lector es necesario responder al submensaje o si es sólo heartbeat de control. El FinalFlag se representa con el literal 'F'. Cuando

$F = 1$ significa que el escritor no requiere una respuesta desde el lector y cuando es igual a 0 significa que el lector debe responder al submensaje Heartbeat.

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **LivelinessFlag**, indica que el escritor de datos DDS asociado con el escritor RTPS se encuentra presente. El LivelinessFlag se representa con el literal 'L'. Cuando $L = 1$ significa que el *DataReader* DDS asociado con el lector de RTPS debe actualizar manualmente el *liveliness* del *DataWriter* DDS asociado con el escritor RTPS del mensaje.

2.2.7.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad RTPS lectora que está siendo informada de los cambios. En este submensaje se puede configurar en *ENTITYID_UNKNOWN* para indicar a todos los lectores que el escritor envió el submensaje.

El **writerId**, identifica la entidad RTPS escritora que realizó el cambio.

El **lastSN**, identifica el último número de secuencia que está disponible en el escritor.

El **Contador**, se incrementa cada vez que se envía un nuevo submensaje *Heartbeat*. Este proporciona los medios para que un lector detecte los submensajes *Heartbeat* duplicados que pueden derivarse de la presencia de las diferentes vías de comunicación..

2.2.7.3. Validez

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el valor del *firstSN* es cero o negativo.
- Cuando el valor del *lastSN* es cero o negativo.
- Cuando el valor *lastSN* es menor que el valor del *firstSN*.

2.2.7.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.7.5. Interpretación lógica

El submensaje Heartbeat tiene dos propósitos:

- Informar al lector de los números de secuencia que están disponibles en el *HistoryCache* del escritor para que el lector pueda solicitar (mediante un *AckNack*) cuales le han faltado.
- Pedir al lector enviar un acuse de recibo para los cambios *CacheChange* que han sido introducidos en la *HistoryCache* del lector para que el escritor conozca el estado del lector.

El lector encontrará siempre el estado del *HistoryCache* del escritor y puede solicitar lo que ha perdido. Normalmente, el lector RTPS sólo enviaría un mensaje *AckNack* si le falta un cambio.

El escritor utiliza el *FinalFlag* para solicitar al lector el envío de un acuse de recibo para los números de secuencia que ha recibido. Si el *Heartbeat* tiene el *FinalFlag* es igual a 1, no es necesario que el lector envíe un mensaje *AckNack*. Sin embargo, si el *FinalFlag* es igual a 0, entonces el lector debe enviar mensaje de *AckNack* que indica que ha recibido el cambio, aunque el *AckNack* indica que ha recibido todos los cambios en *HistoryCache* del escritor.

2.2.7.5.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje *Heartbeat*.

No.	Time	Source	Destination	Protocol	Length	Info
179	14.260170	192.168.3.158	192.168.3.102	RTPS	110	INFO_DST, HEARTBEAT


```

guidPrefix
└─ submessageId: HEARTBEAT (0x07)
  └─ Flags: 0x03 ( _ _ _ _ F E )
    0..... = reserved bit: Not set
    .0..... = reserved bit: Not set
    ..0.... = reserved bit: Not set
    ...0.... = reserved bit: Not set
    ....0... = reserved bit: Not set
    .....0. = Liveliness flag: Not set
    .....1. = Final flag: Set
    .....1 = Endianness bit: Set
  octetsToNextHeader: 28
  readerEntityId: 0x000200c7 (Built-in reader (with key): 0x000200)
    readerEntityKey: 0x000200
    readerEntityKind: Built-in reader (with key) (0xc7)
  writerEntityId: 0x000200c2 (Built-in writer (with key): 0x000200)
    writerEntityKey: 0x000200
    writerEntityKind: Built-in writer (with key) (0xc2)
  firstAvailableSeqNumber: 1
  lastSeqNumber: 0
  count: 1

```

Figura 2-14. Uso del submensaje HEARTBEAT.

Una explicación del uso del *Heartbeat* ha sido descrito dentro del ejemplo 2 y 3 en la sección 3.5.5 de ejemplos de RTPS

2.2.8. HEARTBEATFRAG SUBMESSAGE

En la Figura 2-15 se muestra la estructura del submensaje *HeartBeatFrag*.

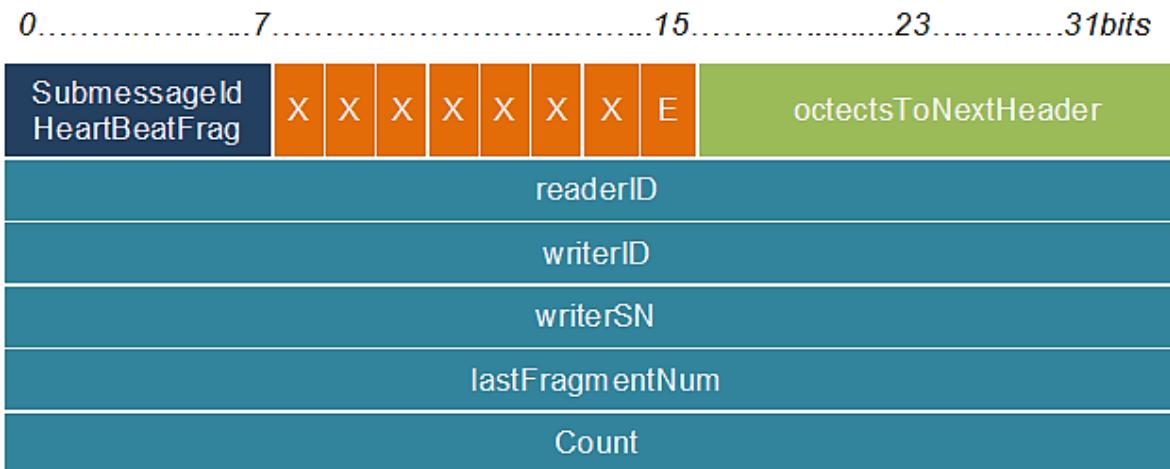


Figura 2-15. Estructura del submensaje *HeartBeatFrag* [41]

El submensaje *HeartbeatFrag* se envía desde un escritor a un lector para comunicar que los fragmentos del escritor están a su disposición. Esto permite una comunicación segura a nivel de fragmento. Una vez que todos los fragmentos están disponibles, se utiliza un mensaje regular de *Heartbeat*.

2.2.8.1. Banderas en el encabezado de submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.8.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica la entidad RTPS lectora que está siendo informada de los cambios.

El **writerId**, identifica la entidad RTPS escritora que realizó el cambio.

El **writerSN**, identifica el número de secuencia de los cambios para que los fragmentos estén disponibles.

El **lastFragmentNum**, indica todos los fragmentos están disponibles en el escritor para el cambio identificado por el *writerSN*.

El **Count**, se incrementa cada vez que se envía un mensaje *HeartbeatFrag*. Este proporciona los medios para que un lector detecte los submensajes *HeartbeatFrag* duplicados que pueden derivarse de la presencia de las diferentes vías de comunicación.

2.2.8.3. Validez

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
- Cuando el *writerSN.value* es cero o negativo.
- Cuando el *lastFragmentNum.value* es cero o negativo.

2.2.8.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor

2.2.8.5. Interpretación lógica

El submensaje de *HeartbeatFrag* tiene el mismo propósito que un submensaje *Heartbeat* regular, pero en lugar de lo que indica la disponibilidad de una gama de números de secuencia, indica la disponibilidad de una gama de fragmentos para el cambio de datos con el número de secuencia *WriterSN*.

El lector RTPS responderá enviando un mensaje *NackFrag*, pero sólo si le falta alguno de los fragmentos disponibles.

2.2.8.5.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje *HeartbeatFrag*.

No.	Time	Source	Destination	Protocol	Length	Info
18	4.3050370	127.0.0.1	127.0.0.1	RTPS	90	HEARTBEAT_FRAG
19	4.8064960	127.0.0.1	127.0.0.1	RTPS	146	TNEO DST ACKNACK NA

Default port mapping. domainName=tor, participantIndex=114, nature=UNICAST_MERITRAN

submessageId: HEARTBEAT_FRAG (0x13)

Flags: 0x01 (_ _ _ _ E)

- 0..... = reserved bit: Not set
- .0..... = reserved bit: Not set
- ..0..... = reserved bit: Not set
- ...0.... = reserved bit: Not set
-0... = reserved bit: Not set
-0.. = reserved bit: Not set
-0. = reserved bit: Not set
-1 = Endianness bit: Set

octetsToNextHeader: 0

readerEntityId: ENTITYID_UNKNOWN (0x00000000)

readerEntityKey: 0x00000000

readerEntityKind: Application-defined unknown kind (0x00)

writerEntityId: 0x00010202 (Application-defined writer (with key): 0x000102)

writerEntityKey: 0x000102

writerEntityKind: Application-defined writer (with key) (0x02)

writerSeqNumber: 6

lastFragmentNum: 2

Count: 1

Figura 2-16. Uso del submensaje HEARTBEAT_FRAG.

2.2.9. INFODESTINATION SUBMESSAGE

En la Figura 2-17 se muestra la estructura del submensaje *InfoDestination*.

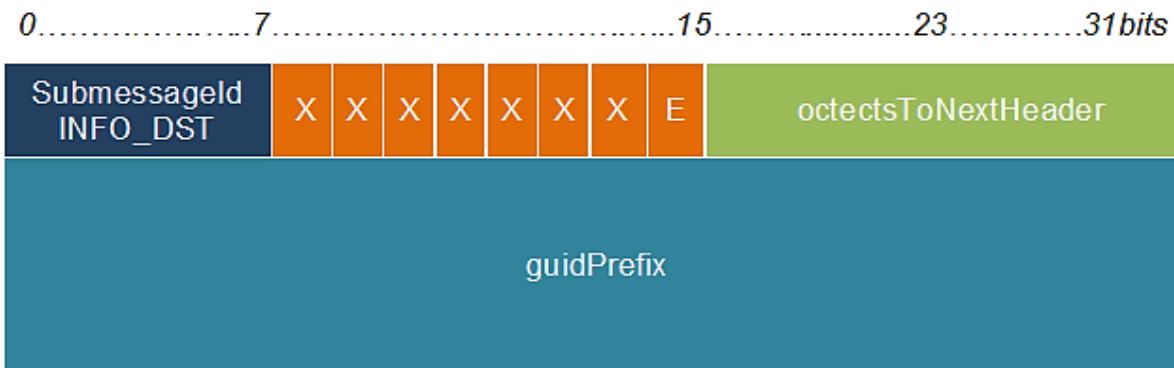


Figura 2-17. Estructura del submensaje *InfoDestination* [41]

Este mensaje se envía desde un escritor RTPS a un lector RTPS para modificar el *GuidPrefix* utilizado para interpretar el *entityId* del lector que aparece en los siguientes submensajes.

2.2.9.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.9.2. Otros elementos en el encabezado del submensaje

El **guidPrefix**, proporciona la identificación que debe utilizarse para reconstruir los GUID de todos los lectores RTPS cuyo *EntityId* aparece en los siguientes submensajes.

2.2.9.3. Validez

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.9.4. Cambio en el estado del receptor

Existe un cambio en el estado del receptor cuando se cumple con (*InfoDestination.guidPrefix*! = *GUIDPREFIX_UNKNOWN*)

2.2.9.4.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje *INFO_DESTINATION*.

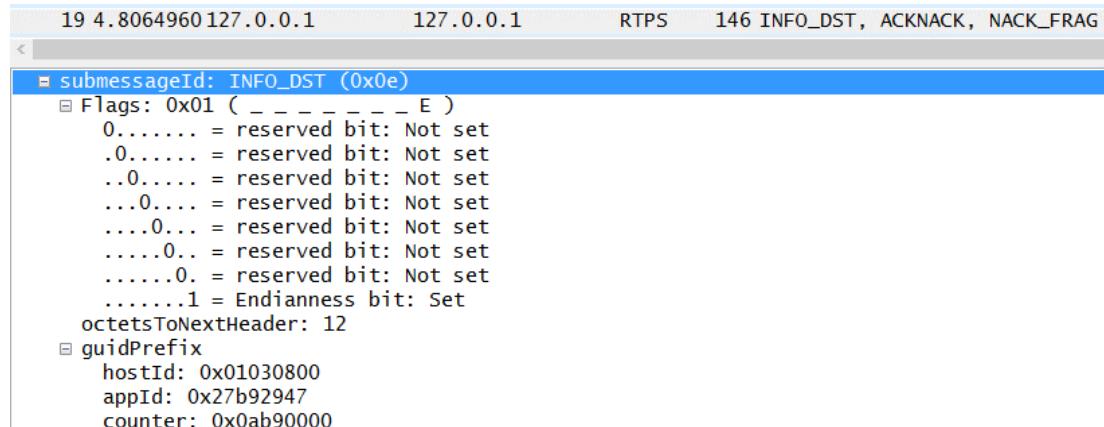


Figura 2-18. Uso del submensaje *INFO_DESTINATION*.

Una explicación del uso del *InfoDestination* se encuentra descrito dentro del ejemplo 2 en la sección 3.5.5 de ejemplos de RTPS.

2.2.10. INFOREPLY SUBMESSAGE

En la Figura 2-19 se muestra la estructura del submensaje *InfoReply*.

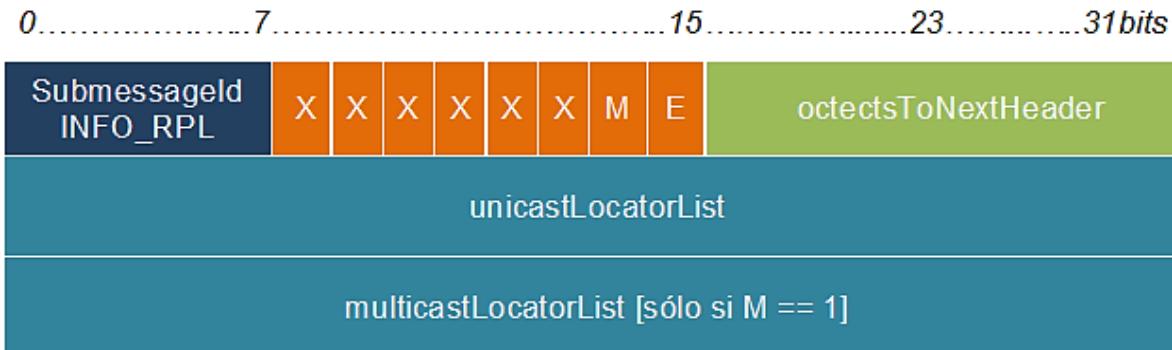


Figura 2-19. Estructura del submensaje *InfoReply* [41]

Este submensaje se envía desde un lector RTPS a un escritor RTPS. Contiene información explícita sobre dónde enviar una respuesta a los submensajes que le siguen en el mismo mensaje.

2.2.10.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **MulticastFlag**, indica si el submensaje también contiene una dirección multicast. El MulticastFlag está representado con el literal M. Si este se encuentra en 1 significa que el submensaje InfoReply también incluye un *multicastLocatorList*.

2.2.10.2. Otros elementos en el encabezado del submensaje

La **unicastLocatorList**, indica un conjunto alternativo de direcciones unicast que el escritor debe utilizar para alcanzara los lectores cuando se replican los submensajes que le siguen.

La **multicastLocatorList**, indica un conjunto alternativo de direcciones de multidifusión que el escritor debe utilizar para llegar a los lectores cuando se replican los submensajes que siguen.

2.2.10.3. Validez

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.10.4. Cambio en el estado del receptor

Se produce un cambio en el estado del receptor cuando la siguiente condición es cumplida:

```
Receiver.unicastReplyLocatorList      =  InfoReply.unicastLocatorList    if
(MulticastFlag)                      {Receiver.multicastReplyLocatorList      =
InfoReply.multicastLocatorList} más {Receiver.multicastReplyLocatorList = < vacío
>}
```

2.2.11. INFOSOURCE SUBMESSAGE

En la Figura 2-20 se muestra la estructura del submensaje *InfoSource*.

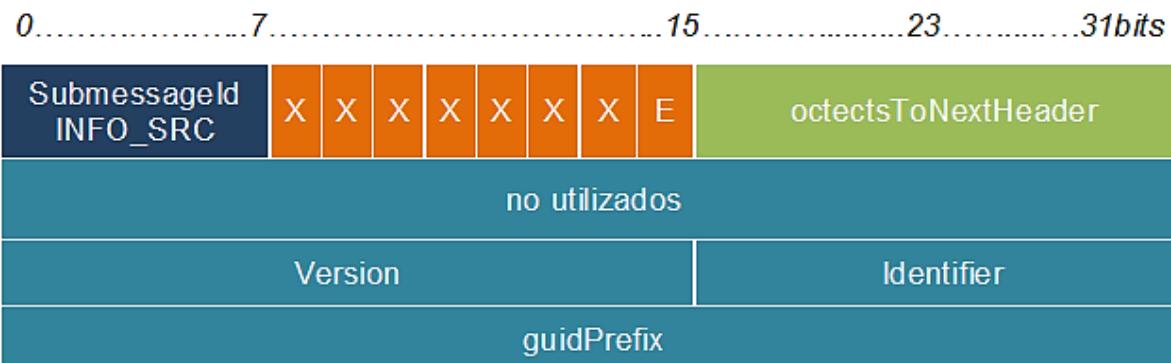


Figura 2-20. Estructura del submensaje *InfoSource* [41]

Este mensaje modifica la fuente de los submensajes que siguen. Es decir este contiene la dirección fuente de los submensajes que se encuentran junto al mismo.

2.2.11.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.11.2. Otros elementos en el encabezado del submensaje

El **protocolVersion**, indica la versión del protocolo utilizado para encapsular submensajes posteriores.

El **identifier**, indica el identificador del fabricante del producto que encapsula submensajes posteriores.

El **guidPrefix**, identifica el participante, es el contenedor del escritor RTPS que representa la fuente de los submensajes posteriores.

2.2.11.3. Validez

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.11.4. Cambio en el estado del receptor

Se produce un cambio en el estado del receptor cuando se cumpla cualquiera de las siguientes condiciones:

- Receiver.sourceGuidPrefix = InfoSource.guidPrefix
- Receiver.sourceVersion = InfoSource.protocolVersion
- Receiver.sourceVendorId = InfoSource.vendorId
- Receiver.unicastReplyLocatorList = {LOCATOR_INVALID}
- Receiver.multicastReplyLocatorList = {LOCATOR_INVALID}
- haveTimestamp = false

2.2.12. INFOTIMESTAMP SUBMESSAGE

En la Figura 2-21 se muestra la estructura del submensaje *InfoTimestamp*.

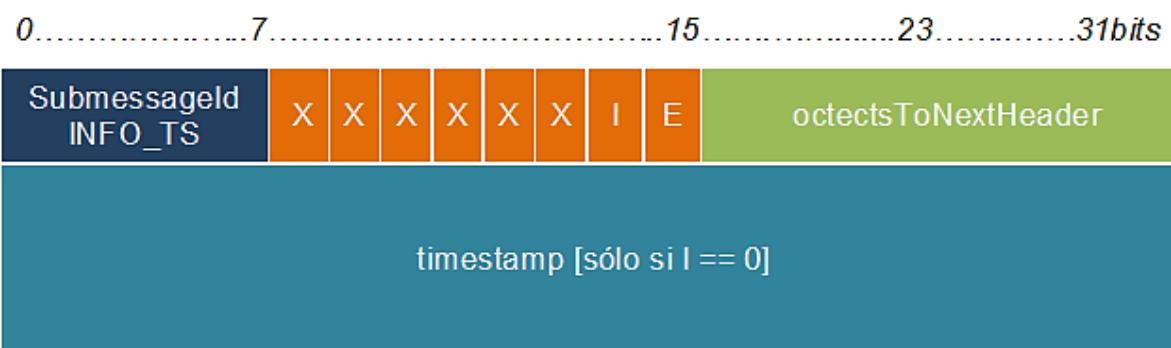


Figura 2-21. Estructura del submensaje *InfoTimestamp* [41]

Este submensaje se utiliza para enviar una marca de tiempo que se aplica a los submensajes que siguen dentro del mismo mensaje.

2.2.12.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **InvalidateFlag**, indica si los submensajes posteriores deben considerar la marca de tiempo o no. El *InvalidateFlag* se representa con el literal '1'. Cuando este es igual a 0 significa que el *InfoTimestamp* también incluye una marca de tiempo. Mientras que si es igual a 1 significa que los submensajes posteriores no deben considerar que tienen una marca de tiempo válida.

2.2.12.2. Otros elementos en el encabezado del submensaje

El **Timestamp**, presenta solamente la marca de tiempo si el *InvalidateFlag* no se encuentra en la cabecera. La marca de tiempo debe utilizarse para interpretar los submensajes posteriores.

2.2.12.3. Validez

Este submensaje es inválido cuando la siguiente condición es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.12.4. Cambio en el estado del receptor

El cambio en el estado del receptor se produce cuando la siguiente condición es verdadera:

```
if (!.InfoTimestamp.InvalidateFlag) {Receiver.haveTimestamp = true
Receiver.timestamp = InfoTimestamp.timestamp} más {Receiver.haveTimestamp =
false}
```

2.2.12.4.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje **INFO_TIMESTAMP**.

No.	Time	Source	Destination	Protocol	Length	Info
44	12.028480	127.0.0.1	127.0.0.1	RTPS	142	GAP, INFO_TS, DATA
45	12.526345	127.0.0.1	127.0.0.1	RTDS	04	HEADTREAT


```

    ☐ submessageId: INFO_TS (0x09)
    ☐ Flags: 0x01 ( _____ E )
      0..... = reserved bit: Not set
      .0..... = reserved bit: Not set
      ..0.... = reserved bit: Not set
      ...0... = reserved bit: Not set
      ....0.. = reserved bit: Not set
      .....0.= Timestamp flag: Not set
      .....1 = Endianness bit: Set
    octetsToNextHeader: 8
    Timestamp: Jan 1, 1970 00:00:00.000000000 UTC
  
```

Figura 2-22. Uso del submensaje INFO_TS.

Una explicación del uso del InfoTimeStamp ha sido descrito dentro del ejemplo 2 en la sección 3.5.5 de ejemplos de RTPS.

2.2.13. NACKFRAG SUBMESSAGE

En la Figura 2-23 se muestra la estructura del submensaje *NackFrag*.

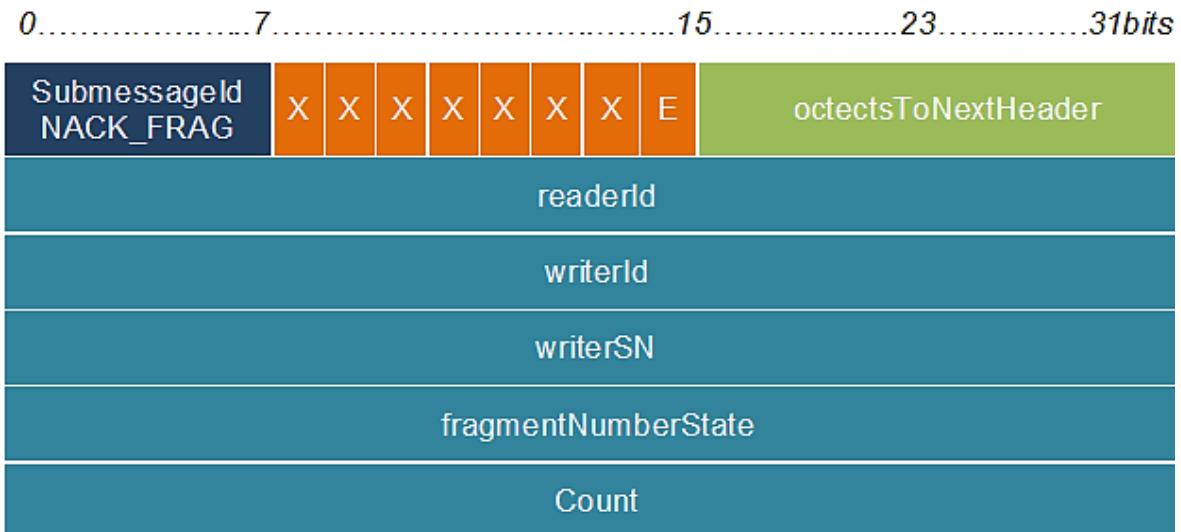


Figura 2-23. Estructura del submensaje *NackFrag* [41]

El submensaje *NackFrag* se utiliza para comunicar el estado de un lector a un escritor. Cuando se envía un cambio con una serie de fragmentos, es decir se envía un dato con un submensaje *DataFrag*, el submensaje *NackFrag* permite al lector dar a conocer al escritor asociado los números fragmento específicos que aún le faltan.

Este submensaje sólo puede contener acuses de recibo negativos. Tenga en cuenta que esto difiere de un submensaje *AckNack*, el cual incluye acuses de recibo tanto positivos como negativos. Las ventajas de este enfoque incluye:

- Eliminar la limitación de ventanas introducida por el submensaje *AckNack*. Dado que el tamaño de un *SequenceNumberSet* se limita a 256 bits, un submensaje *AckNack* se limita solamente a acusar de forma negativa a aquellas muestras cuyo número de secuencia no exceda a aquellos submensajes que primero se perdieron por más de 256 bits. Cualquier muestra que se encuentre por debajo de aquellas que se perdieron primero, son confirmadas. Los submensajes *NackFrag* por otro lado pueden ser utilizados para acusar negativamente cualquier fragmento, incluso fragmentos que tengan mas 256 bits aparte de aquellos que ya han sido acusados negativamente por un submensaje *AckNack* anterior. Esto llega a ser importante cuando manejo las muestras contienen una gran cantidad de fragmentos.
- Los Fragmentos pueden ser acusados negativamente en cualquier orden.

2.2.13.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.13.2. Otros elementos en el encabezado de submensaje

El **readerId**, identifica a la entidad lectora que pide recibir ciertos fragmentos.

El **writerId**, identifica la entidad escritora. Este es el identificador del escritor que pide el reenvío de algunos fragmentos

El **writerSN**, es el número de secuencia de los fragmentos faltantes.

El **fragmentNumberState**, comunica el estado del lector al escritor. Los números de fragmento que aparecen en el conjunto indican fragmentos perdidos en el lado del lector. Los que no aparecen en el conjunto podrían haber sido ya recibidos o no.

El **Count**, se incrementa cada vez que se envía un nuevo submensaje *NackFrag*. Proporciona los medios para que un escritor detecte los submensajes duplicados de *NackFrag* que pueden derivarse de la presencia de las distintas vías de comunicación.

2.2.13.3. Validez

Este submensaje es inválido cuando cualquiera de las siguientes condiciones es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.
 - Cuando el valor del *writerSN* es cero o negativo.
 - Cuando el *fragmentNumberState* no es válido.

2.2.13.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor.

2.2.13.5. Interpretación lógica

El lector envía el submensaje *NackFrag* al escritor para solicitar fragmentos.

2.2.13.5.1. Ejemplo

En la siguiente figura se muestra la captura del submensaje *NACK FRAG*.

Figura 2-24. Uso del submensaje NACK FRAG.

2.2.14. PAP SUBMESSAGE

En la Figura 2-25 se muestra la estructura del submensaje *Pad*.



Figura 2-25. Estructura del submensaje Pad [41]

El propósito de este submensaje es permitir la introducción de cualquier relleno necesario para satisfacer cualquier requisito de alineamiento en la memoria que sea deseado.

2.2.14.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

2.2.14.2. Otros elementos en el encabezado del submensaje

Este submensaje no tienen otros elementos.

2.2.14.3. Validez

Este submensaje siempre es válido.

2.2.14.4. Cambio en el estado del receptor

No provoca cambios en el estado del receptor.

2.2.15. INFOREPLYIP4 SUBMESSAGE

En la Figura 2-26 se muestra la estructura del submensaje *InfoReplyIp4*.



Figura 2-26. Estructura del submensaje *InfoReplyIp4* [41]

Este submensaje se envía desde un lector RTPS a un escritor RTPS. Contiene información explícita sobre dónde enviar una respuesta a los submensajes que le siguen en el mismo mensaje.

Este submensaje por motivos de eficiencia, puede utilizarse en lugar del submensaje *InfoReply* para proporcionar una representación más compacta.

2.2.15.1. Banderas en el encabezado del submensaje

El **EndiannessFlag**, aparece en las banderas de cabecera del submensaje e indica el orden de los bits.

El **MulticastFlag**, indica si el submensaje también contiene una dirección multicast. El MulticastFlag está representado con el literal M. Si este se encuentra

en 1 significa que el submensaje *InfoReply/p4* también incluye un *multicastLocatorList*.

2.2.15.2. Otros elementos en el encabezado del submensaje

La **unicastLocatorList**, indica un conjunto alternativo de direcciones unicast que el escritor debe utilizar para alcanzara los lectores cuando se replican los submensajes que le siguen.

La **multicastLocatorList**, indica un conjunto alternativo de direcciones de multidifusión que el escritor debe utilizar para llegar a los lectores cuando se replican los submensajes que siguen. Sólo esta cuando se establece la MulticastFlag en 1.

2.2.15.3. Validez

Este submensaje es inválido cuando la siguiente condicion es verdadera:

- Cuando el encabezado del submensaje *submessageLength* es demasiado pequeño.

2.2.15.4. Cambio en el estado del receptor

Se produce cambios en el estado del receptor cuando la siguiente condición se cumple:

```
Receiver.unicastReplyLocatorList = InfoReply.unicastLocatorList if
(MulticastFlag) {Receiver.multicastReplyLocatorList =
InfoReply.multicastLocatorList} más {Receiver.multicastReplyLocatorList = < vacío
>}
```

2.3. ANÁLISIS DE REQUISITOS

Una vez realizado el análisis de los paquetes RTPS y en conjunto con los participantes del proyecto *CEPRA VIII-2014-06 Middleware en Tiempo Real basado en el modelo publicación/suscripción*, se definió los requerimientos necesarios para soportar el protocolo RTPS con el Middleware DDS, por lo que se llego a la conclusión que es necesario cumplir con los siguientes requisitos:

- Se deberá implementar los componentes básicos que conforman el DDS, tales como el publicador, suscriptor y el *topic*.
- Se deberá implementar los mecanismos y técnicas para el alcance de la información, es decir se deberá organizar los datos dentro de cada dominio.

- Se deberá implementar los mecanismos de Lectura y Escritura de datos, y el ciclo de vida de los *Topic*.
- Se deberá implementar los componentes necesarios del protocolo RTPS y el mecanismo de descubrimiento proporcionado por el mismo.

Se presenta una breve explicación sobre el contenido de cada requisito.

2.4. MÓDULO DDS

Dentro del módulo DDS se encuentra al Publicador, al Suscriptor, y al Topic.

2.4.1. PUBLICADOR

El **Publicador** será el responsable de la distribución de datos. Podrá publicar los datos de los diferentes tipos de datos. Un *DataWriter* actuará como un *typed*¹³ de acceso a una publicación. Un *DataWriter* se encargará de comunicar a un publicador la existencia de datos de un tipo dado, es decir cuando los valores de los datos hayan sido comunicados al publicador a través de un *DataWriter* apropiado, será responsabilidad del publicador realizar la distribución.

2.4.2. SUSCRIPTOR

El **Suscriptor** será el responsable de recibir los datos publicados y podrá recibir y despachar datos de los diferentes tipos especificados. Para acceder a los datos recibidos, la aplicación deberá utilizar un *typed DataReader* adjunto al suscriptor.

2.4.3. TOPIC

Un **Topic** representará la unidad de información que puede ser producida o consumida; estará compuesta por un tipo, un nombre único y un conjunto de políticas de calidad de servicio, como se muestra en la Figura 2-27, que se utilizará para controlar las propiedades no funcionales asociadas con el Topic. Es decir, que si no se especifica de manera explícita las políticas de QoS, la aplicación DDS utilizará valores predeterminados por la norma.

¹³ Typed, significa que cada objeto *DataWriter* es dedicado a una aplicación de tipo de dato

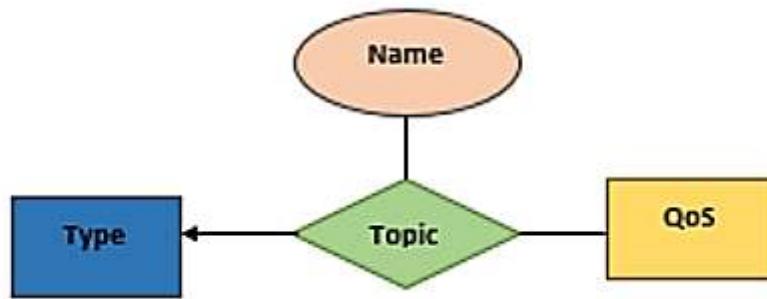


Figura 2-27. Objeto Topic y sus componentes. [37]

Los Topic encajarán entre las publicaciones y suscripciones. Las publicaciones deberán ser conocidas de tal manera que las suscripciones puedan referirse a ellas sin ambigüedades. El Topic tendrá el propósito de:

- Asociar un nombre único en el dominio, es decir, el conjunto de aplicaciones que se comunican entre sí
- Asociar un tipo de datos, y la calidad de servicio en relación con los datos en sí.

Un tipo Topic DDS es descrito por una estructura IDL que contiene un número arbitrario de campos cuyos tipos podrían ser; tipo primitivo como se muestra en la

Tabla 2-1, un tipo *template* como se muestra en la Tabla 2-2, o un tipo compuesto como se muestra en la Tabla 2-3.

Tabla 2-1. Tipos IDL primitivos.

Tipos primitivos
boolean
wchar
short
unsigned short
long
unsigned long
long
float
double
long double

Tabla 2-2. Tipos IDL template.

Tipos Template	Ejemplos
string <length= UNBOUNDED>	string s1; string<32> s2;
wstring<length= UNBOUNDED>	wstring ws1; wstring<64> ws2;
sequence<T, length = UNBOUNDED>	sequence<octect> oseq; sequence<octect, 1024> oseq1k;
	Sequence<MyType> mtseq; Sequence<MyType, 10> mtseq10;
fixed<digits, scale>	fixed<5, 2> fp://d1d2d3.d4d5

Tabla 2-3. Tipos IDL compuestos.

Tipos construidos	Ejemplos
enum	enum Dimension {1D, 2D, 3D, 4D}; struct CoordID {long x;}; struct Coord2D {long x; long y;}; struct Coord3D {long x; long y; long z;}; struct Coord4D {long x; long y; long z; unsigned long long t;};
union	union Coord switch (Dimension) { case 1D: CoordID c1d; case 2D: Coord2D c2d; case 3D: Coord3D c3d; case 4D: Coord4D c4d; };

Los *Topic* deberán ser conocidos por el Middleware DDS. Los objetos del *Topic* serán creados utilizando las operaciones de creación proporcionadas por el *DomainParticipant*.

La interacción será directa por parte del publicador: cuando la aplicación sabe que hacer con los datos disponibles para la publicación, llamará a la operación correspondiente del *DataWriter* relacionado.

La interacción por parte del suscriptor tendrá las siguientes características: la información relevante podrá llegar cuando la aplicación esté ocupada o cuando esté a la espera de esa información. Es decir que dependiendo de la forma en que la aplicación sea diseñada, se utilizarán notificaciones asíncronas o síncronas que son explicadas en la sección 2.6.5.

2.5. MECANISMO Y TÉCNICAS PARA EL ALCANCE DE LA INFORMACIÓN

Los dominios y las particiones serán la manera de organizar los datos. El Topic DDS permitirá crear *topics*, que limitará los valores que pueden tomar sus

instancias. Al suscribirse a un *topic* una aplicación, sólo recibirá, entre todos los valores publicados aquellos valores que coincidan con el filtro del *topic*. Los operadores de los filtros y condiciones de consultas se muestran en la siguiente Tabla 2-4.

Tabla 2-4. Operadores para Filtros DDS y Condiciones de Consulta

Operador	Descripción
=	Igual
◊	Diferente
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
BETWEEN	Entre y rango inclusivo
LIKE	Búsqueda para un patrón

Estos, limitarán la cantidad de memoria utilizada por el Middleware.

2.6. LECTURA Y ESCRITURA DE DATOS

2.6.1. ESCRITURA DE DATOS

Escribir datos dentro del DDS será un proceso simple ya que solo se deberá llamar al método *write* del *DataWriter*. Este permitirá a una aplicación establecer el valor de los datos para ser publicados bajo un determinado Topic.

El ciclo de vida del *topic-instances*, podrá ser manejado implícitamente a través de la semántica implicada por el *DataWriter*, o podrá ser controlada explícitamente por la API *DataWriter*. La transición del ciclo de vida de *topic-instances* puede tener implicaciones en el uso de recursos locales y remotos.

2.6.2. CICLO DE VIDA DE LOS TOPIC-INSTANCES

Los estados disponibles en los *topic-instances* serán:

- *ALIVE*, si por lo menos hay un *DataWriter* escribiendo.
- *NOT_ALIVE_NO_WRITERS*, cuando no haya mas *DataWriters*, escribiendo.
- *NOT_ALIVE_DISPOSED*, si el *DataWriter* ha sido eliminado ya sea de manera implícita debido a un defecto en la calidad de servicio, o de manera explícita mediante una llamada específica en el API *DataWriter*. Este estado indicará que la instancia ya no es relevante para el sistema y que no debe ser escrita más por cualquier escritor.

Como resultado, los recursos asignados por el sistema para almacenar la instancia podrán ser liberados.

2.6.2.1. Administración del ciclo de vida automática

El ciclo de vida automática de las instancias será explicado por medio de un ejemplo. Al observar el código de una supuesta aplicación el cual se muestra en la Tabla 2-5, se puede notar que solamente es para escribir datos, y existen tres operaciones *write*, las cuales crean tres nuevas instancias de *topic* en el sistema, los cuales están asociados a los id=1, 2, 3.

Mientras estas instancias se encuentren en el estado *ALIVE*, estas serán registradas automáticamente, es decir que están asociadas con el *writer*.

Por tanto el comportamiento del DDS será disponer que las instancias del *topic* una vez que se destruya el objeto *DataWriter*, es decir llevando a las instancias a estado *NOT_ALIVE_DISPOSED*.

Tabla 2-5. Administración del Ciclo de Vida Automática

Código de ejemplo del uso del método <i>write()</i>
<pre>int main(int, char**) { dds::Topic<TempSensorType> tsTopic("TempSensorTopic"); dds::DataWriter<TempSensorType> dw(tsTopic); TempSensorType ts; // [NOTE #1]: Instances implicitly registered as part // of the write. // {id, temp hum scale}; ts = {1, 25.0F, 65.0F, CELSIUS}; dw.write(ts); ts = {2, 26.0F, 70.0F, CELSIUS}; dw.write(ts); ts = {3, 27.0F, 75.0F, CELSIUS}; dw.write(ts); sleep(10); // [NOTE #2]: Instances automatically unregistered and // disposed as result of the destruction of the dw object return 0; }</pre>

2.6.2.2. Administración de Ciclo de Vida Explícita

El ciclo de vida de *topic-instances* podrá ser manejado explícitamente por el API definido en el *DataWriter*. En este caso el programador de la aplicación tendrá el control sobre cuándo las instancias son registradas, se dejan de registrar o se eliminan. En esencia, el acto de registrar explícitamente una instancia permitirá al Middleware reservar recursos, así como optimizar la búsqueda de instancias. Finalmente, eliminar un *topic-instances* dará una manera de comunicar al DDS que la instancia no es más relevante para el sistema distribuido, por lo tanto, los

recursos asignados a las instancias específicas deberán ser liberados tanto de forma local como remota.

2.6.2.3. Topic sin claves

Los *topic* sin claves se convierten en únicos. En los *topic* sin clave el estado de transición será vinculado al ciclo de vida del *DataWriter*.

2.6.3. LECTURA DE DATOS

El DDS tendrá dos mecanismos diferentes para acceder a los datos, cada uno de ellos permitirá controlar el acceso a los datos y la disponibilidad de los mismos. El acceso a los datos se realizará a través de la clase *DataReader* exponiendo dos semánticas para acceder a los datos: *read* y *take*.

2.6.3.1. Read y Take

La semántica del *read* implementado por el método *DataReader::read*, dará acceso a los datos recibidos por el *DataReader* sin sacarlo de su caché local. Por lo cual estos datos serán nuevamente legibles mediante una llamada apropiada al *read*. Así mismo, el DDS proporcionará una semántica de *take*, implementado por los métodos *DataReader::take* que permitirá acceder a los datos recibidos por el *DataReader* para removerlo de su caché local. Esto significa que una vez que los datos están tomados, no estarán disponibles para subsecuentes operaciones *read* o *take*.

El *read* y el *take* DDS se encontrarán siempre en modo no bloqueado. Si los datos no están disponibles para leerse, la llamada retornará inmediatamente. Además, si hay menos datos que requieren llamadas reunirán lo disponible y retornará de inmediato. En este modo se asegurará que estos puedan utilizarse con seguridad por aplicaciones que sondan datos.

2.6.4. DATOS Y METADATOS

El ciclo de vida de *topic-instances* junto con otra información que describe las propiedades de las muestras de los datos recibidos estará a disposición del *DataReader* y podrán ser utilizadas para seleccionar los datos accedidos a través del *read* o ya sea del *take*.

Especialmente, para cada muestra de datos recibidos un *DataWriter* será asociado a una estructura, llamada *SampleInfo* describiendo la propiedad de esa muestra. Estas propiedades incluyen información como:

- **Estado de la muestra.** El estado de la muestra podrá ser *READ* o *NOT_READ* dependiendo si la muestra ya ha sido leída o no.
- **Estado de la instancia.** Este indicará el estado de la instancia como *ALIVE*, *NOT_ALIVE_NO_WRITERS*, o *NOT_ALIVE_DISPOSED*.
- **Estado de la vista.** Este podrá ser *NEW* o *NOT_NEW* dependiendo de si es la primera muestra recibida por el *topic-instance*.

El *SampleInfo* también contiene un conjunto de contadores que permitirán reconstruir el número de veces que el *topic-instance* haya realizado cierta transición de estado, convirtiéndose en *alive* después del *disposed*.

Finalmente, el *SampleInfo* contendrá un *timestamp* para los datos y una bandera que dice si la muestra es asociada o no. Esta bandera es importante ya que el DDS podrá generar información válida de las muestras con datos no válidos para informar acerca de las transiciones de estado como una instancia a ser desechara.

2.6.5. NOTIFICACIONES

Una forma de coordinar con DDS será tener un sondeo de uso de datos mediante la realización de un *read* o un *take* de vez en cuando. El sondeo podría ser el mejor enfoque para algunas clases de aplicaciones, el ejemplo más común es en las aplicaciones de control. En general, las aplicaciones podrán ser notificadas de la disponibilidad de datos o tal vez esperar su disponibilidad. El DDS apoyará la coordinación por medio de *waitsets* y los *listeners*.

2.6.5.1. Waitsets

El DDS proporcionará un mecanismo genérico para la espera de eventos. Uno de los tipos soportados de eventos son las condiciones de lectura, las cuales podrán ser usadas para esperar la disponibilidad de los datos de uno o más *DataReaders*.

2.6.5.2. Listeners

Otra manera de encontrar datos a ser leídos, será aprovechar al máximo los eventos planteados por el DDS y asincrónicamente notificar a los *handler* (controladores) registrados. Por lo tanto, si se quiere un *handler* para ser notificado de la disponibilidad de los datos, se deberá conectar el *handler* apropiado con el evento *on_data_available* planteado por el *DataReader*.

Los mecanismos de control de eventos permitirán enlazar cualquier cosa que se quiera a un evento DDS, lo que significa que se puede enlazar una función, un método de clase, etc.. Por ejemplo, cuando se trata con el evento *on_data_available* se tendrá que registrar una entidad exigible que acepte un único parámetro de tipo *DataReader*.

Finalmente, cabe mencionar que el *handler* se ejecutará en un *thread* de Middleware. Como resultado, cuando se utilizan *listeners* se debería minimizar el tiempo del *listener* empleado en el mismo.

2.7. MÓDULO RTPS

Los módulos RTPS están definidos por la PIM. La PIM describe el protocolo en términos de una “máquina virtual.” La estructura de la máquina virtual está construida por clases, las cuales están descritas en el módulo de estructura, además este incluye a los *endpoints* de los *Writer* y *Reader*. Estos extremos se comunican usando los mensajes descritos en el módulo de mensajes. También es necesario describir el comportamiento de la máquina virtual, por medio del módulo de comportamiento, en el cual se observa el intercambio de mensajes que debe tomar lugar entre los extremos. Por último se encuentra el protocolo de descubrimiento usado para configurar la máquina virtual con la información que esta necesita para comunicar a los pares remotos, este protocolo se encuentra descrito en el módulo de descubrimiento.

2.7.1. MÓDULO ESTRUCTURA

El propósito principal de este módulo es describir las clases principales que serán utilizadas por el protocolo RTPS, como se puede observar en la Figura 2-28.

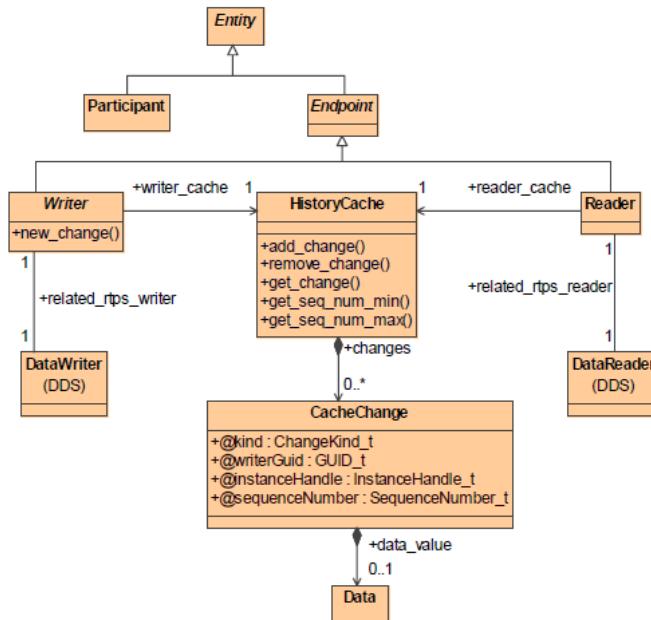


Figura 2-28. Módulo Estructura

Las entidades RTPS muestran los *endpoints* que serán utilizados por las entidades DDS para que se comuniquen entre sí. Cada Entidad RTPS tendrá correspondencia uno a uno con las Entidades DDS. El *HistoryCache* proporcionará una interfaz entre las entidades DDS y su correspondiente entidad RTPS. Por ejemplo, cada operación *write* en un *DataWriter* DDS añadirá un *CacheChange* al *HistoryCache* en su correspondiente RTPS *Writer*. El RTPS *Writer* subsecuentemente transferirá el *CacheChange* al *HistoryCache* de cada *Reader* RTPS asociado. En el lado del receptor, el *DataReader* DDS será notificado por el *Reader* RTPS que un nuevo *CacheChange* ha llegado al *HistoryCache*.

2.7.1.1. Resumen de las clases usadas por la máquina virtual RTPS

Como se puede observar todas las entidades de RTPS son derivadas de la clase *Entity* del RTPS, como se muestra en la Tabla 2-6.

Tabla 2-6. Clases y Entidades RTPS

Entidades y Clases RTPS	
Clase	Propósito
Entity	Es la clase base para todas las entidades RTPS. El <i>Entity</i> representa la clase de objetos que son visibles para otras entidades RTPS en la Red. Estos objetos <i>Entity</i> tienen un identificador único y global llamado <i>GUID</i> y pueden ser referenciados dentro de los mensajes RTPS.
Endpoint	Es la representación de objetos <i>Entity</i> RTPS que pueden ser extremos de comunicación. Es decir, los objetos que pueden ser orígenes o destinos de los mensajes RTPS.

Tabla 2-6. Clases y Entidades RTPS

Participant	Es el contenedor de todas las entidades RTPS que comparten propiedades en común y son localizadas en un espacio de dirección simple.
Writer	Es la representación de objetos <i>Endpoints</i> que puede ser origen de mensajes que comunican <i>CacheChanges</i> .
Reader	Es la representación de objetos <i>Endpoints</i> que puede ser destino de mensajes que comunican <i>CacheChanges</i> .
HistoryCache	<p>Es la clase contenedora usada para almacenar temporalmente y gestionar grupos de cambios.</p> <p>En el lado del escritor este contiene la historia de los cambios en el objeto de datos hechos por el escritor.</p> <p>No todos los cambios deben ser conservados en memoria, por tanto existen reglas para la superposición y acumulamiento para la historia requerida, y también dependerá del estado de comunicaciones y de las políticas de QoS del DDS.</p> <p>En la Figura 2-29 se puede observar un diagrama de clases del <i>HistoryCache</i>.</p>
CacheChange	Representa un cambio individual que se ha hecho al objeto de datos. Estos incluyen la creación, la modificación, y la eliminación de objetos de datos.
Data	Representa a los datos que deben ser asociados con un cambio hecho al objeto de datos.

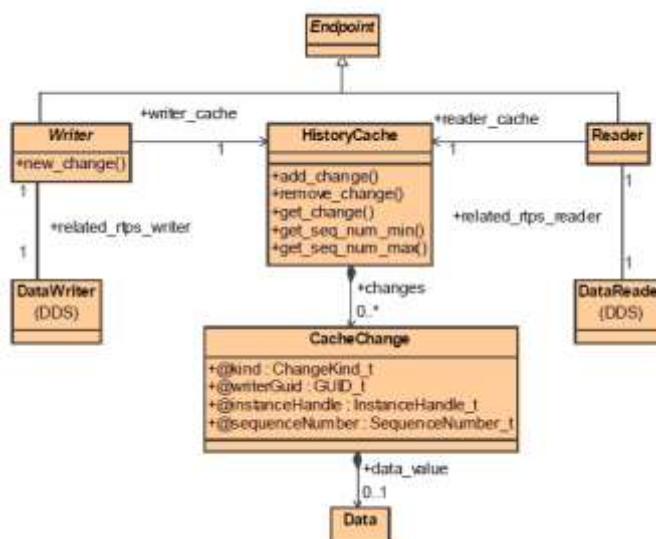


Figura 2-29. HistoryCache

2.7.2. MÓDULO MENSAJES

Este módulo describe la estructura y contenidos lógicos globales de los mensajes que se intercambian entre los puntos finales del *Writer* RTPS y los puntos finales del *Reader* RTPS. Los mensajes RTPS son de diseño modular y se pueden ampliar fácilmente para apoyar tanto nuevas características del protocolo, así como extensiones específicas del proveedor.

2.7.2.1. Estructura general del mensaje RTPS

Consta de una cabecera RTPS de tamaño fijo, seguido de un número variable de Submensajes RTPS. Cada submensaje a su vez consta de un *SubmessageHeader* y un número variable de *SubmessageElements*. Esto se muestra en la Figura 2-30.

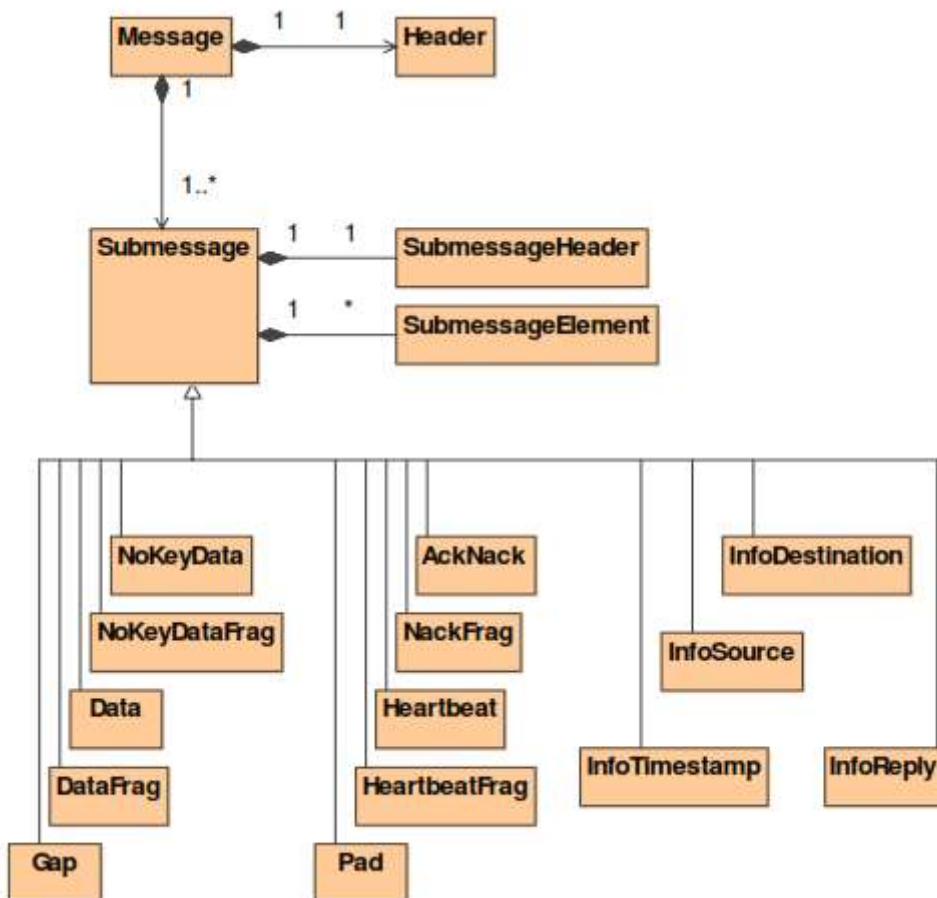


Figura 2-30. Estructura del mensaje RTPS.

Cada mensaje enviado por el protocolo RTPS tendrá una longitud finita. Esta longitud no se envía explícitamente por el protocolo RTPS pero es parte del transporte subyacente con la que se enviarán los mensajes RTPS. En el caso de un transporte orientado a paquetes (como UDP/ IP), la longitud del mensaje ya es proporcionada por la encapsulación del transporte.

2.7.2.1.1. Estructura del Encabezado

El *header* RTPS debe aparecer al principio de cada mensaje. El *header* identifica el mensaje como perteneciente al protocolo RTPS. La cabecera identifica

la versión del protocolo y el *vendor* que envío el mensaje. El *header* contiene los campos que se muestran en la Figura 2-31.

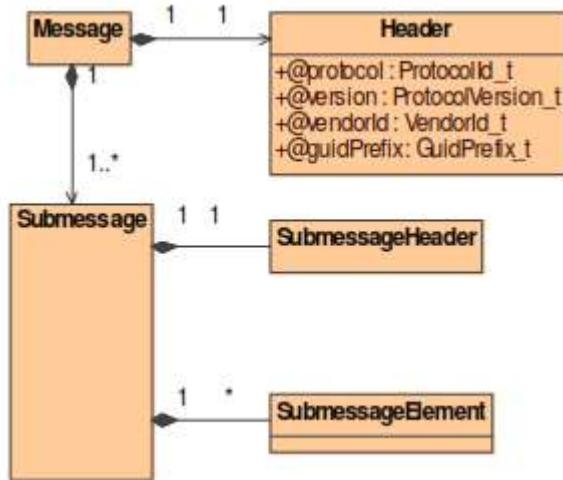


Figura 2-31. Estructura de la cabecera del mensaje RTPS.

2.7.2.1.2. Estructura del submensaje

Cada submensaje RTPS consistirá de un número variable de partes del submensaje RTPS. Todos los submensajes RTPS cuentan con la misma estructura idéntica como se muestra en la Figura 2-32.

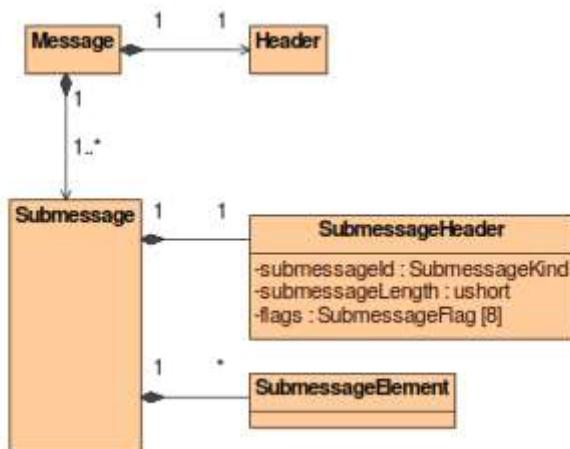


Figura 2-32. Estructura de los submensajes RTPS.

Todos los submensajes empiezan con un *SubmessageHeader* seguido por un *SubmessageElement*. El *header* del submensaje identifica el tipo de mensajes y los elementos opcionales dentro del mismo.

2.7.2.1.2.1. SubmessageId

El *SubmessageId* identifica el tipo del submensaje. A fin de mantener la interoperabilidad con versiones futuras, la plataforma de asignaciones específicas

debe reservar un rango de valores destinados a extensiones de protocolo y un rango de valores que son reservados por el *vendor* de submensajes específicos que nunca serán utilizados por futuras versiones del protocolo RTPS.

2.7.2.1.2.2. Flags

Las *Flags* en la cabecera del submensaje contienen ocho valores booleanos. La primera bandera, el *EndiannessFlag*, está presente y se encuentra en la misma posición en todos los submensajes y representa el orden de bits utilizados para codificar la información en el submensaje.

2.7.2.1.2.3. SubmessageLength

El *SubmessageLength* indica la longitud del submensaje.

Cuando el *SubmessageLength* es mayor a 0, significa que:

- La longitud se mide desde el comienzo de los contenidos del submensaje hasta el comienzo de la cabecera del siguiente submensaje.
- Es la longitud del mensaje restante.

Un intérprete del mensaje puede distinguir entre estos dos casos, ya que conoce la longitud total del mensaje.

Cuando el *SubmessageLength* es igual 0, el submensaje es el último en el mensaje y se extiende hasta el final del mensaje. Esto hace que sea posible enviar submensajes mayores a 64 KB (longitud máxima que se puede almacenar en el campo *submessageLength*), siempre que sean el último submensaje en el mensaje.

2.7.2.2. RTPS Message Receiver

El receptor de un mensaje deberá mantener el estado de los submensajes deserealizados previamente en el mismo mensaje. Este estado se modela como el estado de un receptor RTPS que se reestablece cada vez que un nuevo mensaje se procesa y proporciona un contexto para la interpretación de cada submensaje.

El receptor RTPS se muestra en la Figura 2-33.

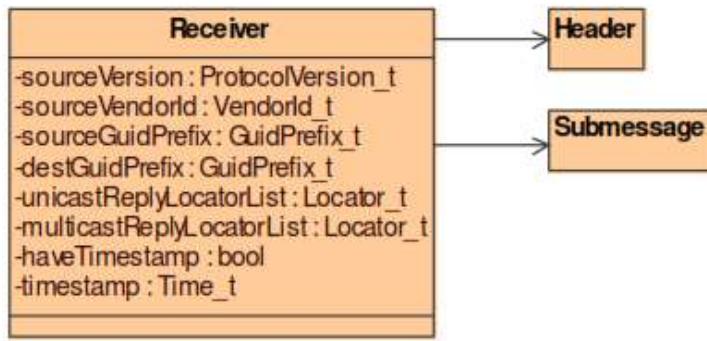


Figura 2-33. Receptor RTPS.

2.7.2.2.1. Reglas seguidas por el Receptor del mensaje

El siguiente algoritmo describe las reglas que un receptor de cualquier mensaje debe seguir:

- Si el encabezado del submensaje no se puede leer, el resto del mensaje se considera no válido.
- El campo *submessageLength* define dónde comienza el siguiente submensaje o indica que el submensaje se extiende hasta el final del mensaje. Si este campo no es válido, el resto del mensaje no es válido.
- Un submensaje con un *SubmessageId* desconocido debe ser ignorado y el análisis debe continuar con el siguiente submensaje.
- En las banderas del submensaje, el receptor de un submensaje debe ignorar banderas desconocidas.
- Un campo *submessageLength* válido siempre debe ser utilizado para encontrar el siguiente submensaje, incluso para submensajes con ID conocidos.
- Un submensaje conocido pero no válido, invalida al resto del mensaje.

La recepción de una cabecera válida y/ o submensaje tiene dos efectos:

- Se puede cambiar el estado del receptor; este estado influye en cómo se interpretan los siguientes submensajes en el mensaje. En esta versión del protocolo, sólo la cabecera y los submensajes *InfoSource*, *InfoReply*, *InfoDestination* e *InfoTimestamp* cambian el estado del receptor.
- Se puede afectar el comportamiento del punto final al que está destinado el mensaje. Esto se aplica a los mensajes básicos RTPS,

tales como: *Data*, *DataFrag*, *HeartBeat*, *AckNack*, *GAP*, *HeartbeatFrag*, *NackFrag*.

2.7.2.3. Elementos del submensaje RTPS

Cada mensaje RTPS contiene un número variable de submensajes RTPS. Cada submensaje RTPS a su vez, se construye a partir de un conjunto de bloques llamados *SubmessageElements*. El RTPS versión 2.2 define los siguientes elementos: submensaje *GuidPrefix*, *entityId*, *sequenceNumber*, *SequenceNumberSet*, *FragmentNumber*, *FragmentNumberSet*, *VendorID*, *ProtocolVersion*, *LocatorList*, *TimeStamp*, *Count*, *SerializedData* y *ParameterList*. A continuación se muestra los elementos del submensaje RTPS en la Figura 2-34.

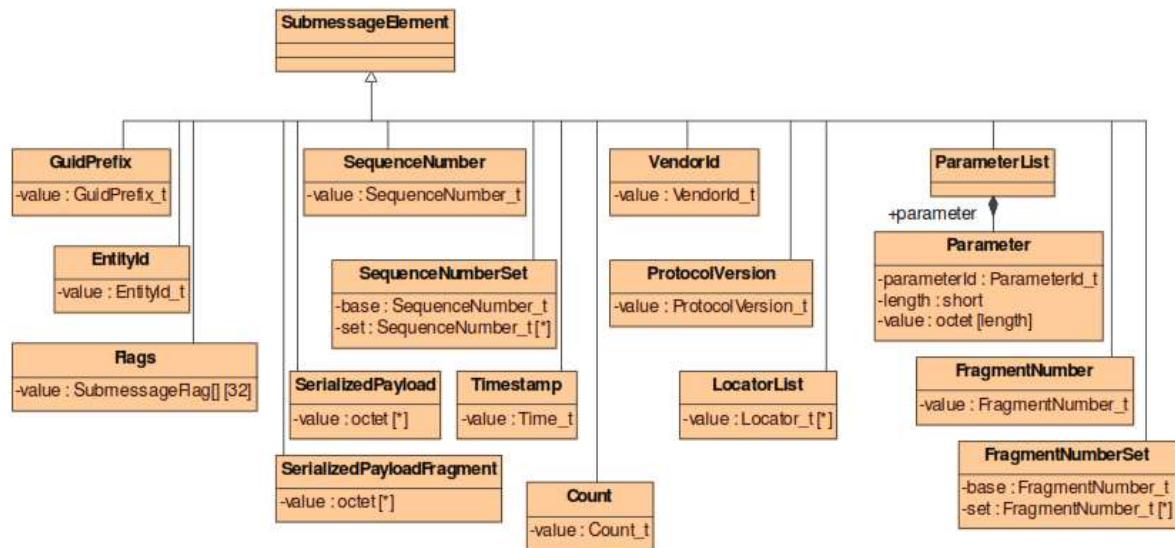


Figura 2-34. Elementos de submensaje RTPS.

El detalle de los elementos del submensaje RTPS se encuentra en la sección del Módulo Mensajes de la norma [41].

2.7.2.4. Submensaje RTPS

El protocolo RTPS de la versión 2.2 define varios tipos de submensajes. Se clasifican en dos grupos: *EntitySubmessages* e *Interpreter-Submessages*.

El *Entity submessage* se dirige a una Entidad RTPS. El *Interpreter-Submessages* modifica el estado del receptor RTPS y proporcionará el contexto que ayuda a los procesos posteriores del *Entity submessage*.

Las entidades del submensaje son:

- El submensaje *Data*, contiene información sobre el valor de un objeto fecha de la aplicación. Los submensajes de datos son enviados por el *Writer* a un *Reader*.
- El *DataFrag*, equivale a los datos, pero sólo contiene una parte del valor (uno o más fragmentos). Permite que los datos se transmitan como varios fragmentos para superar las limitaciones de tamaño de mensajes de transporte.
- El *HeartBeat*, describe la información que está disponible en un *Writer*. Los mensajes *HeartBeat* son enviados por un *Writer* a uno o más *Reader*.
- El *HeartbeatFrag*, sirve para los datos fragmentados, describe que fragmentos están disponibles en un *Writer*. Los submensajes *HeartbeatFrag* son enviados por un *Writer* a uno o más *Reader*.
- El *GAP*, describe la información que ya no es relevante para el *Reader*. Los mensajes *GAP* son enviados por un *Writer* a uno o más *Reader*.
- El *AckNack*, proporciona información sobre el estado de un *Reader* a un *Writer*. Los mensajes *AckNack* son enviados por un *Reader* a uno o más *Writer*.
- El *NackFrag*, proporciona información sobre el estado de un *Reader* a un *Writer*, específicamente los fragmentos de información que siguen perdidos en el *Reader*. Los submensajes *NackFrag* son enviados por un *Reader* a uno o más *Writer*.

Los submensajes de interpretación son:

- El *InfoSource*, proporciona información acerca de la fuente de donde se originaron los Entity Submessage posteriores. Este submensaje se utiliza principalmente para la retransmisión de los submensajes RTPS.
- El *InfoDestination*, proporciona información sobre el destino final de los submensajes que le acompañan. Este submensaje se utiliza principalmente para la retransmisión de submensajes RTPS.
- El *InfoReply*, proporciona información sobre donde responder a las entidades que figuran en submensajes posteriores.

- El *InfoTimestamp*, proporciona una marca de tiempo a los submensajes que le acompañan.
- El *Pad*, se utiliza para agregar relleno a un mensaje, siempre y cuando sea necesaria la alineación de la memoria.

A continuación se muestran los diferentes submensajes RTPS en la Figura 2-35.

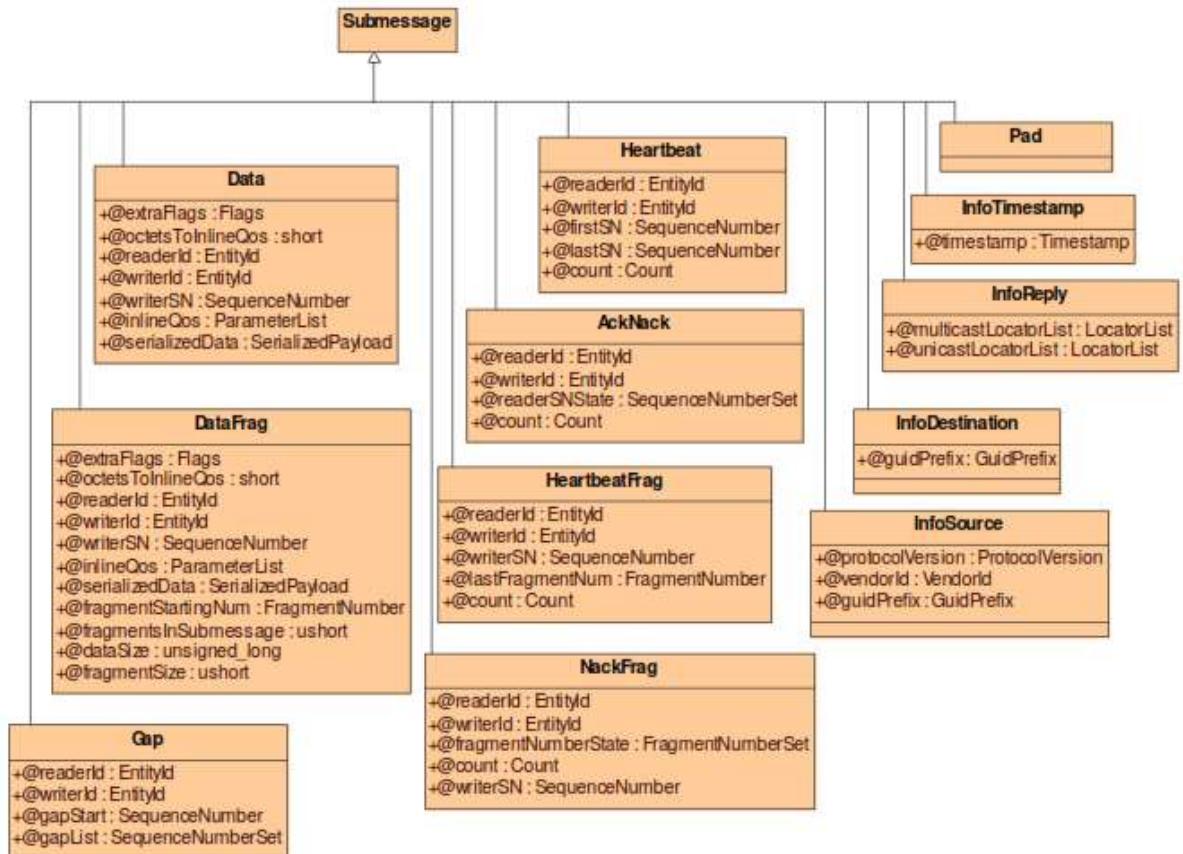


Figura 2-35. Submensajes RTPS.

El detalle de los tipos de submensajes RTPS se encuentra en la sección del Módulo Mensajes de la norma [41].

2.7.3. MÓDULO COMPORTAMIENTO

Una vez que el *Writer* RTPS sea asociado a un *Reader* RTPS, es responsabilidad de ambos, asegurarse que los cambios en el *CacheChange* que existen en la *HistoryCache* de los diferentes *Writers* sean propagados a la *HistoryCache* de los diferentes *Readers*.

Este módulo describe como los pares de *Writer* y *Reader* RTPS asociados deben comportarse para propagar los cambios en el *CacheChange*. Este

comportamiento está definido en términos de los mensajes intercambiados usando a los mensajes RTPS que ya fueron descritos en el punto anterior.

2.7.3.1. Requerimientos Generales

Los siguientes requerimientos aplican a todas las entidades RTPS.

- Todas la comunicaciones deberán tomar lugar usando mensajes RTPS, es decir que ningún otro mensaje que no esta definido en los mensajes RTPS puede ser usado.
- Se debe implementar un *Message Receiver RTPS*, es decir que para interpretar a los submensajes RTPS se deberá usar esta implementación.
- Las características de tiempos en todas las implementaciones deben ser configurables.
- Se debe implementar el protocolo de descubrimiento denominado *Simple Participant and Endpoint Discovery Protocols*, es decir a los protocolos de descubrimientos que cubre el estándar.

2.7.3.2. Comportamiento requerido de los Writer RTPS

- Los *Writer* no deben enviar datos fuera de orden, es decir que estos deben enviar datos en el mismo orden en el que fueron añadidos en la *HistoryCache*.
- Los *Writer* deben incluir valores *in-line QoS* si es requerido por un *Reader*, es decir que un *writer* debe respetar las solicitudes de los *Reader* para recibir mensajes de datos con QoS.
- Los *Writer* deben enviar mensajes *Heartbeat* periódicamente cuando se trabaja en modo confiable, un escritor debe periódicamente informar a cada lector asociado de su disponibilidad de datos, enviando *HeartBeat* periódicos que incluyen el número de secuencia del dato disponible. Si no hay datos disponibles, ningún *Heartbeat* debe ser enviado. Para comunicaciones estrictamente confiables, los escritores deben continuar enviando mensajes Heartbeat a los lectores hasta que los lectores hayan confirmado la recepción de los datos o hayan desparecido.
- Los *Writers* deben eventualmente responder a los acuses de recibo negativos; un acuse de recibo negativo indica que parte de la

información se ha perdido, el escritor debe responder también enviando nuevamente los datos perdidos, enviando un mensaje GAP cuando esta información no es relevante, o enviando un mensaje Heartbeat cuando esta información ya no está disponible.

2.7.3.3. Comportamiento requerido de los Reader RTPS

- Los *Reader* deben responder eventualmente después de recibir un mensaje *Heartbeat* con bandera *final* no establecida con un mensaje *AckNack*, este mensaje debe acusar el recibo de información cuando toda la información ha sido recibida o también podría indicar que algunos datos se han perdido. Además esta respuesta debe ser retardada para evitar rafagas de mensajes.
- Los *Reader* deben responder eventualmente después de recibir *heartbeats* los cuales indican que un dato se ha perdido, hasta recibir un mensaje *Heartbeat*; un lector que está perdiendo información debe responder con un mensaje *AckNack* indicando que información ha perdido. Este requerimiento solamente es aplicado si el lector puede acomodar los datos perdidos en su caché y es independiente de la configuración de la bandera final del submensaje *Heartbeat*.
- Una vez acusado positivamente un mensaje, no se puede acusar negativamente el mismo mensaje.
- Los *Reader* solamente pueden enviar mensajes *AckNack* en respuesta a los mensajes *Heartbeat*.

2.7.3.4. Implementación del Protocolo RTPS

La especificación RTPS establece que una implementación funcional del protocolo debe solamente satisfacer los requerimientos presentados en los dos puntos anteriores. Sin embargo, existen dos implementaciones definidas por el módulo de comportamiento.

- **Implementación sin estado**, la cual esta optimizada para escalabilidad. Esta mantiene virtualmente un no estado en las entidades remotas y por lo tanto escala sin gran problema en sistemas grandes. Además esto implica una escalabilidad mejorada y una disminución en el uso de la memoria, pero un ancho de banda

adicional. La implementación sin estado es ideal para comunicaciones en modo *best-effort* sobre multicast.

- **Implementación con estado**, la cual mantiene el estado de las entidades remotas. Esta minimiza el uso del ancho de banda, pero requiere más memoria y tiene una reducida escalabilidad. También esta garantiza estrictamente comunicaciones confiables y puede aplicar políticas de QoS.

2.7.3.5. Comportamiento de un Writer respecto a Reader asociados

El comportamiento de un escritor RTPS con respecto a sus lectores asociados depende de:

- La configuración del nivel de confiabilidad del escritor y el lector.
- La configuración del tipo de topic usado en el lector y escritor. Es decir controla si los datos que están siendo comunicados corresponden a un topic DDS con una clave definida

No todas las combinaciones de niveles de confiabilidad son posibles con el tipo de topic. En la Tabla 2-7 se muestran las combinaciones posibles.

Tabla 2-7. Combinación de atributos posibles en lectores asociados con escritores [41]

Writer properties	Reader properties	Combination name
topicKind= WITH_KEY reliabilityLevel=BEST EFFORT or reliabilityLevel= RELIABLE	topicKind= WITH_KEY reliabilityLevel=BEST EFFORT	WITH_KEY Best-Effort
topicKind= NO_KEY reliabilityLevel=BEST EFFORT or reliabilityLevel= RELIABLE	topicKind=NO_KEY reliabilityLevel=BEST EFFORT	NO_KEY Best-Effort
topicKind=WITH_KEY reliabilityLevel=RELIABLE	topicKind=WITH_KEY reliabilityLevel=RELIABLE	WITH_KEY Reliable
topicKind=NO_KEY reliabilityLevel=RELIABLE	topicKind=NO_KEY reliabilityLevel=RELIABLE	NO_KEY Reliable

2.7.3.6. Comportamiento de Writer sin estado

2.7.3.6.1. Comportamiento de Writer sin estado con mejor esfuerzo

En la siguiente Figura 2-36 podremos observar el comportamiento de este tipo de escritor.

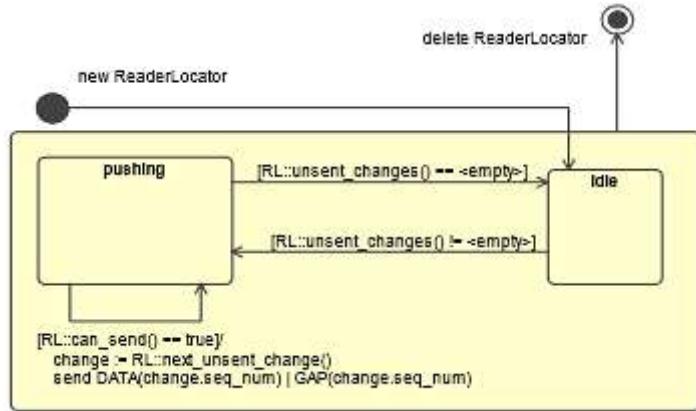


Figura 2-36. Comportamiento de un Writer sin estado con WITH_KEY Best-Effort con respecto a cada ReaderLocator [41]

El listado de estados se encuentra descrito en la Tabla 2-8.

Tabla 2-8. Transiciones del comportamiento en mejor esfuerzo de un Writer sin estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	Initial	El escritor RTPS es configurado con un ReaderLocator.	Idle
T2	Idle	Se indica que hay algunos cambios en el HistoryCache del escritor que aún no han sido enviados al ReaderLocator.	Pushing
T3	Pushing	Se indica que todos los cambios en el HistoryCache del escritor han sido enviados al ReaderLocator.	Idle
T4	Pushing	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al ReaderLocator.	Pushing
T5	Any state	El escritor RTPS es configurado para no mantener más al ReaderLocator.	Final

2.7.3.6.2. Comportamiento de Writer sin estado confiable

En la siguiente Figura 2-37 podremos observar el comportamiento de este tipo de escritor.

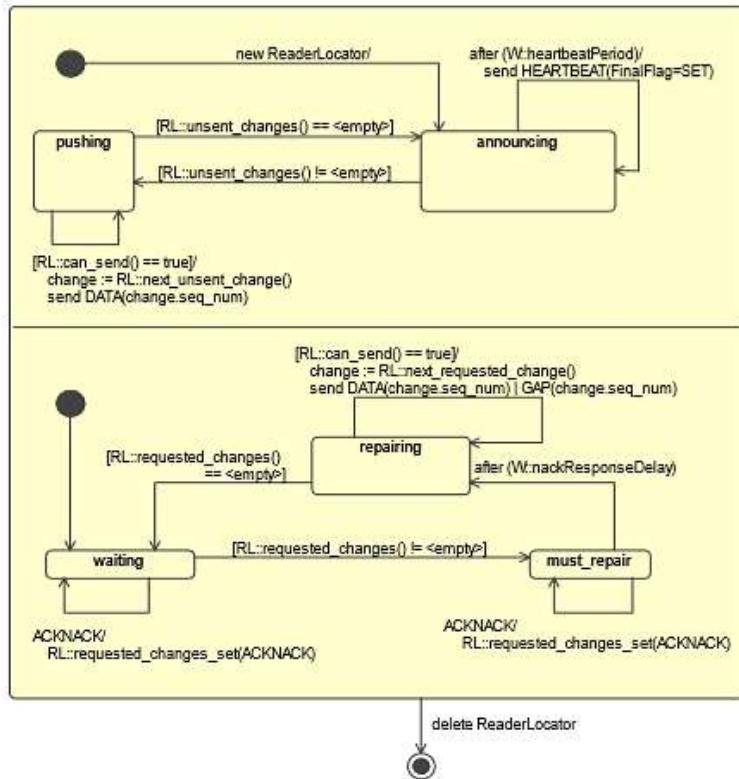


Figura 2-37. Comportamiento de un Writer sin estado con WITH_KEY Reliable con respecto a cada ReaderLocator [41]

El listado de estados se encuentra descrito en la Tabla 2-9.

Tabla 2-9. Transiciones del comportamiento en confiable de un Writer sin estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	Initial	El escritor RTPS es configurado con un ReaderLocator.	Announcing
T2	Announcing	Se indica que hay algunos cambios en el HistoryCache del escritor que no han sido enviados al ReaderLocator.	Pushing
T3	Pushing	Se indica que todos los cambios del HistoryCache del Writer han sido enviados al ReaderLocator.	Announcing
T4	Pushing	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al ReaderLocator.	Pushing
T5	Announcing	Se busca enviar con un temporizador periódico cada Heartbeat.	Announcing
T6	Waiting	Se recepta ACKNACK que han sido destinados al escritor sin estado	Waiting
T7	Waiting	Se indica que hay cambios que han sido solicitados por algún lector RTPS	Must_repair

Tabla 2-9. Transiciones del comportamiento en confiable de un Writer sin estado con respecto a cada ReaderLocator

		alcanzable hacia el ReaderLocator.	
T8	Must_repair	Se recepta ACKNACK que han sido destinados al escritor sin estado	Must_repair
T9	Must_repair	Se busca enviar con un temporizador que la duración del ACKNACK ha caducado mientras se ha entrado a este modo.	Repairing
T10	Repairing	Se indica que el escritor RTPS tiene los recursos necesitados para enviar un cambio al ReaderLocator.	Repairing
T11	Repairing	Se indica que no hay más cambios solicitados por un lector alcanzable para el ReaderLocator.	Waiting
T12	Any state	El escritor RTPS es configurado para no mantener más al ReaderLocator.	Final

2.7.3.7. Comportamiento de Writer con estado

2.7.3.7.1. Comportamiento de Writer con estado con mejor esfuerzo

En la siguiente Figura 2-38 podremos observar el comportamiento de este tipo de escritor.

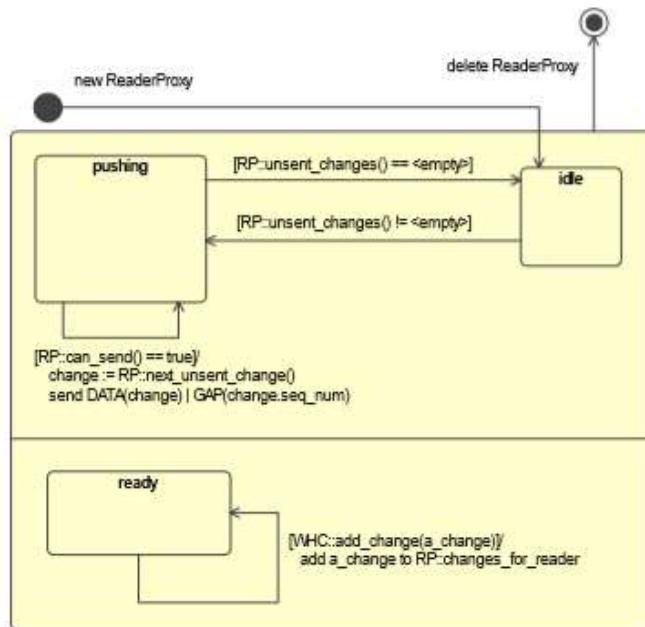


Figura 2-38. Comportamiento de un Writer con estado con WITH_KEY Best-Effort con respecto a cada ReaderLocator [41]

El listado de estados se encuentra descrito en la Tabla 2-10.

Tabla 2-10. Transiciones del comportamiento en mejor esfuerzo de un Writer con estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	Initial	El escritor RTPS es asociado con un <i>Reader</i> .	Idle
T2	Idle	Se indica que hay algunos cambios en el <i>HistoryCache</i> del escritor que aún no han sido enviados al <i>Reader</i> representado por el <i>ReaderProxy</i> .	Pushing
T3	Pushing	Se indica que todos los cambios en el <i>HistoryCache</i> del escritor han sido enviados al <i>Reader</i> representado por el <i>ReaderProxy</i> .	Idle
T4	Pushing	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al <i>Reader</i> representado por el <i>ReaderProxy</i> .	Pushing
T5	Ready	Un nuevo cambio fue añadido al <i>HistoryCache</i> del <i>Writer</i> .	Ready
T6	Any state	El escritor RTPS es configurado para no mantener más al <i>Reader</i> representado por el <i>ReaderProxy</i> .	Final

2.7.3.7.2. Comportamiento de Writer con estado confiable

En la siguiente Figura 2-39 podremos observar el comportamiento de este tipo de escritor.

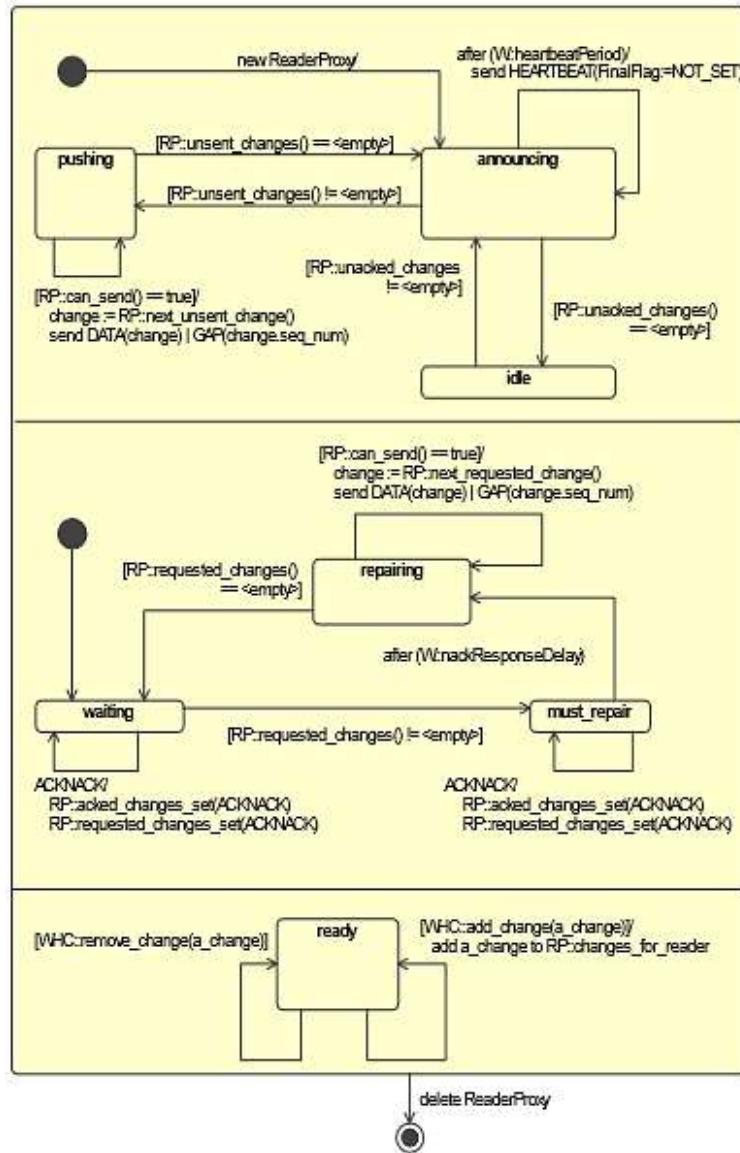


Figura 2-39. Comportamiento de un Writer con estado con WITH_KEY Reliable con respecto a cada ReaderLocator [41]

El listado de estados se encuentra descrito en la Tabla 2-11.

Tabla 2-11. Transiciones del comportamiento en confiable de un

Writer con estado con respecto a cada ReaderLocator

Transición	Estado	Evento	Siguiente Estado
T1	Initial	El escritor RTPS es asociado con un Reader.	Announcing
T2	Announcing	Se indica que hay algunos cambios en el HistoryCache del escritor que no han sido enviados al Reader representado por un ReaderProxy.	Pushing
T3	Pushing	Se indica que todos los cambios del HistoryCache del Writer han sido enviados al Reader representado por el ReaderProxy.	Announcing

Tabla 2-11. Transiciones del comportamiento en confiable de un Writer con estado con respecto a cada ReaderLocator

T4	<i>Pushing</i>	Se indica que el escritor tiene los recursos necesitados para enviar un cambio al Reader representado por el ReaderProxy.	<i>Pushing</i>
T5	<i>Announcing</i>	Se indica que todos los cambios en el HistoryCache del Writer han sido confirmados por el Reader representado por el ReaderProxy.	<i>Idle</i>
T6	<i>Idle</i>	Se indica que hay cambios en el HistoryCache en el Writer que no han sido confirmados por el Reader representado por el ReaderProxy.	<i>Announcing</i>
T7	<i>Announcing</i>	Se busca enviar con un temporizador periódico cada Heartbeat.	<i>Announcing</i>
T8	<i>Waiting</i>	Se recepta ACKNACK que han sido destinados al escritor con estado	<i>Waiting</i>
T9	<i>Waiting</i>	Se indica que hay cambios que han sido solicitados por algún lector RTPS alcanzable hacia el Reader representado por el ReaderProxy.	<i>Must_repair</i>
T10	<i>Must_repair</i>	Se recepta ACKNACK que han sido destinados al escritor con estado	<i>Must_repair</i>
T11	<i>Must_repair</i>	Se busca enviar con un temporizador que la duración del ACKNACK ha caducado mientras se ha entrado a este modo.	<i>Repairing</i>
T12	<i>Repairing</i>	Se indica que el escritor RTPS tiene los recursos necesitados para enviar un cambio al Reader representado por el ReaderProxy.	<i>Repairing</i>
T13	<i>Repairing</i>	Se indica que no hay más cambios solicitados por un lector alcanzable para el Reader representado por el ReaderProxy.	<i>Waiting</i>
T14	<i>Ready</i>	Se añade un nuevo CacheChange al HistoryCache del Writer correspondiente al DDS Writer.	<i>Ready</i>
T15	<i>Ready</i>	Se remueve un CacheChange al HistoryCache del Writer correspondiente al DDS Writer.	<i>Ready</i>
T16	<i>Any state</i>	El escritor RTPS es configurado para no mantener más al Reader representado por el ReaderProxy.	<i>Final</i>

2.7.3.8. Comportamiento de Reader sin estado

2.7.3.8.1. Comportamiento de Reader sin estado con mejor esfuerzo

En la siguiente Figura 2-40 podremos observar el comportamiento de este tipo de lector.

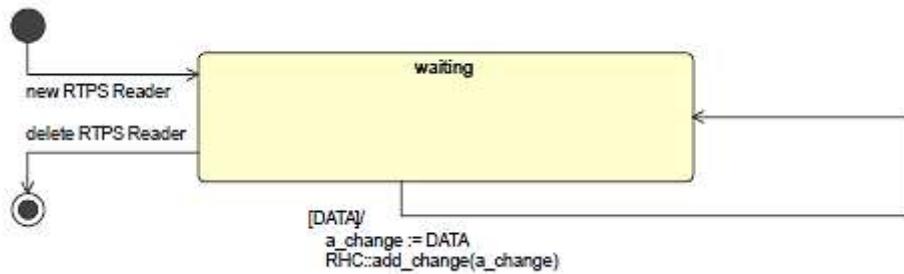


Figura 2-40. Comportamiento de un Reader sin estado con WITH_KEY Best-Effort [41]

El listado de estados se encuentra descrito en la Tabla 2-12.

Tabla 2-12. Transiciones del comportamiento en mejor esfuerzo de un Reader sin estado

Transición	Estado	Evento	Siguiente Estado
T1	Initial	El lector RTPS es creado	Waiting
T2	Waiting	El mensaje DATA es recibido	Waiting
T3	Waiting	El lector RTPS es borrado.	Final

2.7.3.8.2. Comportamiento de Reader sin estado confiable

Esta combinación no es soportada por el protocolo RTPS, es decir que para tener una implementación confiable, el lector RTPS debe mantener algún estado.

2.7.3.9. Comportamiento de Reader con estado

2.7.3.9.1. Comportamiento de Reader con estado con mejor esfuerzo

En la siguiente Figura 2-41 podremos observar el comportamiento de este tipo de lector.

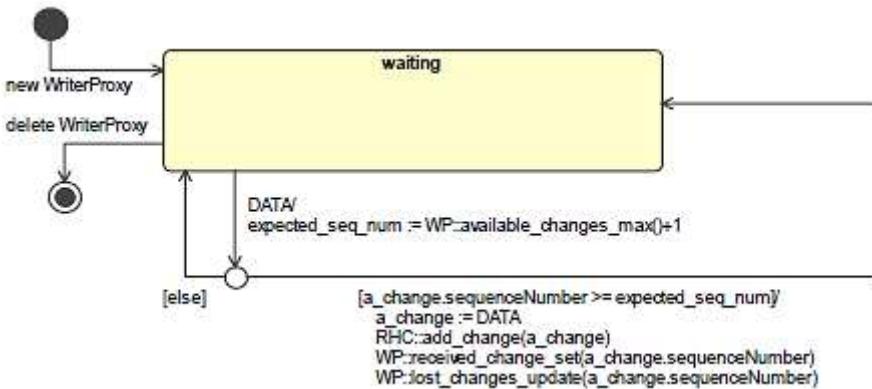


Figura 2-41. Comportamiento de un Reader con estado con WITH_KEY Best-Effort con respecto a cada Writer asociado [41]

El listado de estados se encuentra descrito en la Tabla 2-13.

Tabla 2-13. Transiciones del comportamiento en mejor esfuerzo de un Reader con estado con respecto a cada Writer asociado

Transición	Estado	Evento	Siguiente Estado
T1	Initial	El lector RTPS es configurado con su escritor asociado.	Waiting
T2	Waiting	El mensaje DATA es recibido desde el escritor asociado.	Waiting
T3	Waiting	El lector RTPS es configurado para no estar más asociado con el escritor.	Final

2.7.3.9.2. Comportamiento de Reader con estado con mejor esfuerzo

En la siguiente Figura 2-42 podremos observar el comportamiento de este tipo de lector.

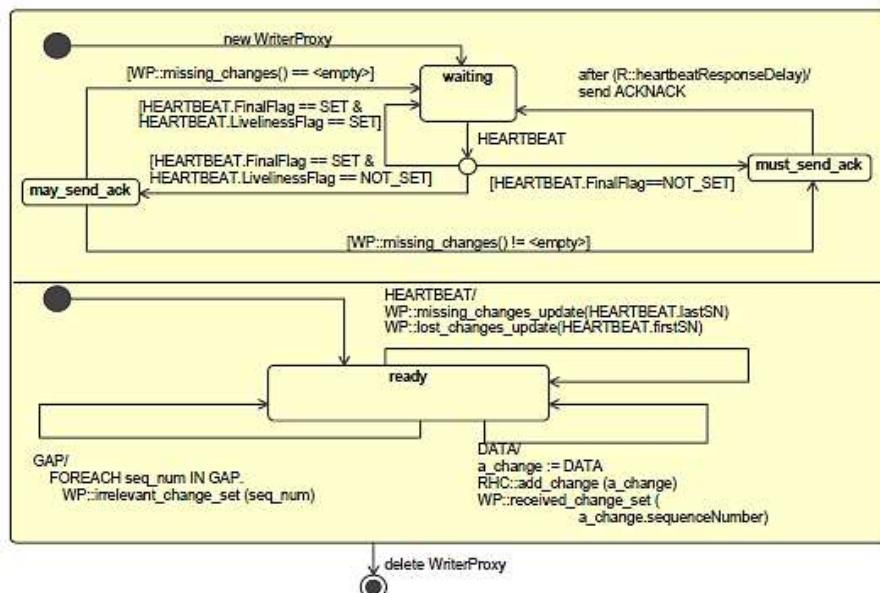


Figura 2-42. Comportamiento de un Reader con estado con WITH_KEY Reliable con respecto a cada Writer asociado [41]

El listado de estados se encuentra descrito en la Tabla 2-14.

Tabla 2-14. Transiciones del comportamiento en confiable de un Reader con estado con respecto a su Writer asociado

Transición	Estado	Evento	Siguiente Estado
T1	Initial	El lector RTPS es configurado con un escritor asociado.	Waiting
T2	Waiting	El mensaje HEARTBEAT es recibido.	Si la finalflag está en 0 Must_send_ack Si la livelinessflag está en 0 May_send_ack Sino Waiting
T3	May_send_ack	Se indica que todos los cambios conocidos a estar en el HistoryCache del Writer	Waiting

Tabla 2-14. Transiciones del comportamiento en confiable de un Reader con estado con respecto a su Writer asociado

		representado por el <i>WriterProxy</i> han sido recibidos por el <i>Reader</i> .	
T4	<i>May_send_ack</i>	Se indica que hay algunos cambios conocidos a estar en el <i>HistoryCache</i> del <i>Writer</i> representado por el <i>WriterProxy</i> , que no han sido recibidos por el <i>Reader</i> .	<i>Must_send_ack</i>
T5	<i>Must_send_ack</i>	Se busca enviar con un temporizador periódico cada Heartbeat.	<i>Waiting</i>
T6	<i>Initial2</i>	Similar a la transición 1.	<i>Ready</i>
T7	<i>Ready</i>	Se recepta un mensaje HEARTBEAT que ha sido destinado al lector con estado.	<i>Ready</i>
T8	<i>Ready</i>	Se recepta un mensaje DATA destinado al lector con estado.	<i>Ready</i>
T9	<i>Ready</i>	Se recepta un mensaje GAP destinado al lector con estado .	<i>Ready</i>
T10	<i>Any state</i>	El Reader RTPS no se encuentra más asociado con el Writer RTPS representado por el <i>WriterProxy</i> .	<i>Final</i>

2.7.4. MÓDULO DESCUBRIMIENTO

El módulo descubrimiento definirá el protocolo de descubrimiento RTPS. El propósito del protocolo de descubrimiento permitirá que cada *participante* RTPS descubra otros relevantes *participantes* y sus *endpoint*. Una vez que el *endpoint* ha sido descubierto, las implementaciones podrán configurar *endpoint* locales para establecer la comunicación.

La especificación RTPS divide al protocolo de descubrimiento en dos protocolos independientes:

- *Participant Discovery Protocol (PDP)*
- *Endpoint Discovery Protocol (EDP)*

Un PDP especifica como los participantes se descubren entre sí en la red. Una vez que dos *participantes* se han descubierto, intercambian información sobre

los *endpoint* que los contienen utilizando un EDP. Aparte de esta relación de causalidad, ambos protocolos se pueden considerar independientes.

A fin de la interoperabilidad, todas las implementaciones RTPS deben proporcionar al menos los siguientes protocolos de descubrimiento:

- Simple Participant Discovery Protocol (SPDP)
- Simple Endpoint Discovery Protocol (SEDP)

Ambos son protocolos básicos de descubrimiento que bastan para pequeñas redes de mediana escala. Los PDP adicionales y EDP que están orientados hacia las redes más grandes se pueden añadir a las futuras versiones de la especificación.

Finalmente, el rol de un protocolo de descubrimiento será proporcionar información sobre *endpoint* remotos descubiertos.

2.7.4.1. Simple Participant Discovery Protocol

El propósito de este protocolo será descubrir la presencia de otros participantes en la red y sus propiedades.

Un participante puede soportar varios PDP, pero para el propósito de interoperabilidad, todas las implementaciones deberían soportar al menos SPDP.

El RTPS SPDP utiliza un enfoque simple para anunciar y detectar la presencia de *participantes* en un dominio.

2.7.4.2. Simple Endpoint Discovery Protocol

Un EDP define la información intercambiada requerida entre dos *participantes* para descubrir *Writer* y *Reader Endpoint*.

Un participante puede soportar varios EDP, pero para el propósito de interoperabilidad, todas las implementaciones soportarían para el caso de *Simple Endpoint Discovery Protocol*.

2.7.4.3. Apoyos alternativos para Protocolos de Descubrimiento

Los requisitos sobre el Participante y Protocolos de Descubrimiento *Endpoint* pueden variar en función del escenario de implementación.

Por ejemplo, un protocolo optimizado para la velocidad y simplicidad (como un protocolo que sea desplegado en dispositivos de una LAN) no pueden escalar bien a los grandes sistemas en un entorno WAN.

Por esta razón, la especificación RTPS permite implementaciones que soportan múltiples PDP y EDP. Hay muchos enfoques posibles para la implementación de un protocolo de descubrimiento que incluye el uso de

descubrimiento estático, el descubrimiento de archivos, servicio de consulta central, etc. El único requisito impuesto por RTPS con el propósito de interoperabilidad es que todas las implementaciones RTPS soporten al menos el SPDP y SEDP.

Si una aplicación soporta múltiples PDP, cada PDP puede ser inicializado de manera diferente y descubrir un conjunto diferente de los participantes remotos. Los *participantes* remotos mediante la implementación RTPS de un *vendor* diferente deben ser contactados usando al menos el SPDP para garantizar la interoperabilidad. No existe tal requisito cuando el participante remoto utiliza la misma implementación RTPS.

Incluso cuando el SPDP es utilizado por todos los participantes, los participantes remotos pueden todavía utilizar diferentes EDP. Los EDP soportan participantes que incluye en la información intercambiada por el SPDP. Todos los participantes deben soportar al menos el SEDP, por lo que siempre tienen al menos un EDP en común. Sin embargo, si dos participantes apoyan a otro EDP, este protocolo alternativo puede ser utilizado en su lugar.

CAPÍTULO 3

DISEÑO E IMPLEMENTACIÓN DE UN MÓDULO QUE PERMITA INTERACTUAR AL PROTOCOLO RTPS CON DDS

3.1. INTRODUCCIÓN

En este capítulo primeramente se diseña un módulo que permita la interacción entre RTPS y DDS y que trabaje con el modelo publicador/suscriptor, por medio de diagramas de clases del módulo RTPS.

Seguidamente se muestra la implementación del diseño, utilizando normas de convención, implementación del protocolo RTPS, implementaciones necesarias DDS, implementación de archivos de configuración.

Finalmente se presenta la interacción entre DDS y RTPS con el modelo publicador/suscriptor por medio de diagramas de secuencia.

3.2. API-RTPS

A partir del API proporcionado por la OMG el cual es el API-RTPS, se muestran los diagramas de clase correspondientes a los modulos que son descritos posteriormente.

El API se encuentra organizado por medio de un modelo con varios niveles lógicos, lo que permite tener un mejor control sobre el código, disminuir su complejidad y dividirlos en submódulos. Estos submódulos son: el submódulo de mensaje y encapsulación, el submódulo de descubrimiento y el submódulo de comportamiento.

El submódulo de mensaje y encapsulación es el encargado de los codificadores, del encapsulamiento y de la administración del encapsulamiento. El submódulo de descubrimiento es el encargado de los mensajes de descubrimiento y de finalizar el descubrimiento de los *participantes*. Finalmente el submódulo de comportamiento es el encargado de la interfaz con el módulo DDS el cual es desarrollado por el grupo de investigación a partir del API-DDS; dentro del submodulo de comportamiento se obseva específicamente la interfaz hacia los *writer* y *reader* y el funcionamiento con estado y sin estado.

3.2.1. DIAGRAMAS DE CLASE A PARTIR DEL API-RTPS

3.2.1.1. Submódulo de mensaje y encapsulamiento

El submódulo de mensaje y encapsulamiento es el encargado de los codificadores, del encapsulamiento y de la administración del encapsulamiento, para lo cual el API-RTPS define una serie de clases que permiten la codificación y decodificación de los diferentes mensajes RTPS, cabeceras, e identificadores. Así como se muestra en la Figura 3-1, Figura 3-2 y Figura 3-3.

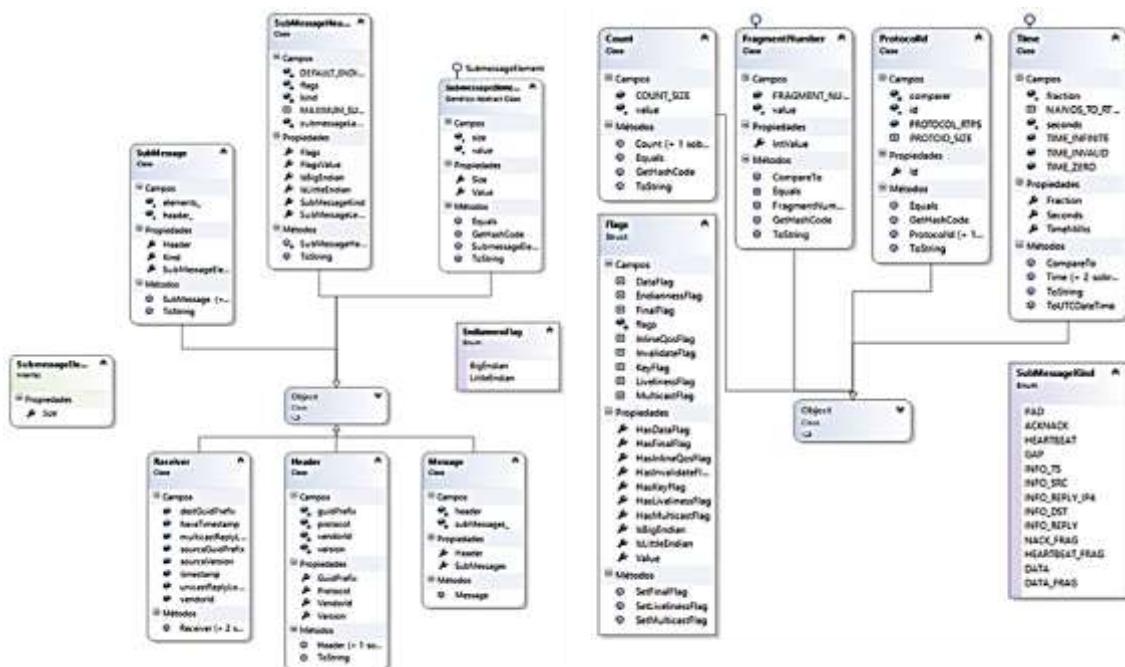


Figura 3-1. Clases del API-RTPS para los mensaje y el encapsulamiento I

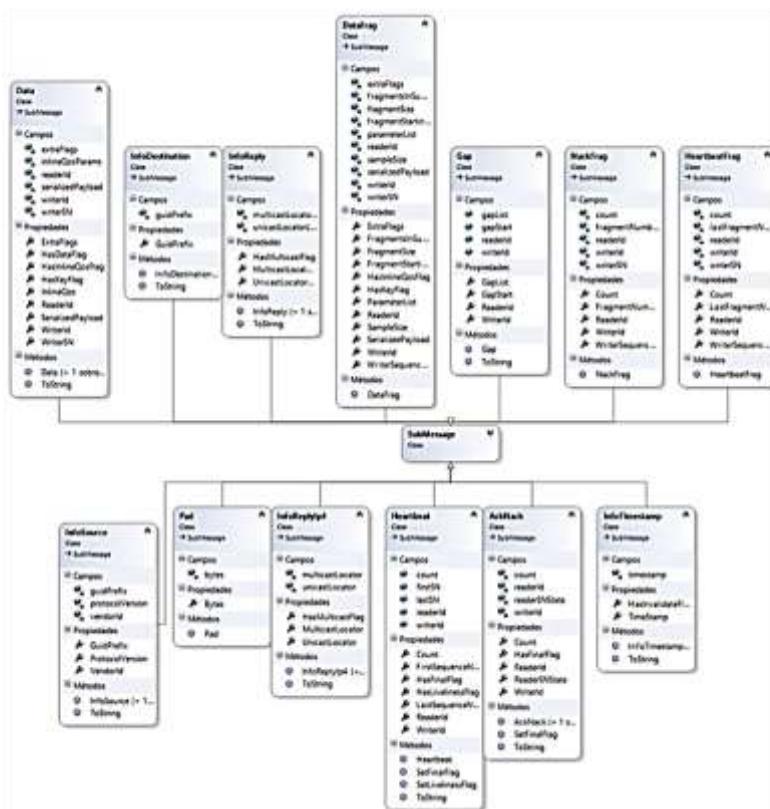


Figura 3-2. Clases del API-RTPS para los mensajes y el encapsulamiento II

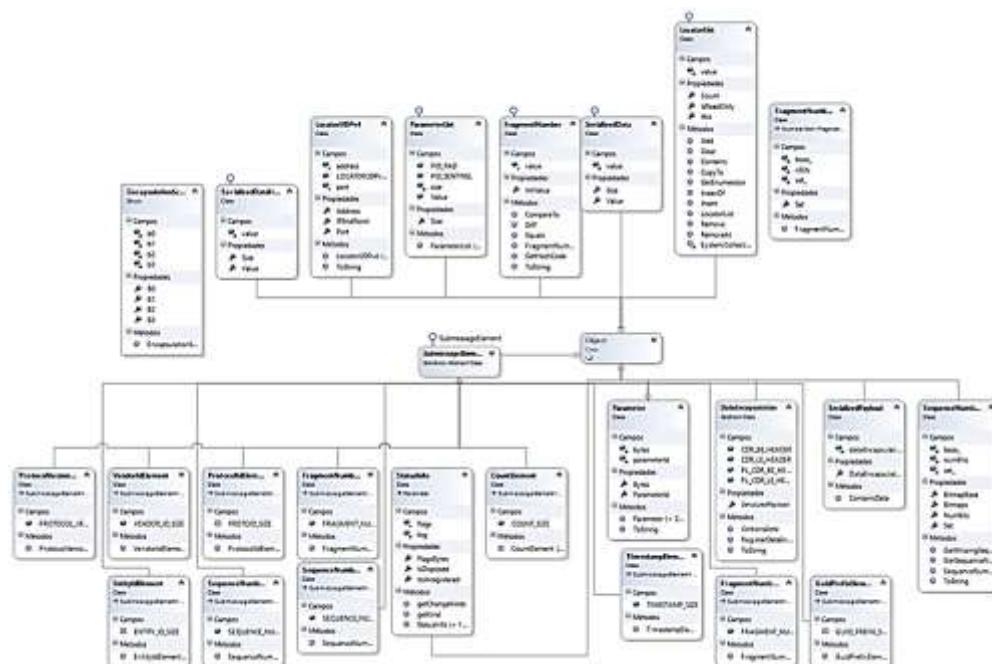


Figura 3-3. Clases del API-RTPS para los mensajes y el encapsulamiento III

3.2.1.2. Submódulo de descubrimiento

El submódulo de descubrimiento es el encargado de los mensajes de descubrimiento y de finalizar el descubrimiento de los participantes, para lo cual el API-RTPS define una serie de clases que permiten el descubrimiento. A continuación en la Figura 3-4 se muestra el diagrama de clase basado en el API-RTPS del submódulo de descubrimiento.

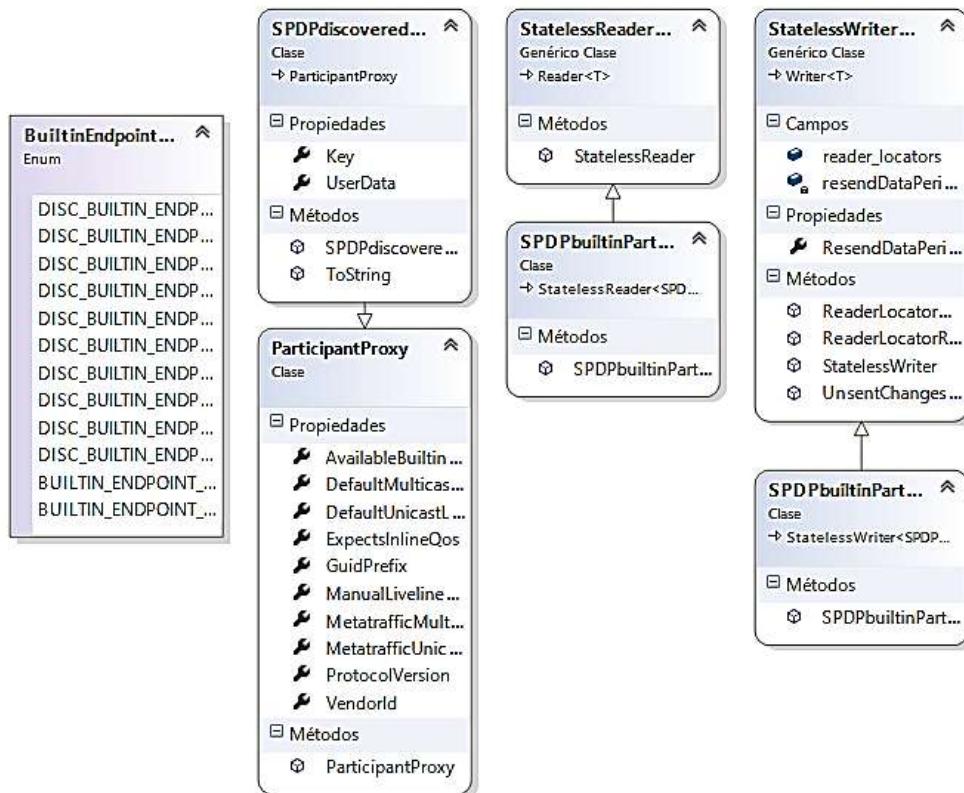


Figura 3-4. Clases del API-RTPS para el descubrimiento

3.2.1.3. Submódulo de comportamiento

El submódulo de comportamiento es el encargado de la interfaz con el módulo DDS, de los *writer* y *reader* y del funcionamiento con estado y sin estado, para lo cual el API-RTPS define una serie de clases que permiten mostrar el comportamiento en la comunicación. A continuación se presenta en la Figura 3-5 el diagrama de clases del submódulo comportamiento.

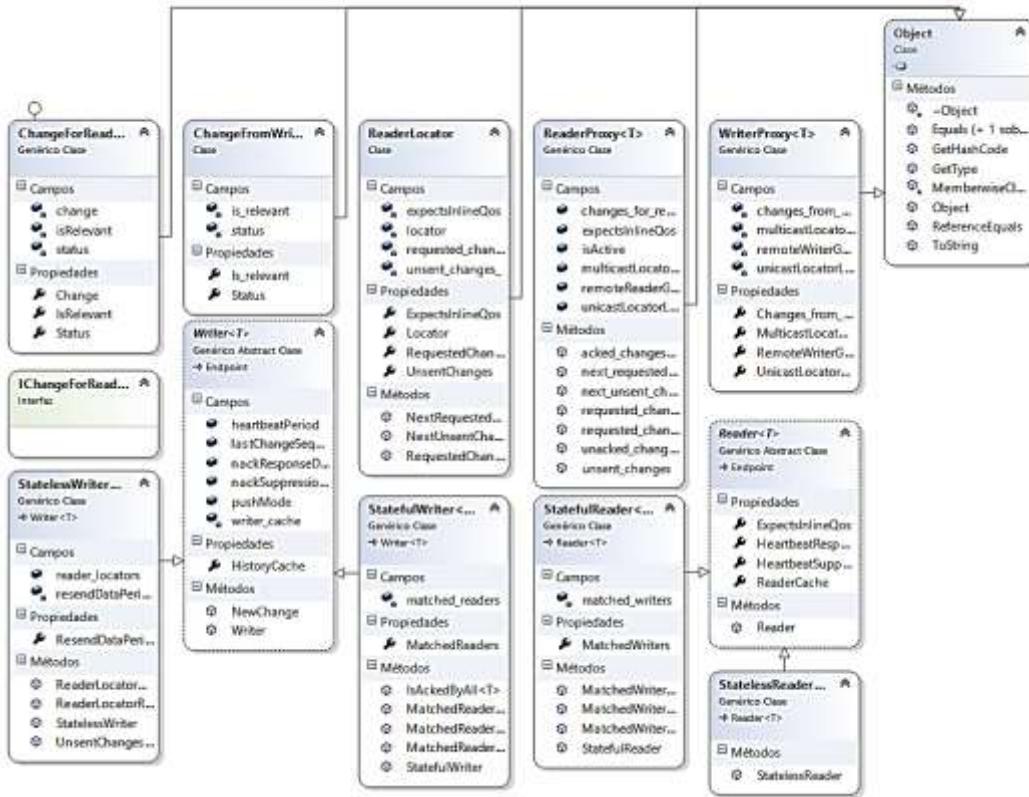


Figura 3-5. Clases del API-RTPS para el comportamiento

3.3. DISEÑO DEL MÓDULO

Una vez recopilada la información necesaria, y establecidos los requerimientos para soportar el protocolo RTPS con el Middleware DDS, en esta sección se realiza el diseño del módulo que permita la interacción entre el RTPS y el DDS.

En el desarrollo del sistema no se sigue un modelo rígido, sino un modelo cambiante el cual se consolida a medida que se avanza en la investigación, es por eso que se presentan diagramas obtenidos al final del desarrollo del proyecto.

En base a los submódulos descritos en el API-RTPS, se adaptó los mismo y además se agrega al diseño 2 submodulos extras, los cuales son: el submódulo de transporte y el submódulo de configuración.

El submódulo de transporte es el encargado del envío y recepción de mensajes RTPS y de mensajes de descubrimiento RTPS. El submódulo de configuración es el encargado de interpretar un archivo de configuración que tiene parámetros configurables tanto del módulo DDS como del módulo RTPS, este submódulo no se encuentra especificado en el API-RTPS.

3.3.1. ADAPTACIÓN DEL API-RTPS PARA INTERACTUAR CON DDS

En relación a la sección 3.2, se procedió con el diseño, por medio de la creación de las clases y métodos necesarios para que RTPS interactúe con DDS.

3.3.1.1. Submódulo de mensaje y encapsulamiento

El submódulo de mensaje y encapsulamiento es el encargado de los codificadores, del encapsulamiento y de la administración del encapsulamiento, para lo cual se define una serie de clases que permiten la codificación y decodificación de los diferentes mensajes RTPS, cabeceras, e identificadores. Así como se muestra en la Figura 3-6 y la Tabla 3-1.

A partir del submódulo de mensaje y encapsulación del API-RTPS, se creó nuevas clases, las cuales son necesarias para la correcta implementación del submódulo, por ejemplo se definió los codificadores y decodificadores de cada tipo de submensaje, como también las clases necesarias para serializar los campos de un mensaje RTPS.



Figura 3-6. Diagrama de clases de la encapsulación de mensajes, cabeceras e identificadores

Tabla 3-1. Métodos de las clases para encapsulamiento de mensajes, cabeceras e identificadores

Clase	Método	Descripción
DataSubmessageEncoder HeartbeatEncoder InfoTimestampEncoder MessageStaticEncoder SubmessageHeaderEncoder SubmessageEncoder DataFragEncoder HeartbeatFragEncoder NackFragEncoder GapEncoder InfoDestinationEncoder InfoSourceEncoder LocatorUDPV4Encoder AckNackEncoder EncapsulationSchemeEncoder PadEncoder InfoReplyEncoder InfoReplyIp4Encoder ParameterEncoder StatusInfoEncoder TimeEncoder ParameterListEncoder HeaderEncoder SequenceNumber SequenceNumberSet MessageCodecFactory	Get	Obtiene el objeto del tipo (nombre de la clase) del buffer
MessageDecoder	Put	Pone un objeto del tipo (nombre de la clase) en el buffer
EntityIdEncoder ProtocolIdEncoder GuidPrefixEncoder VendorIdEncoder	DoDecode	Empieza el proceso de decodificación del mensaje
Sentinel	Get	Obtiene el objeto (nombre de la clase) del buffer
	Put	Pone el objeto (nombre de la clase) en el buffer
	Read	Leer el (nombre de la clase) del buffer
	Write	Escribir o cambia un (nombre de la clase) del buffer
EntityIdSerializer VendorIdSerializer GuidPrefixSerializer GUIDSerializer ProtocolIdSerializer	Sentinel	Crea una instancia en el espacio de memoria correspondiente a la instancia Sentinel
	GetStaticMethods	Crea los métodos delegados (nombre de la clase) de escritura y lectura
	GetSubtypes	Obtiene un subtipo
	Handles	Compara al tipo del tipo de (nombre de la clase)

Una vez que se tiene la posibilidad de codificar y decodificar mensajes, cabeceras e identificadores dentro de un buffer. Es necesario tener un esquema de serialización el cual se encuentra mostrado en la Figura 3-7 y Tabla 3-2.

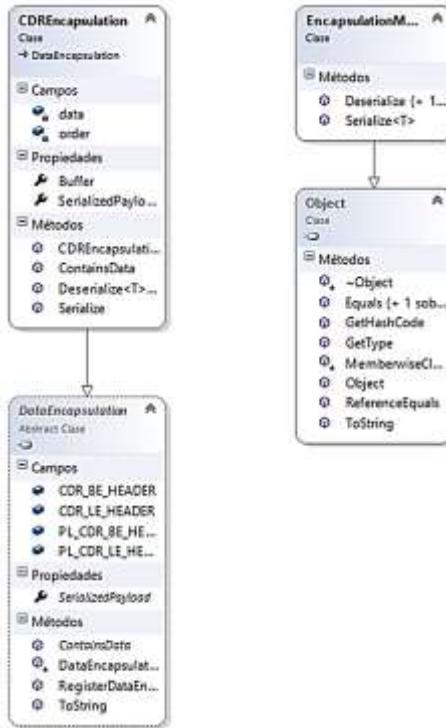


Figura 3-7. Diagrama de clases para la serialización de mensajes

Tabla 3-2. Métodos de las clases para la serialización y deserialización

Clase	Método	Descripción
CDREncapsulation	CDREncapsulation	Es el método que sirve para serializar mensajes de datos
	ContainsData	Informa si contiene Datos o no
	Deserialize	Deserializa desde un buffer
	Serialize	Serializa
EncapsulationManager	Deserialize	Gestiona la deserialización
	Serialize	Gestiona la serialización

3.3.1.2. Submódulo de descubrimiento

El submódulo de descubrimiento es el encargado de los mensajes de descubrimiento y de finalizar el descubrimiento de los participantes, para lo cual se define una serie de clases que permiten el descubrimiento de mensajes RTPS a partir del API-RTPS. A continuación en la Figura 3-8 se muestra el diagrama de clases del submódulo de descubrimiento y en la Tabla 3-3 se realiza una explicación de cada método utilizado en el submódulo.

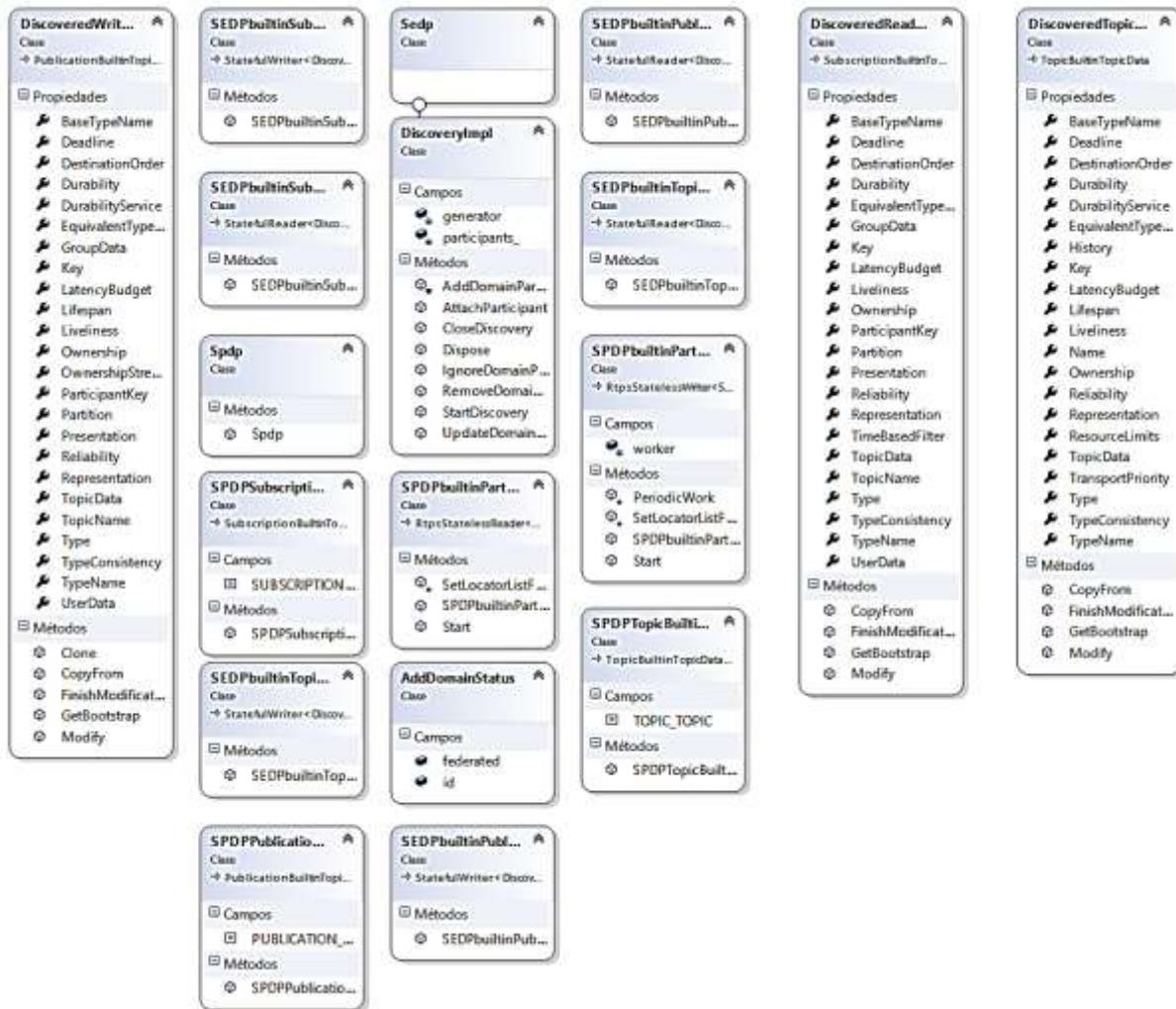


Figura 3-8. Diagrama de clases del submódulo descubrimiento.

Tabla 3-3. Métodos de las clases para el descubrimiento.

Clases	Método	Descripción
SED P builtin Subscriptions Writer	SED P builtin Subscriptions Writer	Se refiere a la suscripción de un Writer por medio del protocolo de descubrimiento SEDP.
SED P builtin Subscriptions Reader	SED P builtin Subscriptions Reader	Se refiere a la suscripción de un Reader por medio del protocolo de descubrimiento SEDP.
SED P builtin Topics Writer	SED P builtin Topics Writer	Se refiere a la representación de un topic Writer en el protocolo SEDP,
SED P builtin Topics Reader	SED P builtin Topics Reader	Se refiere a la representación de un topic Reader en el protocolo SEDP.
Sedp		Se refiere a la representación de una instancia del protocolo SEDP en la implementación del protocolo RTPS.

Tabla 3-3. Métodos de las clases para el descubrimiento.

SEDPbuiltinPublicationsReader	SEDPbuiltinPublicationsReader	Se refiere a las publicaciones descubiertas por el Reader por medio del protocolo SEDP.
SEDPbuiltinPublicationsWriter	SEDPbuiltinPublicationsWriter	Se refiere a las publicaciones descubiertas por el Writer por medio del protocolo SEDP.
Spdp	Spdp	Se refiere a la representación de una instancia del protocolo SPDP en la implementación del protocolo RTPS.
SPDPSsubscriptionBuiltinTopicData	SPDPSsubscriptionBuiltinTopicData	Se refiere a la suscripción en el protocolo SPDP de un topic.
SPDPPublicationBuiltinTopicData	SPDPPublicationBuiltinTopicData	Se refiere a las publicaciones descubiertas por el topic por medio del protocolo SPDP.
SPDPbuiltinParticipantWriterImpl	PeriodicWork	Envía periódicamente objetos de datos.
	SetLocatorListFromConfig	Lista pre configurada con objetos de datos para anunciar la presencia de una participante en la red.
	Start	Inicia el envío de objetos de datos.
SPDPbuiltinParticipantReaderImpl	SetLocatorListFromConfig	Lista pre configurada con objetos de datos para anunciar la presencia de una participante en la red.
	Start	Inicia la recepción de objetos de datos.

3.3.1.3. Submódulo de comportamiento

El submódulo de comportamiento es el encargado de la interfaz con el módulo DDS, de los *writer* y *reader* y del funcionamiento con estado y sin estado, para lo cual se define una serie de clases en base al API-RTPS que permiten mostrar el comportamiento de la comunicación. A continuación se presenta en la Figura 3-9 el diagrama de clases del submódulo comportamiento y en la Tabla 3-4 se muestran los métodos necesarios para las clases para la comunicación entre el protocolo RTPS y el Middleware DDS.

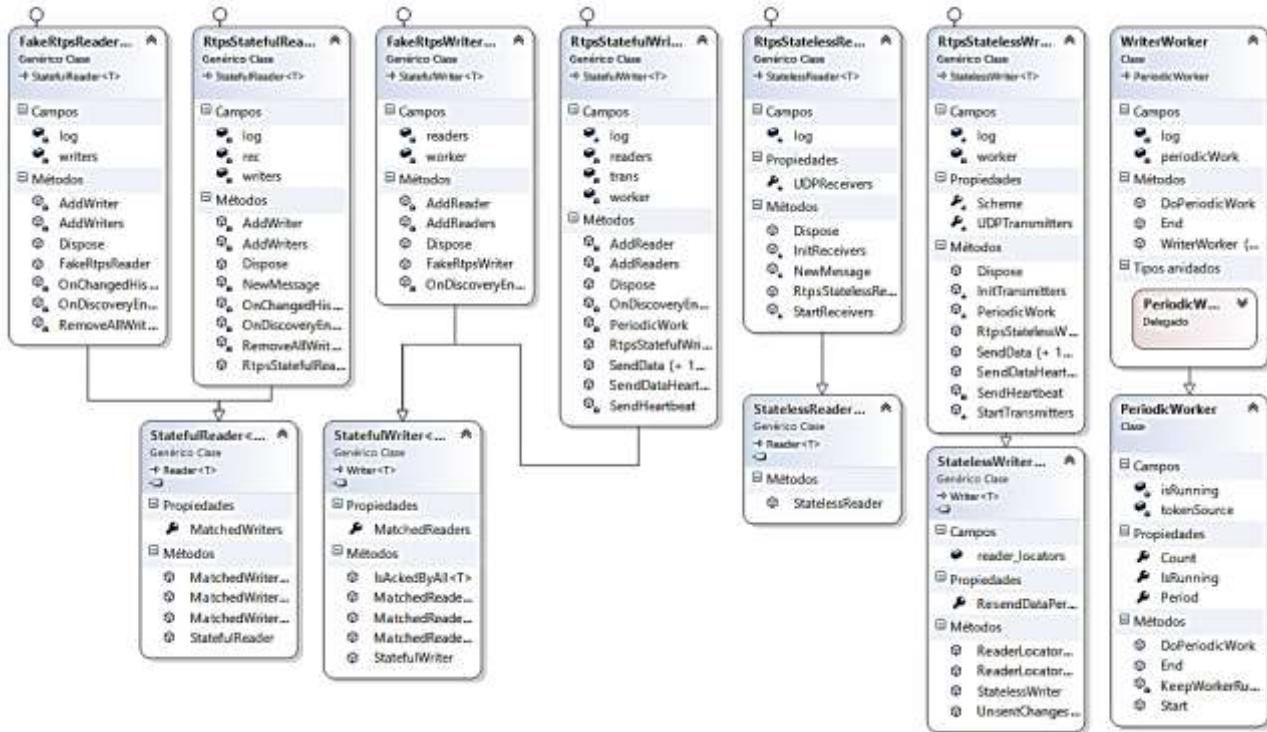


Figura 3-9. Diagrama de clases del submódulo comportamiento.

Tabla 3-4. Métodos de las clases para la comunicación del RTPS con el DDS.

Clases	Métodos	Descripción
FakeRtpsReader	AddWriter	Añade un Writer a las lista de Writer en el lado del Reader.
	AddWriters	Añade al grupo de Writer en los Endpoint por medio de un protocolo de descubrimiento.
	Dispose	Dispone o desregistra todos los Writer de los Endpoint.
	FakeRtpsReader	Representa al Reader RTPS falso.
	OnChangedHistoryCache	Permite añadir, detectar y remover cambios en el HistoryCache.
	OnDiscoveryEndpoints	Añade automáticamente los Writer encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
FakeRtpsWriter	RemoveAllWriters	Limpia la información del HistoryCache de los Writer.
	AddReader	Añade un Reader a las lista de Reader en el lado del Writer.
	AddReaders	Añade al grupo de Reader en los Endpoint por medio de un protocolo de descubrimiento.
	Dispose	Dispone o desregistra todos los Reader de los Endpoint.
	FakeRtpsWriter	Representa al Writer RTPS falso.
RtpsStatefullReader	OnDiscoveryEndpoints	Añade automáticamente los Reader encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
	AddWriter	Añade un Writer a las lista de Writer en el lado del Reader.
	AddWriters	Añade al grupo de Writer en los Endpoint por medio de un protocolo de descubrimiento.
RtpsStatefullWriter	Dispose	Dispone o desregistra todos los Writer de los Endpoint.

Tabla 3-4. Métodos de las clases para la comunicación del RTPS con el DDS.

	NewMessage	Crea los mensajes RTPS.
	OnChangedHistoryCache	Permite añadir, detectar y remover cambios en el HistoryCache.
	OnDiscoveryEndpoints	Añade automáticamente los Writer encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
	RemoveAllWriters	Limpia la información del HistoryCache de los Writer.
	RtpsStatefullReader	Representa al Reader RTPS con estado.
RtpsStatefullWriter	AddReader	Añade un Reader a las lista de Reader en el lado del Writer.
	AddReaders	Añade al grupo de Reader en los Endpoint por medio de un protocolo de descubrimiento.
	Dispose	Dispone o desregistra todos los Reader de los Endpoint.
	OnDiscoveryEndpoints	Añade automáticamente los Reader encontrados dentro de nuevos Endpoint descubiertos por los protocolos de descubrimiento.
	PeriodicWork	Anuncia repetidamente la disponibilidad de datos enviando un mensaje HeartBeat.
	RtpsStatefullWriter	Representa al Writer RTPS con estado.
	SendData	Envía el mensaje RTPS.
RtpsStatelessWriter	SendDataHeartbeat	Crea un mensaje InfoSource y lo envía.
	Dispose	Dispone o desregistra todos los Reader de los Endpoint.
	InitTransmitters	Añade transmisores UDP a una lista de transmisión multicast.
	PeriodicWork	Anuncia repetidamente la disponibilidad de datos enviando un mensaje HeartBeat.
	SendData	Envía el mensaje RTPS.
	SendDataHeartbeat	Crea un mensaje InfoSource y lo envía.
	SendHeartbeat	Crea un mensaje HeartBeat y lo envía.
RtpsStatelessReader	StartTransmitters	Inicia la transmisión UDP.
	RtpsStatelessWriter	Representa al Writer RTPS sin estado.
	Dispose	Dispone o desregistra todos los Writer de los Endpoint.
	InitReceivers	Añade receptores UDP a una lista de transmisión multicast.
	NewMessage	Crea los mensajes RTPS.
WriterWorker	RtpsStatelessReader	Representa al Reader RTPS sin estado.
	StartReceivers	Inicia la recepción UDP.
	DoPeriodicWork	Inicia el PeriodicWork.
	End	Finaliza el PeriodicWork.
	WriterWorker	Publica un delegado de un PeriodicWorker.

3.3.1.4. Submódulo de transporte

En un principio para realizar la codificación de los demás submódulos se diseñó un sistema de transporte local o falso, lo que quiere decir que en realidad se simulaba el transporte de datos, trabajando a nivel local. Esto se realizó tanto para los mensajes de descubrimiento como para los mensajes RTPS. A continuación se presenta en la Figura 3-10 los diagramas de clases del sistema falso de transporte local, los cuales representan a las clases correspondientes *FakeDiscovery* el cual está descrito en la Tabla 3-5, y a los eventos de descubrimiento que interactúan con

DDS; a pesar que estos pertenecen al mismo submódulo de transporte no tienen relación entre sí ya que los eventos de descubrimiento son manejados por el DDS.

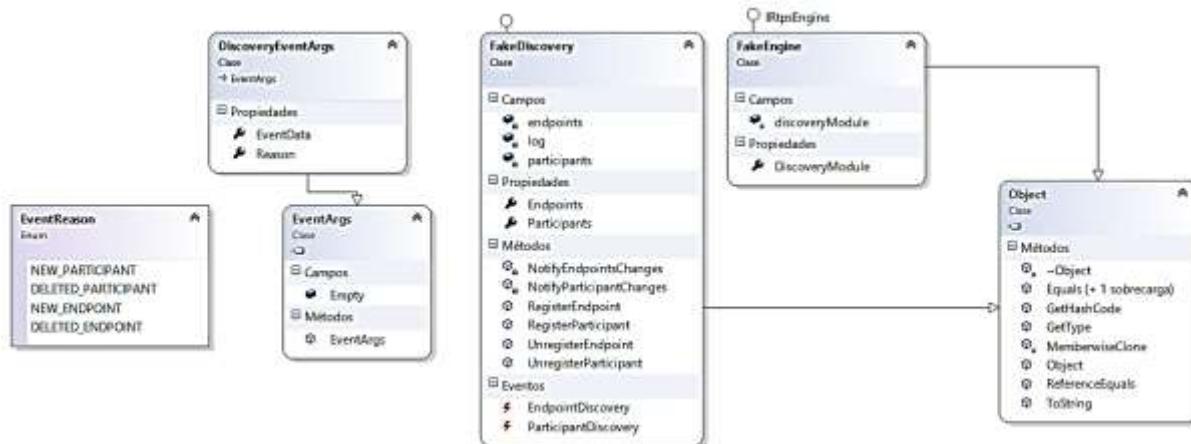


Figura 3-10. Diagrama de clases del sistema falso de transporte.

Tabla 3-5. Métodos de la clase FakeDiscovery.

Clase	Método	Descripción
FakeDiscovery	NotifyEndpointsChanges	Notifica los cambios en los endpoint.
	NotifyParticipantChanges	Notifica los cambios en el participante.
	RegisterEndpoint	Registra los nuevos endpoint.
	RegisterParticipant	Registra los nuevos participantes.
	UnregisterEndpoint	Borra del registro a los endpoint.
	UnregisterParticipant	Borra del registro a los participantes.

Una vez completada la codificación de los demás submódulos se procedió a diseñar el transporte sobre la red en sí. Tomando en cuenta que el protocolo de transporte RTPS trabaja sobre UDP se procedió, en primer lugar a implementar las clases necesarias para enviar y recibir información sobre UDP.

A continuación se presenta en la Figura 3-11 los diagramas de clases del transporte sobre la red, los cuales representan a las clases correspondientes Transmisor y Receptor UDP el cual está descrito en la Tabla 3-6, y a los eventos de entrada y salida de datos; a pesar que estos pertenecen al mismo submódulo de transporte no tienen relación entre sí ya que algunos representan la interfaz al protocolo RTPS

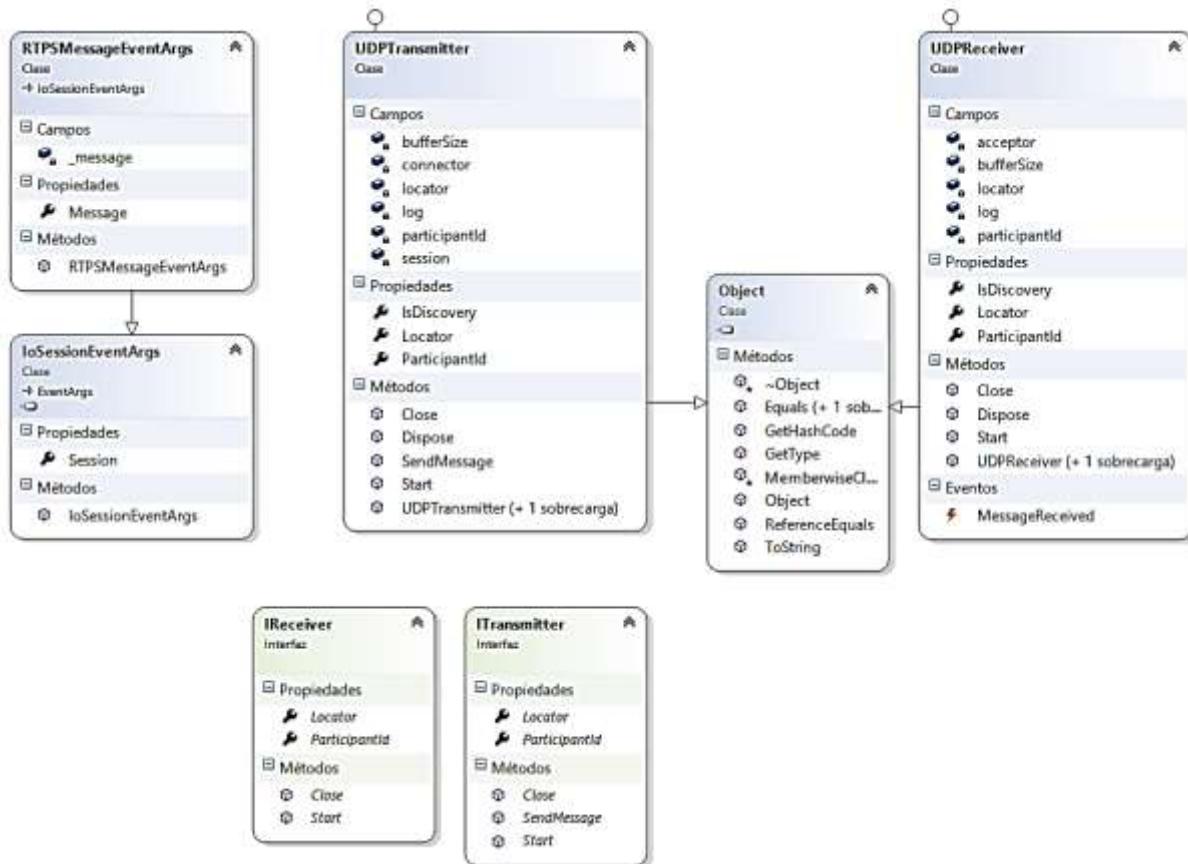


Figura 3-11. Diagrama de clases del sistema de transporte sobre la red.

Tabla 3-6. Métodos de las diferentes clases para la implementación del sistema transporte en red.

Clase	Método	Descripción
UDPTransmitter	Close	Cierra la sesión en el lado del transmisor.
	Dispose	Dispone la sesión en el lado del transmisor.
	SendMessage	Envía un mensaje UDP.
	Start	Inicia la sesión.
	UDPTransmitter	Obtiene la dirección IP desde un DNS para enviar la información en el caso de que no se la tenga.
UDPReceiver	Close	Cierra la sesión en el lado del receptor.
	Dispose	Dispone la sesión en el lado del receptor.
	Start	Acepta una sesión unicast o multicast.
	UDPReceiver	Obtiene direcciones IP desde un DNS antes de recibir información.
ITransmitter	Close	Cierra el transmisor.
	SendMessage	Envía mensajes a su destino.
	Start	Inicia el transmisor.
IReceiver	Close	Cierra el receptor.
	Start	Inicia el receptor.

Finalmente teniendo las clases necesarias para poner mensajes sobre la red se procedió a implementar las clases para enviar mensajes RTPS y mensajes de

descubrimiento RTPS trabajando en conjunto con el envío de mensajes sobre la red.

A continuación se presenta en la Figura 3-12 los diagramas de clases para el envío de mensajes RTPS y mensajes de descubrimiento RTPS, los cuales representan a las clases correspondientes al *EngineFactoryRTPS* y a las interfaces las cuales están descritas en la Tabla 3-7; a pesar que estos pertenecen al mismo submódulo de transporte no tienen relación entre sí ya que algunas clases pertenecen a la interfaz al RTPS.

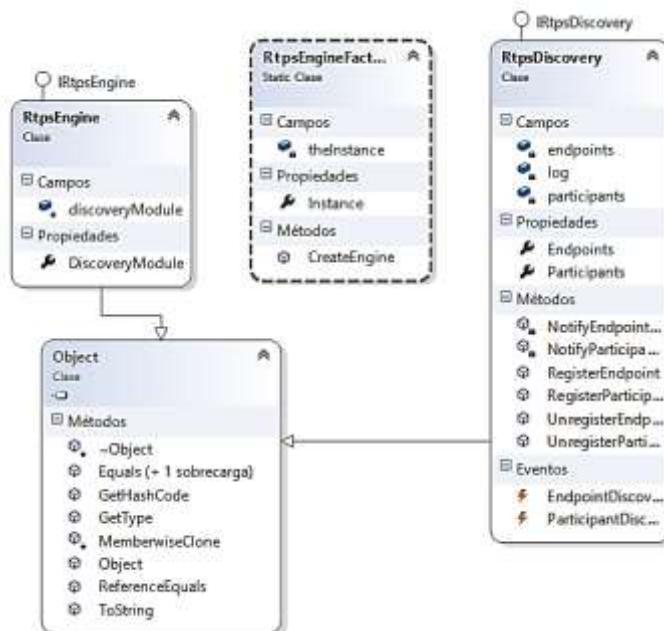


Figura 3-12. Diagrama de clases de los mensajes de descubrimiento y mensajes RTPS.

Tabla 3-7. Métodos de los mensajes de descubrimiento y mensajes RTPS

Clase	Método	Descripción
RtpsDiscovery	NotifyEndpointsChanges	Notifica los cambios en los endpoint.
	NotifyParticipantChanges	Notifica los cambios en el participante.
	RegisterEndpoint	Registra los nuevos endpoint.
	RegisterParticipant	Registra los nuevos participantes.
	UnregisterEndpoint	Borra del registro a los endpoint.
	UnregisterParticipant	Borra del registro a los participantes.
RtpsEngineFactory	CreateEngine	Crea toda la maquinaria necesaria para la comunicación a partir de los parámetros DDS y RTPS encontrados en el archivo de configuración.

3.3.1.5. Submódulo de configuración

El submódulo de configuración es el encargado de interpretar un archivo de configuración, el cual tiene parámetros configurables tanto del módulo DDS como del módulo RTPS. A continuación se muestran en la Figura 3-13 y Figura 3-14 la configuración básica de DDS y RTPS respectivamente.

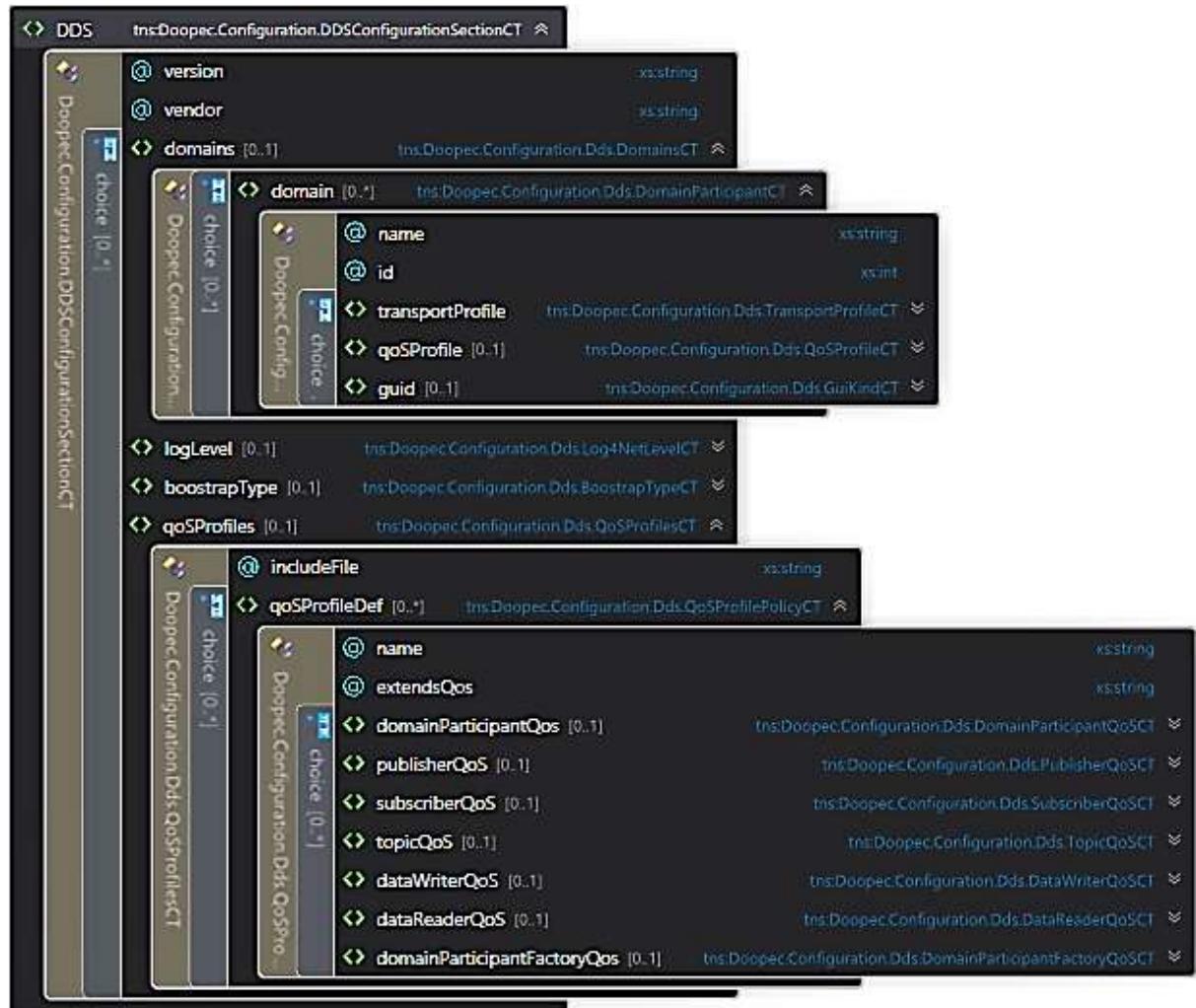


Figura 3-13. Diseño archivo de configuración sección DDS

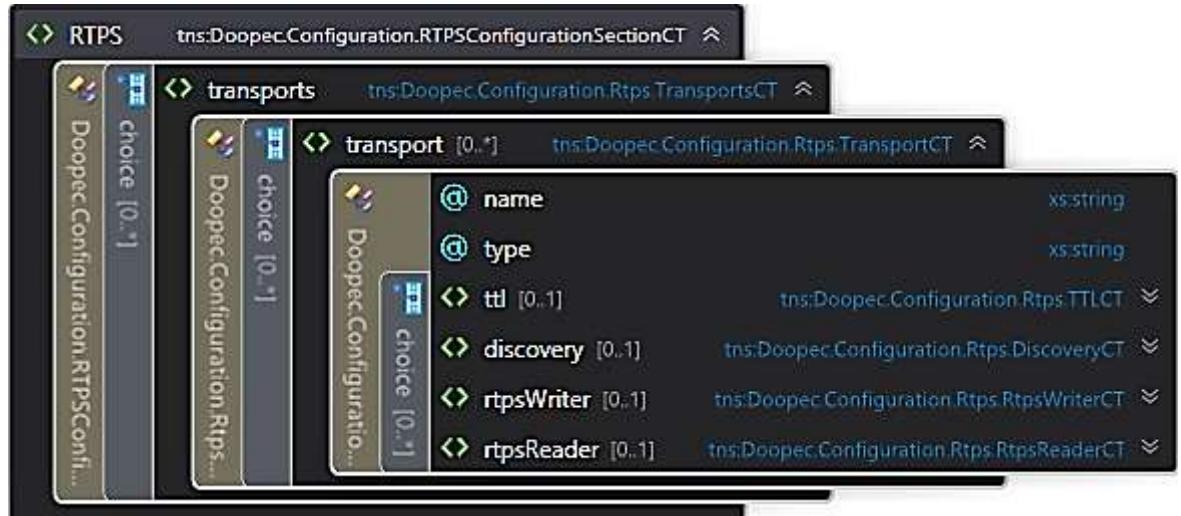


Figura 3-14. Diseño archivo de configuración sección RTPS

3.4. IMPLEMENTACIÓN DEL MÓDULO

En esta sección se explica el desarrollo de todos los submódulos que componen el sistema, tomando en cuenta los diseños presentados en la sección 3.3.

3.4.1. SUBMÓDULO DE TRANSPORTE

A continuación se explica el código más relevante de la implementación de este módulo.

3.4.1.1. Implementación del transmisor UDP

Dentro de la implementación del transmisor UDP, el inicio de la comunicación del lado del transmisor constituye parte vital del proyecto. Se observa que tanto en el transmisor, como en el receptor se utiliza el protocolo que trabaja sobre IP, denominado UDP, el cual solo es útil por su facilidad de uso, ya que RTPS es un protocolo que trabaja en un nivel superior al de la capa transporte.

A continuación se observa parte del código para iniciar la comunicación dentro del transmisor.

Dentro del código mostrado a continuación, se inicia la comunicación. Primeramente se detecta si esta trabaja con direcciones IPv4 o IPv6, y también si se trabaja con tráfico unicast o multicast.

```

public void Start()
{
    IPEndPoint ep = new IPEndPoint(locator.SocketAddress,
locator.Port);
    bool isMultiCastAddr;
    if (ep.AddressFamily == AddressFamily.InterNetwork)
//IP v4
    {
        byte byteIp = ep.Address.GetAddressBytes()[0];
        isMultiCastAddr = (byteIp >= 224 && byteIp <
240) ? true : false;
    }
    else if (ep.AddressFamily ==
AddressFamily.InterNetworkV6)
    {
        isMultiCastAddr = ep.Address.IsIPv6Multicast;
    }
    else
    {
        throw new NotImplementedException("Address family
not supported yet: " + ep.AddressFamily);
    }

    connector = new AsyncDatagramConnector();

    connector.FilterChain.AddLast("RTPS", new
ProtocolCodecFilter(new MessageCodecFactory()));
}

```

Espacio de Código 3-1. Inicio de la comunicación con UDP

En el caso que se trabaje con tráfico multicast, se define un objeto que usa un socket¹⁴, donde finalmente se fuerza a asociar la dirección local IP y obtener una sesión multicast.

```

if (isMultiCastAddr)
{
    // Set the local IP address used by the listener
    // and the sender to
    // exchange multicast messages.
    connector.DefaultLocalEndPoint = new
IPEndPoint(IPAddress.Any, 0);

    // Define a MulticastOption object specifying the
    // multicast group
    // address and the local IP address.
    // The multicast group address is the same as the
    // address used by the listener.
    MulticastOption mcastOption = new
MulticastOption(locator.SocketAddress, IPAddress.Any);
    connector.SessionConfig.MulticastOption =
mcastOption;
}

```

Espacio de Código 3-2. Comunicación Multicast

¹⁴ Socket, Asociación de una dirección IP con un Puerto

Dentro del conector para el caso de UDP se hace referencia al asíncrono, para obtener los posibles eventos de entrada o salida, se utiliza el concepto de futuros dentro de C# donde este, es un sustituto de un cálculo que es inicialmente desconocido, pero vuelve a estar disponible en un momento posterior [42].

```

IoSession session =
connector.Connect(ep).Await().Session;
}
connector.ExceptionCaught += (s, e) =>
{
    log.Error(e.Exception);
};
connector.MessageReceived += (s, e) =>
{
    log.Debug("Session recv...");
};
connector.MessageSent += (s, e) =>
{
    log.Debug("Session sent...");
};
connector.SessionCreated += (s, e) =>
{
    log.Debug("Session created...");
};
connector.SessionOpened += (s, e) =>
{
    log.Debug("Session opened...");
};
connector.SessionClosed += (s, e) =>
{
    log.Debug("Session closed...");
};
connector.SessionIdle += (s, e) =>
{
    log.Debug("Session idle...");
};
IConnectFuture connFuture = connector.Connect(new
IPEndPoint(locator.SocketAddress, locator.Port));
connFuture.Await();

connFuture.Complete += (s, e) =>
{
    IConnectFuture f = (IConnectFuture)e.Future;
    if (f.Connected)
    {
        log.Debug("...connected");
        session = f.Session;
    }
    else
    {
        log.Warn("Not connected...exiting");
    }
};
}
}

```

Espacio de Código 3-3. Conexión mediante futuros.

Como se puede observar en el código final, existen muchas expresiones lambda, las cuales se definen como funciones anónimas [43], las cuales son muy útiles a la hora de utilizar futuros.

3.4.1.2. Implementación del receptor UDP

El código mostrado a continuación hace referencia a la definición de un objeto UDP *Receiver*, en el cual se puede hacer hincapié en que se utiliza un *buffer* para almacenar los mensajes recibidos. También se puede encontrar dentro del código del programa al transmisor.

```
public UDPReceiver(Uri uri, int bufferSize)
{
    this.bufferSize = bufferSize;
    var addresses =
        System.Net.Dns.GetHostAddresses(uri.Host);
    int port = (uri.Port < 0 ? 0 : uri.Port);
    if (addresses != null && addresses.Length >= 1)
        this.locator = new Locator(addresses[0], port);
}
```

Espacio de Código 3-4. Receiver Buffer.

3.4.1.3. Implementación de la maquinaria RTPS

La maquinaria RTPS, se refiere a toda la configuración de perfiles necesarios para trabajar con RTPS, para lo cual primeramente se obtiene del archivo de configuración las instancias, es decir los parámetros del funcionamiento del protocolo, donde se encuentran tiempos, retardos y QoS.

```
public static IRtpsEngine CreateEngine(IDictionary<string,
Object> environment)
{
    DDSConfigurationSection ddsConfig =
        Doopec.Configuration.DDSConfigurationSection.Instance;
    RTPSConfigurationSection rtpsConfig =
        Doopec.Configuration.RTPSConfigurationSection.Instance;
    string transportProfile =
        ddsConfig.Domains[0].TransportProfile.Name;
    string className =
        rtpsConfig.Transports[transportProfile].Type;
    if (string.IsNullOrWhiteSpace(className))
    {
        // no implementation class name specified
        throw new ApplicationException("Please Set the
RTPS engine type property in the settings.");
    }

    Type ctxClass = Type.GetType(className, true);
```

Espacio de Código 3-5. Implementación maquinaria.

Finalmente luego de configurar el RTPS, se llama a la instancia creada por medio de una interfaz.

```
// --- Instantiate new object --- //
try
{
    // First, try a constructor that will accept the
environment.
    object newInstance =
Activator.CreateInstance(ctxClass, environment);
    if (newInstance != null)
        return (IRtpsEngine)newInstance;
```

Espacio de Código 3-6. Interfaz RTPS Engine.

3.4.2. SUBMÓDULO DE MENSAJE Y ENCAPSULAMIENTO

3.4.2.1. Implementación de los mensajes RTPS

Para poner mensajes en la red se utiliza un método que permite alinear el mensaje a los 32 bits como define la norma, luego se define el *Endianess* y finalmente el tipo de submensaje a enviar. En el Espacio de Código 3-7 se muestra como el submensaje es alineado a los 32 bits con respecto al inicio del mensaje por medio del manejo del *buffer*.

```
public static int PutSubMessage(this IoBuffer buffer, SubMessage
msg)
{
    // The PSM aligns each Submessage on a 32-bit
boundary with respect to the start of the Message (page 159).
    buffer.Align(4);
    buffer.Order = (msg.Header.IsLittleEndian ?
ByteOrder.LittleEndian : ByteOrder.BigEndian); // Set the
endianess
    buffer.PutSubMessageHeader(msg.Header);
    int position = buffer.Position;
    switch (msg.Kind)
    {
        case SubMessageKind.PAD:
            buffer.PutPad((Pad)msg);
            break;
        case SubMessageKind.ACKNACK:
            buffer.PutAckNack((AckNack)msg);
            break;
```

Espacio de Código 3-7. Implementación mensajes RTPS.

3.4.2.2. Implementación de la Encapsulación

Para la encapsulación de los mensajes, se obtiene el mensaje luego de alineararlo y se procede a almacenarlo en el buffer de salida como se muestra en el Espacio de Código 3-8.

```

public void Encode(IoSession session, object message,
IProtocolEncoderOutput output)
{
    Message msg = (Message)message;
    IoBuffer buffer = IoBuffer.Allocate(1024);
    buffer.AutoExpand = true;
    buffer.PutMessage(msg);
    buffer.Flip();
    output.Write(buffer);
}

```

Espacio de Código 3-8. Implementación de la Encapsulación.

Ya en el buffer de salida, dependiendo de su *Endianess* se serializa la información como se muestra en el Espacio de Código 3-9

```

public static void Serialize(IoBuffer buffer, object dataObj,
ByteOrder order)
{
    buffer.Order = order;
    if (order == ByteOrder.LittleEndian)
        buffer.PutEncapsulationScheme(CDR_LE_HEADER);
    else
        buffer.PutEncapsulationScheme(CDR_BE_HEADER);
    Doopec.Serializer.Serializer.Serialize(buffer,
dataObj);
}

```

Espacio de Código 3-9. Implementación Serializador.

3.4.3. SUBMÓDULO DE DESCUBRIMIENTO

3.4.3.1. Implementación del descubrimiento RTPS

Para poder realizar el descubrimiento es necesario primeramente añadir a los participantes, estos tienen asociada una QoS, y se envían dentro de algún protocolo de descubrimiento. Estos participantes deben tener una identificación que es autogenerada en el programa. En el Espacio de Código 3-10 se muestra como se añade un participante al dominio y como se autogenera una identificación al participante.

```

internal virtual AddDomainStatus AddDomainParticipant(int domain,
DomainParticipantQos qos)
{
    lock (this)
    {
        AddDomainStatus ads = new AddDomainStatus() { id
= new GUID(), federated = false };
        generator.Populate(ref ads.id);
        ads.id.EntityId = EntityId.ENTITYID_PARTICIPANT;
        try
        {
            if (participants_.ContainsKey(domain) &&
participants_[domain] != null)
            {
                participants_[domain][ads.id] = new
Spdp(domain, ads.id, qos, this);
            }
            else
            {
                participants_[domain] = new
Dictionary<GUID, Spdp>();
                participants_[domain][ads.id] = new
Spdp(domain, ads.id, qos, this);
            }
        }
    }
}

```

Espacio de Código 3-10. Implementación del Descubrimiento RTPS.

3.4.3.2. Implementación de los protocolos SPDP y SEDP

Tanto para los lectores como para los escritores y sus *topics*, es necesario tener la correspondiente clase que implemente el protocolo SEDP en base a un identificador, tal como se muestra en el Espacio de Código 3-11.

```

namespace Doopec.Rtps.Discovery
{
    public class SEDPbuiltinPublicationsWriter :
StatefulWriter<DiscoveredWriterData>
    {
        public SEDPbuiltinPublicationsWriter(GUID guid)
            : base(guid)
        {
            this.guid = new GUID(guid.Prefix,
EntityId.ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER);
        }

    }
}

```

Espacio de Código 3-11. Implementación SEDP.

Así como para el protocolo SEDP, es necesario tener la correspondiente clase que implemente el protocolo SPDP en base a una configuración de transporte predefinida, la cual configura varios parámetros de configuración de calidad de

servicio. En el Espacio de Código 3-12 se implementa el descubrimiento de un *Writer* mediante el protocolo SPDP.

```

public SPDPbuiltinParticipantWriterImpl(Transport
transportconfig, ParticipantImpl participant)
: base(participant.Guid)
{
    SetLocatorListFromConfig(transportconfig,
participant);
    participant.DefaultMulticastLocatorList =
this.MulticastLocatorList as List<Locator>;
    participant.DefaultUnicastLocatorList =
this.UnicastLocatorList as List<Locator>;
    SPDPdiscoveredParticipantData data = new
SPDPdiscoveredParticipantData(participant);
    // TODO Assign UserData from configuration
    CacheChange<SPDPdiscoveredParticipantData> change =
this.NewChange(ChangeKind.ALIVE, new Data(data), null);
    this.HistoryCache.AddChange(change);

    this.TopicKind = TopicKind.WITH_KEY;
    this.ReliabilityLevel = ReliabilityKind.BEST_EFFORT;
    this.ResendDataPeriod = new
Duration(transportconfig.Discovery.ResendPeriod.Val);
    this.heartbeatPeriod = new
Duration(transportconfig.Rtpswriter.HeartbeatPeriod.Val);

        //The following timing-related values are used as
the defaults in order to facilitate
        // ‘out-of-the-box’ interoperability between
implementations.
    this.nackResponseDelay = new
Duration(transportconfig.Rtpswriter.NackResponseDelay.Val);
//200 milliseconds
    this.nackSuppressionDuration = new
Duration(transportconfig.Rtpswriter.NackSuppressionDuration.Val)
;
    this.pushMode =
transportconfig.Rtpswriter.PushMode.Val;

    InitTransmitters();
    foreach (var trans in this.UDPTransmitters)
    {
        trans.IsDiscovery = true;
    }
    this.Scheme = Encapsulation.PL_CDR_BE;

    worker = new WriterWorker(this.PeriodicWork);
}

```

Espacio de Código 3-12. Implementación SPDP.

3.4.4. SUBMÓDULO DE COMPORTAMIENTO

Lo que se observa a continuación muestra lo esencial del código de cada *Reader* y *Writer*.

3.4.4.1. Implementación del *Reader RTPS* con estado

Para la implementación de un lector RTPS con estado, el Espacio de Código 3-13 demuestra la creación de un mensaje, la cual permite escoger el tipo de mensaje, después de almacenar el mensaje en un *buffer*, seguidamente se lo encapsula y serializa. Luego del proceso de creación del mensaje, el *Reader* se encarga de informar al *HistoryCache* que se ha generado un cambio por medio del objeto *CacheChange*.

```
private void NewMessage(object sender, RTPSMessageEventArgs e)
{
    Message msg = e.Message;
    log.DebugFormat("New Message has arrived from {0}",
e.Session.RemoteEndPoint);
    log.DebugFormat("Message Header: {0}", msg.Header);
    foreach (var submsg in msg.SubMessages)
    {
        switch (submsg.Kind)
        {
            case SubMessageKind.DATA:
                Data d = submsg as Data;

                IoBuffer buf =
IoBuffer.Wrap(d.SerializedPayload.DataEncapsulation.SerializedPay
load);
                    buf.Order = ByteOrder.LittleEndian;
//(d.Header.IsLittleEndian ? ByteOrder.LittleEndian :
ByteOrder.BigEndian);
                    object obj =
Doopec.Serializer.Serializer.Deserialize<T>(buf);
                    CacheChange<T> change = new
CacheChange<T>(ChangeKind.ALIVE, new G UID(msg.Header.GuidPrefix,
d.WriterId), d.WriterSN, new DataObj(obj), new InstanceHandle());
                    ReaderCache.AddChange(change);
                break;
        }
    }
}
```

Espacio de Código 3-13. Implementación Reader RTPS con estado.

3.4.4.2. Implementación del *Reader RTPS* sin estado

Una parte esencial de un *Reader* sin estado, es configurar a los receptores, inicializándolos como se muestra en el Espacio de Código 3-14. Esto se realiza de igual manera tanto para tráfico *Unicast* como para tráfico *Multicast*.

```

protected void InitReceivers()
{
    foreach (var locator in MulticastLocatorList)
    {
        UDPReceiver rec = new UDPReceiver(locator, 1024);
        rec.ParticipantId = this.Guid;
        rec.MessageReceived += NewMessage;
        UDPReceivers.Add(rec);
    }

    foreach (var locator in UnicastLocatorList)
    {
        UDPReceiver rec = new UDPReceiver(locator, 1024);
        rec.ParticipantId = this.Guid;
        rec.MessageReceived += NewMessage;
        UDPReceivers.Add(rec);
    }
}

```

Espacio de Código 3-14. Implementación del Reader RTPS sin estado.

3.4.4.3. Implementación del *Writer* RTPS con estado

Tanto para los *Writer* con estado y sin estado, es importante la implementación del método *PeriodicWork*, como se muestra en el Espacio de Código 3-15, el cual se encarga de anunciar repetitivamente la disponibilidad de datos por medio del envío de submensajes *Heartbeat*. Con la diferencia que en los escritores sin estado no se guarda mas que un cambio.

```

private void PeriodicWork()
{
    // the RTPS Writer to repeatedly announce the
    availability of data by sending a Heartbeat Message.
    log.DebugFormat("I have to send a Heartbeat Message,
at {0}", DateTime.Now);
    SendHeartbeat();
    if (HistoryCache.Changes.Count > 0)
    {
        foreach (var change in HistoryCache.Changes)
        {
            //SendHeartbeat();
            //SendData(change);
            SendDataHeartbeat(change);
        }
        HistoryCache.Changes.Clear(); //TODO
    }
}

```

Espacio de Código 3-15. Implementación del *Writer* RTPS con estado.

3.4.4.4. Implementación del Writer RTPS sin estado

Para los *Writer* sin estado aparte de la implementación del método *PeriodicWork*, se implementa un método que permite el envío manual de submensajes *Heartbeat*, como se muestra en el Espacio de Código 3-16.

```
private void SendHeartbeat()
{
    // Create a Message with Heartbeat
    Message m1 = new Message();

    Heartbeat heartbeat = new Heartbeat();
    EntityId id1 = EntityId.ENTITYID_UNKNOWN;
    EntityId id2 = EntityId.ENTITYID_PARTICIPANT;

    heartbeat.readerId = id1;
    heartbeat.writerId = id2;
    heartbeat.firstSN = new SequenceNumber(10);
    heartbeat.lastSN = new SequenceNumber(20);
    heartbeat.count = 5;
    m1.SubMessages.Add(heartbeat);

    SendData(m1);
}
```

Espacio de Código 3-16. Implementación del Writer RTPS sin estado.

3.4.4.5. Implementación del Publicador y el Writer DDS

3.4.4.5.1. Writer

En el Espacio de Código 3-17, se muestra el constructor de un *DataWriter* DDS, y en su inicialización se especifica si el *Writer* RTPS es con estado o sin estado.

```
public DataWriterImpl(Publisher pub, Topic<TYPE> topic,
DataWriterQos qos, DataWriterListener<TYPE> listener,
ICollection<Type> statuses)
{
    this.pub_ = pub;
    this.topic_ = topic;
    this.listener = listener;

    this.rtpsWriter = new
RtpsStatefulWriter<TYPE>((pub.GetParent() as
DomainParticipantImpl).ParticipantGuid);
}
```

Espacio de Código 3-17. Implementación del Writer DDS.

3.4.4.5.2. *Publicador*

A continuación en el Espacio de Código 3-18 se muestra la creación de los *DataWriter DDS* que son agregados a una lista de *DataWriter* y la lista es retornada al publicador.

```
public DataWriter<TYPE> CreateDataWriter<TYPE>(Topic<TYPE> topic)
{
    DataWriter<TYPE> dw = null;
    dw = new DataWriterImpl<TYPE>(this, topic);
    datawriters.Add(dw);
    return dw;
}

public DataWriter<TYPE>
CreateDataWriter<TYPE>(Topic<TYPE> topic, DataWriterQos qos,
DataWriterListener<TYPE> listener, ICollection<Type> statuses)
{
    DataWriter<TYPE> dw = null;
    dw = new DataWriterImpl<TYPE>(this, topic, qos,
listener, statuses);
    datawriters.Add(dw);
    return dw;
}
```

Espacio de Código 3-18. Implementación del Publicador.

3.4.4.6. **Implementación del Suscriptor y del Reader DDS**

3.4.4.6.1. *Reader*

En el Espacio de Código 3-19, se muestra el constructor de un *DataReader DDS*, y en su inicialización se especifica si el *Reader RTPS* es *con estado* o sin estado.

```
public DataReaderImpl(Subscriber sub, TopicDescription<TYPE>
topic, DataReaderQos qos, DataReaderListener<TYPE> listener,
ICollection<Type> statuses)
{
    this.sub_ = sub;
    this.topic_ = topic;
    this.listener = listener;

    RtpsStatefulReader<TYPE> reader = new
RtpsStatefulReader<TYPE>((sub.GetParent() as
DomainParticipantImpl).ParticipantGuid);
    reader.ReaderCache.Changed += NewMessage;
    this.rtpsReader = reader;
}
```

Espacio de Código 3-19. Implementación del Reader DDS.

3.4.4.6.2. Suscriptor

En el Espacio de Código 3-20, se muestra la creación de los *DataReader* DDS que son agregados a una lista de *DataReader*, y la lista es retornada al suscriptor.

```
public DataReader<TYPE>
CreateDataReader<TYPE>(TopicDescription<TYPE> topic)
{
    DataReader<TYPE> dw = null;
    dw = new DataReaderImpl<TYPE>(this, topic);
    datawriters.Add(dw);
    return dw;
}

public DataReader<TYPE>
CreateDataReader<TYPE>(TopicDescription<TYPE> topic,
DataReaderQos qos, DataReaderListener<TYPE> listener,
ICollection<Type> statuses)
{
    DataReader<TYPE> dw = null;
    dw = new DataReaderImpl<TYPE>(this, topic, qos,
listener, statuses);
    datawriters.Add(dw);
    return dw;
}
```

Espacio de Código 3-20. Implementación de Suscriptor.

3.4.5. SUBMÓDULO CONFIGURACIÓN

3.4.5.1. Sección DDS

3.4.5.1.1. Etiqueta DDS

```
<DDS xmlns="urn:Configuration" vendor="Doopec" version="2.1">
```

Espacio de Código 3-21. Etiqueta DDS.

Como se observa en el Espacio de Código 3-21 la etiqueta DDS, necesita tener establecido el *namespace xml* o *xmlns*. Además se define el fabricante dentro del *vendor* y finalmente se define la versión con la que se trabaja.

3.4.5.1.1.1. Etiqueta Bootstrap

Como se observa en el Espacio de Código 3-22, la etiqueta *BootstrapType*, debe tener configurada los atributos *name* y *type*.

```
<bootstrapType name="default"
type="Doopec.Dds.Core.BootstrapImpl, Doopec"/>
```

Espacio de Código 3-22. Etiqueta Bootstrap.

3.4.5.1.1.2. Etiqueta Domains

Como se observa en el Espacio de Código 3-23 la etiqueta Domains, anuncia a los participantes presentes, y con cuales se podrá interactuar por medio del protocolo RTPS, en este caso se puede observar a 3 *domain* configurados, cada uno de estos tiene dentro de su etiqueta *domain*, el atributo *name*, es decir un nombre indistinto, y un atributo *id*, para la identificación única del participante

Etiqueta Transport Profile

Esta etiqueta define el perfil de transporte de información, con el cual trabaja el DDS, para nuestro caso será *defaultRTPS*, es decir que se utiliza el protocolo RTPS como transporte.

Etiqueta QoS Profile

Esta etiqueta definirá el perfil de calidad de servicio, con el cual trabaja DDS, para nuestro caso será *defaultQoS*, es decir que existirá un perfil de calidad de servicio por defecto.

Etiqueta Guid

Para la etiqueta Guid, se tienen 5 posibles valores a tomar:

- *Fixed*
- *Random*
- *Autold*
- *AutoldFromIp*
- *AutoldFromMac*

```

<domains>
    <domain name="Servidor" id="0">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
    </domain>
    <domain name="Servidor" id="3">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
    </domain>
    <domain name="Cliente1" id="1">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
    </domain>
    <domain name="Cliente2" id="2">
        <transportProfile name="defaultRtps"/>
        <qoSProfile name="defaultQoS"/>
        <guid kind="Fixed" val="7F294ABE-33F2-40B9-BFF5-
7D9376EA061C"/>
    </domain>
</domains>

```

Espacio de Código 3-23. Etiqueta Domains.

3.4.5.1.1.3. Etiqueta LogLevel

Como se observa en el Espacio de Código 3-24 la etiqueta *LogLevel*, define un nivel mínimo y un máximo de *logs* con los siguientes posibles valores:

- *DEBUG*
- *ALL*
- *WARN*
- *INFO*
- *ERROR*
- *FATAL*
- *OFF*

```

<logLevel levelMin="DEBUG" levelMax="FATAL"/>

```

Espacio de Código 3-24. Etiqueta LogLevel.

3.4.5.1.1.4. Etiqueta QoS Profiles

QoS Profiles, está compuesto, de un *QosProfileDef*, el cual tiene un *domainParticipantFactory*, un *domainParticipantQoS*, un *topicQoS*, un *publisherQoS*, un *subscriberQoS*, un *dataWriterQoS*, y un *dataReaderQoS*.

Etiqueta domainParticipantFactoryQoS

En el Espacio de Código 3-25, además de asignar un nombre a esta etiqueta, tiene una etiqueta interna *autoenableCreatedEntities*, la cual define si se auto habilita a los participantes creados.

```
<qoSProfiles>
    <qoSProfileDef name="defaultQoS">
        <domainParticipantFactoryQos
name="defaultDomainParticipantFactoryQoS">
            <entityFactory autoenableCreatedEntities="true"/>
        </domainParticipantFactoryQos>
```

Espacio de Código 3-25. Etiqueta domainParticipantFactoryQoS.

Etiqueta domainParticipantQos

En el Espacio de Código 3-26, además de asignar un nombre a esta etiqueta, se tiene una etiqueta interna *autoenableCreatedEntity*, la cual define si se auto habilita a los participantes creados, y tiene una etiqueta *userData*, en la cual se puede poner valores de acuerdo a la política *userDataQoS*.

```
<domainParticipantQos name="defaultDomainParticipantQoS">
    <entityFactory autoenableCreatedEntities="true"/>
    <userData value="" />
</domainParticipantQos>
```

Espacio de Código 3-26. Etiqueta domainParticipantQoS.

Etiqueta topicQoS

En el Espacio de Código 3-27, además de asignar un nombre a esta etiqueta, se tiene una etiqueta interna llamada *topicData*, a la cual se le puede asignar un valor, aparte se puede agregar varias políticas de QoS, como en este caso *deadline* y *durability*.

```
<topicQoS name="defaultTopicQoS">
    <topicData value="" />
    <deadline period="100" />
    <durability kind="VOLATILE" />
</topicQoS>
```

Espacio de Código 3-27. Etiqueta topicQoS.

Etiqueta publisherQos

En el Espacio de Código 3-28, además de asignar un nombre a esta etiqueta, se tiene una etiqueta interna llamada *entityFactory*, a la cual se le puede asignar un valor, aparte se puede agregar varias políticas de QoS, como en este caso *groupData*, *partition*, y *presentation*.

```
<publisherQoS name="defaultPublisherQoS">
    <entityFactory autoenableCreatedEntities="true"/>
    <groupData value="" />
    <partition value="" />
    <presentation accessScope="INSTANCE"
coherentAccess="true" orderedAccess="true"/>
</publisherQoS>
```

Espacio de Código 3-28. Etiqueta publisherQoS.

Etiqueta subscriberQos

En el Espacio de Código 3-29, además de asignar un nombre a esta etiqueta, se tiene una etiqueta interna llamada *entityFactory*, a la cual se le puede asignar un valor, aparte se puede agregar varias políticas de QoS, como en este caso *groupData*, *partition*, y *presentation*.

```
<subscriberQoS name="defaultSubscriberQoS">
    <entityFactory autoenableCreatedEntities="true"/>
    <groupData value="" />
    <partition value="" />
    <presentation accessScope="INSTANCE"
coherentAccess="true" orderedAccess="true"/>
</subscriberQoS>
```

Espacio de Código 3-29. Etiqueta subscriberQoS.

Etiqueta dataWriterQos

En el Espacio de Código 3-30, además de asignar un nombre a esta etiqueta, se puede agregar todas políticas de QoS utilizadas en un escritor.

```

<dataWriterQoS name="defaultDataWriterQoS">
    <deadline period="1"/>
    <destinationOrder kind="BY_SOURCE_TIMESTAMP"/>
    <durability kind="VOLATILE"/>
    <durabilityService historyDepth="0"
historyKind="KEEP_LAST" maxInstances="1" maxSamples="1"
maxSamplesPerInstance="1" serviceCleanupDelay="100"/>
    <history kind="KEEP_LAST" depth="1"/>
    <latencyBudget duration="100"/>
    <lifespan duration="100"/>
    <liveliness kind="AUTOMATIC" leaseDuration="100"/>
    <ownership kind="SHARED"/>
    <ownershipStrength value="100"/>
    <reliability kind="BEST EFFORT"
maxBlockingTime="1000"/>
    <resourceLimits maxInstances="1" maxSamples="1"
maxSamplesPerInstance="1"/>
    <transportPriority value="1"/>
    <userData value="" />
    <writerDataLifecycle
autodisposeUnregisteredInstances="true"/>
</dataWriterQoS>

```

Espacio de Código 3-30. Etiqueta dataWriterQoS.

Etiqueta dataReaderQoS

En el Espacio de Código 3-31, además de asignar un nombre a esta etiqueta, se puede agregar todas políticas de QoS utilizadas en un lector.

```

<dataReaderQoS name="defaultDataReaderQoS">
    <deadline period="1"/>
    <destinationOrder kind="BY_SOURCE_TIMESTAMP"/>
    <durability kind="VOLATILE"/>
    <history kind="KEEP_LAST" depth="1"/>
    <latencyBudget duration="100"/>
    <liveliness kind="AUTOMATIC" leaseDuration="100"/>
    <ownership kind="SHARED"/>
    <reliability kind="BEST EFFORT"
maxBlockingTime="1000"/>
    <resourceLimits maxInstances="1" maxSamples="1"
maxSamplesPerInstance="1"/>
    <readerDataLifecycle
autopurgeDisposedSamplesDelay="1000"
autopurgeNowriterSamplesDelay="1000"/>
    <timeBasedFilter minimumSeparation="1000"/>
    <userData value="" />
</dataReaderQoS>
</qoSProfileDef>
</qoSProfiles>
</DDS>

```

Espacio de Código 3-31. Etiqueta dataReaderQoS.

3.4.5.2. Sección RTPS

3.4.5.2.1. Etiqueta RTPS

```
<RTPS xmlns="urn:Configuration">
```

Espacio de Código 3-32. Etiqueta RTPS.

Como se observa en el Espacio de Código 3-32 la etiqueta RTPS, necesita tener establecido el *namespace xml* o *xmlns*.

3.4.5.2.1.1. Etiqueta Transports

La etiqueta *transport* trae los atributos *name* y *type*, el segundo especifica el Motor RTPS, y además hay una etiqueta *ttl*, la cual cumple las funciones de *time to live*.

```
<transports>
    <transport name="defaultRtps"
type="Doopec.Rtps.RtpsTransport.RtpsEngine, Doopec">
        <ttl val="1"/>
```

Espacio de Código 3-33. Etiqueta transport.

Etiqueta Discovery

En el Espacio de Código 3-34 la etiqueta Discovery introduce la configuración de los paquetes de descubrimiento, aquí se puede configurar el periodo de reenvío, además se define si se usan paquetes multicast del tipo SEDP, el puerto base con el que se trabaja, el domainGain y el participantGain; también se configura *offsets* tanto para tráfico unicast y multicast, y se define una lista de IP con las que se trabaja tanto en modo multicast y unicast.

```
<discovery name="defaultDiscovery">
    <resendPeriod val="30000"/>
    <useSedpMulticast val="true"/>
    <portBase val="7400"/>
    <domainGain val="250"/>
    <participantGain val="2"/>
    <offsetMetatrafficMulticast val="0"/>
    <offsetMetatrafficUnicast val="10"/>
    <metatrafficUnicastLocatorList val="localhost"/>
    <metatrafficMulticastLocatorList val="239.255.0.1"/>
</discovery>
```

Espacio de Código 3-34. Etiqueta Discovery.

Etiqueta rtpsWriter

En el Espacio de Código 3-35 la etiqueta *rtpsWriter* introduce la configuración de los *writer* RTPS y su comportamiento, aquí se puede configurar el periodo de envío de los submensajes *Heartbeat*, se define los retardos de las respuestas por medio de los submensajes *Nack*, también el tiempo para que se suprima una respuesta *Nack*, y si estamos trabajando con modo *push*.

```
<rtpsWriter>
    <heartbeatPeriod val="1000"/>
    <nackResponseDelay val="200"/>
    <nackSuppressionDuration val="0"/>
    <pushMode val="true"/>
</rtpsWriter>
```

Espacio de Código 3-35. Etiqueta rtpsWriter.

Etiqueta rtpsReader

En el Espacio de Código 3-36 la etiqueta *rtpsReader* introduce la configuración de los *Reader* RTPS y su comportamiento, se define los retardos de las respuestas por medio de los submensajes *Heartbeat*, también el tiempo para que se suprima una respuesta *Heartbeat*.

```
<rtpsReader>
    <heartbeatResponseDelay val="500"/>
    <heartbeatSuppressionDuration val="0"/>
</rtpsReader>
</transport>
</transports>
</RTPS>
<appSettings>
    <add key="org.omg.dds.serviceClassName"
value="Doopec.Dds.Core.BootstrapImpl, Doopec" />
</appSettings>
<startup>
    <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.5.1" />
</startup>
</configuration>
```

Espacio de Código 3-36. Etiqueta rtpsReader.

3.5. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS

A continuación se muestra la descripción de cada diagrama mostrado en la sección 3.6

3.5.1. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS CON ESTADO

3.5.1.1. Interacción en base a la QoS Best Effort

3.5.1.1.1. Best Effort Reader – Best Effort Writer

La siguiente descripción corresponde a la Figura 3-20.

- 1) El usuario DDS escribe datos por medio de la llamada a la operación *write* en el *DataWriter* DDS.
- 2) El *DataWriter* DDS llama a la operación *new_change* en el *Writer* RTPS para crear un nuevo *CacheChange*. Cada uno de estos cambios es identificado únicamente por un *SequenceNumber*.
- 3) La operación *new_change* termina.
- 4) El *DataWriter* DDS utiliza la operación *add_change* para almacenar el *CacheChange* dentro de *HistoryCache* del *Writer* RTPS.
- 5) El *HistoryCache* del *Writer* RTPS notifica el cambio por medio de la operación *notify_change* al *Publisher* DDS.
- 6) La operación *notify_change* termina.
- 7) La operación *add_change* termina.
- 8) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
- 9) El *HistoryCache* del *Writer* DDS utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
- 10) La operación *unsent_changes* termina.
- 11) El *HistoryCache* del *Writer* DDS utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
- 12) La operación *can_send* termina.
- 13) El *DataWriter* DDS utiliza la operación *remove_change* en el *HistoryCache* del *Writer* DDS para limpiar la cache. Esta operación puede ser realizada posteriormente.
- 14) La operación *remove_change* termina.

- 15) El *ReaderProxy* serializa la información mediante la operación *serialize* en el *Serializer*.
- 16) La operación *serialize* termina.
- 17) El *ReaderProxy* envía el submensaje DATA al *MessageEncoder* para que sea encapsulado.
- 18) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 19) La operación *encoded_message* termina.
- 20) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 21) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 22) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 23) La operación *doDecode* termina.
- 24) El *MessageDecoder* envía el submensaje DATA al *WriterProxy*.
- 25) El *WriterProxy* llama a la operación *deserialize_data* al *Deserializer*
- 26) La operación *deserialize_data* termina.
- 27) El *WriterProxy* llama a la operación *available_change* dentro del *HistoryCache* del *Reader RTPS*, para la verificación de números de secuencia recibidos.
- 28) La operación *available_change* termina.
- 29) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader RTPS*.
- 30) La operación *new_change* termina.
- 31) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación *add_change*.
- 32) El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 33) La operación *notify_change* termina.
- 34) La operación *add_change* termina.
- 35) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.

- 36) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
- 37) La operación *get_change* termina.
- 38) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 39) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
- 40) La operación *remove_change* termina.

3.5.1.1.2. Best Effort Reader – BestEffort Writer (Packet Failure)

La siguiente descripción corresponde a la Figura 3-21.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS
- 2) El DataWriter DDS crea un CacheChange mediante la operación *new_change* al Stateful Writer.
- 3) La operación *new_change* termina.
- 4) El DataWriter DDS añade el cambio mediante la operación *add_change* al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación *notify_change*.
- 6) La operación *notify_change* termina.
- 7) La operación *add_change* termina.
- 8) La operación *write* termina.
- 9) El HistoryCache del Writer RTPS envía los cambios no enviados mediante la operación *unsent_changes* al ReaderProxy.
- 10) La operación *unsent_changes* termina.
- 11) El HistoryCache del Writer RTPS le informa que todos los cambios han sido enviados por medio de la operación *can_send*.
- 12) La operación *can_send* termina.
- 13) El ReaderProxy serializa los datos mediante la operación *serialize*.
- 14) La operación *serialize* termina.
- 15) El DataWriter DDS elimina el cambio enviado mediante la operación *remove_change*.

- 16) La operación *remove_change* termina.
- 17) El ReaderProxy envía el submensaje DATA al MessageEncoder, para que el mensaje sea encapsulado.
- 18) El MessageEncoder envía el mensaje encapsulado mediante la operación *encoded_message*
- 19) La operación *encoded_message* termina.
- 20) El UDPTransmitter envía el mensaje UDP a la red.
- 21) El usuario intenta obtener datos mediante la operación *take* al DataReader DDS.
- 22) El DataReader DDS intenta obtener los cambios mediante la operación *get_change* al HistoryCache del Reader RTPS.
- 23) La operación *get_change* termina.
- 24) La operación *take* termina.

3.5.1.2. Interacción en base a la QoS Reliable

3.5.1.2.1. Reliable Reader—Reliable Writer

La siguiente descripción corresponde a la Figura 3-22.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS
- 2) El DataWriter DDS crea un CacheChange mediante la operación *new_change* al Stateful Writer.
- 3) La operación *new_change* termina.
- 4) El DataWriter DDS añade el cambio mediante la operación *add_change* al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación *notify_change*.
- 6) La operación *notify_change* termina.
- 7) La operación *add_change* termina.
- 8) La operación *write* termina.
- 9) El HistoryCache del Writer RTPS envía los cambios no enviados mediante la operación *unsent_changes* al ReaderProxy.
- 10) La operación *unsent_changes* termina.
- 11) El HistoryCache del Writer RTPS le informa que todos los cambios han sido enviados por medio de la operación *can_send*.

- 12) La operación *can_send* termina.
- 13) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
- 14) La operación *serialize* termina.
- 15) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 16) El *HistoryCache* del *Writer RTPS* reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.
- 17) La operación *unacked_changes* termina.
- 18) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT.
- 19) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 20) La operación *encoded_message* termina.
- 21) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 22) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 23) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 24) La operación *doDecode* termina.
- 25) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.
- 26) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 27) La operación *deserialize_data* termina.
- 28) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del *Reader RTPS*.
- 29) La operación *missing_changes* termina.

- 30) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize*.
- 31) La operación *serialize* termina.
- 32) El *WriterProxy* envía los submensajes *INFO_DESTINATION* y *ACKNACK* con la confirmación de recepción o pérdida de paquetes.
- 33) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 34) La operación *encoded_message* termina.
- 35) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 36) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 37) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 38) La operación *doDecode* termina.
- 39) El *ReaderProxy* recibe los submensajes *INFO_DESTINATION* y *ACKNACK* desde el *MessageEncoder*. El submensaje *INFO_DESTINATION* contiene el destino el cual ha confirmado el cambio.
- 40) El *ReaderProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 41) La operación *deserialize_data* termina.
- 42) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación *acked_changes*.
- 43) La operación *acked_changes* termina.
- 44) El *DataWriter DDS* consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
- 45) La operación *is_acked_by_all* termina.
- 46) El *DataWriter DDS* elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer RTPS*.
- 47) La operación *remove_change* termina.

- 48)Este literal toma lugar después del punto 25, luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder*.
- 49) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader RTPS*.
- 50)La operación *new_change* termina.
- 51)El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación *add_change*.
- 52)El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 53)La operación *notify_change* termina.
- 54)La operación *add_change* termina.
- 55)El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.
- 56)El *DataReader DDS* solicita los cambios por medio de la operación *get_change*.
- 57)La operación *get_change* termina.
- 58)La operación *take* termina. Los datos recibidos son entregados al usuario.
- 59)El usuario indica al *DataReader DDS* que ya obtuvo el cambio mediante la operación *return_loan*.
- 60)El *DataReader DDS* pregunta al *HistoryCache* del *Reader RTPS* si el cambio indicado es relevante mediante la operación *a_change_is_relevant*.
- 61)La operación *a_change_is_relevant* termina.
- 62)Dependiendo si el cambio es relevante el *DataReader DDS* elimina los cambios mediante la operación *remove_change*.
- 63)La operación *remove_change* termina.
- 64)La operación *return_loan* termina.

3.5.1.2.2. Reliable Reader—Reliable Writer con fragmentación

La siguiente descripción corresponde a la Figura 3-23.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS

- 2) El DataWriter DDS crea un CacheChange mediante la operación new_change al Stateful Writer.
- 3) La operación new_change termina.
- 4) El DataWriter DDS añade el cambio mediante la operación add_change al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación notify_change.
- 6) La operación notify_change termina.
- 7) La operación add_change termina.
- 8) La operación write termina.
- 9) El HistoryCache del Writer RTPS envía los cambios no enviados mediante la operación unsent_changes al ReaderProxy.
- 10) La operación unsent_changes termina.
- 11) El HistoryCache del Writer RTPS le informa que todos los cambios han sido enviados por medio de la operación can_send.
- 12) La operación can_send termina.
- 13) El ReaderProxy serializa los datos mediante la operación serialize, y se realiza el proceso de fragmentación de la información.
- 14) La operación serialize termina.
- 15) El ReaderProxy envía al MessageEncoder los submensajes GAP, DATA_FRAG y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 16) El HistoryCache del Writer RTPS reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación unacked_changes, para que sean añadidos a los submensajes HEARTBEAT_FRAG.
- 17) La operación unacked_changes termina.
- 18) El ReaderProxy envía al MessageEncoder los submensajes HEARTBEAT_FRAG.
- 19) El MessageEncoder encapsula el mensaje y lo envía mediante la operación encoded_message al UDPTransmitter.
- 20) La operación encoded_message termina.

- 21) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 22) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 23) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 24) La operación *doDecode* termina.
- 25) El *WriterProxy* recibe los submensajes *HEARTBEAT_FRAG* desde el *MessageDecoder*.
- 26) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 27) La operación *deserialize_data* termina.
- 28) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del Reader RTPS.
- 29) La operación *missing_changes* termina.
- 30) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize*.
- 31) La operación *serialize* termina.
- 32) El *WriterProxy* envía los submensajes *INFO_DESTINATION* y *NACK_FRAG* con la confirmación de recepción o pérdida de paquetes.
- 33) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 34) La operación *encoded_message* termina.
- 35) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 36) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 37) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 38) La operación *doDecode* termina.
- 39) El *ReaderProxy* recibe los submensajes *INFO_DESTINATION* y *NACK_FRAG* desde el *MessageEncoder*. El submensaje

`INFO_DESTINATION` contiene el destino el cual ha confirmado el cambio.

- 40) El *ReaderProxy* deserializa el submensaje por medio de la operación `deserialize_data`.
- 41) La operación `deserialize_data` termina.
- 42) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación `acked_changes`.
- 43) La operación `acked_changes` termina.
- 44) El *DataWriter DDS* consulta mediante la operación `is_acked_by_all` al *Stateful Writer* que todos los suscriptores tienen los cambios.
- 45) La operación `is_acked_by_all` termina.
- 46) El *DataWriter DDS* elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación `remove_change` al *HistoryCache* del *Writer RTPS*.
- 47) La operación `remove_change` termina.
- 48) Este literal toma lugar después del punto 25, luego de recibir el `HEARTBEAT` en el lado del suscriptor. El *WriterProxy* recibe los submensajes `GAP`, `INFO_TIMESTAMP` y `DATA_FRAG` del *MessageDecoder*.
- 49) El *WriterProxy* genera un nuevo *CacheChange* con la operación `new_change` en el *Reader RTPS*.
- 50) La operación `new_change` termina.
- 51) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación `add_change`.
- 52) El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación `notify_change`.
- 53) La operación `notify_change` termina.
- 54) La operación `add_change` termina.
- 55) El Usuario llama a la operación `take` para obtener los datos desde el *DataReader DDS*.
- 56) El *DataReader DDS* solicita los cambios por medio de la operación `get_change`.
- 57) La operación `get_change` termina.

- 58) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 59) El usuario indica al *DataReader* DDS que ya obtuvo el cambio mediante la operación *return_loan*.
- 60) El *DataReader* DDS pregunta al *HistoryCache* del *Reader RTPS* si el cambio indicado es relevante mediante la operación *a_change_is_relevant*.
- 61) La operación *a_change_is_relevant* termina.
- 62) Dependiendo si el cambio es relevante el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
- 63) La operación *remove_change* termina.
- 64) La operación *return_loan* termina.

3.5.1.2.3. Reliable Reader—Reliable Writer (Communication Error)

La siguiente descripción corresponde a la Figura 3-24.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS
- 2) El DataWriter DDS crea un CacheChange mediante la operación *new_change* al Stateful Writer.
- 3) La operación *new_change* termina.
- 4) El DataWriter DDS añade el cambio mediante la operación *add_change* al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación *notify_change*.
- 6) El Publisher DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación *data_available*.
- 7) El *ReaderProxy* serializa la notificación mediante la operación *serialize*.
- 8) La operación *serialize* termina.
- 9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder*.
- 10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 11) La operación *encoded_message* termina.

- 12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 15) La operación *doDecode* termina.
- 16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder*. El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
- 17) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data*.
- 18) La operación *deserialize_data* termina.
- 19) La operación *notify_change* termina.
- 20) La operación *add_change* termina.
- 21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
- 22) El *HistoryCache* del *Writer DDS* utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
- 23) La operación *unsent_changes* termina.
- 24) El *HistoryCache* del *Writer DDS* utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
- 25) La operación *can_send* termina.
- 26) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
- 27) La operación *serialize* termina.
- 28) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 29) El *HistoryCache* del *Writer RTPS* reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.
- 30) La operación *unacked_changes* termina.

- 31) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
- 32) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 33) La operación *encoded_message termina*.
- 34) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos. En este punto se ha simulado una falla en la comunicación por tanto el mensaje no llega a su destino.
- 35) El *ReaderProxy* reenvía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 36) El *ReaderProxy* reenvía al *MessageEncoder* el submensaje HEARTBEAT.
- 37) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 38) La operación *encoded_message termina*.
- 39) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 40) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 41) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 42) La operación *doDecode* termina.
- 43) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.
- 44) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 45) La operación *deserialize_data* termina.
- 46) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del Reader RTPS.
- 47) La operación *missing_changes* termina.

- 48) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize*.
- 49) La operación *serialize* termina.
- 50) El *WriterProxy* envía los submensajes *INFO_DESTINATION* y *ACKNACK* con la confirmación de recepción o pérdida de paquetes.
- 51) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 52) La operación *encoded_message* termina.
- 53) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 54) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 55) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 56) La operación *doDecode* termina.
- 57) El *ReaderProxy* recibe los submensajes *INFO_DESTINATION* y *ACKNACK* desde el *MessageEncoder*. El submensaje *INFO_DESTINATION* contiene el destino el cual ha confirmado el cambio.
- 58) El *ReaderProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 59) La operación *deserialize_data* termina.
- 60) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación *acked_changes*.
- 61) La operación *acked_changes* termina.
- 62) El *DataWriter DDS* consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
- 63) La operación *is_acked_by_all* termina.
- 64) El *DataWriter DDS* elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer RTPS*.
- 65) La operación *remove_change* termina.

- 66)Este literal toma lugar después del punto 43, luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder*.
- 67) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader RTPS*.
- 68)La operación *new_change* termina.
- 69)El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación *add_change*.
- 70)El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 71)La operación *notify_change* termina.
- 72)La operación *add_change* termina.
- 73)El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.
- 74)El *DataReader DDS* solicita los cambios por medio de la operación *get_change*.
- 75)La operación *get_change* termina.
- 76)La operación *take* termina. Los datos recibidos son entregados al usuario.
- 77)El usuario indica al *DataReader DDS* que ya obtuvo el cambio mediante la operación *return_loan*.
- 78)El *DataReader DDS* pregunta al *HistoryCache* del *Reader RTPS* si el cambio indicado es relevante mediante la operación *a_change_is_relevant*.
- 79)La operación *a_change_is_relevant* termina.
- 80)Dependiendo si el cambio es relevante el *DataReader DDS* elimina los cambios mediante la operación *remove_change*.
- 81)La operación *remove_change* termina.
- 82)La operación *return_loan* termina.

3.5.1.2.4. Reliable Reader—Reliable Writer (Packet Failure)

La siguiente descripción corresponde a la Figura 3-25.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS

- 2) El DataWriter DDS crea un CacheChange mediante la operación `new_change` al Stateful Writer.
- 3) La operación `new_change` termina.
- 4) El DataWriter DDS añade el cambio mediante la operación `add_change` al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación `notify_change`.
- 6) El Publisher DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación `data_available`.
- 7) El *ReaderProxy* serializa la notificación mediante la operación `serialize`.
- 8) La operación `serialize` termina.
- 9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder*, que será dirigido al participante uno.
- 10) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder*, que será dirigido al participante dos.
- 11) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación `encoded_message` al *UDPTransmitter*.
- 12) La operación `encoded_message` termina.

Del punto 13 al 18 pertenecen al participante 2

- 13) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos, dirigido hacia el participante dos.
- 14) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 15) El *UDPReceiver* desencapsula el mensaje mediante la operación `doDecode` en el *MessageDecoder*
- 16) La operación `doDecode` termina.
- 17) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder*. El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
- 18) El *WriterProxy* deserializa el submensaje mediante la operación `deserialize_data`.
- 19) La operación `deserialize_data` termina.

Del punto 20 al 26 pertenecen al participante 1

- 20) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos, dirigido hacia el participante dos.
- 21) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 22) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 23) La operación *doDecode* termina.
- 24) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder*. El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
- 25) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data*.
- 26) La operación *deserialize_data* termina.
- 27) La operación *notify_change* termina.
- 28) La operación *add_change* termina.
- 29) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
- 30) El *HistoryCache* del *Writer DDS* utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
- 31) La operación *unsent_changes* termina.
- 32) El *HistoryCache* del *Writer DDS* utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
- 33) La operación *can_send* termina.
- 34) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
- 35) La operación *serialize* termina.
- 36) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP, va dirigido al suscriptor uno.
- 37) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP, va dirigido al suscriptor dos.

38) El *HistoryCache* del *Writer RTPS* reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.

39) La operación *unacked_changes* termina.

40) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones, va dirigido al suscriptor uno.

41) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones, va dirigido al suscriptor dos.

Del punto 42 al 59 pertenece al suscriptor 2

42) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.

43) La operación *encoded_message* termina.

44) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.

45) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.

46) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*

47) La operación *doDecode* termina.

48) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.

49) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data*.

50) La operación *deserialize_data* termina.

51) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del *Reader RTPS*.

52) La operación *missing_changes* termina.

- 53) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize*.
- 54) La operación *serialize* termina.
- 55) El *WriterProxy* envía los submensajes *INFO_DESTINATION* y *ACKNACK* con la confirmación de recepción o pérdida de paquetes.
- 56) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 57) La operación *encoded_message* termina.
- 58) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 59) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- Del punto 60 al 77 pertenece al suscriptor 1**
- 60) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 61) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos. El mensaje se corrompe.
- 62) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 63) La operación *doDecode* termina.
- 64) El *WriterProxy* recibe el submensaje *HEARTBEAT* desde el *MessageDecoder*.
- 65) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 66) La operación *deserialize_data* termina.
- 67) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del Reader RTPS.
- 68) La operación *missing_changes* termina.
- 69) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize*.
- 70) La operación *serialize* termina.

- 71) El *WriterProxy* envía los submensajes *INFO_DESTINATION* y *ACKNACK* con la confirmación de recepción o pérdida de paquetes.
- 72) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 73) La operación *encoded_message* termina.
- 74) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 75) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 76) Una vez recibido los mensajes UDP, el *UDPReceiver* desencapsula los mensaje mediante la operación *doDecode* en el *MessageDecoder*.
- 77) La operación *doDecode* termina.
- 78) El *ReaderProxy* recibe los submensajes *INFO_DESTINATION* y *ACKNACK* desde el *MessageEncoder*, del suscriptor uno. El submensaje *INFO_DESTINATION* contiene el destino el cual ha confirmado el cambio.
- 79) El *ReaderProxy* recibe los submensajes *INFO_DESTINATION* y *ACKNACK* desde el *MessageEncoder*, del suscriptor dos. El submensaje *INFO_DESTINATION* contiene el destino el cual ha confirmado el cambio.
- 80) El *ReaderProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 81) La operación *deserialize_data* termina.
- 82) El *HistoryCache* del *Writer RTPS* pregunta si hay cambios confirmados mediante la operación *requested_changes* al *ReaderProxy*.
- 83) El *ReaderProxy* envía los cambios confirmados y no confirmados al *Stateful Writer* mediante la operación *acked_changes*.
- 84) La operación *acked_changes* termina.
- 85) La operación *requested_changes* termina.

Del punto 86 al 102 pertenece al suscriptor 2

- 86) Luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder*.
- 87) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader RTPS*.
- 88) La operación *new_change* termina.
- 89) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación *add_change*.
- 90) El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 91) La operación *notify_change* termina.
- 92) La operación *add_change* termina.
- 93) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.
- 94) El *DataReader DDS* solicita los cambios por medio de la operación *get_change*.
- 95) La operación *get_change* termina.
- 96) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 97) El usuario indica al *DataReader DDS* que ya obtuvo el cambio mediante la operación *return_loan*.
- 98) El *DataReader DDS* pregunta al *HistoryCache* del *Reader RTPS* si el cambio indicado es relevante mediante la operación *a_change_is_relevant*.
- 99) La operación *a_change_is_relevant* termina.
- 100) Dependiendo si el cambio es relevante el *DataReader DDS* elimina los cambios mediante la operación *remove_change*.
- 101) La operación *remove_change* termina.
- 102) La operación *return_loan* termina.
- Desde el punto 103 pertenece al suscriptor 1**
- 103) Luego de recibir el HEARTBEAT en el lado del suscriptor. El *WriterProxy* recibe los submensajes GAP, INFO_TIMESTAMP y DATA del *MessageDecoder*.

- 104) El *HistoryCache* del *Writer RTPS* reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT.
- 105) La operación *unacked_changes* termina.
- 106) El *ReaderProxy* reenvía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 107) El *ReaderProxy* reenvía al *MessageEncoder* el submensaje HEARTBEAT.
- 108) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 109) La operación *encoded_message* termina.
- 110) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 111) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 112) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 113) La operación *doDecode* termina.
- 114) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.
- 115) El *WriterProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 116) La operación *deserialize_data* termina.
- 117) El *WriterProxy* obtiene una lista de los cambios que se han perdido por medio de la operación *missing_changes* al *HistoryCache* del *Reader RTPS*.
- 118) La operación *missing_changes* termina.
- 119) Los números de secuencia faltantes son serializados en el *WriterProxy* mediante la operación *serialize*.
- 120) La operación *serialize* termina.

- 121) El *WriterProxy* envía los submensajes *INFO_DESTINATION* y *ACKNACK* con la confirmación de recepción o pérdida de paquetes.
- 122) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 123) La operación *encoded_message termina*.
- 124) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 125) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 126) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 127) La operación *doDecode* termina.
- 128) El *ReaderProxy* recibe los submensajes *INFO_DESTINATION* y *ACKNACK* desde el *MessageEncoder*. El submensaje *INFO_DESTINATION* contiene el destino el cual ha confirmado el cambio.
- 129) El *ReaderProxy* deserializa el submensaje por medio de la operación *deserialize_data*.
- 130) La operación *deserialize_data* termina.
- 131) El *ReaderProxy* envía al *Stateful Writer* los cambios que han sido confirmados mediante la operación *acked_changes*.
- 132) La operación *acked_changes* termina.
- 133) El *DataWriter DDS* consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
- 134) La operación *is_acked_by_all* termina.
- 135) El *DataWriter DDS* elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer RTPS*.
- 136) La operación *remove_change* termina.
- 137) Este literal toma lugar después del punto 114, luego de recibir el *HEARTBEAT* en el lado del suscriptor. El *WriterProxy* recibe los submensajes *GAP*, *INFO_TIMESTAMP* y *DATA* del *MessageDecoder*.

- 138) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader RTPS*.
- 139) La operación *new_change* termina.
- 140) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación *add_change*.
- 141) El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 142) La operación *notify_change* termina.
- 143) La operación *add_change* termina.
- 144) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.
- 145) El *DataReader DDS* solicita los cambios por medio de la operación *get_change*.
- 146) La operación *get_change* termina.
- 147) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 148) El usuario indica al *DataReader DDS* que ya obtuvo el cambio mediante la operación *return_loan*.
- 149) El *DataReader DDS* pregunta al *HistoryCache* del *Reader RTPS* si el cambio indicado es relevante mediante la operación *a_change_is_relevant*.
- 150) La operación *a_change_is_relevant* termina.
- 151) Dependiendo si el cambio es relevante el *DataReader DDS* elimina los cambios mediante la operación *remove_change*.
- 152) La operación *remove_change* termina.
- 153) La operación *return_loan* termina.

3.5.1.3. Interacción en base a la combinación QoS reliable – Best Effort

3.5.1.3.1. Reliable Writer—Best Effort Reader

La siguiente descripción corresponde a la Figura 3-26.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS

- 2) El DataWriter DDS crea un CacheChange mediante la operación `new_change` al Stateful Writer.
- 3) La operación `new_change` termina.
- 4) El DataWriter DDS añade el cambio mediante la operación `add_change` al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación `notify_change`.
- 6) El Publisher DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación `data_available`.
- 7) El *ReaderProxy* serializa la notificación mediante la operación `serialize`.
- 8) La operación `serialize` termina.
- 9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder*.
- 10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación `encoded_message` al *UDPTransmitter*.
- 11) La operación `encoded_message` termina.
- 12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 14) El *UDPReceiver* desencapsula el mensaje mediante la operación `doDecode` en el *MessageDecoder*
- 15) La operación `doDecode` termina.
- 16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder*. El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
- 17) El *WriterProxy* deserializa el submensaje mediante la operación `deserialize_data`.
- 18) La operación `deserialize_data` termina.
- 19) La operación `notify_change` termina.
- 20) La operación `add_change` termina.

- 21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
- 22) El *HistoryCache* del *Writer DDS* utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
- 23) La operación *unsent_changes* termina.
- 24) El *HistoryCache* del *Writer DDS* utiliza la operación *can_send* para informar al *ReaderProxy* que puede enviar los cambios.
- 25) La operación *can_send* termina.
- 26) El *ReaderProxy* serializa los datos mediante la operación *serialize*.
- 27) La operación *serialize* termina.
- 28) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 29) El *HistoryCache* del *Writer RTPS* reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación *unacked_changes*, para que sean añadidos al submensaje HEARTBEAT. Esta operación asume que todos los mensajes son confirmados cuando se trabaja con suscriptores con mejor esfuerzo.
- 30) La operación *unacked_changes* termina.
- 31) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
- 32) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 33) La operación *encoded_message* termina.
- 34) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 35) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 36) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 37) La operación *doDecode* termina.

- 38) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.
- 39) El *WriterProxy* recibe el submensaje GAP Y DATA desde el *MessageDecoder*.
- 40) El *WriterProxy* llama a la operación *deserialize_data* al *Deserializer*
- 41) La operación *deserialize_data* termina.
- 42) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el *Reader RTPS*.
- 43) La operación *new_change* termina.
- 44) El *Stateful Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación *add_change*.
- 45) El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 46) La operación *notify_change* termina.
- 47) La operación *add_change* termina.
- 48) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.
- 49) El *DataReader DDS* solicita los cambios por medio de la operación *get_change*.
- 50) La operación *get_change* termina.
- 51) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 52) Una vez obtenido los cambios el *DataReader DDS* elimina los cambios mediante la operación *remove_change*.
- 53) La operación *remove_change* termina.
- 54) El *DataWriter DDS* consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
- 55) La operación *is_acked_by_all* termina.
- 56) El *DataWriter DDS* elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer RTPS*.
- 57) La operación *remove_change* termina.

3.5.1.3.2. Reliable Writer—Best Effort Reader (Packet Failure)

La siguiente descripción corresponde a la Figura 3-27.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS
- 2) El DataWriter DDS crea un CacheChange mediante la operación *new_change* al Stateful Writer.
- 3) La operación *new_change* termina.
- 4) El DataWriter DDS añade el cambio mediante la operación *add_change* al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación *notify_change*.
- 6) El Publisher DDS indica la disponibilidad de datos al *ReaderProxy* mediante la operación *data_available*.
- 7) El *ReaderProxy* serializa la notificación mediante la operación *serialize*.
- 8) La operación *serialize* termina.
- 9) El *ReaderProxy* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder*.
- 10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 11) La operación *encoded_message* termina.
- 12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 15) La operación *doDecode* termina.
- 16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder*. El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
- 17) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data*.

- 18) La operación `deserialize_data` termina.
- 19) La operación `notify_change` termina.
- 20) La operación `add_change` termina.
- 21) La operación `write` termina. El usuario ha completado la acción de escritura de datos.
- 22) El *HistoryCache* del *Writer DDS* utiliza la operación `unsent_changes` para informar al *ReaderProxy* que hay cambios o información no enviada.
- 23) La operación `unsent_changes` termina.
- 24) El *HistoryCache* del *Writer DDS* utiliza la operación `can_send` para informar al *ReaderProxy* que puede enviar los cambios.
- 25) La operación `can_send` termina.
- 26) El *ReaderProxy* serializa los datos mediante la operación `serialize`.
- 27) La operación `serialize` termina.
- 28) El *ReaderProxy* envía al *MessageEncoder* los submensajes GAP, DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 29) El *HistoryCache* del *Writer RTPS* reenvía los números de secuencia de los cambios que no han sido confirmados mediante la operación `unacked_changes`, para que sean añadidos al submensaje HEARTBEAT. Esta operación asume que todos los mensajes son confirmados cuando se trabaja con suscriptores con mejor esfuerzo.
- 30) La operación `unacked_changes` termina.
- 31) El *ReaderProxy* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
- 32) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación `encoded_message` al *UDPTransmitter*.
- 33) La operación `encoded_message` termina.
- 34) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 35) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos. Se simula un paquete corrupto.

- 36) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 37) La operación *doDecode* termina.
- 38) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.
- 39) El *WriterProxy* recibe el submensaje GAP Y DATA desde el *MessageDecoder*.
- 40) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.
- 41) El *DataReader DDS* solicita los cambios por medio de la operación *get_change*.
- 42) La operación *get_change* termina.
- 43) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 44) El *DataWriter DDS* consulta mediante la operación *is_acked_by_all* al *Stateful Writer* que todos los suscriptores tienen los cambios.
- 45) La operación *is_acked_by_all* termina.
- 46) El *DataWriter DDS* elimina los cambios cuando todos suscriptores han recibido los cambios por medio de la operación *remove_change* al *HistoryCache* del *Writer RTPS*.
- 47) La operación *remove_change* termina.

3.5.2. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS SIN ESTADO

3.5.2.1. Interacción en base a la QoS Best Effort

3.5.2.1.1. BestEffort Reader -- Best Effort Writer

La siguiente descripción corresponde a la Figura 3-28.

- 1) El usuario DDS escribe datos por medio de la llamada a la operación *write* en el *DataWriter DDS*.
- 2) El *DataWriter DDS* llama a la operación *new_change* en el *Writer RTPS* para crear un nuevo *CacheChange*. Cada uno de estos cambios es identificado únicamente por un *SequenceNumber*.
- 3) La operación *new_change* termina.

- 4) El *DataWriter* DDS utiliza la operación *add_change* para almacenar el *CacheChange* dentro de *HistoryCache* del *Writer RTPS*.
- 5) El *HistoryCache* del *Writer RTPS* notifica el cambio por medio de la operación *notify_change* al *Publisher DDS*.
- 6) La operación *notify_change* termina.
- 7) La operación *add_change* termina.
- 8) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
- 9) El *HistoryCache* del *Writer DDS* utiliza la operación *unsent_changes* para informar al *ReaderLocator* que hay cambios o información no enviada.
- 10) La operación *unsent_changes* termina.
- 11) El *HistoryCache* del *Writer DDS* utiliza la operación *can_send* para informar al *ReaderLocator* que puede enviar los cambios.
- 12) La operación *can_send* termina.
- 13) El *DataWriter DDS* utiliza la operación *remove_change* en el *HistoryCache* del *Writer DDS* para limpiar la cache. Esta operación puede ser realizada posteriormente.
- 14) La operación *remove_change* termina.
- 15) El *ReaderLocator* serializa la información mediante la operación *serialize* en el *Serializer*.
- 16) La operación *serialize* termina.
- 17) El *ReaderLocator* envía al *MessageEncoder* el submensaje DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP.
- 18) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 19) La operación *encoded_message* termina.
- 20) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 21) El *UDPRceiver* recibe el mensaje *UDP_Message* desde la red de Datos.

- 22) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*.
- 23) La operación *doDecode* termina.
- 24) El *MessageDecoder* envía el submensaje DATA y dependiendo de la política de QoS se envía también un INFO_TIMESTAMP al Reader RTPS.
- 25) El Reader RTPS llama a la operación *deserialize_data* en el *Deserializer*.
- 26) La operación *deserialize_data* termina.
- 27) El *Stateless Reader* añade el cambio al *HistoryCache* del Reader RTPS por medio de la operación *add_change*.
- 28) El *HistoryCache* del Reader RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 29) La operación *notify_change* termina.
- 30) La operación *add_change* termina.
- 31) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
- 32) El *DataReader* DDS solicita los cambios por medio de la operación *get_change* al *HistoryCache* del Reader RTPS.
- 33) La operación *get_change* termina.
- 34) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 35) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
- 36) La operación *remove_change* termina.

3.5.2.2. Interacción en base a la QoS Reliable – Best Effort

3.5.2.2.1. Reliable Writer – Best Effort Reader

La siguiente descripción corresponde a la Figura 3-29.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS
- 2) El DataWriter DDS crea un CacheChange mediante la operación *new_change* al Stateful Writer.
- 3) La operación *new_change* termina.

- 4) El DataWriter DDS añade el cambio mediante la operación *add_change* al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación *notify_change*.
- 6) El Publisher DDS indica la disponibilidad de datos al *ReaderLocator* mediante la operación *data_available*.
- 7) El *ReaderLocator* serializa la notificación mediante la operación *serialize*.
- 8) La operación *serialize* termina.
- 9) El *ReaderLocator* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder*.
- 10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 11) La operación *encoded_message* termina.
- 12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 15) La operación *doDecode* termina.
- 16) El *Stateless Reader* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder*. El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
- 17) El *Stateless Reader* deserializa el submensaje mediante la operación *deserialize_data*.
- 18) La operación *deserialize_data* termina.
- 19) La operación *notify_change* termina.
- 20) La operación *add_change* termina.
- 21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.

- 22) El *HistoryCache* del *Writer DDS* utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
- 23) La operación *unsent_changes* termina.
- 24) El *HistoryCache* del *Writer DDS* utiliza la operación *can_send* para informar al *ReaderLocator* que puede enviar los cambios.
- 25) La operación *can_send* termina.
- 26) El *ReaderLocator* serializa los datos mediante la operación *serialize*.
- 27) La operación *serialize* termina.
- 28) El *ReaderLocator* envía al *MessageEncoder* los submensajes DATA.
- 29) El *ReaderLocator* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
- 30) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 31) La operación *encoded_message* termina.
- 32) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 33) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 34) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 35) La operación *doDecode* termina.
- 36) El *Stateless Reader* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.
- 37) El *Stateless Reader* recibe el submensaje DATA desde el *MessageDecoder*.
- 38) El *Stateless Reader* llama a la operación *deserialize_data* al *Deserializer*
- 39) La operación *deserialize_data* termina.
- 40) El *Stateless Reader* añade el cambio al *HistoryCache* del *Reader RTPS* por medio de la operación *add_change*.

- 41) El *HistoryCache* del *Reader RTPS* notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 42) La operación *notify_change* termina.
- 43) La operación *add_change* termina.
- 44) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader DDS*.
- 45) El *DataReader DDS* solicita los cambios por medio de la operación *get_change*.
- 46) La operación *get_change* termina.
- 47) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 48) El *HistoryCache* del *Writer RTPS* solicita los cambios no confirmados al *ReaderLocator* mediante la operación *requested_changes*. Como se está trabajando con un *Reader* sin estado con mejor esfuerzo, este no confirma ningún cambio por lo cual en esta operación se assume que todo está confirmado.
- 49) La operación *requested_changes* termina.
- 50) El *ReaderLocator* envía al *MessageEncoder* el submensaje GAP. Este submensaje en este caso no solicitará ningún número de secuencia.
- 51) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 52) La operación *encoded_message* termina.
- 53) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 54) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 55) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 56) La operación *doDecode* termina.
- 57) El *Stateless Reader* recibe el submensaje GAP desde el *MessageDecoder*.
- 58) El *Stateless Reader* llama a la operación *deserialize_data* al *Deserializer*

- 59) La operación *deserialize_data* termina.
- 60) Una vez obtenido los cambios el *DataReader* DDS elimina los cambios mediante la operación *remove_change*.
- 61) La operación *remove_change* termina.

3.5.3. INTERACCIÓN DEL DDS CON EL PROTOCOLO RTPS HÍBRIDOS (CON ESTADO Y SIN ESTADO)

3.5.3.1.1. Reliable Stateless Writer – Reliable Stateful Reader

La siguiente descripción corresponde a la Figura 3-30.

- 1) El usuario DDS escribe datos mediante la operación *write* en el DataWriter DDS
- 2) El DataWriter DDS crea un CacheChange mediante la operación *new_change* al Stateful Writer.
- 3) La operación *new_change* termina.
- 4) El DataWriter DDS añade el cambio mediante la operación *add_change* al HistoryCache del Writer RTPS.
- 5) El HistoryCache del Writer RTPS notifica al Publisher mediante la operación *notify_change*.
- 6) El Publisher DDS indica la disponibilidad de datos al *ReaderLocator* mediante la operación *data_available*.
- 7) El *ReaderLocator* serializa la notificación mediante la operación *serialize*.
- 8) La operación *serialize* termina.
- 9) El *ReaderLocator* envía un submensaje HEARTBEAT en modo *waiting* al *MessageEncoder*.
- 10) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 11) La operación *encoded_message* termina.
- 12) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 13) El *UDPReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 14) El *UDPReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*

- 15) La operación *doDecode* termina.
- 16) El *WriterProxy* recibe el HEARTBEAT en modo *waiting* del *MessageDecoder*. El envío de este tipo HEARTBEAT sirve solamente para confirmar la presencia de un participante.
- 17) El *WriterProxy* deserializa el submensaje mediante la operación *deserialize_data*.
- 18) La operación *deserialize_data* termina.
- 19) La operación *notify_change* termina.
- 20) La operación *add_change* termina.
- 21) La operación *write* termina. El usuario ha completado la acción de escritura de datos.
- 22) El *HistoryCache* del *Writer DDS* utiliza la operación *unsent_changes* para informar al *ReaderProxy* que hay cambios o información no enviada.
- 23) La operación *unsent_changes* termina.
- 24) El *HistoryCache* del *Writer DDS* utiliza la operación *can_send* para informar al *ReaderLocator* que puede enviar los cambios.
- 25) La operación *can_send* termina.
- 26) El *ReaderLocator* serializa los datos mediante la operación *serialize*.
- 27) La operación *serialize* termina.
- 28) El *ReaderLocator* envía al *MessageEncoder* los submensajes DATA.
- 29) El *ReaderLocator* envía al *MessageEncoder* el submensaje HEARTBEAT. Este HEARTBEAT tiene un temporizador llamado *Heartbeat period* en el cual se debería recibir las confirmaciones.
- 30) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 31) La operación *encoded_message* termina.
- 32) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 33) El *UDPRReceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 34) El *UDPRReceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*

- 35) La operación *doDecode* termina.
- 36) El *WriterProxy* recibe el submensaje HEARTBEAT desde el *MessageDecoder*.
- 37) El *WriterProxy* recibe el submensaje DATA desde el *MessageDecoder*.
- 38) El *WriterProxy* deserializa los datos mediante la operación *deserialize_data*.
- 39) La operación *deserialize_data* termina.
- 40) El *WriterProxy* genera un nuevo *CacheChange* con la operación *new_change* en el Reader RTPS.
- 41) La operación *new_change* termina.
- 42) El *Stateful Reader* añade el cambio al *HistoryCache* del Reader RTPS por medio de la operación *add_change*.
- 43) El *HistoryCache* del Reader RTPS notifica el cambio al Suscriptor DDS por medio de la operación *notify_change*.
- 44) La operación *notify_change* termina.
- 45) La operación *add_change* termina.
- 46) El Usuario llama a la operación *take* para obtener los datos desde el *DataReader* DDS.
- 47) El *DataReader* DDS solicita los cambios por medio de la operación *get_change*.
- 48) La operación *get_change* termina.
- 49) La operación *take* termina. Los datos recibidos son entregados al usuario.
- 50) El *WriterProxy* envía la confirmación de los datos mediante un submensaje ACKNACK e indica el destinatario mediante el submensaje INFO_REPLY. No se utiliza el submensaje INFO_DESTINATION ya que el publicador es sin estado.
- 51) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 52) La operación *encoded_message* termina.
- 53) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.

- 54) El *UDPRceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 55) El *UDPRceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 56) La operación *doDecode* termina.
- 57) El *ReaderLocator* recibe los submensajes *INFO_REPLY* y *ACKNACK* desde el *MessageEncoder*. El submensaje *INFO_DESTINATION* contiene el destino el cual ha confirmado el cambio.
- 58) El *ReaderLocator* deserializa el submensaje por medio de la operación *deserialize_data*.
- 59) La operación *deserialize_data* termina.
- 60) El *HistoryCache* del *Writer RTPS* solicita los cambios no confirmados al *ReaderLocator* mediante la operación *requested_changes*.
- 61) La operación *requested_changes* termina.
- 62) El *ReaderLocator* envía al *MessageEncoder* el submensaje *GAP*. Este submensaje en este caso no solicitará ningún número de secuencia.
- 63) El *MessageEncoder* encapsula el mensaje y lo envía mediante la operación *encoded_message* al *UDPTransmitter*.
- 64) La operación *encoded_message* termina.
- 65) El *UDPTransmitter* envía el mensaje *UDP_Message* hacia la red de Datos.
- 66) El *UDPRceiver* recibe el mensaje *UDP_Message* desde la red de Datos.
- 67) El *UDPRceiver* desencapsula el mensaje mediante la operación *doDecode* en el *MessageDecoder*
- 68) La operación *doDecode* termina.
- 69) El *WriterProxy* recibe el submensaje *GAP* desde el *MessageDecoder*.
- 70) El *WriterProxy* llama a la operación *deserialize_data* al *Deserializer*
- 71) La operación *deserialize_data* termina.
- 72) Una vez obtenido los cambios el *DataReader DDS* elimina los cambios mediante la operación *remove_change*.
- 73) La operación *remove_change* termina.

3.5.4. PROTOCOLO DESCUBRIMIENTO

3.5.4.1. Tráfico de Descubrimiento

3.5.4.1.1. Fase de Descubrimiento de Participantes.

La siguiente descripción corresponde a la Figura 3-31.

- 1) El participante 1 ha sido creado.
- 2) El participante 1 se anuncia enviando mensajes SPDP.
- 3) El Participante 2 es creado.
- 4) El participante 2 se anuncia enviando mensajes SPDP.
- 5) El participante 1 recibe los paquetes SPDP y en este caso añade al participante 2 a la base de datos.
- 6) El participante 1 se anuncia enviando mensajes SPDP.
- 7) El participante 2 recibe los paquetes SPDP y en este caso añade al participante 1 a la base de datos.
- 8) El participante 2 se anuncia enviando mensajes SPDP.
- 9) El participante 1 crea un *DataWriter*.
- 10) El participante 1 envía su publicación por medio de mensajes SEDP.
- 11) El participante 2 recibe el mensaje SEDP y añade al *DataWriter* remoto a su base de datos.
- 12) El participante 1 continúa enviando su publicación mediante mensajes SEDP.
- 13) El participante 1 destruye su *DataWriter*.
- 14) El participante 1 informa que el *DataWriter* ha sido eliminado enviando mensajes SEDP.
- 15) El participante 1 es destruido.
- 16) El participante 1 informa que ha sido eliminado enviando mensajes SPDP.
- 17) El participante 2 recibe el mensaje SPDP y remueve al participante 1 de la base de datos.
- 18) El participante 2 es destruido.
- 19) El participante 2 informa que ha sido eliminado enviando mensajes SPDP.

3.5.5. INTERCAMBIO DE MENSAJES RTPS SOBRE LA RED

Una vez presentado el comportamiento mediante los diagramas de secuencia de la interacción del protocolo RTPS con DDS, se muestra varios ejemplos sobre el intercambio de mensajes RTPS.

3.5.5.1. Ejemplo 1

El siguiente gráfico ilustra el flujo de datos que han generado un par de entidades que se comunican entre sí.



Figura 3-15. Intercambio de mensajes RTPS sobre la red Ejemplo 1

- Los primeros 3 paquetes capturados muestran la interacción entre entidades independientes con el DDS.
- Los 3 paquetes siguientes son mensajes equivalentes a un saludo.
- El paquete resaltado en azul representa al suscriptor buscando un servicio por medio del *Topic*, el mensaje no va al publicador en primer lugar, porque no sabe que una de las entidades ya ha

publicado ese servicio, el mensaje va directamente al Middleware para consultar si alguna entidad dispone de un servicio para la solicitud.

- El Middleware anuncia al publicador que una entidad requiere sus servicios, y envía la información de la entidad que quiere el servicio, de forma transparente.
- El siguiente mensaje, el cual esta resaltado es un mensaje ping que anuncia la presencia de la entidad que tiene el servicio.
- El siguiente mensaje tiene 2 submensajes, el primero un submensaje *infoDestination*, que indica el identificador del servicio publicado a la entidad que ha consultado el servicio. El otro submensaje corresponde a un submensaje *GAP*, que indica que los siguientes mensajes vienen con un número de secuencia y tienen que mantener un orden.
- El siguiente grupo de submensajes representa los datos enviados por el servicio y que tienen un orden, hasta cualquier cambio de la continuidad de los mensajes, para que la entidad que utiliza el servicio, informe a otra entidad con un submensaje *Heartbeat*.
- Con el submensaje *Acknack* la entidad que tiene el servicio indica que este ya no se encuentra disponible.

3.5.5.2. Ejemplo 2

11 1.5118040 127.0.0.1	127.0.0.1	RTPS	94 DATA
12 1.5119690 127.0.0.1	127.0.0.1	RTPS	98 GAP
13 2.009720 127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
14 2.5113380 127.0.0.1	127.0.0.1	RTPS	110 INFO_DST, ACKNACK
15 3.0120970 127.0.0.1	127.0.0.1	RTPS	1150 DATA_FRAG
16 3.0122350 127.0.0.1	127.0.0.1	RTPS	1122 DATA_FRAG
17 4.0137800 127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
18 4.0138470 127.0.0.1	127.0.0.1	RTPS	90 HEARTBEAT_FRAG
19 4.5158290 127.0.0.1	127.0.0.1	RTPS	146 INFO_DST, ACKNACK, NACK_FRAG
20 4.5160960 127.0.0.1	127.0.0.1	RTPS	1122 DATA_FRAG
21 5.0158690 127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
22 5.5121540 127.0.0.1	127.0.0.1	RTPS	110 INFO_DST, ACKNACK
23 6.0173590 127.0.0.1	127.0.0.1	RTPS	90 HEARTBEAT_FRAG
24 6.5188610 127.0.0.1	127.0.0.1	RTPS	146 INFO_DST, ACKNACK, NACK_FRAG
25 7.0187010 127.0.0.1	127.0.0.1	RTPS	1150 DATA_FRAG
26 7.0188060 127.0.0.1	127.0.0.1	RTPS	94 DATA
27 7.0188440 127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
28 7.5209270 127.0.0.1	127.0.0.1	RTPS	146 INFO_DST, ACKNACK, NACK_FRAG

Figura 3-16. Intercambio de Mensajes RTPS Ejemplo 2

La captura corresponde al flujo de datos intercambiados por 2 participantes que trabajan con RTPS sobre el Middleware DDS.

Al inicio de la captura, se puede observar que todos los participantes envían un mensaje con el mismo formato, y que todos son enviados a la IP del Middleware, 239.255.0.1, este mensaje incluye los submensajes *InfoTimeStamp* y *Data*. El

submensaje *InfoTimeStamp* tiene el propósito de dar una referencia de tiempo o marca a los siguientes submensajes. El submensaje *Data* solo envía cambios en los objetos de datos.

En el mensaje resaltado, se puede observar que el *host* con IP 192.168.5.1 envía al host con IP 192.168.5.2 un mensaje RTPS con los submensajes, que corresponden a un *InfoDestination*, el cual tiene el único propósito de enviar información sobre el *guidPrefix* para ser identificado, y un submensaje *AckNack*.

Como ya fue explicado anteriormente el submensaje *AckNack* es usado para comunicar el estado de un Lector a un Escritor, y anteriormente se pudo observar que el *host* 192.168.5.2, envía al *host* 192.168.5.1 un submensaje *Heartbeat*, lo que significa que el participante está saludando y el otro participante responde, con el submensaje *AckNack*, diciendo ¿necesitas algo?, y como la bandera estaba seteada en 1, el *host* con 192.168.5.1 no está obligado a responder.

3.5.5.3. Ejemplo 3

11 1.5118040127.0.0.1	127.0.0.1	RTPS	94 DATA
12 1.511950127.0.0.1	127.0.0.1	RTPS	98 GAP
13 2.0097220127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
14 2.5113380127.0.0.1	127.0.0.1	RTPS	110 INFO_DST, ACKNACK
15 3.0124070127.0.0.1	127.0.0.1	RTPS	1150 DATA_FRAG
16 3.0122330127.0.0.1	127.0.0.1	RTPS	1122 DATA_FRAG
17 4.0137804127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
18 4.0138470127.0.0.1	127.0.0.1	RTPS	90 HEARTBEAT_FRAG
19 4.5158290127.0.0.1	127.0.0.1	RTPS	146 INFO_DST, ACKNACK, NACK_FRAG
20 4.5160960127.0.0.1	127.0.0.1	RTPS	1122 DATA_FRAG
21 5.0158690127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
22 5.5171540127.0.0.1	127.0.0.1	RTPS	110 INFO_DST, ACKNACK
23 6.0173590127.0.0.1	127.0.0.1	RTPS	90 HEARTBEAT_FRAG
24 6.5188610127.0.0.1	127.0.0.1	RTPS	146 INFO_DST, ACKNACK, NACK_FRAG
25 7.0187050127.0.0.1	127.0.0.1	RTPS	1150 DATA_FRAG
26 7.0188660127.0.0.1	127.0.0.1	RTPS	94 DATA
27 7.0188440127.0.0.1	127.0.0.1	RTPS	94 HEARTBEAT
28 7.5209770127.0.0.1	127.0.0.1	RTPS	146 INFO_DST, ACKNACK, NACK_FRAG

Figura 3-17. Intercambio de Mensajes RTPS Ejemplo 3

La captura corresponde a un flujo de datos intercambiados trabajando de manera local.

Al inicio de la captura, se puede observar un Submensaje *GAP*, el cual indica que el siguiente submensaje viene un número de secuencia para mantener un orden. A continuación, se puede observar el submensaje *HeartBeat* que significa que el participante está saludando y responden con los submensajes *InfoDestination* y *AckNack*; el primero, tiene como propósito enviar información sobre el *guidPrefix* para ser identificado localmente y el submensaje *AckNack* es usado para comunicar el estado del Lector al Escritor.

El siguiente submensaje es el *DataFrag*, el cual fragmenta los datos y los envía en varios submensajes *DataFrag*. Los fragmentos contenidos en los submensajes *DataFrag* se vuelven a ensamblar en el *Reader RTPS*. El siguiente

submensaje es el *HeartbeatFrag*, el cual se envía desde un *Writer RTPS* a un *Reader RTPS*, lo que indica cuales fragmentos están disponibles.

Por último, se puede observar los submensajes *InfoDestination*, *AckNack* y el *NackFrag*. El submensaje *NackFrag* permite al lector informar al *Writer* acerca del número de fragmentos que se han perdido.

3.5.5.4. Ejemplo 4

En este ejemplo se procede a realizar la misma prueba del ejemplo 1, con la diferencia que en este caso se trabajará con 3 *host*

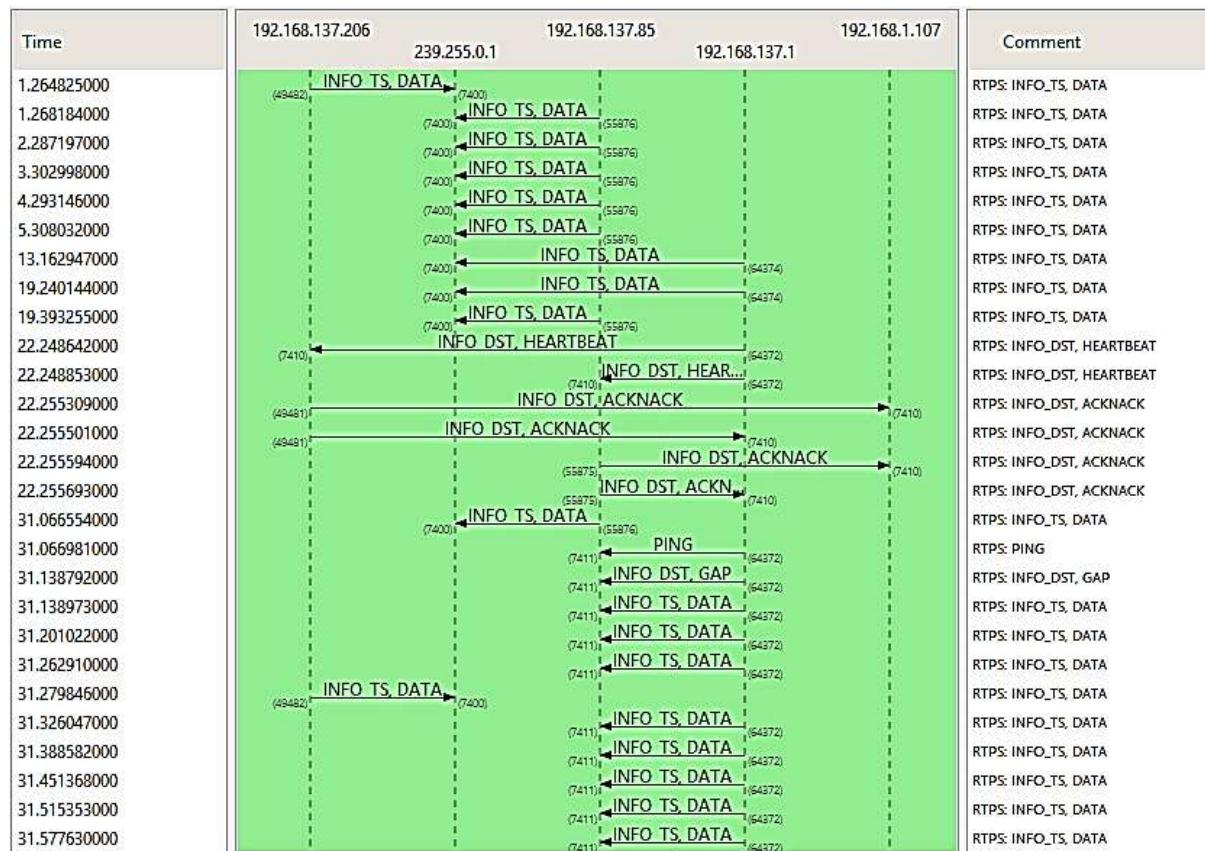


Figura 3-18. Intercambio de Mensajes RTPS Ejemplo 4.1

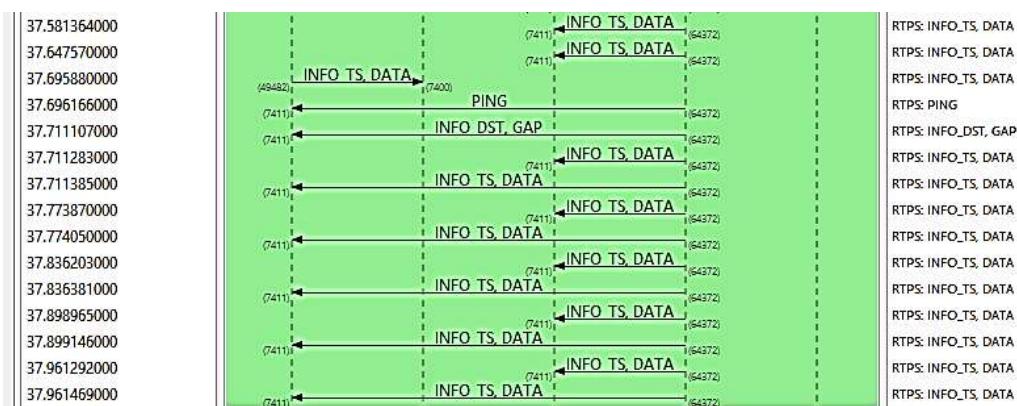


Figura 3-19. Intercambio de Mensajes RTPS Ejemplo 4.2

Como se puede observar en la primera figura, el comportamiento es el mismo que en el ejemplo1, al suscribirse un segundo computador al servicio, como se observa en la segunda imagen, primeramente, el publicador hace un ping al subscriptor, luego le informa con el *GAP*, que los datos que vienen deben mantener un orden. Y el mensaje que antes se enviaba a un solo *host*, ahora es enviado a dos *host*, es decir se enviarán tantos mensajes como suscriptores hayan.

3.6. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS

3.6.1. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS CON ESTADO

3.6.1.1. Diagramas de secuencia basados en la QoS Best Effort

3.6.1.1.1. Best Effort Reader – Best Effort Writer

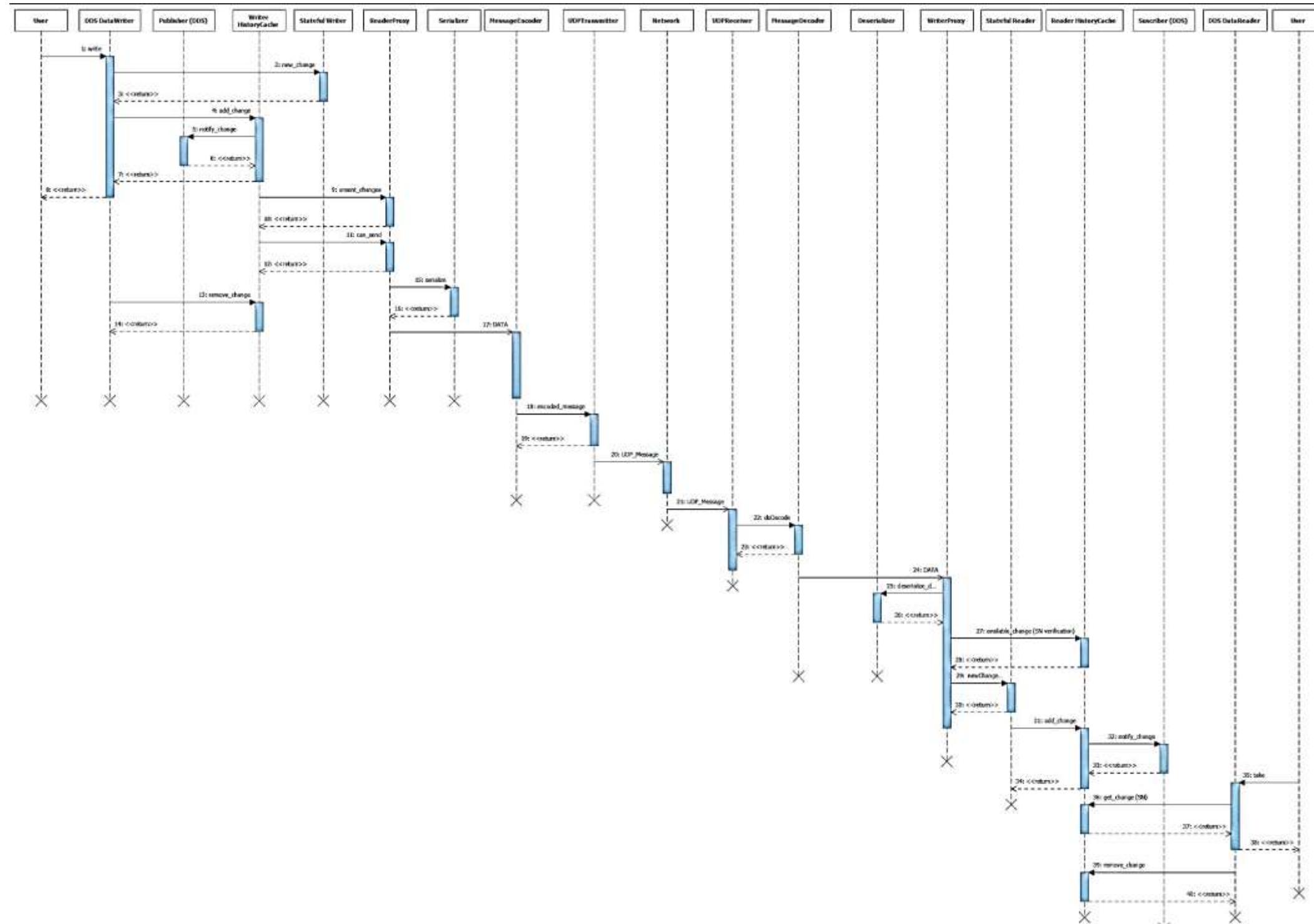


Figura 3-20. Comportamiento Best Effort Reader – Best Effort Writer en interacción con estado.

3.6.1.1.2. Best Effort Reader – BestEffort Writer (Packet Failure)

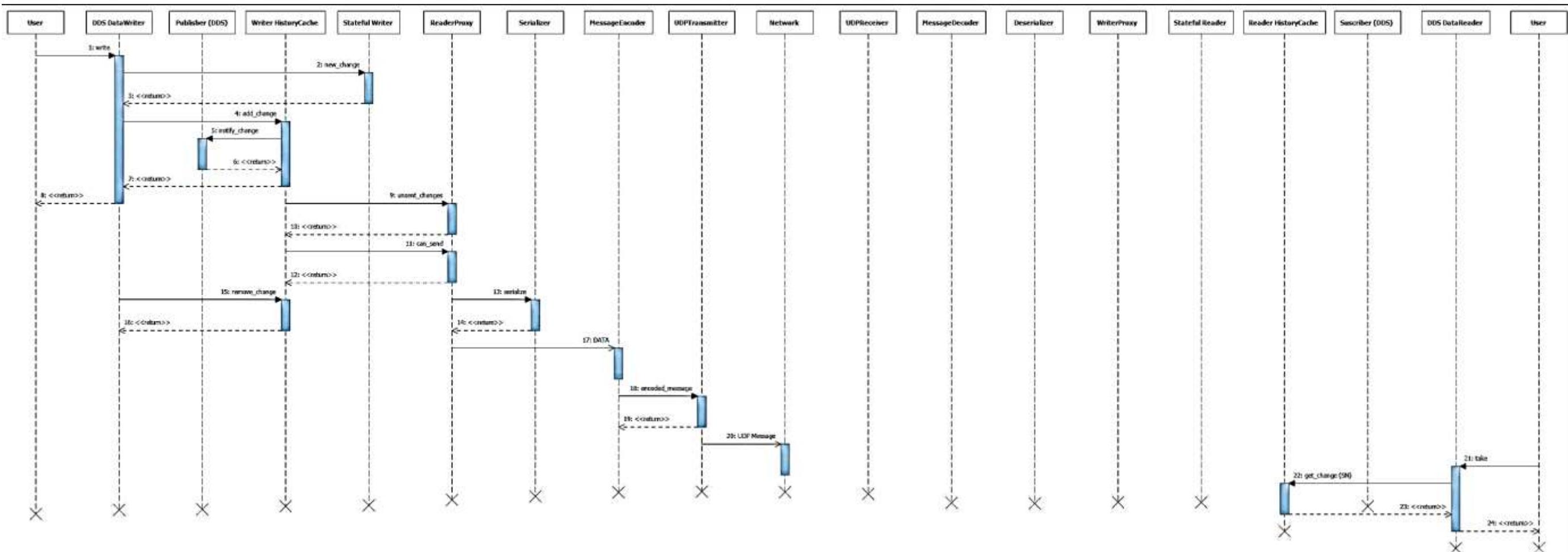


Figura 3-21. Comportamiento Best Effort Reader – Best Effort Writer en interacción con estado con falla de envío de paquete.

3.6.1.2. Diagramas de secuencia basados en la QoS Reliable

3.6.1.2.1. Reliable Reader – Reliable Writer

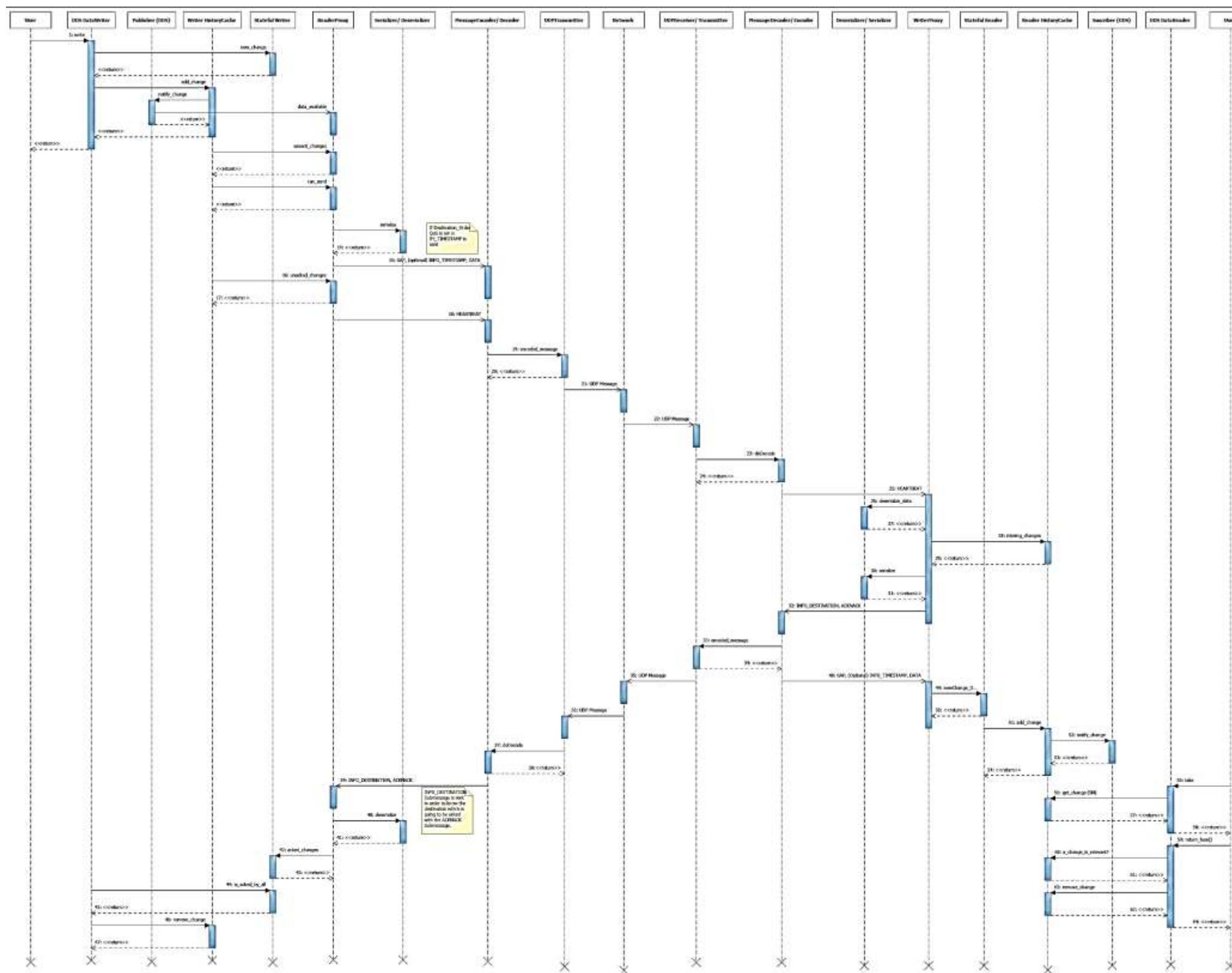


Figura 3-22. Comportamiento Reliable Reader – Reliable Writer en interacción con estado.

3.6.1.2.2. Reliable Reader – Reliable Writer con fragmentación

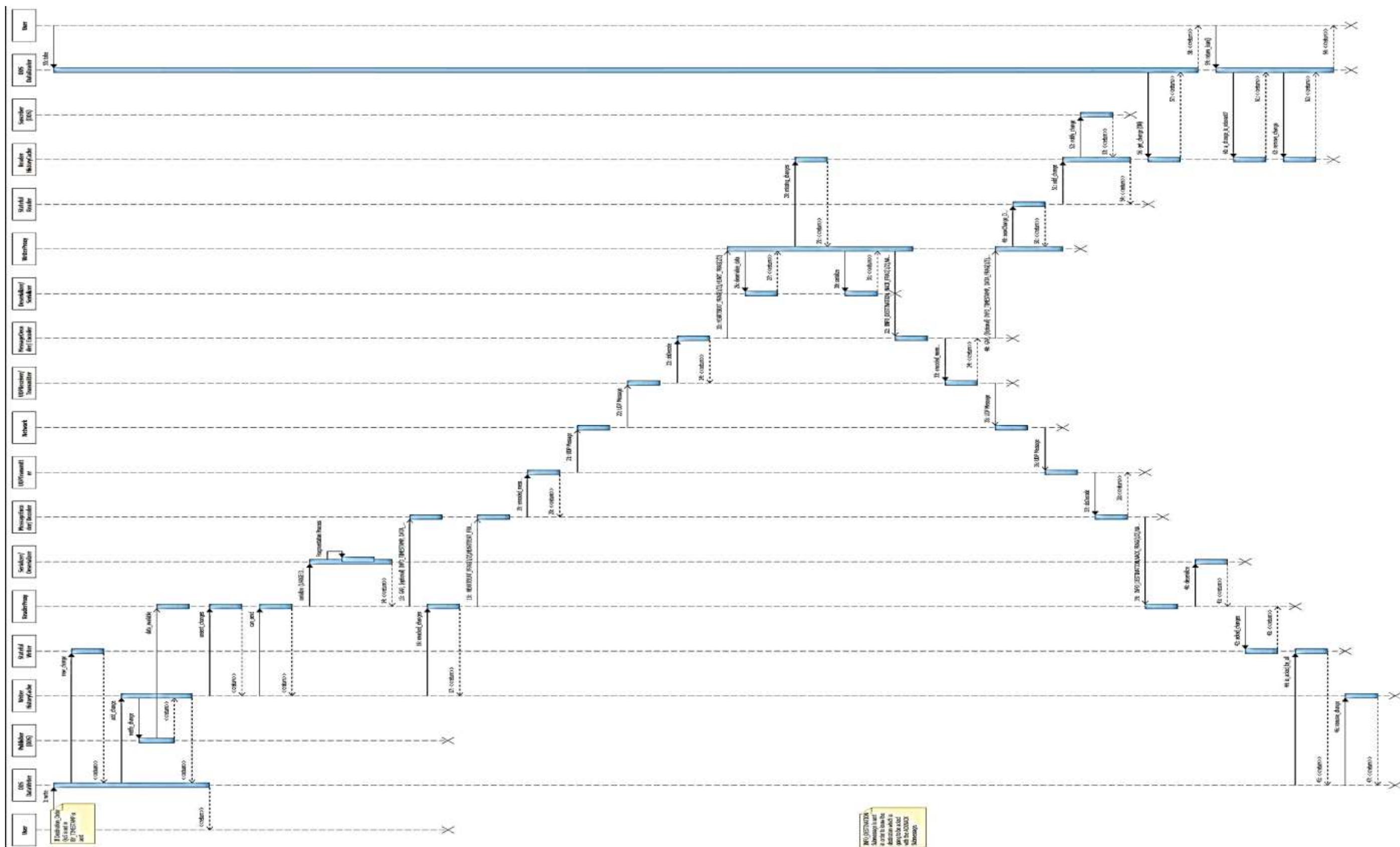


Figura 3-23. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con fragmentación de datos.

3.6.1.2.3. Reliable Reader – Reliable Writer (Communication Error)

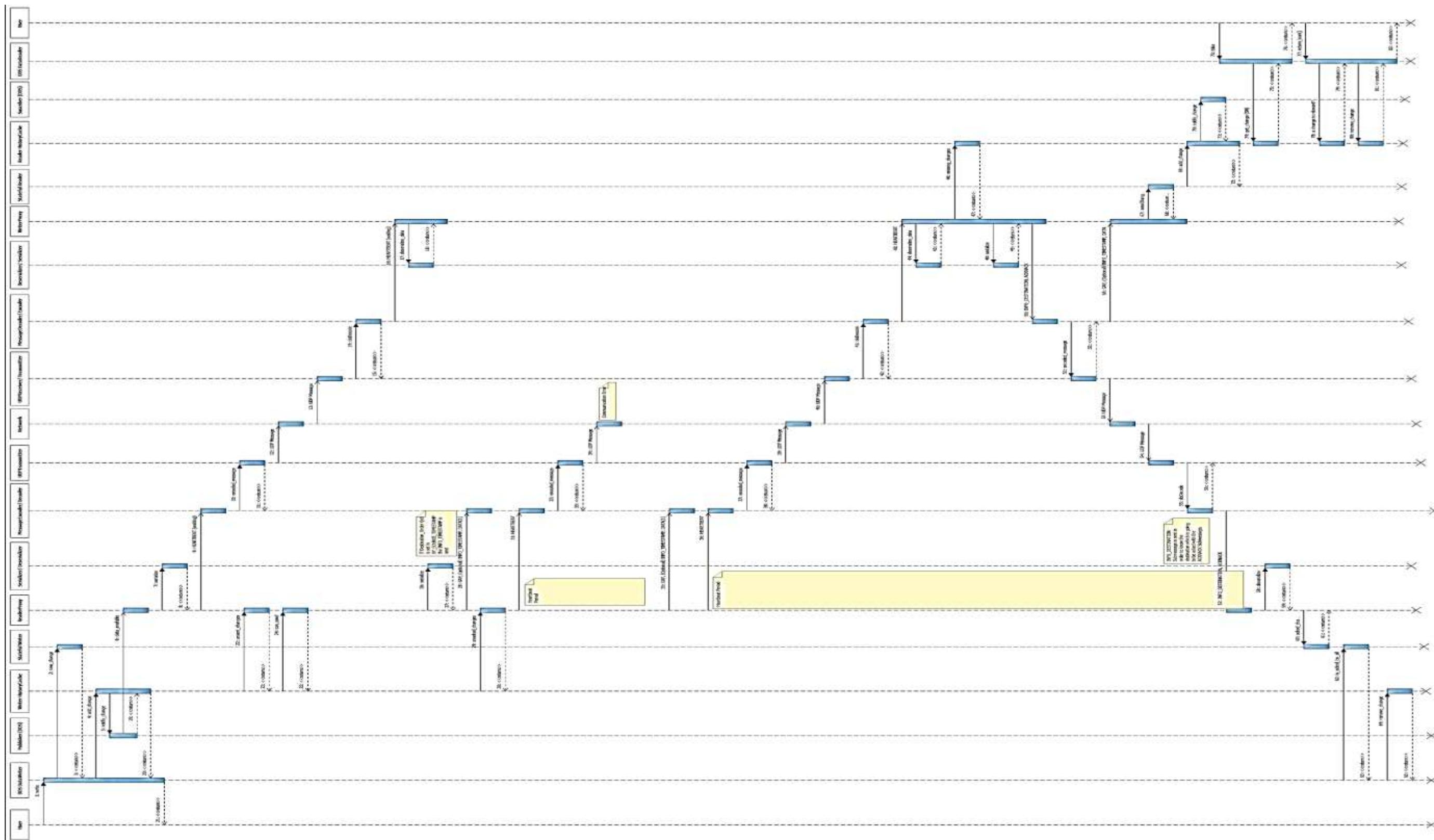


Figura 3-24. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con falla en la comunicación.

3.6.1.2.4. Reliable Reader – Reliable Writer (Packet Failure)

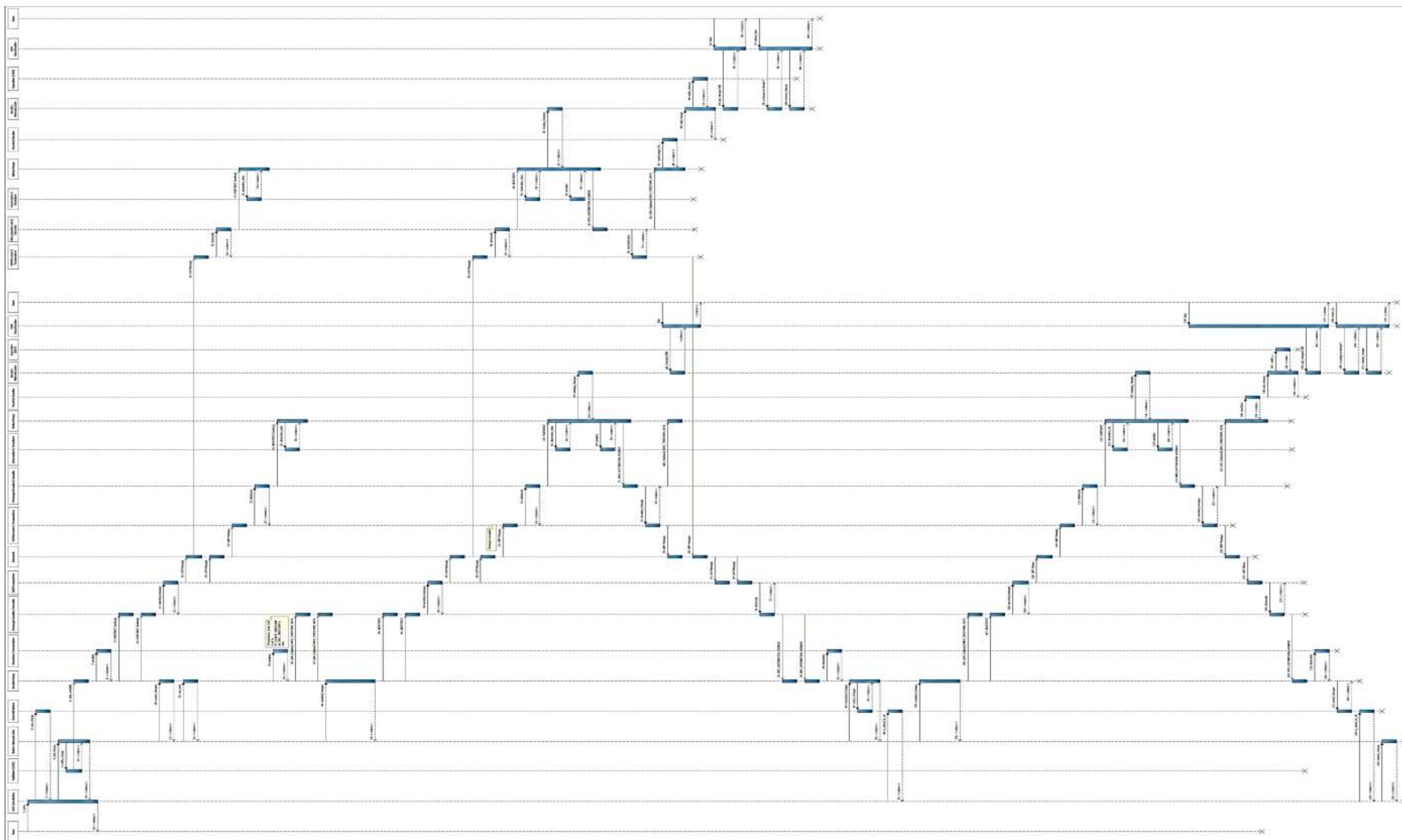


Figura 3-25. Comportamiento Reliable Reader – Reliable Writer en interacción con estado con falla de envío de paquete y con tres participantes.

3.6.1.3. Diagramas de secuencia basados en la combinación de la QoS Reliable – Best Effort

3.6.1.3.1. Reliable Writer—Best Effort Reader

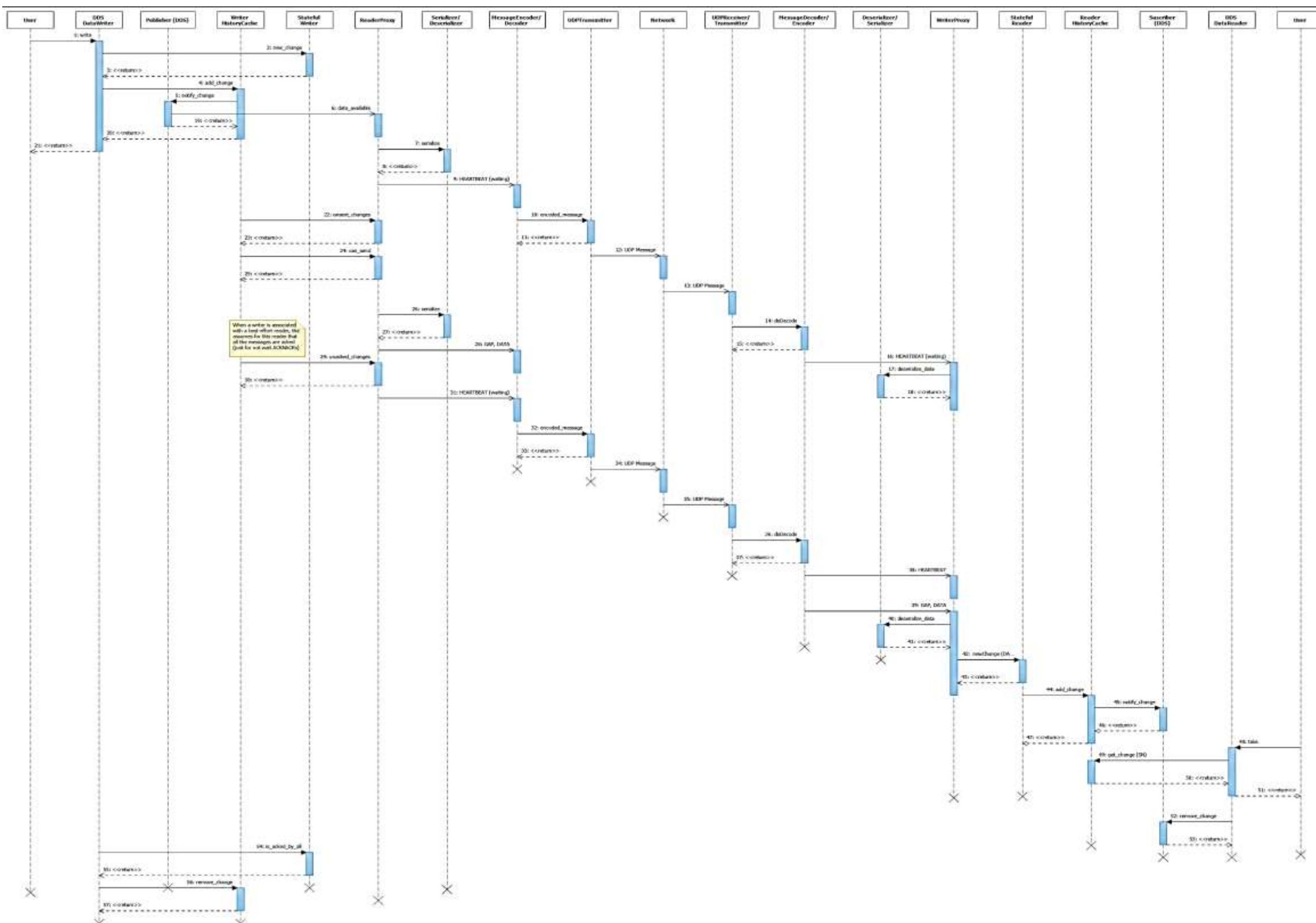


Figura 3-26. Comportamiento Reliable Writer – Best Effort Reader en interacción con estado.

3.6.1.3.2. Reliable Writer—Best Effort Reader (Packet Failure)

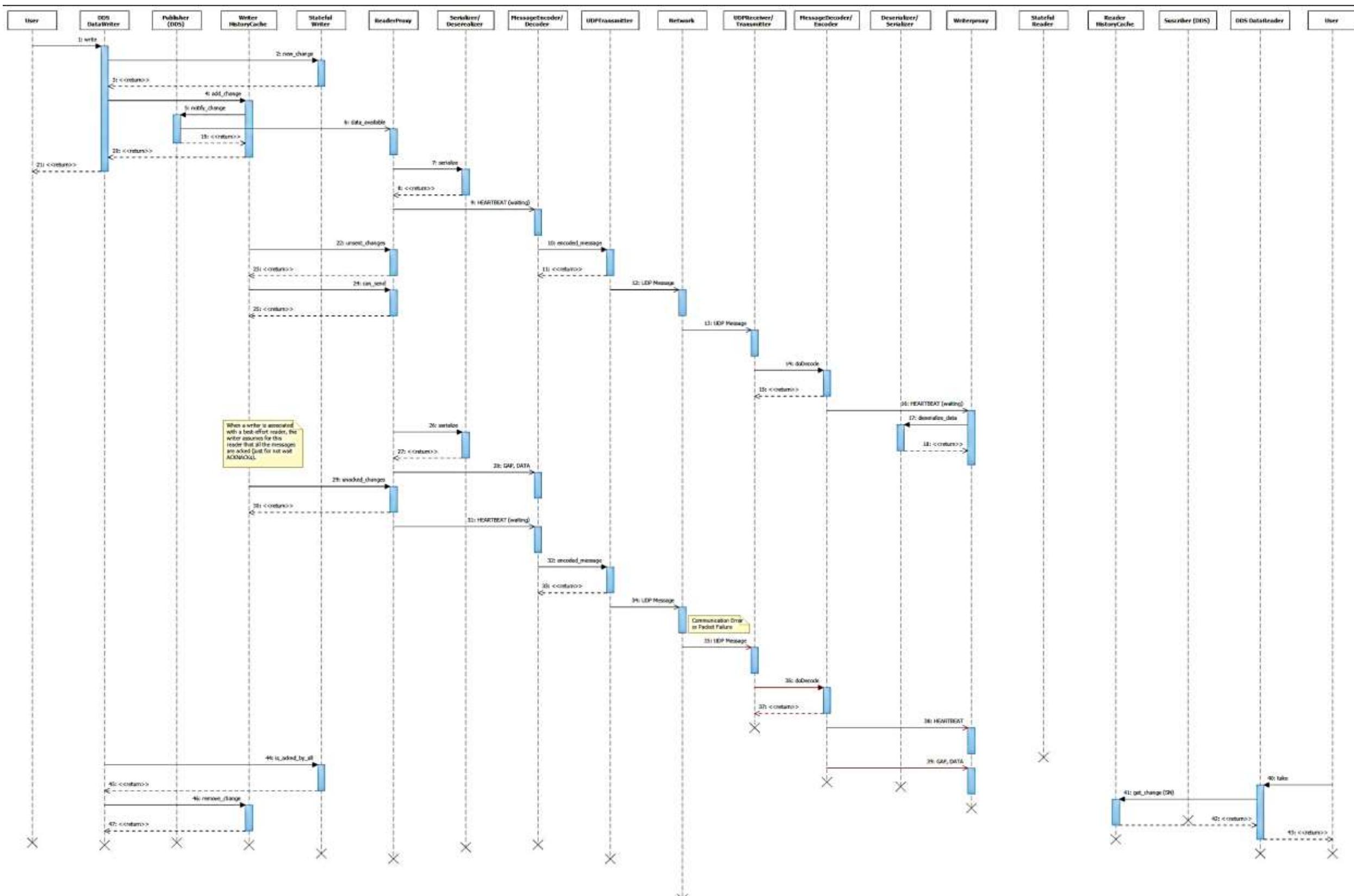


Figura 3-27. Comportamiento Reliable Writer – Best Effort Reader en interacción con estado con falla en el envío de paquetes.

3.6.2. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS SIN ESTADO

3.6.2.1. Diagrama de secuencia basado en la QoS Best Effort

3.6.2.1.1. Best Effort Reader -- Best Effort Writer

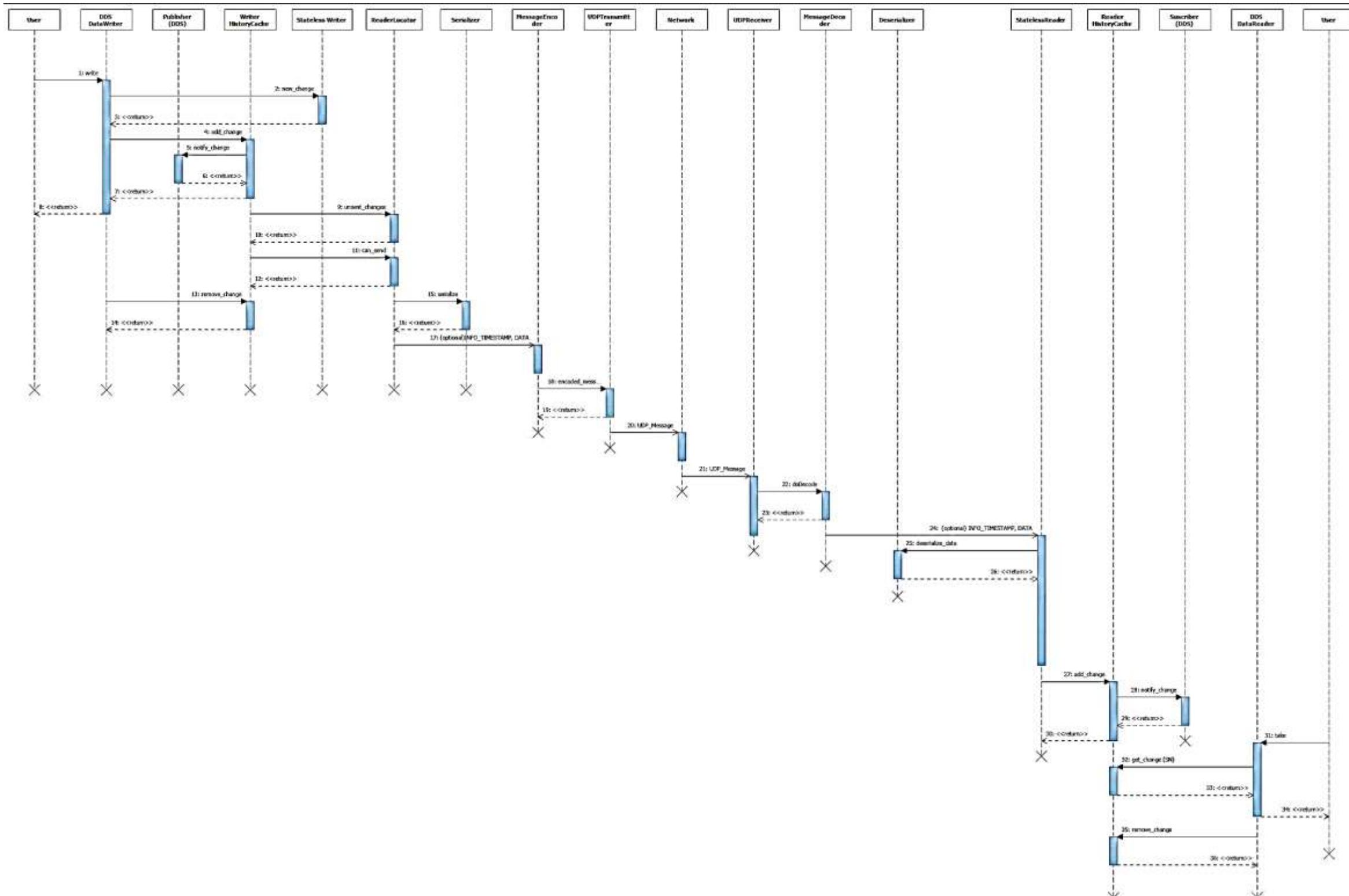


Figura 3-28. Comportamiento Best Effort Reader – Best Effort Writer en interacción sin estado.

3.6.2.2. Diagrama de secuencia basado en la QoS Reliable – Best Effort

3.6.2.2.1. Reliable Writer – Best Effort Reader

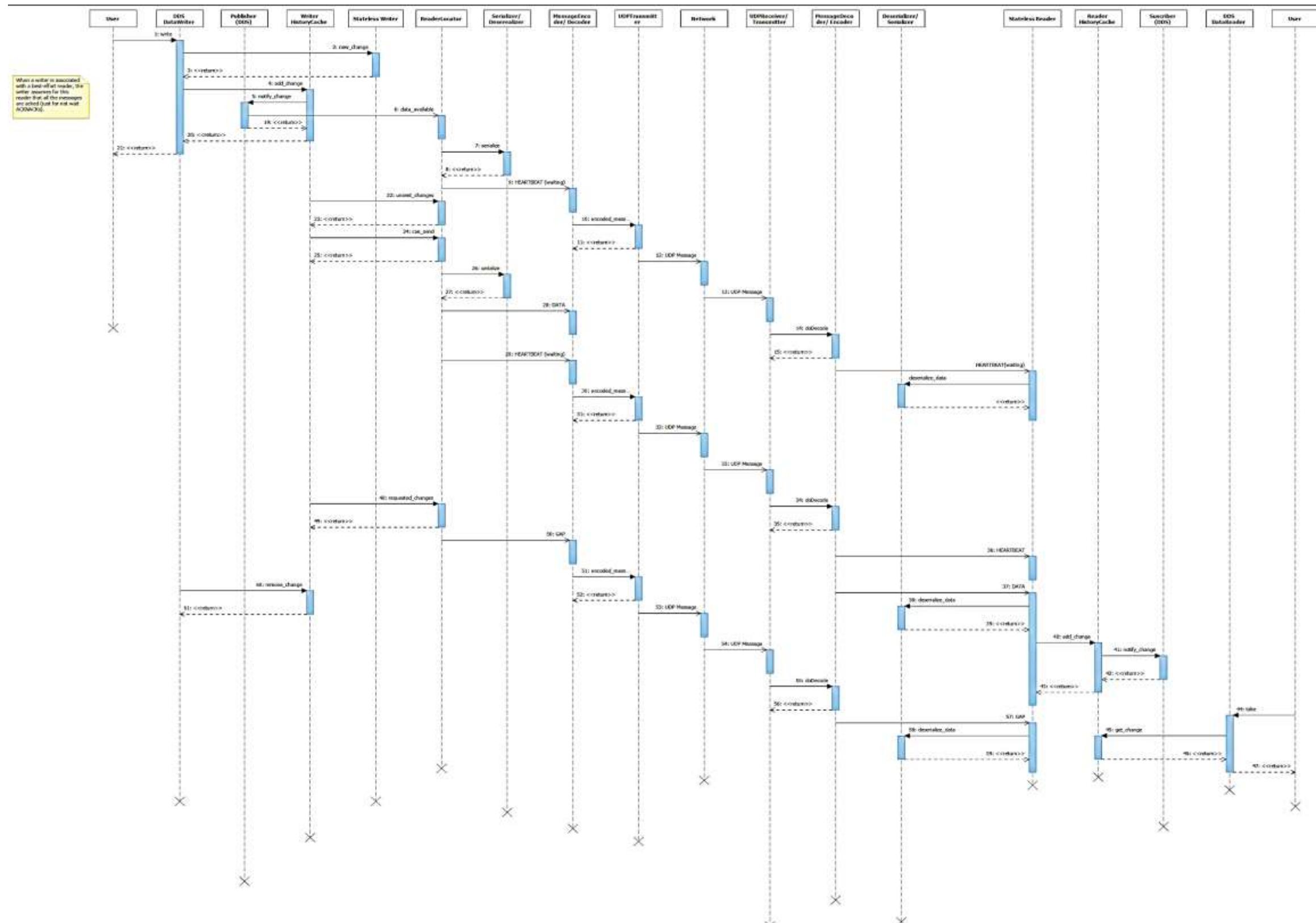


Figura 3-29. Comportamiento Reliable Writer – Best Effort Reader en interacción sin estado.

3.6.3. DIAGRAMAS DE SECUENCIA DE LA INTERACCIÓN DE DDS CON RTPS HÍBRIDOS (CON ESTADO Y SIN ESTADO)

3.6.3.1.1. Reliable Stateless Writer – Reliable Stateful Reader

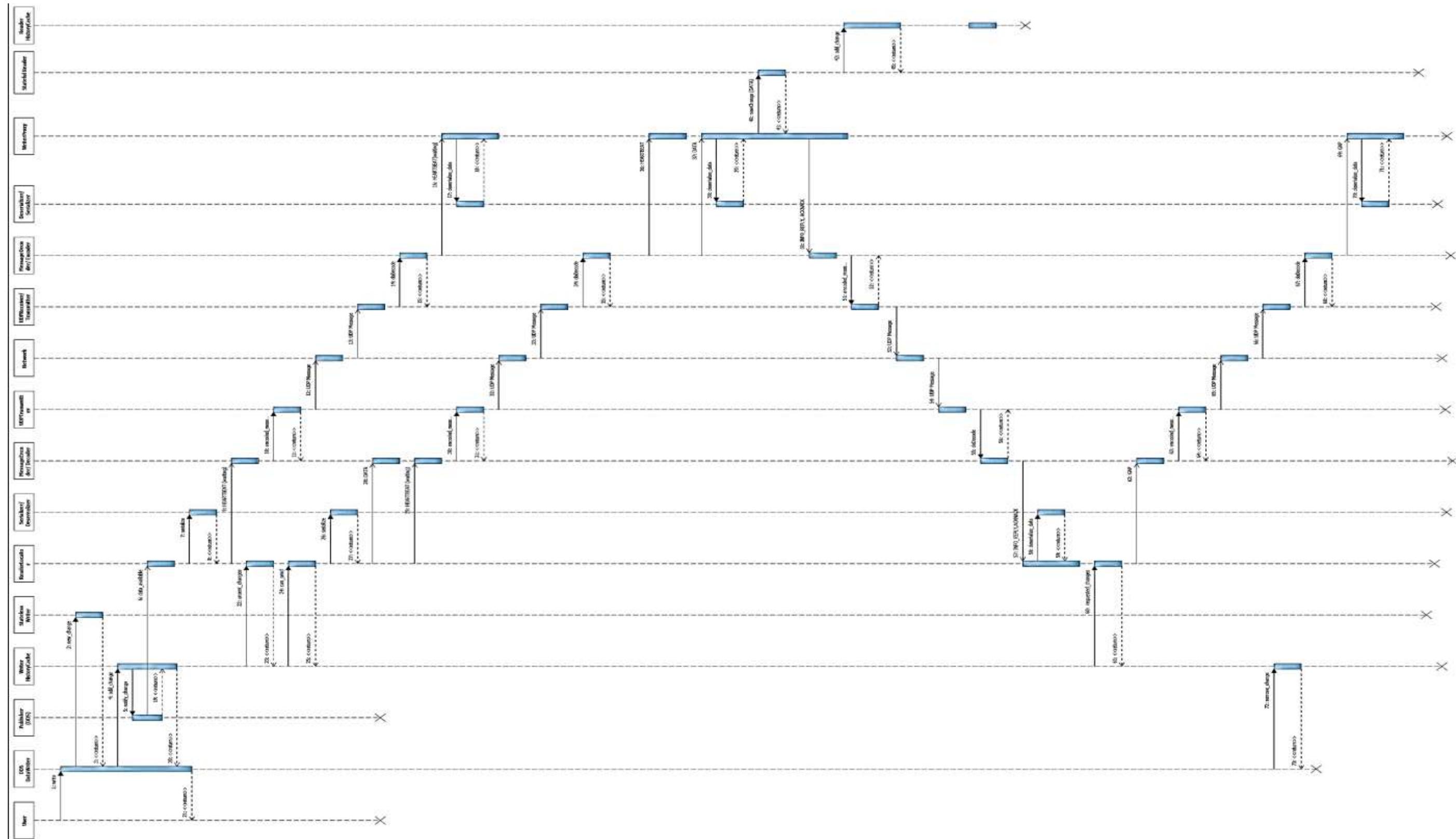


Figura 3-30. Comportamiento Reliable Stateless Writer sin estado – Reliable Stateful Reader con estado.

3.6.4. PROTOCOLO DESCUBRIMIENTO

3.6.4.1. Tráfico de Descubrimiento

3.6.4.1.1. *Fase de Descubrimiento de Participantes.*

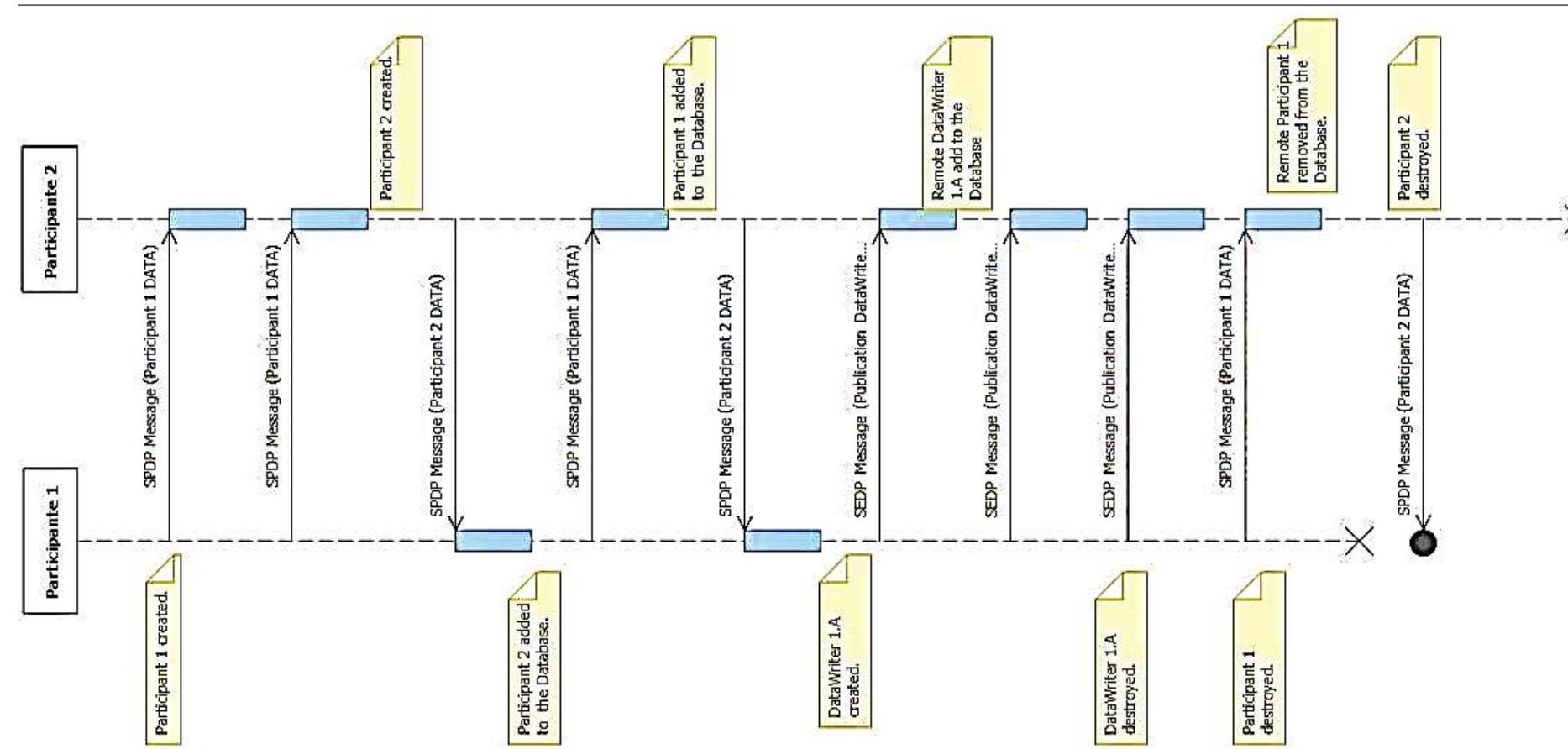


Figura 3-31. Fases de descubrimiento de participantes.

CAPÍTULO 4

PRUEBAS

4.1. INTRODUCCIÓN

En este capítulo primeramente se realizarán pruebas unitarias para la comprobación de distintos componentes del programa tales como clases y métodos, las cuales estarán documentadas mediante una estructura de tablas donde se incluye la llamada, la descripción, la entrada, la referencia, el código y la salida.

A continuación se describirá el ambiente de pruebas, el cual tendrá como base varios computadores comunicándose entre si por medio del protocolo RTPS y además se realizará una prueba con el protocolo RT-CORBA o Ada-DSA o DRTSJ, esta prueba obtiene los paquetes correspondientes al protocolo y verifica el tiempo de respuesta del mismo. Además se adjunta capturas de pantalla tanto de la aplicación utilizando los protocolos y de capturas del flujo de datos con la herramienta wireshark, y se presenta un manual de usuario de las aplicaciones y del protocolo RTPS.

Finalmente, se realizará una comparación midiendo tiempos de respuesta y eficiencia del protocolo dentro de nuestro ambiente de pruebas.

4.2. PRUEBAS UNITARIAS DE LA IMPLEMENTACIÓN

4.2.1. CODIFICADORES

4.2.1.1. Prueba de los Elementos de los Mensajes.

En estas pruebas se verifica el funcionamiento del serializador y deserializador de los elementos del módulo de mensajes y encapsulación; los elementos a comprobar son los siguientes: *Locator*, *ClassWithLocator*, *EntityId*, *ProtocolVersion*, *ClassWithProtocolVersion*.

- *Locator*, representa la información de la ubicación del *endpoint* RTPS para el envío de los mensajes usando una dirección IPv4 y un puerto.
- *ClassWithLocator*, representa la información de la ubicación del *endpoint* RTPS para el envío de los mensajes usando una dirección IPv6 y un puerto.
- *EntityId*, representa al identificador de cada entidad a la cual se envía los mensajes.
- *ProtocolVersion*, representa la versión del protocolo RTPS.

- ClassWithProtocolVersion, es una clase que presenta a la versión del protocolo.

Tabla 4-1. TestLocatorIpV4CDR_BE

Llamada: public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)	
Descripción	En esta prueba se verifica los <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	[TestMethod] public void TestLocatorIpV4CDR_BE() { Encapsulation Scheme = Encapsulation.CDR_BE; int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("10.20.30.40"), 2700); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 00 00 00 00 00 00 00 01 00 00 0A 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 14 1E 28", buffer.GetHexDump()); Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); }
Salida	Nombre de la prueba: <i>TestLocatorIpV4CDR_BE</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:02.9120332

Tabla 4-2. TestLocatorIpV4CDR_LE

Llamada: public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)	
Descripción	En esta prueba se verifica los <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	[TestMethod] public void TestLocatorIpV4CDR_LE() { Encapsulation Scheme = Encapsulation.CDR_LE;

Tabla 4-2. TestLocatorIpV4CDR_LE

	<pre> int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("10.20.30.40"), 2700); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 01 00 00 00 8C 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 14 1E 28", buffer.GetHexDump()); Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV4CDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 00:00:00.1384335</p>

Tabla 4-3. TestLocatorIpV6CDR_BE

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica los <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestLocatorIpV6CDR_BE() { Encapsulation Scheme = Encapsulation.CDR_BE; int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse "::1", 2700); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 00 00 00 00 00 00 00 02 00 00 0A 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01", buffer.GetHexDump()); } </pre>

Tabla 4-3. TestLocatorIpV6CDR_BE

	<pre> Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV6CDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1594283</p>

Tabla 4-4. TestLocatorIpV6CDR_LE

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica los <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestLocatorIpV6CDR_LE() { Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("::1"), 2700); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 02 00 00 00 8C 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01", buffer.GetHexDump()); Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV6CDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1602378</p>

Tabla 4-5. TestLocatorIpV6CDR_BE2

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica los <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorIpV6CDR_BE2() { Encapsulation Scheme = Encapsulation.CDR_BE; int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("FF00:4501:0:0:0:0:0:32"), 2700); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 00 00 00 00 00 00 02 00 00 0A 8C FF 00 45 01 00 00 00 00 00 00 00 00 00 00 00 32", buffer.GetHexDump()); Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV6CDR_BE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1492697</p>

Tabla 4-6. TestLocatorIpV6CDR_LE2

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica los <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorIpV6CDR_LE2() { Encapsulation Scheme = Encapsulation.CDR_LE;</pre>

Tabla 4-6. TestLocatorIpV6CDR_LE2

	<pre> int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("FF00:4501:0:0:0:0:32"), 2700); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 02 00 00 00 8C 0A 00 00 FF 00 45 01 00 00 00 00 00 00 00 00 00 00 00 32", buffer.GetHexDump()); Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV6CDR_LE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1643104</p>

Tabla 4-7. TestLocatorFromSample1

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestLocatorFromSample1() { Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("172.16.0.128"), 36945); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 01 00 00 00 51 90 00 00 00 00 00 00 00 00 00 00 00 00 00 AC 10 00 80", buffer.GetHexDump()); } </pre>

Tabla 4-7. TestLocatorFromSample1

	<pre> Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorFromSample1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.172042</p>

Tabla 4-8. TestLocatorFromSample2

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorFromSample2() { Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("239.255.0.1"), 9652); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 01 00 00 00 B4 25 00 00 00 00 00 00 00 00 00 00 00 00 EF FF 00 01", buffer.GetHexDump()); Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorFromSample2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1519815</p>

Tabla 4-9. *TestLocatorFromSample3*

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorFromSample3() { Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 16 + 4 + 4 + CDRHeaderSize; Locator v1 = new Locator(IPAddress.Parse("127.0.0.1"), 12345); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<Locator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 01 00 00 00 39 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7F 00 00 01", buffer.GetHexDump()); Locator v2 = EncapsulationManager.Deserialize<Locator>(buffer); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorFromSample3</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1650963</p>

Tabla 4-10. *TestLocatorIpV4PL_CDR_BE*

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorIpV4PL_CDR_BE() { Encapsulation Scheme = Encapsulation.PL_CDR_BE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize;</pre>

Tabla 4-10. *TestLocatorIpV4PL_CDR_BE*

	<pre> ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse("10.20.30.40"), 2700) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 02 00 00 00 48 00 18 00 00 00 01 00 00 0A 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 14 1E 28 00 01 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV4PL_CDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1744933</p>

Tabla 4-11. *TestLocatorIpV4PL_CDR_LE*

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestLocatorIpV4PL_CDR_LE() { Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse("10.20.30.40"), 2700) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); } </pre>

Tabla 4-11. *TestLocatorIpV4PL_CDR_LE*

	<pre> Assert.AreEqual("00 03 00 00 48 00 18 00 01 00 00 00 8C 0A 00 00 00 00 00 00 00 00 00 00 00 00 0A 14 1E 28 01 00 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV4PL_CDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1634546</p>

Tabla 4-12. *TestLocatorIpV6PL_CDR_BE*

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestLocatorIpV6PL_CDR_BE() { Encapsulation Scheme = Encapsulation.PL_CDR_BE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse(":1"), 2700) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 02 00 00 00 48 00 18 00 00 00 02 00 00 0A 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV6PL_CDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1639482</p>

Tabla 4-13. *TestLocatorIpV6PL_CDR_LE*

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorIpV6PL_CDR_LE() { Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse("::1"), 2700) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPay load); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 03 00 00 48 00 18 00 02 00 00 00 8C 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 01 00 00 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(bu ffer); Assert.AreEqual(v1.Locator, v2.Locator); }</pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV6PL_CDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1558723</p>

Tabla 4-14. TestLocatorIpV6PL_CDR_BE2

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorIpV6PL_CDR_BE2() { Encapsulation Scheme = Encapsulation.PL_CDR_BE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse("FF00:4501:0:0:0:0:32"), 2700) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 02 00 00 00 48 00 18 00 00 00 02 00 00 0A 8C FF 00 45 01 00 00 00 00 00 00 00 00 00 00 32 00 01 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); }</pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpV6PL_CDR_BE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 00:00:00.1640392</p>

Tabla 4-15. TestLocatorIpV6PL_CDR_LE2

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestLocatorIpV6PL_CDR_LE2()</pre>

Tabla 4-15. *TestLocatorIpv6PL_CDR_LE2*

	<pre> Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse("FF00:4501:0:0:0:0:32"), 2700) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 03 00 00 48 00 18 00 02 00 00 00 8C 0A 00 00 FF 00 45 01 00 00 00 00 00 00 00 32 01 00 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorIpv6PL_CDR_LE2</i></p> <p>Resultado de la prueba: ✓ 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1546355</p>

Tabla 4-16. *TestLocatorFromSample1PL_CDR_LE*

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestLocatorFromSample1PL_CDR_LE() { Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse("172.16.0.128"), 36945) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); </pre>

Tabla 4-16. *TestLocatorFromSample1PL_CDR_LE*

	<pre> Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 03 00 00 48 00 18 00 01 00 00 00 51 90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 AC 10 00 80 01 00 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorFromSample1PL_CDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1686305</p>

Tabla 4-17. *TestLocatorFromSample2PL_CDR_LE*

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestLocatorFromSample2PL_CDR_LE() { Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator() { Locator = new Locator(IPAddress.Parse("239.255.0.1"), 9652) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 03 00 00 48 00 18 00 01 00 00 00 B4 25 00 00 00 00 00 00 00 00 00 00 00 00 00 00 EF FF 00 01 01 00 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); } </pre>
Salida	<p>Nombre de la prueba: <i>TestLocatorFromSample2PL_CDR_LE</i></p>

Tabla 4-17. *TestLocatorFromSample2PL_CDR_LE*

	Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00.1738197
--	--

Tabla 4-18. *TestLocatorFromSample3PL_CDR_LE*

Llamada:	
	public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)
Descripción	En esta prueba se verifica el <i>ClassWithLocator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	[TestMethod] public void TestLocatorFromSample3PL_CDR_LE() { Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 16 + 4 + 4 + PL_CDRHeaderSize; ClassWithLocator v1 = new ClassWithLocator(); Locator = new Locator(IPAddress.Parse("127.0.0.1"), 12345); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithLocator>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 03 00 00 48 00 18 00 01 00 00 00 39 30 00 00 00 00 00 00 00 00 00 00 00 00 7F 00 00 01 01 00 00 00", buffer.GetHexDump()); ClassWithLocator v2 = EncapsulationManager.Deserialize<ClassWithLocator>(buffer); Assert.AreEqual(v1.Locator, v2.Locator); }
Salida	Nombre de la prueba: <i>TestLocatorFromSample3PL_CDR_LE</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00.1984551

Tabla 4-19. *TestGUIDCDR_BE*

Llamada:	
	public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)
Descripción	En esta prueba se verifica el <i>EntityId</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .

Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[public void TestGUIDCDR_BE() { Encapsulation Scheme = Encapsulation.CDR_BE; int bufferSize = 16 + CDRHeaderSize; GUID v1 = new GUID(new GuidPrefix(new byte[] { 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB }),, new EntityId(new byte[] { 0x0, 0x1, 0x2 },, EntityKinds.BUILT_IN_PARTICIPANT)); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<GUID>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPay load); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 00 00 00 00 01 02 03 04 05 06 07 08 09 0A 0B 00 01 02 C1", buffer.GetHexDump()); GUID v2 = EncapsulationManager.Deserialize<GUID>(buffer); Assert.AreEqual(v1, v2); }]</pre>
Salida	<p>Nombre de la prueba: <i>TestGUIDCDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 00:00:00.2030336</p>

Tabla 4-20. *TestGUIDCDR_LE*

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>EntityId</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestGUIDCDR_LE() { Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 16 + CDRHeaderSize; GUID v1 = new GUID(new GuidPrefix(new byte[] { 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB }),,</pre>

Tabla 4-20. TestGUICDR_LE

	<pre> new EntityId(new byte[] { 0x0, 0x1, 0x2 }, EntityKinds.BUILT_IN_PARTICIPANT)); SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<GUID>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 00 01 02 03 04 05 06 07 08 09 0A 0B 00 01 02 C1", buffer.GetHexDump()); GUID v2 = EncapsulationManager.Deserialize<GUID>(buffer); Assert.AreEqual(v1, v2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestGUICDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.2267873</p>

Tabla 4-21. TestGUIDPL_CDR_BE

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>EntityId</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestGUIDPL_CDR_BE() { Encapsulation Scheme = Encapsulation.PL_CDR_BE; int bufferSize = 24 + CDRHeaderSize; ClassWithGUID v1 = new ClassWithGUID() { Key = new GUID(new GuidPrefix(new byte[] { 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB }), new EntityId(new byte[] { 0x0, 0x1, 0x2 }, EntityKinds.BUILT_IN_PARTICIPANT)) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithGUID>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); } </pre>

Tabla 4-21. TestGUIDPL_CDR_BE

	<pre> Assert.AreEqual("00 02 00 00 00 50 00 10 00 01 02 03 04 05 06 07 08 09 0A 0B 00 01 02 C1 00 01 00 00", buffer.GetHexDump()); ClassWithGUID v2 = EncapsulationManager.Deserialize<ClassWithGUID>(buffer); Assert.AreEqual(v1.Key, v2.Key); } </pre>
Salida	<p>Nombre de la prueba: <i>TestGUIDPL_CDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.2057796</p>

Tabla 4-22. TestGUIDPL_CDR_LE

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>EntityId</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestGUIDPL_CDR_LE() { Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 24 + CDRHeaderSize; ClassWithGUID v1 = new ClassWithGUID() { Key = new GUID(new GuidPrefix(new byte[] { 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB })), new EntityId(new byte[] { 0x0, 0x1, 0x2 }, EntityKinds.BUILT_IN_PARTICIPANT) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithGUID>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 03 00 00 50 00 10 00 00 01 02 03 04 05 06 07 08 09 0A 0B 00 01 02 C1 01 00 00 00", buffer.GetHexDump()); ClassWithGUID v2 = EncapsulationManager.Deserialize<ClassWithGUID>(buffer); Assert.AreEqual(v1.Key, v2.Key); } </pre>
Salida	<p>Nombre de la prueba: <i>TestGUIDPL_CDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p>

Tabla 4-22. TestGUIDPL_CDR_LE

	Duración de la prueba: 0:00:00.2031448
--	--

Tabla 4-23. TestVendorIdCDR_BE

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>EntityId</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestVendorIdCDR_BE() { Encapsulation Scheme = Encapsulation.CDR_BE; int bufferSize = 2 + CDRHeaderSize; VendorId v1 = VendorId.Doopec; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<VendorId>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 00 00 00 01 0F", buffer.GetHexDump()); VendorId v2 = EncapsulationManager.Deserialize<VendorId>(buffer); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: TestVendorIdCDR_BE</p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1845436</p>

Tabla 4-24. TestVendorIdCDR_LE

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>EntityId</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestVendorIDCDR_LE()</pre>

Tabla 4-24. TestVendorIdCDR_LE

	<pre>{ Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 2 + CDRHeaderSize; VendorId v1 = VendorId.Doopec; ; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<VendorId>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 01 0F", buffer.GetHexDump()); VendorId v2 = EncapsulationManager.Deserialize<VendorId>(buffer); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestVendorIdCDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1874632</p>

Tabla 4-25. TestProtocolVersionCDR_BE

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>Protocol/Version</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestProtocolVersionCDR_BE() { Encapsulation Scheme = Encapsulation.CDR_BE; int bufferSize = 2 + CDRHeaderSize; ProtocolVersion v1 = ProtocolVersion.PROTOCOLVERSION_2_1; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ProtocolVersion>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 00 00 00 02 01", buffer.GetHexDump()); ProtocolVersion v2 = EncapsulationManager.Deserialize<ProtocolVersion>(buffer);</pre>

Tabla 4-25. TestProtocolVersionCDR_BE

	<pre> Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestProtocolVersionCDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.2274154</p>

Tabla 4-26. TestProtocolVersionCDR_LE

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>Protocol/Version</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestProtocolVersionCDR_LE() { Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 2 + CDRHeaderSize; ProtocolVersion v1 = ProtocolVersion.PROTOCOLVERSION_2_1; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ProtocolVersion>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 02 01", buffer.GetHexDump()); ProtocolVersion v2 = EncapsulationManager.Deserialize<ProtocolVersion>(buffer); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestProtocolVersionCDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1825883</p>

Tabla 4-27. *TestProtocolVersionPL_CDR_BE*

Llamada: <pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>	
Descripción	En esta prueba se verifica el <i>ClassWithProtocolVersion</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestProtocolVersionPL_CDR_BE() { Encapsulation Scheme = Encapsulation.PL_CDR_BE; int bufferSize = 12 + CDRHeaderSize; ClassWithProtocolVersion v1 = new ClassWithProtocolVersion() { ProtocolVersion = ProtocolVersion.PROTOCOLVERSION_2_1 }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithProtocolVersion>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 02 00 00 00 15 00 04 02 01 00 00 00 01 00 00", buffer.GetHexDump()); ClassWithProtocolVersion v2 = EncapsulationManager.Deserialize<ClassWithProtocolVersion>(bu ffer); Assert.AreEqual(v1.ProtocolVersion, v2.ProtocolVersion); }</pre>
Salida	<p>Nombre de la prueba: <i>TestProtocolVersionPL_CDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.200578</p>

Tabla 4-28. *TestProtocolVersionPL_CDR_LE*

Llamada: <pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>	
Descripción	En esta prueba se verifica el <i>ClassWithProtocolVersion</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .

Tabla 4-28. TestProtocolVersionPL_CDR_LE

Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestProtocolVersionPL_CDR_LE() { Encapsulation Scheme = Encapsulation.PL_CDR_LE; int bufferSize = 12 + CDRHeaderSize; ClassWithProtocolVersion v1 = new ClassWithProtocolVersion() { ProtocolVersion = ProtocolVersion.PROTOCOLVERSION_2_1 }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<ClassWithProtocolVersion>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 03 00 00 15 00 04 00 02 01 00 00 01 00 00 00", buffer.GetHexDump()); ClassWithProtocolVersion v2 = EncapsulationManager.Deserialize<ClassWithProtocolVersion>(bu ffer); Assert.AreEqual(v1.ProtocolVersion, v2.ProtocolVersion); }</pre>
Salida	<p>Nombre de la prueba: <i>TestProtocolVersionPL_CDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.200578</p>

Tabla 4-29. TestListLocatorCDR_BE

Llamada:	<pre>public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length)</pre>
Descripción	En esta prueba se verifica el <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre>[TestMethod] public void TestListLocatorCDR_BE() { Encapsulation Scheme = Encapsulation.CDR_BE; int bufferSize = 16 + 4 + 4 + 4 + CDRHeaderSize; List<Locator> v1 = new List<Locator>() { new Locator(IPAddress.Parse("10.20.30.40"), 2700) };</pre>

Tabla 4-29. *TestListLocatorCDR_BE*

	<pre> SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<List<Locator>>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 00 00 00 00 00 00 00 01 00 00 00 01 00 00 0A 8C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 14 1E 28", buffer.GetHexDump()); List<Locator> v2 = EncapsulationManager.Deserialize<List<Locator>>(buffer); Assert.AreEqual(v1.Count, v2.Count); Assert.AreEqual(v1[0], v2[0]); } </pre>
Salida	<p>Nombre de la prueba: <i>TestListLocatorCDR_BE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.200578</p>

Tabla 4-30. *TestListLocatorCDR_LE*

Llamada:	<pre> public static DataEncapsulation Serialize<T>(T obj, Encapsulation scheme = Encapsulation.CDR_BE) public static DataEncapsulation Deserialize(IoBuffer buffer, int length) </pre>
Descripción	En esta prueba se verifica el <i>Locator</i> con el <i>Serializador</i> y el <i>Deserealizador</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataEncapsulation</i> .
Código	<pre> [TestMethod] public void TestListLocatorCDR_LE() { Encapsulation Scheme = Encapsulation.CDR_LE; int bufferSize = 16 + 4 + 4 + 4 + CDRHeaderSize; List<Locator> v1 = new List<Locator>() { new Locator(IPAddress.Parse("10.20.30.40"), 2700) }; SerializedPayload payload = new SerializedPayload(); payload.DataEncapsulation = EncapsulationManager.Serialize<List<Locator>>(v1, Scheme); IoBuffer buffer = IoBuffer.Wrap(payload.DataEncapsulation.SerializedPayload); Assert.AreEqual(bufferSize, buffer.Remaining); Assert.AreEqual("00 01 00 00 01 00 00 00 01 00 00 00 8C 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0A 14 1E 28", buffer.GetHexDump()); } </pre>

Tabla 4-30. *TestListLocatorCDR_LE*

	<pre> List<Locator> v2 = EncapsulationManager.Deserialize<List<Locator>>(buffer); Assert.AreEqual(v1.Count, v2.Count); Assert.AreEqual(v1[0], v2[0]); } </pre>
Salida	<p>Nombre de la prueba: <i>TestListLocatorCDR_LE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.2074474</p>

4.2.1.2. Prueba de Mensajes

En estas pruebas se verifica el funcionamiento de todos los submensajes, comprobando que se pueda escribir y leer el submensaje correctamente y verificando que no haya pérdida de información.

Tabla 4-31. *TestInfoDestination*

Llamada:	<pre> public InfoDestination(GuidPrefix guidPrefix) </pre>
Descripción	En esta prueba se verifica el submensaje <i>InfoDestination</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>InfoDestination</i> .
Código	<pre> [TestMethod] public void TestInfoDestination() { // Create a Message with InfoDestination Message m1 = new Message(); m1.SubMessages.Add(new InfoDestination(GuidPrefix.GUIDPREFIX_UNKNOWN)); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestInfoDestination</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1653231</p>

Tabla 4-32. TestInfoSource

Llamada:	<pre>public InfoSource(GuidPrefix guidPrefix)</pre>
Descripción	En esta prueba se verifica el submensaje <i>InfoSource</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>InfoDestination</i> .
Código	<pre>[TestMethod] public void TestInfoSource() { // Create a Message with InfoSource Message m1 = new Message(); m1.SubMessages.Add(new InfoSource(GuidPrefix.GUIDPREFIX_UNKNOWN)); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestInfoSource</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1801263</p>

Tabla 4-33. TestInfoReply

Llamada:	<pre>public InfoReply(IList<Locator> unicastLocators, IList<Locator> multicastLocators)</pre>
Descripción	En esta prueba se verifica el submensaje <i>InfoReply</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>InfoReply</i> .
Código	<pre>[TestMethod] public void TestInfoReply() { // Create a Message with InfoReply Message m1 = new Message(); Locator loc1 = new Locator(IPAddress.Loopback, 7111); Locator loc2 = new Locator(IPAddress.Loopback, 7222); IList<Locator> unicastLocators = new List<Locator>(); unicastLocators.Add(loc1);</pre>

Tabla 4-33. TestInfoReply

	<pre> IList<Locator> multicastLocators = new List<Locator>(); multicastLocators.Add(loc2); m1.SubMessages.Add(new InfoReply(unicastLocators, multicastLocators)); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestInfoReply</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.2084802</p>

Tabla 4-34. TestInfoReplyIp4

Llamada:	<pre> public InfoReplyIp4(LocatorUDPV4 unicastLocator, LocatorUDPV4 multicastLocator) </pre>
Descripción	En esta prueba se verifica el submensaje <i>InfoReplyIp4</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>InfoReplyIp4</i> .
Código	<pre> [TestMethod] public void TestInfoReplyIp4() { // Create a Message with InfoReplyIp4 Message m1 = new Message(); LocatorUDPV4 lc1 = new LocatorUDPV4(IPAddress.Loopback, 7111); LocatorUDPV4 lc2 = LocatorUDPV4.LOCATORUDPV4_INVALID; m1.SubMessages.Add(new InfoReplyIp4(lc1, lc2)); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal </pre>

Tabla 4-34. *TestInfoReplyIp4*

	AssertArrayEquals(bytes1, bytes2); }
Salida	Nombre de la prueba: <i>TestInfoReplyIp4</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00.169154

Tabla 4-35. *TestInfoTimestamp*

Llamada: <code>public InfoTimestamp(long systemCurrentMillis)</code>	
Descripción	En esta prueba se verifica el submensaje <i>InfoTimestamp</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>InfoTimestamp</i> .
Código	<pre>[TestMethod] public void TestInfoTimestamp() { // Create a Message with InfoDestination Message m1 = new Message(); m1.SubMessages.Add(new InfoTimestamp(123)); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); }</pre>
Salida	Nombre de la prueba: <i>TestInfoTimestamp</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00.1588509

Tabla 4-36. *TestGAP*

Llamada: <code>public GAP()</code>	
Descripción	En esta prueba se verifica el submensaje <i>GAP</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>GAP</i> .
Código	<pre>[TestMethod] public void TestGAP() { // Create a Message with GAP</pre>

Tabla 4-36. TestGAP

	<pre> Message m1 = new Message(); GAP GAP = new GAP(); EntityId id1 = EntityId.ENTITYID_UNKNOWN; EntityId id2 = EntityId.ENTITYID_UNKNOWN; GAP.ReaderId = id1; GAP.WriterId = id2; GAP.GAPStart = new SequenceNumber(10); GAP.GAPList = new SequenceNumberSet(10, new int[] { 12, 15, 19 }); m1.SubMessages.Add(GAP); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestGAP</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.166937</p>

Tabla 4-37. TestAckNack

Llamada: public AckNack()	
Descripción	En esta prueba se verifica el submensaje <i>AckNack</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>AckNack</i> .
Código	<pre> [TestMethod] public void TestAckNack() { // Create a Message with AckNack Message m1 = new Message(); AckNack ackNack = new AckNack(); EntityId id1 = EntityId.ENTITYID_UNKNOWN; EntityId id2 = EntityId.ENTITYID_UNKNOWN; ackNack.ReaderId = id1; ackNack.WriterId = id2; ackNack.ReaderSNSState = new SequenceNumberSet(10, new int[] { 12, 15, 19 }); ackNack.Count = 10; m1.SubMessages.Add(ackNack); } </pre>

Tabla 4-37. TestAckNack

	<pre> // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestAckNack</i></p> <p>Resultado de la prueba:</p>  1 Prueba superada <p>Duración de la prueba: 0:00:00.1699033</p>

Tabla 4-38. TestPad

Llamada: public Pad()	
Descripción	En esta prueba se verifica el submensaje <i>Pad</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>Pad</i> .
Código	<pre> [TestMethod] public void TestPad() { // Create a Message with Pad Message m1 = new Message(); Pad pad = new Pad(); pad.Bytes = new byte[] { 12, 15, 19 }; m1.SubMessages.Add(pad); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestPad</i></p> <p>Resultado de la prueba:</p>  1 Prueba superada <p>Duración de la prueba: 0:00:00.1486317</p>

Tabla 4-39. *TestHeartbeat*

Llamada: public Heartbeat()	
Descripción	En esta prueba se verifica el submensaje <i>Heartbeat</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>Heartbeat</i> .
Código	<pre>[TestMethod] public void TestHeartbeat() { // Create a Message with Heartbeat Message m1 = new Message(); Heartbeat heartbeat = new Heartbeat(); EntityId id1 = EntityId.ENTITYID_UNKNOWN; EntityId id2 = EntityId.ENTITYID_UNKNOWN; heartbeat.readerId = id1; heartbeat.writerId = id2; heartbeat.firstSN = new SequenceNumber(10); heartbeat.lastSN = new SequenceNumber(20); heartbeat.count = 5; m1.SubMessages.Add(heartbeat); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestHeartbeat</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1949603</p>

Tabla 4-40. *TestNackFrag*

Llamada: public NackFrag()	
Descripción	En esta prueba se verifica el submensaje <i>NackFrag</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>NackFrag</i> .
Código	<pre>[TestMethod] public void TestNackFrag() { // Create a Message with Heartbeat Message m1 = new Message();</pre>

Tabla 4-40. TestNackFrag

	<pre> NackFrag nackFrag = new NackFrag(); EntityId id1 = EntityId.ENTITYID_UNKNOWN; EntityId id2 = EntityId.ENTITYID_UNKNOWN; nackFrag.ReaderId = id1; nackFrag.WriterId = id2; nackFrag.FragmentNumberState = new SequenceNumberSet(5, new int[] { 6, 7, 21 }); nackFrag.WriterSequenceNumber = new SequenceNumber(20); nackFrag.Count = 2; m1.SubMessages.Add(nackFrag); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestNackFrag</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.2558301</p>

Tabla 4-41. TestHeartbeatFrag

Llamada:	
	<pre>public HeartbeatFrag()</pre>
Descripción	En esta prueba se verifica el submensaje <i>HeartbeatFrag</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>HeartbeatFrag</i> .
Código	<pre> /// <summary> /// Tests, that reading and writing of HeartbeatFrag is symmetrical. /// </summary> [TestMethod] public void TestHeartbeatFrag() { // Create a Message with Heartbeat Message m1 = new Message(); HeartbeatFrag heartbeatFrag = new HeartbeatFrag(); EntityId id1 = EntityId.ENTITYID_UNKNOWN; EntityId id2 = EntityId.ENTITYID_UNKNOWN; heartbeatFrag.ReaderId = id1; heartbeatFrag.WriterId = id2; heartbeatFrag.WriterSequenceNumber = new SequenceNumber(10); } </pre>

Tabla 4-41. TestHeartbeatFrag

	<pre> heartbeatFrag.LastFragmentNumber = 30; heartbeatFrag.Count = 50; m1.SubMessages.Add(heartbeatFrag); // Write Message to bytes1 array byte[] bytes1 = Write(m1); // Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal Assert.AreEqual(bytes1, bytes2); } </pre>
Salida	<p>Nombre de la prueba: <i>TestHeartbeatFrag</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1587616</p>

Tabla 4-42. TestDataFrag

Llamada: <code>public DataFrag()</code>	
Descripción	En esta prueba se verifica el submensaje <i>DataFrag</i> .
Entrada	Inicialmente no existe ninguna instancia del <i>DataFrag</i> .
Código	<pre> /// <summary> /// Tests, that reading and writing of DataFrag is symmetrical. /// </summary> [TestMethod] public void TestDataFrag() { // Create a Message with DataFrag Message m1 = new Message(); DataFrag dataFrag = new DataFrag(); EntityId id1 = EntityId.ENTITYID_UNKNOWN; EntityId id2 = EntityId.ENTITYID_UNKNOWN; dataFrag.ReaderId = id1; dataFrag.WriterId = id2; dataFrag.WriterSequenceNumber = new SequenceNumber(10); dataFrag.FragmentStartingNumber = 30; dataFrag.FragmentsInSubmessage = 1; dataFrag.FragmentSize = 4; dataFrag.SerializedPayload = new byte[] { 100, 10, 1, 0 }; m1.SubMessages.Add(dataFrag); // Write Message to bytes1 array byte[] bytes1 = Write(m1); </pre>

Tabla 4-42. *TestDataFrag*

	<pre>// Read from bytes1 array - tests reading Message m2 = Read(bytes1); // Write the message Read to bytes2 byte[] bytes2 = Write(m2); // Test, that bytes1 and bytes2 are equal AssertArrayEquals(bytes1, bytes2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestDataFrag</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00.1957048</p>

4.2.2. TRANSPORTE

4.2.2.1. Prueba de Detección de paquetes RTPS.

En esta prueba se verifica el funcionamiento de los UDP *Receiver*, por medio de la generación de mensajes RTPS.

Tabla 4-43. *TestPublishData*

Llamada: public UDPReceiver(Uri uri, int bufferSize)	
Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS, a los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert
Entrada	Inicialmente no se tiene inicializado al <i>Receiver UDP</i>
Código	<pre>[TestMethod] public void TestPublishData() { object key = new object(); UDPReceiver rec = new UDPReceiver(new Uri("udp://" + + Host + ":" + Port), 1024); rec.MessageReceived += (s, m) => { Message msg = m.Message; Debug.WriteLine("New Message has arrived from {0}", m.Session.RemoteEndPoint); Debug.WriteLine("Message Header: {0}", msg.Header); Assert.AreEqual(ProtocolId.PROTOCOL_RTPS, msg.Header.Protocol); Assert.AreEqual(VendorId.OCI, msg.Header.VendorId); Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1, msg.Header.Version); Assert.AreEqual(2, msg.SubMessages.Count); }; }</pre>

Tabla 4-43. TestPublishData

4.2.2.2. Pruebas de paquetes RTPS.

En estas pruebas se verifica el funcionamiento de los UDP *Receiver*, pero a diferencia de la prueba anterior en estas se utilizan transmisores de otros fabricantes, es decir se comprueba que los receptores pueden recibir de una manera adecuada los mensajes RTPS. El fabricante que se utiliza para las pruebas corresponde a *OpenDDS*.

Tabla 4-44. TestPublishPacket2

Llamada: public UDPReceiver(Uri uri, int bufferSize)	
Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS de otros vendors con los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert
Entrada	Inicialmente no se tiene inicializado al <i>Receiver UDP</i>
Código	<pre>[TestMethod] public void TestPublishPacket2() { object key = new object(); UDPReceiver rec = new UDPReceiver(new Uri("udp://" + + Host + ":" + Port), 1024); rec.MessageReceived += (s, m) => { Message msg = m.Message; Debug.WriteLine("New Message has arrived from {0}", m.Session.RemoteEndPoint); Debug.WriteLine("Message Header: {0}", msg.Header); Assert.AreEqual(ProtocolId.PROTOCOL_RTPS, msg.Header.Protocol); Assert.AreEqual(VendorId.OCI, msg.Header.VendorId); Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1, msg.Header.Version); Assert.AreEqual(2, msg.SubMessages.Count); foreach (var submsg in msg.SubMessages) { Debug.WriteLine("SubMessage: {0}", submsg); switch (submsg.Kind) { case SubMessageKind.DATA: Data d = submsg as Data; foreach (var par in d.InlineQos.Value) Debug.WriteLine("InlineQos: {0}", par); break; case SubMessageKind.INFO_TS: InfoTimestamp its = submsg as InfoTimestamp; Debug.WriteLine("The TimeStampFlag value state is: {0}", its.HasInvalidateFlag); } } }; } }</pre>

Tabla 4-44. *TestPublishPacket2*

	<pre> Debug.WriteLine("The EndiannessFlag value state is: {0}", its.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", its.Header.SubMessageLength); if (its.HasValueFlag == false) { Debug.WriteLine("The Timestamp value is: {0}", its.TimeStamp); } break; default: Assert.Fail("Only Timestamp and Data submessages are expected"); break; } } lock (key) Monitor.Pulse(key); }; rec.Start(); simulator.SendUDPPacket("SamplePackets/packet3.dat", Host, Port); lock (key) { Assert.IsTrue(Monitor.Wait(key, 10000), "Time- out. Message has not arrived or there is an error on it."); } rec.Close(); } </pre>
Salida	<p>Nombre de la prueba: <i>TestPublishPacket2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2431361</p> <p>Salida estándar de Result:</p> <p>Trace du débogage :</p> <pre>no configuration section <common/logging> found - suppressing logging output Sent 196/196 bytes to 224.0.1.111:7400 New Message has arrived from 172.30.82.26:63179 Message Header: [RTPS, 2.1, 01-03, 01-03-00-00-01- 23-45-67-89-AB-CD-EF] SubMessage: InfoTimestamp:header[9, 1, 8], 01/01/1900 00:00:00 [0:0] The TimeStampFlag value state is: False The EndiannessFlag value state is: True The octetsToNextHeader value is: 8 The Timestamp value is: 01/01/1900 00:00:00 [0:0] SubMessage: Data:header[21, 11, 0], Payload[Rtps.Messages.Submessages.Elements.SerializedP ayload]</pre>

Tabla 4-44. TestPublishPacket2

	InlineQos: ParameterId=PID_STATUS_INFO, Content=00-00-00-01 InlineQos: ParameterId=PID_TOPIC_NAME, Content=0A-00-00-00-4D-79-20-54-6F-70-69-63-20-00-00-00 InlineQos: ParameterId=PID_PRESENTATION, Content=E7-03-00-00-00-00-00-00-00 InlineQos: ParameterId=PID_PARTITION, Content=01-00-00-06-00-00-00-48-65-6C-6C-6F-00-00-00 InlineQos: ParameterId=PID_OWNERSHIP_STRENGTH, Content=0C-00-00-00 InlineQos: ParameterId=PID_LIVELINESS, Content=02-00-00-00-FF-FF-FF-7F-FF-FF-FF-7F InlineQos: ParameterId=PID_RELIABILITY, Content=00-00-00-00-00-00-00-00-00-E1-F5-05 InlineQos: ParameterId=PID_TRANSPORT_PRIORITY, Content=0D-00-00-00 InlineQos: ParameterId=PID_LIFESPAN, Content=0E-00-00-00-FF-FF-FF-7F InlineQos: ParameterId=PID_DESTINATION_ORDER, Content=01-00-00-00 InlineQos: ParameterId=PID_SENTINEL, Content=
--	---

Tabla 4-45. GeneralRTSPMessageTesterMethod

Llamada:	public UDPReceiver(Uri uri, int bufferSize)
Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS de otros vendors con los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert. De esta prueba se pueden derivar otras.
Entrada	Inicialmente no se tiene inicializado al Receiver UDP

Tabla 4-45. GeneralRTPSMessageterMethod

	<pre> Assert.AreEqual(ProtocolId.PROTOCOL_RTPS, msg.Header.Protocol); Debug.WriteLine("The Header Protocol is: {0}", msg.Header.Protocol); Assert.AreEqual(VendorId.OCI, msg.Header.VendorId); Debug.WriteLine("The VendorId value state is: {0}", msg.Header.VendorId); Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1, msg.Header.Version); Debug.WriteLine("The Protocol Version value state is: {0}", msg.Header.Version); Debug.WriteLine("The number of SubMessages in the message is: {0}", msg.SubMessages.Count); //Assert.AreEqual(2, msg.SubMessages.Count); foreach (var submsg in msg.SubMessages) { Debug.WriteLine("SubMessage: {0}", submsg.Kind); switch (submsg.Kind) { case SubMessageKind.DATA: { Data d = submsg as Data; Debug.WriteLine("The KeyFlag value state is: {0}", d.HasKeyFlag); Debug.WriteLine("The DataFlag value state is: {0}", d.HasDataFlag); Debug.WriteLine("The InlineQoSFlag value state is: {0}", d.HasInlineQosFlag); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The extraFlags value is: {0}", d.ExtraFlags.Value); Debug.WriteLine("The octetsToInlineQos value is: Aun no logro"); Debug.WriteLine("The readerID is: {0}", d.ReaderId); Debug.WriteLine("The writerID is: {0}", d.WriterId); Debug.WriteLine("The writerSN is: {0}", d.WriterSN); if (d.HasInlineQosFlag) { foreach (var par in d.InlineQos.Value) { Debug.WriteLine("InlineQos: {0}", par); } } } } } </pre>
--	---

Tabla 4-45. GeneralRTPSMessageTesterMethod

	<pre> if (d.HasDataFlag d.Header.Flags.IsLittleEndian) { for (int i = 0; i <= d.SerializedPayload.DataEncapsulation.SerializedPayload.Length - 1; i++) { Debug.WriteLine("SerializedPayload: {0}", d.SerializedPayload.DataEncapsulation.SerializedPayload.GetValue(i)); } break; } case SubMessageKind.ACKNACK: { AckNack d = submsg as AckNack; Debug.WriteLine("The FinalFlag value state is: {0}", d.HasFinalFlag); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The readerID is: {0}", d.ReaderId); Debug.WriteLine("The writerID is: {0}", d.WriterId); Debug.WriteLine("The readerSNState is: {0}", d.ReaderSNState); Debug.WriteLine("The Count is: {0}", d.Count); break; } case SubMessageKind.NACK_FRAG: { NackFrag d = submsg as NackFrag; Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The readerID is: {0}", d.ReaderId); Debug.WriteLine("The writerID is: {0}", d.WriterId); Debug.WriteLine("The writerSN is: {0}", d.WriterSequenceNumber); Debug.WriteLine("The fragmentNumberState value is: {0}", d.FragmentNumberState); break; } case SubMessageKind.DATA_FRAG: { </pre>
--	---

Tabla 4-45. GeneralRTPSMessageTesterMethod

	<pre> DataFrag d = submsg as DataFrag; Debug.WriteLine("The KeyFlag value state is: {0}", d.HasKeyFlag); Debug.WriteLine("The InlineQoSFlag value state is: {0}", d.HasInlineQosFlag); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The extraFlags value is: {0}", d.ExtraFlags); Debug.WriteLine("The octetsToInlineQos value is: Aun no logro"); Debug.WriteLine("The readerID is: {0}", d.ReaderId); Debug.WriteLine("The writerID is: {0}", d.WriterId); Debug.WriteLine("The writerSN is: {0}", d.WriterSequenceNumber); Debug.WriteLine("The FragmentNumber is: {0}", d.FragmentStartingNumber); Debug.WriteLine("The fragmentsInSubmessage is: {0}", d.FragmentsInSubmessage); Debug.WriteLine("The samplesize is: {0}", d.SampleSize); if (d.HasInlineQosFlag) { foreach (var par in d.ParameterList.Value) { Debug.WriteLine("InlineQos: {0}", par); } } for (int i = 0; i <= d.SerializedPayload.Length - 1; i++) { Debug.WriteLine("SerializedPayload: {0}", d.SerializedPayload.GetValue(i)); } break; } case SubMessageKind.GAP: { GAP d = submsg as GAP; Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The readerID is: {0}", d.ReaderId); Debug.WriteLine("The writerID is: {0}", d.WriterId); } </pre>
--	--

Tabla 4-45. GeneralRTPSMessageTesterMethod

	<pre> Debug.WriteLine("The GAPStart number is: {0}", d.GAPStart); Debug.WriteLine("The GAPList value is: {0}", d.GAPList); break; } case SubMessageKind.HEARTBEAT: { Heartbeat d = submsg as Heartbeat; Debug.WriteLine("The LivelinessFlag value state is: {0}", d.HasLivelinessFlag); Debug.WriteLine("The FinalFlag value state is: {0}", d.HasFinalFlag); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The readerID is: {0}", d.ReaderId); Debug.WriteLine("The writerID is: {0}", d.WriterId); Debug.WriteLine("The firstSN is: {0}", d.FirstSequenceNumber); Debug.WriteLine("The lastSN is: {0}", d.LastSequenceNumber); Debug.WriteLine("The Count is: {0}", d.Count); break; } case SubMessageKind.HEARTBEAT_FRAG: { HeartbeatFrag d = submsg as HeartbeatFrag; Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The readerID is: {0}", d.ReaderId); Debug.WriteLine("The writerID is: {0}", d.WriterId); Debug.WriteLine("The writerSN is: {0}", d.WriterSequenceNumber); Debug.WriteLine("The FragmentNumber is: {0}", d.LastFragmentNumber); Debug.WriteLine("The Count is: {0}", d.Count); break; } case SubMessageKind.INFO_DST: { InfoDestination d = submsg as InfoDestination; </pre>
--	--

Tabla 4-45. GeneralRTPSMessageTesterMethod

	<pre> Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The guidPrefix value is: {0}", d.GuidPrefix); break; } case SubMessageKind.INFO_TS: { InfoTimestamp d = submsg as InfoTimestamp; Debug.WriteLine("The TimeStampFlag value state is: {0}", d.HasInvalidateFlag); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); if (d.HasInvalidateFlag == false) { Debug.WriteLine("The Timestamp value is: {0}", d.TimeStamp); } break; } case SubMessageKind.INFO_SRC: { InfoSource d = submsg as InfoSource; Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The ProtocolVersion value is: {0}", d.ProtocolVersion); Debug.WriteLine("The vendorId value is: {0}", d.VendorId); Debug.WriteLine("The guidPrefix value is: {0}", d.GuidPrefix); break; } case SubMessageKind.INFO_REPLY: { InfoReply d = submsg as InfoReply; Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); } } </pre>
--	--

Tabla 4-45. GeneralRTPSMessageTesterMethod

	<pre> Debug.WriteLine("The MulticastFlag value state is: {0}", d.HasMulticastFlag); Debug.WriteLine("The unicastLocatorList value state is: {0}", d.UnicastLocatorList); if (d.HasMulticastFlag) { Debug.WriteLine("The multicastLocatorList value state is: {0}", d.MulticastLocatorList); } break; } case SubMessageKind.INFO_REPLY_IP4: { InfoReplyIp4 d = submsg as InfoReplyIp4; Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Debug.WriteLine("The MulticastFlag value state is: {0}", d.HasMulticastFlag); Debug.WriteLine("The unicastLocatorList value state is: {0}", d.UnicastLocator); if (d.HasMulticastFlag) { Debug.WriteLine("The multicastLocatorList value state is: {0}", d.MulticastLocator); } break; } case SubMessageKind.PAD: { Pad d = submsg as Pad; Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); break; } } lock (key) Monitor.Pulse(key); }; rec.Start(); simulator.SendUDPPacket("SamplePackets/TestOpenDDS_rtps_reliability_runttest_local/Packet04.dat", Host, Port); lock (key) { Assert.IsTrue(Monitor.Wait(key, 10000), "Time- out. Message has not arrived or there is an error on it."); } </pre>
--	--

Tabla 4-45. GeneralRTPSMessageTesterMethod

	<pre> rec.Close(); } </pre>
Salida	<p>Nombre de la prueba: <i>GeneralRTPSMessageTesterMethod</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2446508</p> <p>Nombre de la prueba: <i>GeneralRTPSMessageTesterMethod</i></p> <p>Resultado de la prueba: Superada</p> <p>Salida estándar de Result:</p> <p>Trace du débogage :</p> <pre> no configuration section <common/logging> found - suppressing logging output </pre> <p>Sent 52/52 bytes to 224.0.1.111:7400</p> <p>New Message has arrived from 172.30.82.26:54070</p> <p>Message Header: [RTPS, 2.1, 01-03, 01-03-08-00-27- B9-29-47-0A-AF-00-00]</p> <p>The Header Protocol is: RTPS</p> <p>The VendorId value state is: 01-03</p> <p>The Protocol Version value state is: 2.1</p> <p>The number of SubMessages in the message is: 1</p> <p>SubMessage: DATA</p> <p>The KeyFlag value state is: False</p> <p>The DataFlag value state is: True</p> <p>The InlineQoSFlag value state is: False</p> <p>The EndiannessFlag value state is: True</p> <p>The octetsToNextHeader value is: 0</p> <p>The extraFlags value is: 0</p> <p>The octetsToInlineQos value is: Aun no logro</p> <p>The readerID is: 0-USER_DEFINED_UNKNOWN</p> <p>The writerID is: 3-USER_DEFINED_WRITER_W_KEY</p> <p>The writerSN is: 3</p> <p>SerializedPayload: 205</p> <p>SerializedPayload: 171</p> <p>SerializedPayload: 205</p> <p>SerializedPayload: 171</p>

Tabla 4-46. TestOpenDDS_rtps_reliability_runttest_localPacket01

Llamada:	
	public UDPReceiver(Uri uri, int bufferSize)
Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS de otros

Tabla 4-46. TestOpenDDS_rtps_reliability_runttest_localPacket01

	vendedores con los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert.
Entrada	Inicialmente no se tiene inicializado al Receiver UDP
Código	<pre>[TestMethod] public void TesOpenDDS_rtps_reliability_runttest_localPacket01() { object key = new object(); UDPReceiver rec = new UDPReceiver(new Uri("udp://" + + Host + ":" + Port), 1024); rec.MessageReceived += (s, m) => { Message msg = m.Message; Debug.WriteLine("New Message has arrived from {0}", m.Session.RemoteEndPoint); Debug.WriteLine("Message Header: {0}", msg.Header); Assert.AreEqual(ProtocolId.PROTOCOL_RTPS.ToString(), msg.Header.Protocol.ToString()); Debug.WriteLine("The Header Protocol is: {0}", msg.Header.Protocol); Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1.ToString(), msg.Header.Version.ToString()); Debug.WriteLine("The Protocol Version value state is: {0}", msg.Header.Version); Assert.AreEqual(VendorId.OCI.ToString(), msg.Header.VendorId.ToString()); Debug.WriteLine("The VendorId value state is: {0}", msg.Header.VendorId); Assert.AreEqual("01-03-08-00-27-B9-29-47-0A-AF- 00-00", msg.Header.GuidPrefix.ToString()); Debug.WriteLine("The guidPrefix value state is: {0}", msg.Header.GuidPrefix); Assert.AreEqual(1, msg.SubMessages.Count); Debug.WriteLine("The number of SubMessages in the message is: {0}", msg.SubMessages.Count); foreach (var submsg in msg.SubMessages) { Assert.AreEqual(SubMessageKind.DATA, submsg.Kind); Debug.WriteLine("SubMessage: {0}", submsg.Kind); switch (submsg.Kind) { case SubMessageKind.DATA: { Data d = submsg as Data; </pre>

Tabla 4-46. TestOpenDDS_rtps_reliability_runttest_localPacket01

	<pre> d.HasKeyFlag); Assert.AreEqual(false, Debug.WriteLine("The KeyFlag value state is: {0}", d.HasKeyFlag); Assert.AreEqual(true, d.HasDataFlag); Debug.WriteLine("The DataFlag value state is: {0}", d.HasDataFlag); Assert.AreEqual(false, d.HasInlineQosFlag); Debug.WriteLine("The InlineQoSFlag value state is: {0}", d.HasInlineQosFlag); Assert.AreEqual(true, d.Header.Flags.IsLittleEndian); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Assert.AreEqual(0, d.Header.SubMessageLength); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Assert.AreEqual(0, d.ExtraFlags .Value); Debug.WriteLine("The extraFlags value is: {0}", d.ExtraFlags.Value); Debug.WriteLine("The octetsToInlineQos value is: "); Assert.AreEqual(0, d.ReaderId.EntityKey0); Assert.AreEqual(0, d.ReaderId.EntityKey1); Assert.AreEqual(0, d.ReaderId.EntityKey2); Debug.WriteLine("The readerIDEntityKey is: {0}-{1}-{2}", d.ReaderId.EntityKey0,d.ReaderId.EntityKey1,d.ReaderId.EntityKey2); Assert.AreEqual(0,(int) d.ReaderId.TypeID); Debug.WriteLine("The readerIDEntityKind value is: {0} ",(int)d.ReaderId.TypeID); Assert.AreEqual(0, d.WriterId.EntityKey0); Assert.AreEqual(1, d.WriterId.EntityKey1); Assert.AreEqual(2, d.WriterId.EntityKey2); Debug.WriteLine("The writerID is: {0}-{1}-{2}", d.WriterId.EntityKey0, d.WriterId.EntityKey1, d.WriterId.EntityKey2); Assert.AreEqual(2, (int)d.WriterId.TypeID); Debug.WriteLine("The writerIDEntityKind value is:{0} ",(int) d.WriterId.TypeID); Assert.AreEqual("1", d.WriterSN.ToString()); Debug.WriteLine("The writerSN is: {0}", d.WriterSN); </pre>
--	---

Tabla 4-46. TestOpenDDS_rtps_reliability_runttest_localPacket01

	<pre> if (d.HasInlineQosFlag) { /*foreach (var par in d.InlineQos.Value) Debug.WriteLine("InlineQos: {0}", par); }*/ } if (d.HasDataFlag d.Header.Flags.IsLittleEndian) { for (int i = 0; i <= d.SerializedPayload.DataEncapsulation.SerializedPayload.Length - 1; i++) { Debug.WriteLine("SerializedPayload: {0}", d.SerializedPayload.DataEncapsulation.SerializedPayload.GetValue(i)); } } break; } } lock (key) Monitor.Pulse(key); }; rec.Start(); simulator.SendUDPPacket("SamplePackets/TestOpenDDS_rtps_reliability_runttest_local/Packet01.dat", Host, Port); lock (key) { Assert.IsTrue(Monitor.Wait(key, 10000), "Time- out. Message has not arrived or there is an error on it."); } rec.Close(); } </pre>
Salida	<p>Nombre de la prueba: <i>TestOpenDDS_rtps_reliability_runttest_localPacket01</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2312824</p> <p>Salida estándar de Result:</p> <p>Trace du débogage :</p>

Tabla 4-46. TestOpenDDS_rtps_reliability_runttest_localPacket01

	<p>no configuration section <common/logging> found - - suppressing logging output</p> <p>Sent 52/52 bytes to 224.0.1.111:7400</p> <p>New Message has arrived from 172.30.82.26:63701</p> <p>Message Header: [RTPS, 2.1, 01-03, 01-03-08-00-27-B9-29-47-0A-AF-00-00]</p> <p>The Header Protocol is: RTPS</p> <p>The Protocol Version value state is: 2.1</p> <p>The VendorId value state is: 01-03</p> <p>The guidPrefix value state is: 01-03-08-00-27-B9-29-47-0A-AF-00-00</p> <p>The number of SubMessages in the message is: 1</p> <p>SubMessage: DATA</p> <p>The KeyFlag value state is: False</p> <p>The DataFlag value state is: True</p> <p>The InlineQoSFlag value state is: False</p> <p>The EndiannessFlag value state is: True</p> <p>The octetsToNextHeader value is: 0</p> <p>The extraFlags value is: 0</p> <p>The octetsToInlineQos value is:</p> <p>The readerIDEntityKey is: 0-0-0</p> <p>The readerIDEntityKind value is: 0</p> <p>The writerID is: 0-1-2</p> <p>The writerIDEntityKind value is: 2</p> <p>The writerSN is: 1</p> <p>SerializedPayload: 205</p> <p>SerializedPayload: 171</p> <p>SerializedPayload: 205</p> <p>SerializedPayload: 171</p>
--	---

Tabla 4-47. TestOpenDDS_rtps_reliability_runttest_localPacket02

Llamada:	public UDPReceiver(Uri uri, int bufferSize)
Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS de otros vendors con los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert.
Entrada	Inicialmente no se tiene inicializado al Receiver UDP
Código	[TestMethod] public void TesOpenDDS_rtps_reliability_runttest_localPacket02() { object key = new object();

Tabla 4-47. TestOpenDDS_rtps_reliability_runttest_localPacket02

	<pre> UDPReceiver rec = new UDPReceiver(new Uri("udp://" + + Host + ":" + Port), 1024); rec.MessageReceived += (s, m) => { Message msg = m.Message; Debug.WriteLine("New Message has arrived from {0}", m.Session.RemoteEndPoint); Debug.WriteLine("Message Header: {0}", msg.Header); Assert.AreEqual(ProtocolId.PROTOCOL_RTPS.ToString(), msg.Header.Protocol.ToString()); Debug.WriteLine("The Header Protocol is: {0}", msg.Header.Protocol); Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1.ToString(), msg.Header.Version.ToString()); Debug.WriteLine("The Protocol Version value state is: {0}", msg.Header.Version); Assert.AreEqual(VendorId.OCI.ToString(), msg.Header.VendorId.ToString()); Debug.WriteLine("The VendorId value state is: {0}", msg.Header.VendorId); Assert.AreEqual("01-03-08-00-27-B9-29-47-0A- AF-00-00", msg.Header.GuidPrefix.ToString()); Debug.WriteLine("The guidPrefix value state is: {0}", msg.Header.GuidPrefix); Assert.AreEqual(1, msg.SubMessages.Count); Debug.WriteLine("The number of SubMessages in the message is: {0}", msg.SubMessages.Count); foreach (var submsg in msg.SubMessages) { Assert.AreEqual(SubMessageKind.HEARTBEAT, submsg.Kind); Debug.WriteLine("SubMessage: {0}", submsg.Kind); switch (submsg.Kind) { case SubMessageKind.HEARTBEAT: { Heartbeat d = submsg as Heartbeat; Assert.AreEqual(false, d.HasLivelinessFlag); Debug.WriteLine("The LivelinessFlag value state is: {0}", d.HasLivelinessFlag); Assert.AreEqual(false, d.HasFinalFlag); Debug.WriteLine("The FinalFlag value state is: {0}", d.HasFinalFlag); Assert.AreEqual(true, d.Header.Flags.IsLittleEndian); } } } } } </pre>
--	---

Tabla 4-47. TestOpenDDS_rtps_reliability_runttest_localPacket02

	<pre> Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Assert.AreEqual(0, d.Header.SubMessageLength); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Assert.AreEqual(0, d.ReaderId.EntityKey0); Assert.AreEqual(0, d.ReaderId.EntityKey1); Assert.AreEqual(0, d.ReaderId.EntityKey2); Debug.WriteLine("The readerIDEntityKey is: {0}-{1}-{2}", d.ReaderId.EntityKey0, d.ReaderId.EntityKey1, d.ReaderId.EntityKey2); Assert.AreEqual(0, (int)d.ReaderId.TypeID); Debug.WriteLine("The readerIDEntityKind value is: {0} ", (int)d.ReaderId.TypeID); Assert.AreEqual(0, d.WriterId.EntityKey0); Assert.AreEqual(1, d.WriterId.EntityKey1); Assert.AreEqual(2, d.WriterId.EntityKey2); Debug.WriteLine("The writerID is: {0}-{1}-{2}", d.WriterId.EntityKey0, d.WriterId.EntityKey1, d.WriterId.EntityKey2); Assert.AreEqual(2, (int)d.WriterId.TypeID); Debug.WriteLine("The writerIDEntityKind value is:{0} ", (int)d.WriterId.TypeID); Assert.AreEqual(1,d.FirstSequenceNumber); Debug.WriteLine("The firstSN is: {0}", d.FirstSequenceNumber); Assert.AreEqual(1,d.LastSequenceNumber); Debug.WriteLine("The lastSN is: {0}", d.LastSequenceNumber); Assert.AreEqual(1,d.Count); Debug.WriteLine("The Count is: {0}", d.Count); break; } } lock (key) Monitor.Pulse(key); }; rec.Start(); simulator.SendUDPPacket("SamplePackets/TestOpenDDS_rtps_reliability_runttest_local/Packet02.dat", Host, Port); </pre>
--	---

Tabla 4-47. TestOpenDDS_rtps_reliability_runttest_localPacket02

	<pre> lock (key) { Assert.IsTrue(Monitor.Wait(key, 10000), "Time- out. Message has not arrived or there is an error on it."); } rec.Close(); } </pre>
Salida	<p>Nombre de la prueba: <i>TestOpenDDS_rtps_reliability_runttest_localPacket02</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2411492</p> <p>Nombre de la prueba: <i>TesOpenDDS_rtps_reliability_runttest_localPacket02</i></p> <p>Resultado de la prueba: Superada</p> <p>Salida estándar de Resultado:</p> <p>Trace du débogage :</p> <p>no configuration section <common/logging> found - suppressing logging output</p> <p>Sent 52/52 bytes to 224.0.1.111:7400</p> <p>New Message has arrived from 172.30.82.26:59411</p> <p>Message Header: [RTPS, 2.1, 01-03, 01-03-08-00-27-B9-29-47-0A-AF-00-00]</p> <p>The Header Protocol is: RTPS</p> <p>The Protocol Version value state is: 2.1</p> <p>The VendorId value state is: 01-03</p> <p>The guidPrefix value state is: 01-03-08-00-27-B9-29-47-0A-AF-00-00</p> <p>The number of SubMessages in the message is: 1</p> <p>SubMessage: HEARTBEAT</p> <p>The LivelinessFlag value state is: False</p> <p>The FinalFlag value state is: False</p> <p>The EndianessFlag value state is: True</p> <p>The octetsToNextHeader value is: 0</p> <p>The readerIDEntityKey is: 0-0-0</p> <p>The readerIDEntityKind value is: 0</p> <p>The writerID is: 0-1-2</p> <p>The writerIDEntityKind value is: 2</p> <p>The firstSN is: 1</p> <p>The lastSN is: 1</p> <p>The Count is: 1</p>

Tabla 4-48. TestOpenDDS_rtps_reliability_runttest_localPacket03

Llamada:	
	public UDPReceiver(Uri uri, int bufferSize)
Descripción	En esta prueba se verifica el correcto funcionamiento de los receptores UDP, utilizando mensajes RTPS de otros vendors con los cuales se verifica que sus datos sean correctos con pequeñas pruebas assert.
Entrada	Inicialmente no se tiene inicializado al Receiver UDP
Código	<pre>[TestMethod] public void TestOpenDDS_rtps_reliability_runttest_localPacket03() { object key = new object(); UDPReceiver rec = new UDPReceiver(new Uri("udp://" + Host + ":" + Port), 1024); rec.MessageReceived += (s, m) => { Message msg = m.Message; Debug.WriteLine("New Message has arrived from {0}", m.Session.RemoteEndPoint); Debug.WriteLine("Message Header: {0}", msg.Header); Assert.AreEqual(ProtocolId.PROTOCOL_RTPS.ToString(), msg.Header.Protocol.ToString()); Debug.WriteLine("The Header Protocol is: {0}", msg.Header.Protocol); Assert.AreEqual(ProtocolVersion.PROTOCOLVERSION_2_1.ToString(), msg.Header.Version.ToString()); Debug.WriteLine("The Protocol Version value state is: {0}", msg.Header.Version); Assert.AreEqual(VendorId.OCI.ToString(), msg.Header.VendorId.ToString()); Debug.WriteLine("The VendorId value state is: {0}", msg.Header.VendorId); Assert.AreEqual("01-03-08-00-27-B9-29-47-0A-AF- 00-01", msg.Header.GuidPrefix.ToString()); Debug.WriteLine("The guidPrefix value state is: {0}", msg.Header.GuidPrefix); Assert.AreEqual(2, msg.SubMessages.Count); Debug.WriteLine("The number of SubMessages in the message is: {0}", msg.SubMessages.Count); foreach (var submsg in msg.SubMessages) { Debug.WriteLine("SubMessage: {0}", submsg.Kind); } }; }</pre>

Tabla 4-48. TestOpenDDS_rtps_reliability_runtest_localPacket03

	<pre> switch (submsg.Kind) { case SubMessageKind.INFO_DST: { InfoDestination d = submsg as InfoDestination; Assert.AreEqual(true, d.Header.Flags.IsLittleEndian); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Assert.AreEqual(12, d.Header.SubMessageLength); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Assert.AreEqual("01-03-08-00-27- B9-29-47-0A-AF-00-00", d.GuidPrefix.ToString()); Debug.WriteLine("The guidPrefix value is: {0}", d.GuidPrefix); break; } case SubMessageKind.ACKNACK: { AckNack d = submsg as AckNack; Assert.AreEqual(true, d.HasFinalFlag); Debug.WriteLine("The FinalFlag value state is: {0}", d.HasFinalFlag); Assert.AreEqual(true, d.Header.Flags.IsLittleEndian); Debug.WriteLine("The EndiannessFlag value state is: {0}", d.Header.Flags.IsLittleEndian); Debug.WriteLine("The octetsToNextHeader value is: {0}", d.Header.SubMessageLength); Assert.AreEqual(0, d.ReaderId.EntityKey0); Assert.AreEqual(1, d.ReaderId.EntityKey1); Assert.AreEqual(5, d.ReaderId.EntityKey2); Debug.WriteLine("The readerIDEntityKey is: {0}-{1}-{2}", d.ReaderId.EntityKey0, d.ReaderId.EntityKey1, d.ReaderId.EntityKey2); Assert.AreEqual(7, (int)d.ReaderId.TypeID); Debug.WriteLine("The readerIDEntityKind value is: {0} ", (int)d.ReaderId.TypeID); Assert.AreEqual(0, d.WriterId.EntityKey0); Assert.AreEqual(1, d.WriterId.EntityKey1); Assert.AreEqual(2, d.WriterId.EntityKey2); Debug.WriteLine("The writerID is: {0}-{1}-{2}", d.WriterId.EntityKey0, d.WriterId.EntityKey1, d.WriterId.EntityKey2); } } </pre>
--	--

Tabla 4-48. TestOpenDDS_rtps_reliability_runttest_localPacket03

	<pre> Assert.AreEqual(2, (int)d.WriterId.TypeID); Debug.WriteLine("The writerIDEntityKind value is:{0} ", (int)d.WriterId.TypeID); Assert.AreEqual("2", d.ReaderSNState.BitmapBase.ToString()); Assert.AreEqual(1, d.ReaderSNState.NumBits); Assert.AreEqual(0, d.ReaderSNState.Bitmaps[0]); Debug.WriteLine("The readerSNState is: {0}", d.ReaderSNState); Debug.WriteLine("The Count is: {0}", d.Count); break; } } lock (key) Monitor.Pulse(key); }; rec.Start(); simulator.SendUDPPacket("SamplePackets/TestOpenDDS_rtps_reliability_runttest_local/Packet03.dat", Host, Port); lock (key) { Assert.IsTrue(Monitor.Wait(key, 1000), "Time- out. Message has not arrived or there is an error on it."); } rec.Close(); } } </pre>
Salida	<p>Nombre de la prueba: <i>TestOpenDDS_rtps_reliability_runttest_localPacket03</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2464427</p> <p>Salida estándar de Result:</p> <p>Trace du débogage :</p> <pre>no configuration section <common/logging> found - suppressing logging output Sent 68/68 bytes to 224.0.1.111:7400 New Message has arrived from 172.30.82.26:63366 Message Header: [RTPS, 2.1, 01-03, 01-03-08-00-27-B9- 29-47-0A-AF-00-01] The Header Protocol is: RTPS</pre>

Tabla 4-48. TestOpenDDS_rtps_reliability_runttest_localPacket03

	<p>The Protocol Version value state is: 2.1 The VendorId value state is: 01-03 The guidPrefix value state is: 01-03-08-00-27-B9-29-47-0A-AF-00-01 The number of SubMessages in the message is: 2 SubMessage: INFO_DST The EndiannessFlag value state is: True The octetsToNextHeader value is: 12 The guidPrefix value is: 01-03-08-00-27-B9-29-47-0A-AF-00-00 SubMessage: ACKNACK The FinalFlag value state is: True The EndiannessFlag value state is: True The octetsToNextHeader value is: 28 The readerIDEntityKey is: 0-1-5 The readerIDEntityKind value is: 7 The writerID is: 0-1-2 The writerIDEntityKind value is: 2 The readerSNState is: 2/1:[0x0000] The Count is: 1 </p>
--	---

4.2.3. UTILS

4.2.3.1. Pruebas del generador de identidad.

En estas pruebas se verifica el correcto funcionamiento del *Guid Generator*, el cual es el generador de identidades.

Tabla 4-49. TestGenerator1

Llamada: static GuidGenerator()	
Descripción	En esta prueba se verifica el correcto funcionamiento del Guid Generator.
Entrada	Inicialmente no se tiene inicializado al <i>GuidGenerator</i>
Código	<pre>[TestMethod] public void TestGeneration1() { GuidGenerator generator = new GuidGenerator(); GUID guid = generator.GenerateGuid(); Assert.AreEqual(GuidGenerator.VENDORID_DOOPEC[0], guid.Prefix.Prefix[0]); Assert.AreEqual(GuidGenerator.VENDORID_DOOPEC[1], guid.Prefix.Prefix[1]); }</pre>

Salida	Nombre de la prueba: <i>TestGeneration1</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,172692
---------------	---

Tabla 4-50. TestGeneration2

Llamada: <code>static GuidGenerator()</code>	
Descripción	En esta prueba se verifica el correcto funcionamiento del Guid Generator.
Entrada	Inicialmente no se tiene inicializado al <i>GuidGenerator</i>
Código	<pre>[TestMethod] public void TestGeneration2() { GuidGenerator generator = new GuidGenerator(); GUID guid1 = generator.GenerateGuid(); GUID guid2 = generator.GenerateGuid(); Assert.AreNotEqual(guid2.Prefix.ToString(), guid1.Prefix.ToString()); }</pre>
Salida	Nombre de la prueba: <i>TestGeneration2</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1552052

4.2.3.2. Pruebas del PeriodicWorker.

En estas pruebas se verifica el funcionamiento del *PeriodicWorker*, el cual está encargado del envío de mensajes con submensajes *Heartbeat* los cuales sirven para mantener la conexión entre entidades.

Tabla 4-51. TestWorkerVerySlow

Llamada: <code>private void KeepWorkerRunning()</code>	
Descripción	En esta prueba se verifica el correcto funcionamiento del Worker en el cual se realiza tareas de actualización y descubrimiento
Entrada	Inicialmente no se tiene inicializado al Worker
Código	<pre>[TestMethod] public void TestWorkerVerySlow()</pre>

Tabla 4-51. TestWorkerVerySlow

	<pre>{ int period = 2 * 1000; int sleepTime = 20 * 1000+90; PeriodicWorker worker = new PeriodicWorker(); worker.Start(period); Thread.Sleep(sleepTime); worker.End(); Assert.AreEqual(sleepTime / period, worker.Count); }</pre>
Salida	<p>Nombre de la prueba: <i>TestWorkerVerySlow</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:20,2217009</p>

Tabla 4-52. TestWorkerSlow

Llamada: private void KeepWorkerRunning()	
Descripción	En esta prueba se verifica el correcto funcionamiento del Worker en el cual se realiza tareas de actualización y descubrimiento
Entrada	Inicialmente no se tiene inicializado al Worker
Código	<pre>[TestMethod] public void TestWorkerSlow() { int period = 2 * 100; int sleepTime = 20 * 100 + 50; PeriodicWorker worker = new PeriodicWorker(); worker.Start(period); Thread.Sleep(sleepTime); worker.End(); Assert.AreEqual(sleepTime / period, worker.Count); }</pre>
Salida	<p>Nombre de la prueba: <i>TestWorkerSlow</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:02,1763206</p>

Tabla 4-53. TestWorkerQuick

Llamada: private void KeepWorkerRunning()	
Descripción	En esta prueba se verifica el correcto funcionamiento del Worker en el cual se realiza tareas de actualización y descubrimiento
Entrada	Inicialmente no se tiene inicializado al Worker
Código	[TestMethod] public void TestWorkerQuick() { int period = 2 * 10; int sleepTime = 20 * 10 + 50; PeriodicWorker worker = new PeriodicWorker(); worker.Start(period); Thread.Sleep(sleepTime); worker.End(); Assert.AreEqual(sleepTime / period, worker.Count); }
Salida	Nombre de la prueba: <i>TestWorkerQuick</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,3822312

4.2.3.3. Prueba de tiempo

En esta prueba se verifica la conversión del tipo *timeMillis* hacia el tipo de dato *long*.

Tabla 4-54. TestTimeSeconds

Llamada: public Time(long systemCurrentMillis)	
Descripción	En esta prueba se verifica el correcto funcionamiento del temporizador
Entrada	Inicialmente no se tiene inicializado al <i>Time</i>
Código	[TestMethod] public void TestTimeSeconds() { long timeMillis = 1000; // 1 sec Time t = new Time(timeMillis); long timeConverted = t.TimeMillis; Assert.AreEqual(timeMillis, timeConverted); }
Salida	Nombre de la prueba: <i>TestTimeSeconds</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:02,8221745

4.2.4. SERIALIZADOR

4.2.4.1. Pruebas del BuiltinTopic

En estas pruebas se verifica el correcto funcionamiento del serializador de la implementación del API-DDS, la cual es encapsulada en los mensajes RTPS.

Tabla 4-55. TestParticipantBuiltinTopicData

Llamada: public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador del DDS en el Builtin Data Participant
Entrada	Inicialmente no se tiene inicializado al ddsType
Código	[TestMethod] public void TestParticipantBuiltinTopicData() { var ddsType = TypeExplorer.ExploreType(typeof(ParticipantBuiltin TopicData)); Assert.IsNotNull(ddsType); Assert.IsNotNull(ddsType.GetProperty()); var propInfo = ddsType.GetProperty(); Assert.AreEqual("org.omg.dds.topic.ParticipantBuil tinTopicData", propInfo.Name); Assert.IsInstanceOfType(ddsType, typeof(StructureType)); StructureType structType = ddsType as StructureType; var members = structType.GetMember(); Assert.IsNotNull(members); Assert.AreEqual(2, members.Count); Assert.AreEqual("Key", members[0].GetProperty().Name); Assert.AreEqual("UserData", members[1].GetProperty().Name); Assert.AreEqual((uint)0x0050, members[0].GetProperty().MemberId); Assert.AreEqual((uint)0x002C, members[1].GetProperty().MemberId); }
Salida	Nombre de la prueba: <i>TestParticipantBuiltinTopicData</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1871316

Tabla 4-56. TestPublicationBuiltinTopicData

Llamada: public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador de DDS en el Builtin Data Publication
Entrada	Inicialmente no se tiene inicializado al ddsType
Código	[TestMethod] public void TestPublicationBuiltinTopicData() { var ddsType = TypeExplorer.ExploreType(typeof(PublicationBuiltin TopicData)); Assert.IsNotNull(ddsType); Assert.IsNotNull(ddsType.GetProperty()); var propInfo = ddsType.GetProperty(); Assert.AreEqual("org.omg.dds.topic.PublicationBuil tinTopicData", propInfo.Name); Assert.IsInstanceOfType(ddsType, typeof(StructureType)); StructureType structType = ddsType as StructureType; var members = structType.GetMember(); Assert.IsNotNull(members); Assert.AreEqual(24, members.Count); Assert.AreEqual("Key", members[0].GetProperty().Name); Assert.AreEqual("ParticipantKey", members[1].GetProperty().Name); Assert.AreEqual("TopicName", members[2].GetProperty().Name); Assert.AreEqual("TypeName", members[3].GetProperty().Name); Assert.AreEqual("EquivalentTypeName", members[4].GetProperty().Name); Assert.AreEqual("BaseTypeName", members[5].GetProperty().Name); Assert.AreEqual("Type", members[6].GetProperty().Name); Assert.AreEqual("Durability", members[7].GetProperty().Name); Assert.AreEqual("DurabilityService", members[8].GetProperty().Name); Assert.AreEqual("Deadline", members[9].GetProperty().Name); Assert.AreEqual("LatencyBudget", members[10].GetProperty().Name); Assert.AreEqual("Liveliness", members[11].GetProperty().Name); Assert.AreEqual("Reliability", members[12].GetProperty().Name);

Tabla 4-56. TestPublicationBuiltinTopicData

	<pre> Assert.AreEqual("Lifespan", members[13].GetProperty().Name); Assert.AreEqual("UserData", members[14].GetProperty().Name); Assert.AreEqual("Ownership", members[15].GetProperty().Name); Assert.AreEqual("OwnershipStrength", members[16].GetProperty().Name); Assert.AreEqual("DestinationOrder", members[17].GetProperty().Name); Assert.AreEqual("Presentation", members[18].GetProperty().Name); Assert.AreEqual("Partition", members[19].GetProperty().Name); Assert.AreEqual("TopicData", members[20].GetProperty().Name); Assert.AreEqual("GroupData", members[21].GetProperty().Name); Assert.AreEqual("Representation", members[22].GetProperty().Name); Assert.AreEqual("TypeConsistency", members[23].GetProperty().Name); Assert.AreEqual((uint)0x005A, members[0].GetProperty().MemberId); Assert.AreEqual((uint)0x0050, members[1].GetProperty().MemberId); Assert.AreEqual((uint)0x0005, members[2].GetProperty().MemberId); Assert.AreEqual((uint)0x0007, members[3].GetProperty().MemberId); Assert.AreEqual((uint)0x0075, members[4].GetProperty().MemberId); Assert.AreEqual((uint)0x0076, members[5].GetProperty().MemberId); Assert.AreEqual((uint)0x0072, members[6].GetProperty().MemberId); Assert.AreEqual((uint)0x001D, members[7].GetProperty().MemberId); Assert.AreEqual((uint)0x001E, members[8].GetProperty().MemberId); Assert.AreEqual((uint)0x0023, members[9].GetProperty().MemberId); Assert.AreEqual((uint)0x0027, members[10].GetProperty().MemberId); Assert.AreEqual((uint)0x001B, members[11].GetProperty().MemberId); Assert.AreEqual((uint)0x001A, members[12].GetProperty().MemberId); Assert.AreEqual((uint)0x002B, members[13].GetProperty().MemberId); Assert.AreEqual((uint)0x002C, members[14].GetProperty().MemberId); Assert.AreEqual((uint)0x001F, members[15].GetProperty().MemberId); Assert.AreEqual((uint)0x0006, members[16].GetProperty().MemberId); Assert.AreEqual((uint)0x0025, members[17].GetProperty().MemberId); </pre>
--	--

Tabla 4-56. TestPublicationBuiltinTopicData

	<pre> Assert.AreEqual((uint)0x0021, members[18].GetProperty().MemberId); Assert.AreEqual((uint)0x0029, members[19].GetProperty().MemberId); Assert.AreEqual((uint)0x002E, members[20].GetProperty().MemberId); Assert.AreEqual((uint)0x002D, members[21].GetProperty().MemberId); Assert.AreEqual((uint)0x0073, members[22].GetProperty().MemberId); Assert.AreEqual((uint)0x0074, members[23].GetProperty().MemberId); } </pre>
Salida	<p>Nombre de la prueba: <i>TestPublicationBuiltinTopicData</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1681182</p>

Tabla 4-57. TestSubscriptionBuiltinTopicData

Llamada: public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador de DDS en el Builtin Data Suscriber
Entrada	Inicialmente no se tiene inicializado al ddsType
Código	<pre> [TestMethod] public void TestSubscriptionBuiltinTopicData() { var ddsType = TypeExplorer.ExploreType(typeof(SubscriptionBuiltinTopicData)); Assert.IsNotNull(ddsType); Assert.IsNotNull(ddsType.GetProperty()); var propInfo = ddsType.GetProperty(); Assert.AreEqual("org.omg.dds.topic.SubscriptionBuiltinTopicData", propInfo.Name); Assert.IsInstanceOfType(ddsType, typeof(StructureType)); StructureType structType = ddsType as StructureType; var members = structType.GetMember(); Assert.IsNotNull(members); Assert.AreEqual(22, members.Count); Assert.AreEqual("Key", members[0].GetProperty().Name); Assert.AreEqual("ParticipantKey", members[1].GetProperty().Name); } </pre>

Tabla 4-57. TestSubscriptionBuiltinTopicData

	<pre> Assert.AreEqual("TopicName", members[2].GetProperty().Name); Assert.AreEqual("TypeName", members[3].GetProperty().Name); Assert.AreEqual("EquivalentTypeName", members[4].GetProperty().Name); Assert.AreEqual("BaseTypeName", members[5].GetProperty().Name); Assert.AreEqual("Type", members[6].GetProperty().Name); Assert.AreEqual("Durability", members[7].GetProperty().Name); Assert.AreEqual("Deadline", members[8].GetProperty().Name); Assert.AreEqual("LatencyBudget", members[9].GetProperty().Name); Assert.AreEqual("Liveliness", members[10].GetProperty().Name); Assert.AreEqual("Reliability", members[11].GetProperty().Name); Assert.AreEqual("Ownership", members[12].GetProperty().Name); Assert.AreEqual("DestinationOrder", members[13].GetProperty().Name); Assert.AreEqual("UserData", members[14].GetProperty().Name); Assert.AreEqual("TimeBasedFilter", members[15].GetProperty().Name); Assert.AreEqual("Presentation", members[16].GetProperty().Name); Assert.AreEqual("Partition", members[17].GetProperty().Name); Assert.AreEqual("TopicData", members[18].GetProperty().Name); Assert.AreEqual("GroupData", members[19].GetProperty().Name); Assert.AreEqual("Representation", members[20].GetProperty().Name); Assert.AreEqual("TypeConsistency", members[21].GetProperty().Name); Assert.AreEqual((uint)0x005A, members[0].GetProperty().MemberId); Assert.AreEqual((uint)0x0050, members[1].GetProperty().MemberId); Assert.AreEqual((uint)0x0005, members[2].GetProperty().MemberId); Assert.AreEqual((uint)0x0007, members[3].GetProperty().MemberId); Assert.AreEqual((uint)0x0075, members[4].GetProperty().MemberId); Assert.AreEqual((uint)0x0076, members[5].GetProperty().MemberId); Assert.AreEqual((uint)0x0072, members[6].GetProperty().MemberId); Assert.AreEqual((uint)0x001D, members[7].GetProperty().MemberId); Assert.AreEqual((uint)0x0023, members[8].GetProperty().MemberId); </pre>
--	--

Tabla 4-57. TestSubscriptionBuiltinTopicData

	<pre> Assert.AreEqual((uint)0x0027, members[9].GetProperty().MemberId); Assert.AreEqual((uint)0x001B, members[10].GetProperty().MemberId); Assert.AreEqual((uint)0x001A, members[11].GetProperty().MemberId); Assert.AreEqual((uint)0x001F, members[12].GetProperty().MemberId); Assert.AreEqual((uint)0x0025, members[13].GetProperty().MemberId); Assert.AreEqual((uint)0x002C, members[14].GetProperty().MemberId); Assert.AreEqual((uint)0x0004, members[15].GetProperty().MemberId); Assert.AreEqual((uint)0x0021, members[16].GetProperty().MemberId); Assert.AreEqual((uint)0x0029, members[17].GetProperty().MemberId); Assert.AreEqual((uint)0x002E, members[18].GetProperty().MemberId); Assert.AreEqual((uint)0x002D, members[19].GetProperty().MemberId); Assert.AreEqual((uint)0x0073, members[20].GetProperty().MemberId); Assert.AreEqual((uint)0x0074, members[21].GetProperty().MemberId); } </pre>
Salida	<p>Nombre de la prueba: <i>TestSubscriptionBuiltinTopicData</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1438964</p>

Tabla 4-58. TestSubscriptionBuiltinTopicData

Llamada: public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador de DDS en el Builtin Data Suscriber
Entrada	Inicialmente no se tiene inicializado al ddsType
Código	<pre> [TestMethod] public void TestSubscriptionBuiltinTopicData() { var ddsType = TypeExplorer.ExploreType(typeof(SubscriptionBuiltinTopicData)); Assert.IsNotNull(ddsType); Assert.IsNotNull(ddsType.GetProperty()); var propInfo = ddsType.GetProperty(); </pre>

Tabla 4-58. TestSubscriptionBuiltinTopicData

	<pre> Assert.AreEqual("org.omg.dds.topic.SubscriptionBui ltinTopicData", propInfo.Name); Assert.IsInstanceOfType(ddsType, typeof(StructureType)); StructureType structType = ddsType as StructureType; var members = structType.GetMember(); Assert.IsNotNull(members); Assert.AreEqual(22, members.Count); Assert.AreEqual("Key", members[0].GetProperty().Name); Assert.AreEqual("ParticipantKey", members[1].GetProperty().Name); Assert.AreEqual("TopicName", members[2].GetProperty().Name); Assert.AreEqual("TypeName", members[3].GetProperty().Name); Assert.AreEqual("EquivalentTypeName", members[4].GetProperty().Name); Assert.AreEqual("BaseTypeName", members[5].GetProperty().Name); Assert.AreEqual("Type", members[6].GetProperty().Name); Assert.AreEqual("Durability", members[7].GetProperty().Name); Assert.AreEqual("Deadline", members[8].GetProperty().Name); Assert.AreEqual("LatencyBudget", members[9].GetProperty().Name); Assert.AreEqual("Liveliness", members[10].GetProperty().Name); Assert.AreEqual("Reliability", members[11].GetProperty().Name); Assert.AreEqual("Ownership", members[12].GetProperty().Name); Assert.AreEqual("DestinationOrder", members[13].GetProperty().Name); Assert.AreEqual("UserData", members[14].GetProperty().Name); Assert.AreEqual("TimeBasedFilter", members[15].GetProperty().Name); Assert.AreEqual("Presentation", members[16].GetProperty().Name); Assert.AreEqual("Partition", members[17].GetProperty().Name); Assert.AreEqual("TopicData", members[18].GetProperty().Name); Assert.AreEqual("GroupData", members[19].GetProperty().Name); Assert.AreEqual("Representation", members[20].GetProperty().Name); Assert.AreEqual("TypeConsistency", members[21].GetProperty().Name); Assert.AreEqual((uint)0x005A, members[0].GetProperty().MemberId); Assert.AreEqual((uint)0x0050, members[1].GetProperty().MemberId); </pre>
--	--

Tabla 4-58. TestSubscriptionBuiltinTopicData

	<pre> Assert.AreEqual((uint)0x0005, members[2].GetProperty().MemberId); Assert.AreEqual((uint)0x0007, members[3].GetProperty().MemberId); Assert.AreEqual((uint)0x0075, members[4].GetProperty().MemberId); Assert.AreEqual((uint)0x0076, members[5].GetProperty().MemberId); Assert.AreEqual((uint)0x0072, members[6].GetProperty().MemberId); Assert.AreEqual((uint)0x001D, members[7].GetProperty().MemberId); Assert.AreEqual((uint)0x0023, members[8].GetProperty().MemberId); Assert.AreEqual((uint)0x0027, members[9].GetProperty().MemberId); Assert.AreEqual((uint)0x001B, members[10].GetProperty().MemberId); Assert.AreEqual((uint)0x001A, members[11].GetProperty().MemberId); Assert.AreEqual((uint)0x001F, members[12].GetProperty().MemberId); Assert.AreEqual((uint)0x0025, members[13].GetProperty().MemberId); Assert.AreEqual((uint)0x002C, members[14].GetProperty().MemberId); Assert.AreEqual((uint)0x0004, members[15].GetProperty().MemberId); Assert.AreEqual((uint)0x0021, members[16].GetProperty().MemberId); Assert.AreEqual((uint)0x0029, members[17].GetProperty().MemberId); Assert.AreEqual((uint)0x002E, members[18].GetProperty().MemberId); Assert.AreEqual((uint)0x002D, members[19].GetProperty().MemberId); Assert.AreEqual((uint)0x0073, members[20].GetProperty().MemberId); Assert.AreEqual((uint)0x0074, members[21].GetProperty().MemberId); } </pre>
Salida	<p>Nombre de la prueba: <i>TestSubscriptionBuiltinTopicData</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1438964</p>

Tabla 4-59. TestTopicBuiltinTopicData

Llamada: public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador de DDS en el Builtin Data Topic

Tabla 4-59. TestTopicBuiltinTopicData

Entrada	Inicialmente no se tiene inicializado al ddsType
Código	<pre>[TestMethod] public void TestTopicBuiltinTopicData() { var ddsType = TypeExplorer.ExploreType(typeof(TopicBuiltinTopicD ata)); Assert.IsNotNull(ddsType); Assert.IsNotNull(ddsType.GetProperty()); var propInfo = ddsType.GetProperty(); Assert.AreEqual("org.omg.dds.topic.TopicBuiltinTop icData", propInfo.Name); Assert.IsInstanceOfType(ddsType, typeof(StructureType)); StructureType structType = ddsType as StructureType; var members = structType.GetMember(); Assert.IsNotNull(members); Assert.AreEqual(21, members.Count); Assert.AreEqual("Key", members[0].GetProperty().Name); Assert.AreEqual("Name", members[1].GetProperty().Name); Assert.AreEqual("TypeName", members[2].GetProperty().Name); Assert.AreEqual("EquivalentTypeName", members[3].GetProperty().Name); Assert.AreEqual("BaseTypeName", members[4].GetProperty().Name); Assert.AreEqual("Type", members[5].GetProperty().Name); Assert.AreEqual("Durability", members[6].GetProperty().Name); Assert.AreEqual("DurabilityService", members[7].GetProperty().Name); Assert.AreEqual("Deadline", members[8].GetProperty().Name); Assert.AreEqual("LatencyBudget", members[9].GetProperty().Name); Assert.AreEqual("Liveliness", members[10].GetProperty().Name); Assert.AreEqual("Reliability", members[11].GetProperty().Name); Assert.AreEqual("TransportPriority", members[12].GetProperty().Name); Assert.AreEqual("Lifespan", members[13].GetProperty().Name); Assert.AreEqual("DestinationOrder", members[14].GetProperty().Name); Assert.AreEqual("History", members[15].GetProperty().Name); Assert.AreEqual("ResourceLimits", members[16].GetProperty().Name); Assert.AreEqual("Ownership", members[17].GetProperty().Name);</pre>

Tabla 4-59. TestTopicBuiltInTopicData

	<pre> Assert.AreEqual("TopicData", members[18].GetProperty().Name); Assert.AreEqual("Representation", members[19].GetProperty().Name); Assert.AreEqual("TypeConsistency", members[20].GetProperty().Name); Assert.AreEqual((uint)0x005A, members[0].GetProperty().MemberId); Assert.AreEqual((uint)0x0005, members[1].GetProperty().MemberId); Assert.AreEqual((uint)0x0007, members[2].GetProperty().MemberId); Assert.AreEqual((uint)0x0075, members[3].GetProperty().MemberId); Assert.AreEqual((uint)0x0076, members[4].GetProperty().MemberId); Assert.AreEqual((uint)0x0072, members[5].GetProperty().MemberId); Assert.AreEqual((uint)0x001D, members[6].GetProperty().MemberId); Assert.AreEqual((uint)0x001E, members[7].GetProperty().MemberId); Assert.AreEqual((uint)0x0023, members[8].GetProperty().MemberId); Assert.AreEqual((uint)0x0027, members[9].GetProperty().MemberId); Assert.AreEqual((uint)0x001B, members[10].GetProperty().MemberId); Assert.AreEqual((uint)0x001A, members[11].GetProperty().MemberId); Assert.AreEqual((uint)0x0049, members[12].GetProperty().MemberId); Assert.AreEqual((uint)0x002B, members[13].GetProperty().MemberId); Assert.AreEqual((uint)0x0025, members[14].GetProperty().MemberId); Assert.AreEqual((uint)0x0040, members[15].GetProperty().MemberId); Assert.AreEqual((uint)0x0041, members[16].GetProperty().MemberId); Assert.AreEqual((uint)0x001F, members[17].GetProperty().MemberId); Assert.AreEqual((uint)0x002E, members[18].GetProperty().MemberId); Assert.AreEqual((uint)0x0073, members[19].GetProperty().MemberId); Assert.AreEqual((uint)0x0074, members[20].GetProperty().MemberId); } </pre>
Salida	<p>Nombre de la prueba: <i>TestTopicBuiltInTopicData</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1909715</p>

4.2.4.2. Pruebas de Encapsulación CDR.

En estas pruebas se procede a verificar la encapsulación CDR, es decir que se comprueba la forma en la que se ordenan los *bytes* en los diferentes tipos de datos que se muestran de la forma *LittleEndian* y *BigEndian*.

Tabla 4-60. TestBoolPacketLE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el BoolPacket Little Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestBoolPacketLE() { BoolPacket v1 = new BoolPacket(true); int bufferSize = sizeof(bool) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 01", buffer.GetHexDump()); BoolPacket v2 = CDREncapsulation.Deserialize<BoolPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestBoolPacketLE</i></p> <p>Resultado de la prueba: ✓ 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1651349</p>

Tabla 4-61. TestCharPacketLE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order)</pre>
-----------------	---

Tabla 4-61. TestCharPacketLE

	<pre>public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el CharPacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestCharPacketLE() { CharPacket v1 = new CharPacket('A'); int bufferSize = sizeof(char) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 41 00", buffer.GetHexDump()); CharPacket v2 = CDREncapsulation.Deserialize<CharPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestCharPacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2107132</p>

Tabla 4-62. TestU8PacketLE

Llamada: <pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>	
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el U8Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestU8PacketLE() { U8Packet v1 = new U8Packet(0xA); int bufferSize = sizeof(byte) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize);</pre>

Tabla 4-62. TestU8PacketLE

	<pre> CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 0A", buffer.GetHexDump()); U8Packet v2 = CDREncapsulation.Deserialize<U8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } } </pre>
Salida	<p>Nombre de la prueba: <i>TestU8PacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2791979</p>

Tabla 4-63. TestU16PacketLE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el U16Packet Little Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestU16PacketLE() { U16Packet v1 = new U16Packet(0xAB); int bufferSize = sizeof(ushort) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 AB 00", buffer.GetHexDump()); U16Packet v2 = CDREncapsulation.Deserialize<U16Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>

Tabla 4-63. TestU16PacketLE

Salida	Nombre de la prueba: <i>TestU16PacketLE</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1831424
---------------	--

Tabla 4-64. TestU32PacketLE

Llamada: public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)	
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el U32Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	[TestMethod] public void TestU32PacketLE() { U32Packet v1 = new U32Packet(0xABA); int bufferSize = sizeof(uint) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 BA 0A 00 00", buffer.GetHexDump()); U32Packet v2 = CDREncapsulation.Deserialize<U32Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }
Salida	Nombre de la prueba: <i>TestU32PacketLE</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,171796

Tabla 4-65. TestU64PacketLE

Llamada: public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order)	
--	--

Tabla 4-65. TestU64PacketLE

	<pre>public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el U64Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestU64PacketLE() { U64Packet v1 = new U64Packet(0xABCD); int bufferSize = sizeof(ulong) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 EF CD AB 00 00 00 00 00", buffer.GetHexDump()); U64Packet v2 = CDREncapsulation.Deserialize<U64Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestU64PacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1623851</p>

Tabla 4-66. TestS8PacketLE1

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S8Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS8PacketLE1() { S8Packet v1 = new S8Packet(-1); int bufferSize = sizeof(sbyte) + CDRHeaderSize;</pre>

Tabla 4-66. TestS8PacketLE1

	<pre> var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 FF", buffer.GetHexDump()); S8Packet v2 = CDREncapsulation.Deserialize<S8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestS8PacketLE1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1973871</p>

Tabla 4-67. TestS8PacketLE2

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S8Packet Little Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestS8PacketLE2() { S8Packet v1 = new S8Packet(+1); int bufferSize = sizeof(sbyte) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 01", buffer.GetHexDump()); S8Packet v2 = CDREncapsulation.Deserialize<S8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>

Tabla 4-67. TestS8PacketLE2

Salida	} Nombre de la prueba: <i>TestS8PacketLE2</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1755769
---------------	---

Tabla 4-68. TestS16PacketLE1

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el S16Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS16PacketLE1() { S16Packet v1 = new S16Packet(-10); int bufferSize = sizeof(short) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 F6 FF", buffer.GetHexDump()); S16Packet v2 = CDREncapsulation.Deserialize<S16Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	Nombre de la prueba: <i>TestS16PacketLE1</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1916198

Tabla 4-69. TestS16PacketLE2

Llamada:

Tabla 4-69. TestS16PacketLE2

	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S16Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS16PacketLE2() { S16Packet v1 = new S16Packet(+10); int bufferSize = sizeof(short) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 0A 00", buffer.GetHexDump()); S16Packet v2 = CDREncapsulation.Deserialize<S16Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS16PacketLE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2151697</p>

Tabla 4-70. TestS32PacketLE1

	Llamada: <pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S32Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS32PacketLE1() { S32Packet v1 = new S32Packet(-0xABA);</pre>

Tabla 4-70. TestS32PacketLE1

	<pre> int bufferSize = sizeof(int) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 46 F5 FF FF", buffer.GetHexDump()); S32Packet v2 = CDREncapsulation.Deserialize<S32Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestS32PacketLE1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,15711</p>

Tabla 4-71. TestS32PacketLE2

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el S32Packet Little Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestS32PacketLE2() { S32Packet v1 = new S32Packet(0xABA); int bufferSize = sizeof(int) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 BA 0A 00 00", buffer.GetHexDump()); S32Packet v2 = CDREncapsulation.Deserialize<S32Packet>(buffer); Assert.AreEqual(v1, v2); } </pre>

Tabla 4-71. TestS32PacketLE2

	<pre> Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS32PacketLE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1565519</p>

Tabla 4-72. TestS64PacketLE1

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S4Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS64PacketLE1() { S64Packet v1 = new S64Packet(-0xABCD); int bufferSize = sizeof(long) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 33 54 FF FF FF FF FF", buffer.GetHexDump()); S64Packet v2 = CDREncapsulation.Deserialize<S64Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS64PacketLE1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1781261</p>

Tabla 4-73. TestS64PacketLE2

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order)</pre>
-----------------	--

Tabla 4-73. TestS64PacketLE2

	<pre>public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S4Packet LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS64PacketLE2() { S64Packet v1 = new S64Packet(0xABCD); int bufferSize = sizeof(long) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 CD AB 00 00 00 00 00 00", buffer.GetHexDump()); S64Packet v2 = CDREncapsulation.Deserialize<S64Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS64PacketLE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1868797</p>

Tabla 4-74. TestSinglePacketLE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el SinglePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestSinglePacketLE() { SinglePacket v1 = new SinglePacket(0.1f);</pre>

Tabla 4-74. TestSinglePacketLE

	<pre> int bufferSize = sizeof(float) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 CD CC CC 3D", buffer.GetHexDump()); SinglePacket v2 = CDREncapsulation.Deserialize<SinglePacket>(buffer) ; Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestSinglePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1841175</p>

Tabla 4-75. TestDoublePacketLE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el DoublePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestDoublePacketLE() { DoublePacket v1 = new DoublePacket(0.1); int bufferSize = sizeof(double) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 9A 99 99 99 99 99 B9 3F", buffer.GetHexDump()); } </pre>

Tabla 4-75. TestDoublePacketLE

	<pre> DoublePacket v2 = CDREncapsulation.Deserialize<DoublePacket>(buffer) ; Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestDoublePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1841175</p>

Tabla 4-76. TestBoolSequencePacketLE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el BoolSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestBoolSequencePacketLE() { BoolSequencePacket v1 = new BoolSequencePacket(new bool[] { true, false, false, true }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 04 00 00 00 01 00 00 01", buffer.GetHexDump()); BoolSequencePacket v2 = CDREncapsulation.Deserialize<BoolSequencePacket>(b uffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestBoolSequencePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1841175</p>

Tabla 4-77. TestShortSequencePacketLE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el ShortSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestShortSequencePacketLE() { ShortSequencePacket v1 = new ShortSequencePacket(new short[] { 0xFA1, 0xFF0, 0xB2F, 0x001 }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 04 00 00 00 A1 0F F0 0F 2F 0B 01 00", buffer.GetHexDump()); ShortSequencePacket v2 = CDREncapsulation.Deserialize<ShortSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestShortSequencePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1904143</p>

Tabla 4-78. TestEnumSequencePacketLE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
-----------------	---

Tabla 4-78. TestEnumSequencePacketLE

Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el EnumSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestEnumSequencePacketLE() { EnumSequencePacket v1 = new EnumSequencePacket(new MyEnum[] { MyEnum.Four, MyEnum.Three, MyEnum.Three, MyEnum.Zero, MyEnum.One, MyEnum.Five }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 06 00 00 00 04 00 00 00 03 00 00 00 03 00 00 00 00 00 00 01 00 00 00 05 00 00 00", buffer.GetHexDump()); EnumSequencePacket v2 = CDREncapsulation.Deserialize<EnumSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestEnumSequencePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1877371</p>

Tabla 4-79. TestIntSequencePacketLE

Llamada: public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)	
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el IntSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestIntSequencePacketLE() {</pre>

Tabla 4-79. TestIntSequencePacketLE

	<pre> IntSequencePacket v1 = new IntSequencePacket(new int[] { 0xFFA1F0, 0xFF230F, 0xB200F, 0xFFFFF01 }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 04 00 00 00 F0 A1 FF 00 0F 23 FF 00 0F 00 B2 00 01 FF FF 0F", buffer.GetHexDump()); IntSequencePacket v2 = CDREncapsulation.Deserialize<IntSequencePacket>(bu ffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestIntSequencePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1932013</p>

Tabla 4-80. TestEmptyBoolSequencePacketLE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el EmptyBoolSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestEmptyBoolSequencePacketLE() { BoolSequencePacket v1 = new BoolSequencePacket(new bool[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); } </pre>

Tabla 4-80. TestEmptyBoolSequencePacketLE

	<pre> buffer.Rewind(); Assert.AreEqual("00 01 00 00 00 00 00 00 00", buffer.GetHexDump()); BoolSequencePacket v2 = CDREncapsulation.Deserialize<BoolSequencePacket>(b uffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestEmptyBoolSequencePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1851345</p>

Tabla 4-81. TestEmptyShortSequencePacketLE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el EmptyShortSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestEmptyShortSequencePacketLE() { ShortSequencePacket v1 = new ShortSequencePacket(new short[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 00 00 00 00", buffer.GetHexDump()); ShortSequencePacket v2 = CDREncapsulation.Deserialize<ShortSequencePacket>(b uffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestEmptyShortSequencePacketLE</i></p>

Tabla 4-81. TestEmptyShortSequencePacketLE

	Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,2004389
--	--

Tabla 4-82. TestEmptyEnumSequencePacketLE

Llamada: <pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>	
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el EmptyEnumtSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestEmptyEnumSequencePacketLE() { EnumSequencePacket v1 = new EnumSequencePacket(new MyEnum[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 00 00 00 00 00", buffer.GetHexDump()); EnumSequencePacket v2 = CDREncapsulation.Deserialize<EnumSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	Nombre de la prueba: <i>TestEmptyEnumSequencePacketLE</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 0:00:00,1785581

Tabla 4-83. TestEmptyIntSequencePacketLE

Llamada: <pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order)</pre>
--

Tabla 4-83. TestEmptyIntSequencePacketLE

	<pre>public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el EmptyEnumtSequencePacket LittleEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestEmptyIntSequencePacketLE() { IntSequencePacket v1 = new IntSequencePacket(new int[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.LittleEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 01 00 00 00 00 00 00", buffer.GetHexDump()); IntSequencePacket v2 = CDREncapsulation.Deserialize<IntSequencePacket>(bu ffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestEmptyIntSequencePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1852795</p>

Tabla 4-84. TestBoolPacketBE

Llamada: <pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>	
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el BoolPacket BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestBoolPacketBE() { BoolPacket v1 = new BoolPacket(true); int bufferSize = sizeof(bool) + CDRHeaderSize;</pre>

Tabla 4-84. TestBoolPacketBE

	<pre> var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 01", buffer.GetHexDump()); BoolPacket v2 = CDREncapsulation.Deserialize<BoolPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestBoolPacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,204271</p>

Tabla 4-85. TestCharPacketBE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el CharPacket Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestCharPacketBE() { CharPacket v1 = new CharPacket('A'); int bufferSize = sizeof(char) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 41", buffer.GetHexDump()); CharPacket v2 = CDREncapsulation.Deserialize<CharPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>

Tabla 4-85. TestCharPacketBE

	}
Salida	<p>Nombre de la prueba: <i>TestCharPacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1953506</p>

Tabla 4-86. TestU8PackeBE

Llamada: public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)	
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el U8Packet Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	[TestMethod] public void TestU8PacketBE() { U8Packet v1 = new U8Packet(0xA); int bufferSize = sizeof(byte) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 0A", buffer.GetHexDump()); U8Packet v2 = CDREncapsulation.Deserialize<U8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }
Salida	<p>Nombre de la prueba: <i>TestU8PackeBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2004217</p>

Tabla 4-87. TestU16PacketBE

Llamada: public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order)
--

Tabla 4-87. TestU16PacketBE

	<pre>public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el U16Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestU16PacketBE() { U16Packet v1 = new U16Packet(0xAB); int bufferSize = sizeof(ushort) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 AB", buffer.GetHexDump()); U16Packet v2 = CDREncapsulation.Deserialize<U16Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestU16PacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2097027</p>

Tabla 4-88. TestU32PacketBE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el U32Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestU32PacketBE() { U32Packet v1 = new U32Packet(0xABA); int bufferSize = sizeof(uint) + CDRHeaderSize;</pre>

Tabla 4-88. TestU32PacketBE

	<pre> var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 0A BA", buffer.GetHexDump()); U32Packet v2 = CDREncapsulation.Deserialize<U32Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestU32PacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1931363</p>

Tabla 4-89. TestU64PacketBE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el U64Packet Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestU64PacketBE() { U64Packet v1 = new U64Packet(0xABCD); int bufferSize = sizeof(ulong) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 00 00 AB CD EF", buffer.GetHexDump()); U64Packet v2 = CDREncapsulation.Deserialize<U64Packet>(buffer); Assert.AreEqual(v1, v2); } </pre>

Tabla 4-89. TestU64PacketBE

	<pre> Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestU64PacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2023983</p>

Tabla 4-90. TestS8PacketBE1

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el S8Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS8PacketBE1() { S8Packet v1 = new S8Packet(-1); int bufferSize = sizeof(sbyte) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 FF", buffer.GetHexDump()); S8Packet v2 = CDREncapsulation.Deserialize<S8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS8PacketBE1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2144957</p>

Tabla 4-91. TestS8PacketBE2

Llamada:

Tabla 4-91. TestS8PacketBE2

	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S8Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS8PacketBE2() { S8Packet v1 = new S8Packet(+1); int bufferSize = sizeof(sbyte) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 01", buffer.GetHexDump()); S8Packet v2 = CDREncapsulation.Deserialize<S8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS8PacketBE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1883842</p>

Tabla 4-92. TestS16PacketBE1

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S16Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS16PacketBE1() { S16Packet v1 = new S16Packet(-10);</pre>

Tabla 4-92. TestS16PacketBE1

	<pre> int bufferSize = sizeof(short) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 FF F6", buffer.GetHexDump()); S16Packet v2 = CDREncapsulation.Deserialize<S16Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestS16PacketBE1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2190627</p>

Tabla 4-93. TestS16PacketBE2

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el S16Packet Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestS16PacketBE2() { S16Packet v1 = new S16Packet(+10); int bufferSize = sizeof(short) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 0A", buffer.GetHexDump()); S16Packet v2 = CDREncapsulation.Deserialize<S16Packet>(buffer); Assert.AreEqual(v1, v2); } </pre>

Tabla 4-93. TestS16PacketBE2

	<pre> Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS16PacketBE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1791405</p>

Tabla 4-94. TestS32PacketBE1

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el S32Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS32PacketBE1() { S32Packet v1 = new S32Packet(-0xABA); int bufferSize = sizeof(int) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 FF FF F5 46", buffer.GetHexDump()); S32Packet v2 = CDREncapsulation.Deserialize<S32Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS32PacketBE1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2366255</p>

Tabla 4-95. TestS32PacketBE2

Llamada:

Tabla 4-95. TestS32PacketBE2

	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S32Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS32PacketBE2() { S32Packet v1 = new S32Packet(0xABA); int bufferSize = sizeof(int) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 0A BA", buffer.GetHexDump()); S32Packet v2 = CDREncapsulation.Deserialize<S32Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS32PacketBE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1962171</p>

Tabla 4-96. TestS64PacketBE1

	Llamada: <pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el S4Packet BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestS64PacketBE1() { S64Packet v1 = new S64Packet(-0xABCD);</pre>

Tabla 4-96. TestS64PacketBE1

	<pre> int bufferSize = sizeof(long) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 FF FF FF FF FF FF 54 33", buffer.GetHexDump()); S64Packet v2 = CDREncapsulation.Deserialize<S64Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestS64PacketBE1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2031737</p>

Tabla 4-97. TestS64PacketLE2

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el S4Packet Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestS64PacketBE2() { S64Packet v1 = new S64Packet(0xABCD); int bufferSize = sizeof(long) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 00 00 00 AB CD", buffer.GetHexDump()); S64Packet v2 = CDREncapsulation.Deserialize<S64Packet>(buffer); Assert.AreEqual(v1, v2); } </pre>

Tabla 4-97. TestS64PacketLE2

	<pre> Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS64PacketLE2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1994617</p>

Tabla 4-98. TestSinglePacketBE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el SinglePacket BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestSinglePacketBE() { SinglePacket v1 = new SinglePacket(0.1f); int bufferSize = sizeof(float) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 3D CC CC CD", buffer.GetHexDump()); SinglePacket v2 = CDREncapsulation.Deserialize<SinglePacket>(buffer) ; Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestSinglePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1785456</p>

Tabla 4-99. TestDoublePacketLE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el DoublePacket Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestDoublePacketBE() { DoublePacket v1 = new DoublePacket(0.1); int bufferSize = sizeof(double) + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 3F B9 99 99 99 99 99 9A", buffer.GetHexDump()); DoublePacket v2 = CDREncapsulation.Deserialize<DoublePacket>(buffer) ; Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestDoublePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,199862</p>

Tabla 4-100. TestBoolSequencePacketBE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el BoolSequencePacket Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador

Tabla 4-100. TestBoolSequencePacketBE

Código	<pre>[TestMethod] public void TestBoolSequencePacketBE() { BoolSequencePacket v1 = new BoolSequencePacket(new bool[] { true, false, false, true }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 04 01 00 00 01", buffer.GetHexDump()); BoolSequencePacket v2 = CDREncapsulation.Deserialize<BoolSequencePacket>(b uffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestBoolSequencePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1890411</p>

Tabla 4-101. TestShortSequencePacketBE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el ShortSequencePacket Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestShortSequencePacketBE() { ShortSequencePacket v1 = new ShortSequencePacket(new short[] { 0xFA1, 0xFF0, 0xB2F, 0x001 }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize);</pre>

Tabla 4-101. TestShortSequencePacketBE

	<pre> CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 04 0F A1 0F F0 0B 2F 00 01", buffer.GetHexDump()); ShortSequencePacket v2 = CDREncapsulation.Deserialize<ShortSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } } </pre>
Salida	<p>Nombre de la prueba: <i>TestShortSequencePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1981403</p>

Tabla 4-102. TestEnumSequencePacketBE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el EnumSequencePacket Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestEnumSequencePacketBE() { EnumSequencePacket v1 = new EnumSequencePacket(new MyEnum[] { MyEnum.Four, MyEnum.Three, MyEnum.Three, MyEnum.Zero, MyEnum.One, MyEnum.Five }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 06 00 00 00 04 00 00 00 03 00 00 00 03 00 00 00 00 00 00 00 01 00 00 00 00 05", buffer.GetHexDump()); } </pre>

Tabla 4-102. TestEnumSequencePacketBE

	<pre> EnumSequencePacket v2 = CDREncapsulation.Deserialize<EnumSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestEnumSequencePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1841432</p>

Tabla 4-103. TestIntSequencePacketBE

Llamada:	<pre> public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer) </pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el IntSequencePacket Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestIntSequencePacketBE() { IntSequencePacket v1 = new IntSequencePacket(new int[] { 0xFFA1F0, 0xFF230F, 0xB200F, 0xFFFFF01 }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 04 00 FF A1 F0 00 FF 23 0F 00 B2 00 0F 0F FF FF 01", buffer.GetHexDump()); IntSequencePacket v2 = CDREncapsulation.Deserialize<IntSequencePacket>(bu ffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>

Tabla 4-103. TestIntSequencePacketBE

Salida	<p>Nombre de la prueba: <i>TestIntSequencePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1821422</p>
---------------	--

Tabla 4-104. TestEmptyBoolSequencePacketBE

Llamada:	<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el EmptyBoolSequencePacket BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestEmptyBoolSequencePacketBE() { BoolSequencePacket v1 = new BoolSequencePacket(new bool[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 00 00", buffer.GetHexDump()); BoolSequencePacket v2 = CDREncapsulation.Deserialize<BoolSequencePacket>(b uffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestEmptyBoolSequencePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1865145</p>

Tabla 4-105. TestEmptyShortSequencePacketBE

Llamada:

Tabla 4-105. TestEmptyShortSequencePacketBE

<pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>	
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el EmptyShortSequencePacket BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre>[TestMethod] public void TestEmptyShortSequencePacketBE() { ShortSequencePacket v1 = new ShortSequencePacket(new short[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 00 00", buffer.GetHexDump()); ShortSequencePacket v2 = CDREncapsulation.Deserialize<ShortSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestEmptyShortSequencePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,2001416</p>

Tabla 4-106. TestEmptyEnumSequencePacketBE

Llamada: <pre>public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)</pre>	
Descripción	En esta prueba se muestra el corrector funcionamiento del CDREncapsulation para el EmptyEnumtSequencePacket BigEndian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	[TestMethod]

Tabla 4-106. TestEmptyEnumSequencePacketBE

	<pre> public void TestEmptyEnumSequencePacketBE() { EnumSequencePacket v1 = new EnumSequencePacket(new MyEnum[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 00 00", buffer.GetHexDump()); EnumSequencePacket v2 = CDREncapsulation.Deserialize<EnumSequencePacket>(b uffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestEmptyEnumSequencePacketBE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1854031</p>

Tabla 4-107. TestEmptyIntSequencePacketLE

Llamada: public CDREncapsulation(IoBuffer buffer, object dataObj, ByteOrder order) public static void Serialize(IoBuffer buffer, object dataObj, ByteOrder order) public static T Deserialize<T>(IoBuffer buffer)	
Descripción	En esta prueba se muestra el correcto funcionamiento del CDREncapsulation para el EmptyEnumtSequencePacket Big Endian
Entrada	Dentro del Test Initialize se inicializa el serializador
Código	<pre> [TestMethod] public void TestEmptyIntSequencePacketBE() { IntSequencePacket v1 = new IntSequencePacket(new int[] { }); int bufferSize = v1.Size + ArrayHeader + CDRHeaderSize; var buffer = ByteBufferAllocator.Instance.Allocate(bufferSize); CDREncapsulation.Serialize(buffer, v1, ByteOrder.BigEndian); Assert.AreEqual(bufferSize, buffer.Position); } </pre>

Tabla 4-107. TestEmptyIntSequencePacketLE

	<pre> buffer.Rewind(); Assert.AreEqual("00 00 00 00 00 00 00 00 00", buffer.GetHexDump()); IntSequencePacket v2 = CDREncapsulation.Deserialize<IntSequencePacket>(bu ffer); Assert.AreEqual(v1, v2); Assert.AreEqual(bufferSize, buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestEmptyIntSequencePacketLE</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1957202</p>

4.2.4.3. Pruebas de exploración de tipo.

En esta prueba se verifica que la serialización de la clase *ddsType* funciona de manera adecuada.

Tabla 4-108. TestExploreMyClass1

Llamada: public static org.omg.dds.type.typeobject.Type ExploreType(System.Type type)	
Descripción	En esta prueba se verifica el correcto funcionamiento del serializador de DDS en una Clase
Entrada	Inicialmente no se tiene inicializado al <i>ddsType</i>
Código	<pre> [TestMethod] public void TestExplore MyClass1() { var ddsType = TypeExplorer.ExploreType(typeof(X MyClass1)); Assert.IsNotNull(ddsType); Assert.IsNotNull(ddsType.GetProperty()); var propInfo = ddsType.GetProperty(); Assert.AreEqual("SerializerTests.X MyClass1", propInfo.Name); Assert.IsInstanceOfType(ddsType, typeof(StructureType)); StructureType structType = ddsType as StructureType; var members = structType.GetMember(); Assert.IsNotNull(members); Assert.AreEqual(3, members.Count); Assert.AreEqual("m_byte", members[0].GetProperty().Name); } </pre>

Tabla 4-108. TestExploreMyClass1

	<pre> Assert.AreEqual("m_int", members[1].GetProperty().Name); Assert.AreEqual("m_short", members[2].GetProperty().Name); Assert.AreEqual((uint)0x8001, members[0].GetProperty().MemberId); Assert.AreEqual((uint)0x8002, members[1].GetProperty().MemberId); Assert.AreEqual((uint)0x8003, members[2].GetProperty().MemberId); </pre>
Salida	<p>Nombre de la prueba: <i>TestExplore MyClass1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:00,1996965</p>

4.2.4.4. Pruebas de Paquetes.

En estas pruebas se verifica que los diferentes tipos de datos se están serializando y deserializando de manera adecuada.

Tabla 4-109. TestBoolPacket

Llamada:	<code>public BoolPacket(bool v)</code>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>BoolPacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre> [TestMethod] public void TestBoolPacket() { BoolPacket v1 = new BoolPacket(true); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(bool)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(bool), buffer.Position); buffer.Rewind(); BoolPacket v2 = Serializer.Deserialize<BoolPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(bool), buffer.Position); } </pre>
Salida	<p>Nombre de la prueba: <i>TestBoolPacket</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 0:00:39.9332615</p>

Tabla 4-110. TestCharPacket

Llamada:
<code>public CharPacket(char v)</code>

Tabla 4-110. TestCharPacket

Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>CharPacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestCharPacket() { CharPacket v1 = new CharPacket('A'); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(char)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(char), buffer.Position); buffer.Rewind(); CharPacket v2 = Serializer.Deserialize<CharPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(char), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestCharPacket</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-111. TestU8Packet

Llamada:	public U8Packet(byte v)
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>U8Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Referencia	
Código	<pre>[TestMethod] public void TestU8Packet() { U8Packet v1 = new U8Packet(0xA); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(byte)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(byte), buffer.Position); buffer.Rewind(); U8Packet v2 = Serializer.Deserialize<U8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(byte), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestU8Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-112. TestU16Packet

Llamada:

Tabla 4-112. TestU16Packet

	public U16Packet(ushort v)
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>U16Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestU16Packet() { U16Packet v1 = new U16Packet(0xAB); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(ushort)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(ushort), buffer.Position); buffer.Rewind(); U16Packet v2 = Serializer.Deserialize<U16Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(ushort), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestU16Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-113. TestU32Packet

	Llamada: public U32Packet(uint v)
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>U32Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestU32Packet() { U32Packet v1 = new U32Packet(0xABA); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(uint)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(uint), buffer.Position); buffer.Rewind(); U32Packet v2 = Serializer.Deserialize<U32Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(uint), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestU32Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-114. TestU64Packet

Llamada:	<pre>public U64Packet(ulong v)</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>U64Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestU64Packet() { U64Packet v1 = new U64Packet(0xABCD); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(ulong)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(ulong), buffer.Position); buffer.Rewind(); U64Packet v2 = Serializer.Deserialize<U64Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(ulong), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestU64Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-115. TestS8Packet

Llamada:	<pre>public S8Packet(sbyte v)</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>S8Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestS8Packet() { S8Packet v1 = new S8Packet(0xA); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(sbyte)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(sbyte), buffer.Position); buffer.Rewind(); S8Packet v2 = Serializer.Deserialize<S8Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(sbyte), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS8Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-116. TestS16Packet

Llamada:	<pre>public S16Packet(short v)</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>S16Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestS16Packet() { S16Packet v1 = new S16Packet(0xAB); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(short)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(short), buffer.Position); buffer.Rewind(); S16Packet v2 = Serializer.Deserialize<S16Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(short), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS16Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-117. TestS32Packet

Llamada:	<pre>public S32Packet(int v)</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>S32Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestS32Packet() { S32Packet v1 = new S32Packet(0xABA); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(int)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(int), buffer.Position); buffer.Rewind(); S32Packet v2 = Serializer.Deserialize<S32Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(int), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS32Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-118. TestS64Packet

Llamada:	<pre>public S64Packet(long v)</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>S64Packet</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestS64Packet() { S64Packet v1 = new S64Packet(0xABCD); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(long)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(long), buffer.Position); buffer.Rewind(); S64Packet v2 = Serializer.Deserialize<S64Packet>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(long), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestS64Packet</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-119. TestSinglePacket

Llamada:	<pre>public SinglePacket(float v)</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>SinglePacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestSinglePacket() { SinglePacket v1 = new SinglePacket(0.1f); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(float)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(float), buffer.Position); buffer.Rewind(); SinglePacket v2 = Serializer.Deserialize<SinglePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(float), buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestSinglePacket</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-120. TestDoublePacket

Llamada:	public DoublePacket(double v)
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>DoublePacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	[TestMethod] public void TestDoublePacket() { DoublePacket v1 = new DoublePacket(0.1); var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(double)); Serializer.Serialize(buffer, v1); Assert.AreEqual(sizeof(double), buffer.Position); buffer.Rewind(); DoublePacket v2 = Serializer.Deserialize<DoublePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(sizeof(double), buffer.Position); }
Salida	Nombre de la prueba: <i>TestDoublePacket</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: < 1 ms

Tabla 4-121. TestPrimitivesPacket

Llamada:	public PrimitivesPacket(int seed)
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>PrimitivesPacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	[TestMethod] public void TestPrimitivesPacket() { int size = PrimitivesPacket.Size(); PrimitivesPacket v1 = new PrimitivesPacket(15); var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); PrimitivesPacket v2 = Serializer.Deserialize<PrimitivesPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }
Salida	Nombre de la prueba: <i>TestPrimitivesPacket</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 1 ms

Tabla 4-122. TestEnumPacket

Llamada:	public EnumPacket(MyEnum v)
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>EnumPacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestEnumPacket() { int size = EnumPacket.Size(); EnumPacket v1 = new EnumPacket(MyEnum.Three); var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); EnumPacket v2 = Serializer.Deserialize<EnumPacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestEnumPacket</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-123. TestStruct1Packet

Llamada:	public struct MyStruct1
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>MyStruct1</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestStruct1Packet() { int size = MyStruct1.Size(); MyStruct1 v1 = new MyStruct1(); v1.m_byte = 1; v1.m_int = 2; v1.m_long = 3; var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); MyStruct1 v2 = Serializer.Deserialize<MyStruct1>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	Nombre de la prueba: <i>TestStruct1Packet</i>

Tabla 4-123. TestStruct1Packet

	Resultado de la prueba: 1 Prueba superada Duración de la prueba: 1 ms
--	---

Tabla 4-124. TestStructMessagePacket

Llamada: <code>public StructMessage()</code>	
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>StructMessage</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestStructMessagePacket() { int size = StructMessage.Size(); StructMessage v1 = new StructMessage(); var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); StructMessage v2 = Serializer.Deserialize<StructMessage>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	Nombre de la prueba: <i>TestStructMessagePacket</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: 1 ms

Tabla 4-125. TestSequenceMessagePacket

Llamada: <code>public IntSequencePacket(int[] v)</code>	
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>IntSequencePacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestSequenceMessagePacket() { IntSequencePacket v1 = new IntSequencePacket(new int[] { 1, 2, 3 }); int size = (1 + v1.m_val.Length) * sizeof(int); var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); IntSequencePacket v2 = Serializer.Deserialize<IntSequencePacket>(buffer);</pre>

Tabla 4-125. TestSequenceMessagePacket

	<pre>Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestSequenceMessagePacket</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-126. TestSequenceMessagePacket2

Llamada:	<code>public IntSequencePacket(int[] v)</code>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>IntSequencePacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestSequenceMessagePacket2() { IntSequencePacket v1 = new IntSequencePacket(new int[] { }); int size = (1 + v1.m_val.Length) * sizeof(int); var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); IntSequencePacket v2 = Serializer.Deserialize<IntSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestSequenceMessagePacket2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-127. TestSequenceMessagePacket3

Llamada:	<code>public IntSequencePacket(int[] v)</code>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>IntSequencePacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestSequenceMessagePacket3() { IntSequencePacket v1 = new IntSequencePacket(null); int size = (1) * sizeof(int); var buffer = ByteBufferAllocator.Instance.Allocate(size);</pre>

Tabla 4-127. TestSequenceMessagePacket3

	<pre> Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); IntSequencePacket v2 = Serializer.Deserialize<IntSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); } } </pre>
Salida	<p>Nombre de la prueba: <i>TestSequenceMessagePacket3</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-128. TestSequenceMessagePacket3

Llamada:	<pre>public IntSequencePacket(int[] v)</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente el <i>IntSequencePacket</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre> [TestMethod] public void TestSequenceMessagePacket3() { IntSequencePacket v1 = new IntSequencePacket(null); int size = (1) * sizeof(int); var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); IntSequencePacket v2 = Serializer.Deserialize<IntSequencePacket>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); } } </pre>
Salida	<p>Nombre de la prueba: <i>TestSequenceMessagePacket3</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-129. TestMyClassListMessagePacket1

Llamada:	<pre>public class MyClassList</pre>
Descripción	En esta prueba se verifica que se está serializando y deserializando correctamente la clase <i>MyClassList</i> .
Entrada	En la prueba no se tiene inicializado el fichero.

Tabla 4-129. TestMyClassListMessagePacket1

Código	<pre>[TestMethod] public void TestMyClassListMessagePacket1() { MyClassList v1 = new MyClassList(); v1.m_intlist = new List<int>() { 5, 6, 7 }; int size = (v1.m_intlist.Count + 1) * 4 + 1 * 4; var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); MyClassList v2 = Serializer.Deserialize<MyClassList>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestMyClassListMessagePacket1</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-130. TestMyClassListMessagePacket2

Llamada: public class MyClassList	
Descripción	En esta prueba se verifica que está serializando y deserializando correctamente la clase <i>MyClassList</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestMyClassListMessagePacket2() { MyClassList v1 = new MyClassList(); v1.m_intlist = new List<int>(); int size = (v1.m_intlist.Count + 1) * 4 + 1 * 4; var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); MyClassList v2 = Serializer.Deserialize<MyClassList>(buffer); Assert.AreEqual(v1, v2); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestMyClassListMessagePacket2</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-131. TestMyClassListMessagePacket3

Llamada:	<pre>public class MyClassList</pre>
Descripción	En esta prueba se verifica que está serializando y deserializando correctamente la clase <i>MyClassList</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestMyClassListMessagePacket3() { MyClassList v1 = new MyClassList(); v1.m_intlist = null; int size = 1 * 4 + 1 * 4; var buffer = ByteBufferAllocator.Instance.Allocate(size); Serializer.Serialize(buffer, v1); Assert.AreEqual(size, buffer.Position); buffer.Rewind(); MyClassList v2 = Serializer.Deserialize<MyClassList>(buffer); Assert.IsNotNull(v1.m_intlist); Assert.IsNotNull(v2.m_intlist); Assert.AreEqual(v1.m_int, v2.m_int); Assert.AreEqual(size, buffer.Position); }</pre>
Salida	<p>Nombre de la prueba: <i>TestMyClassListMessagePacket3</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

4.2.4.5. Pruebas de Primitivas.

En estas pruebas se verifica que los tipos de datos primitivos están siendo serializados y deserializados de manera adecuada.

Tabla 4-132. TestDouble

Llamada:	<pre>public struct Double : IComparable, IFormattable, IConvertible, IComparable<double>, IEquatable<double></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>doublé</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestDouble() { double v1 = 1.0; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(double)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); double v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2);</pre>

Tabla 4-132. TestDouble

	<pre> Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestDouble</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-133. TestSingle

Llamada:	<pre>public struct Single : IComparable, IFormattable, IConvertible, IComparable<float>, IEquatable<float></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>float</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestSingle() { float v1 = 1.0f; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(float)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); float v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestSingle</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-134. TestShort

Llamada:	<pre>public struct Int16 : IComparable, IFormattable, IConvertible, IComparable<short>, IEquatable<short></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>short</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestShort() { short v1 = 1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(short)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); short v2;</pre>

Tabla 4-134. TestShort

	<pre>Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	Nombre de la prueba: <i>TestShort</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: < 1 ms

Tabla 4-135. TestUShort

Llamada:	<pre>public struct UInt16 : IComparable, IFormattable, IConvertible, IComparable<ushort>, IEquatable<ushort></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>UShort</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestUShort() { ushort v1 = 1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(ushort)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); ushort v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	Nombre de la prueba: <i>TestUShort</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: < 1 ms

Tabla 4-136. TestInt

Llamada:	<pre>public struct Int32 : IComparable, IFormattable, IConvertible, IComparable<int>, IEquatable<int></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>int</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestInt() { int v1 = 1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(int)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1);</pre>

Tabla 4-136. TestInt

	<pre>buffer.Rewind(); int v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestInt</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-137. TestUInt

Llamada:	<pre>public struct UInt32 : IComparable, IFormattable, IConvertible, IComparable<uint>, IEquatable<uint></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>uint</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestUInt() { uint v1 = 1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(uint)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); uint v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestUInt</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-138. TestLong

Llamada:	<pre>public struct Int64 : IComparable, IFormattable, IConvertible, IComparable<long>, IEquatable<long></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>long</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestLong()</pre>

Tabla 4-138. TestLong

	<pre>{ long v1 = 1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(long)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); long v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	Nombre de la prueba: <i>TestLong</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: < 1 ms

Tabla 4-139. TestULong

Llamada:	public struct UInt64 : IComparable, IFormattable, IConvertible, IComparable<ulong>, IEquatable<ulong>
Descripción	En esta prueba se verifica el estado de la primitiva <i>ulong</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestULong() { ulong v1 = 1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(ulong)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); ulong v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	Nombre de la prueba: <i>TestULong</i> Resultado de la prueba: 1 Prueba superada Duración de la prueba: < 1 ms

Tabla 4-140. TestChar

Llamada:	public struct Char : IComparable, IConvertible, IComparable<char>, IEquatable<char>
Descripción	En esta prueba se verifica el estado de la primitiva <i>char</i> .

Tabla 4-140. TestChar

Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestChar() { char v1 = 'a'; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(char)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); char v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestChar</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-141. TestByte

Llamada:	public struct Byte : IComparable, IFormattable, IConvertible, IComparable<byte>, IEquatable<byte>
Descripción	En esta prueba se verifica el estado de la primitiva byte.
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestByte() { byte v1 = 1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(byte)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); byte v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestByte</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-142. TestSByte

Llamada:	public struct SByte : IComparable, IFormattable, IConvertible, IComparable<sbyte>, IEquatable<sbyte>
Descripción	En esta prueba se verifica el estado de la primitiva sbyte.

Tabla 4-142. TestSByte

Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestSByte() { sbyte v1 = -1; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(sbyte)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); sbyte v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestSByte</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-143. TestBool

Llamada:	<pre>public struct Boolean : IComparable, IConvertible, IComparable<bool>, IEquatable<bool></pre>
Descripción	En esta prueba se verifica el estado de la primitiva <i>bool</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestBool() { bool v1 = true; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(bool)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); bool v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestBool</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: < 1 ms</p>

Tabla 4-144. TestDateTime

Llamada:	<pre>public struct DateTime : IComparable, IFormattable, IConvertible, ISerializable, IComparable<DateTime>, IEquatable<DateTime></pre>
-----------------	---

Tabla 4-144. TestDateTime

Descripción	En esta prueba se verifica el estado de la primitiva <i>DateTime</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestDateTime() { DateTime v1 = DateTime.Now; var buffer = ByteBufferAllocator.Instance.Allocate(sizeof(long)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); DateTime v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestDateTime</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 1 ms</p>

Tabla 4-145. TestString

Llamada:	
<pre>public sealed class String : IComparable, ICloneable, IConvertible, IComparable<string>, IEnumerable<char>, IEnumerable, IEquatable<string></pre>	
Descripción	En esta prueba se verifica el estado de la primitiva <i>string</i> .
Entrada	En la prueba no se tiene inicializado el fichero.
Código	<pre>[TestMethod] public void TestString() { string v1 = "this is a string!"; var buffer = ByteBufferAllocator.Instance.Allocate(1 + v1.Length * sizeof(char)); Doopec.Serializer.Primitives.WritePrimitive(buffer, v1); buffer.Rewind(); string v2; Doopec.Serializer.Primitives.ReadPrimitive(buffer, out v2); Assert.AreEqual(v1, v2); }</pre>
Salida	<p>Nombre de la prueba: <i>TestString</i></p> <p>Resultado de la prueba: 1 Prueba superada</p> <p>Duración de la prueba: 2 ms</p>

4.3. PRUEBA APLICACIÓN RTPS

A continuación, se muestra los requerimientos necesarios para el uso de un chat trabajando bajo el Middleware DDS-RTPS, se presenta además la interfaz de aplicación.

4.3.1. REQUERIMIENTOS PARA EL USO DE LA APLICACIÓN

Para usar la aplicación se debe tener en cuenta los siguientes requerimientos:

- Sistema Operativo Windows 7, 8, 8.1 y 10 o superior.
- Copiar el paquete del CD.

4.3.2. APLICACIÓN CHAT CON TECNOLOGÍA DDS-RTPS

Se muestra la interfaz de usuario y las capturas de paquetes.

4.3.2.1. Interfaz de Usuario.

- 1) Ejecutar la aplicación

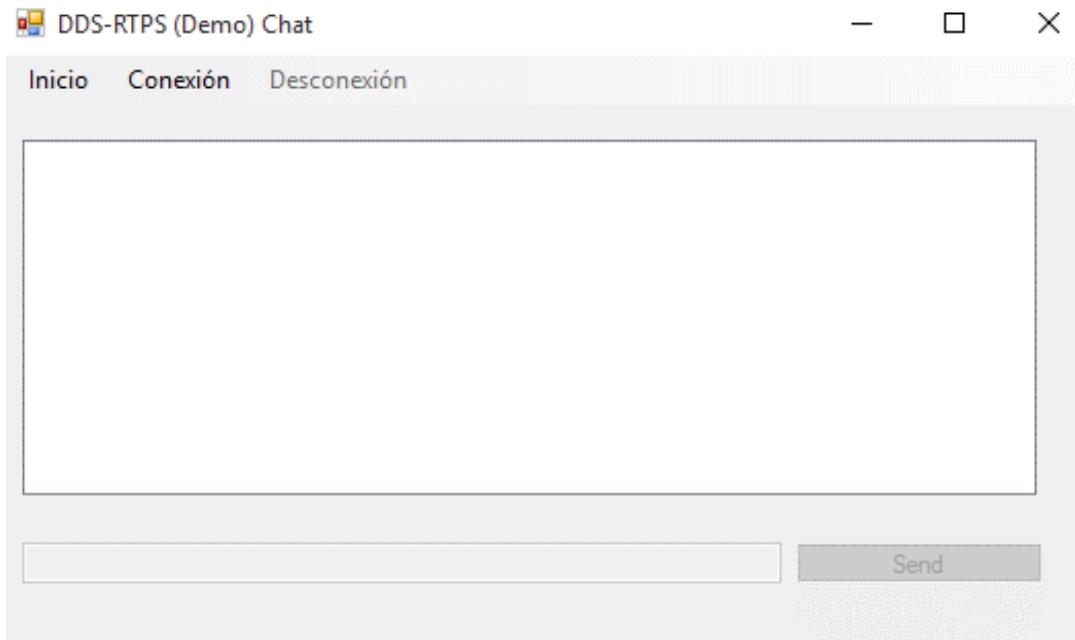


Figura 4-1. Ejecución del Chat RTPS.

- 2) Conectarse al servicio de chat

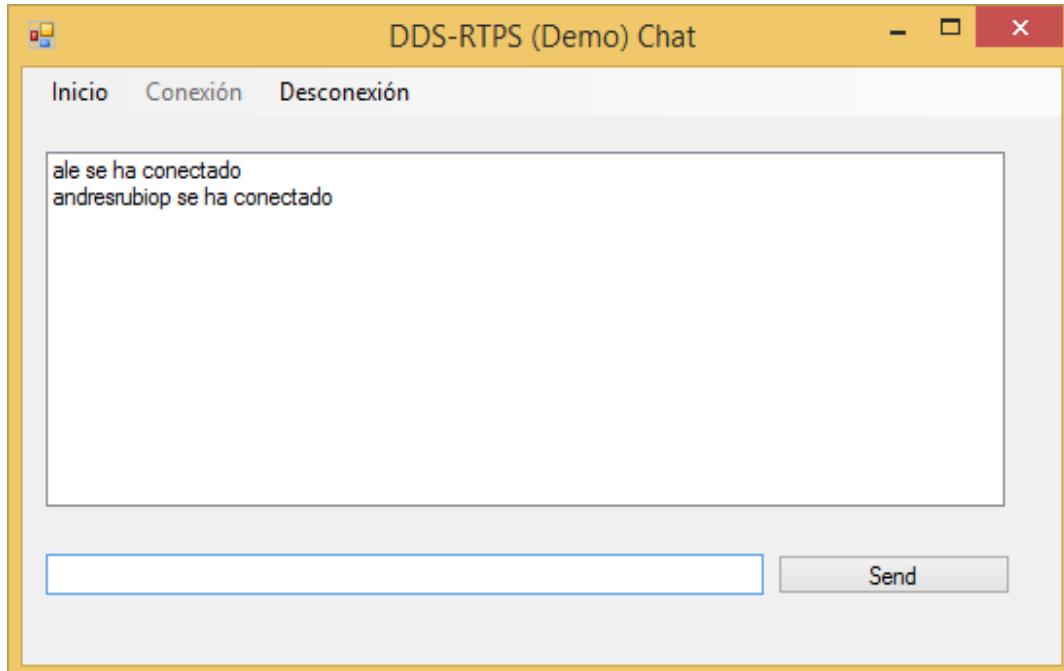


Figura 4-2. Conexión al servicio de Chat.

3) Intercambio de mensajes

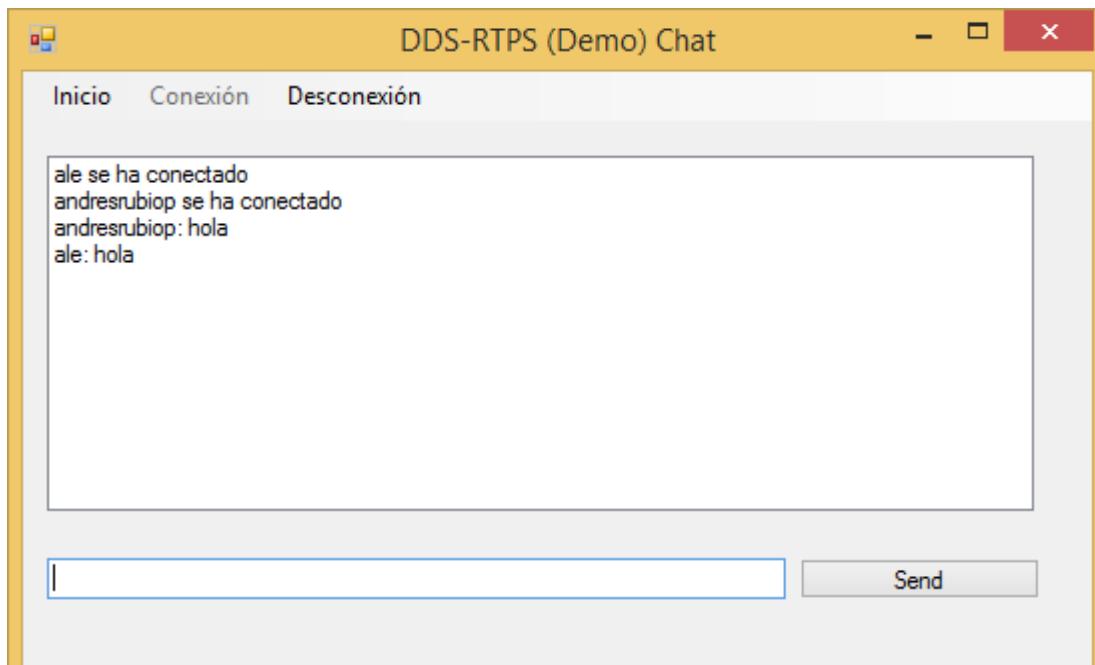


Figura 4-3. Intercambio de mensajes.

4) Desconexión del servicio de chat

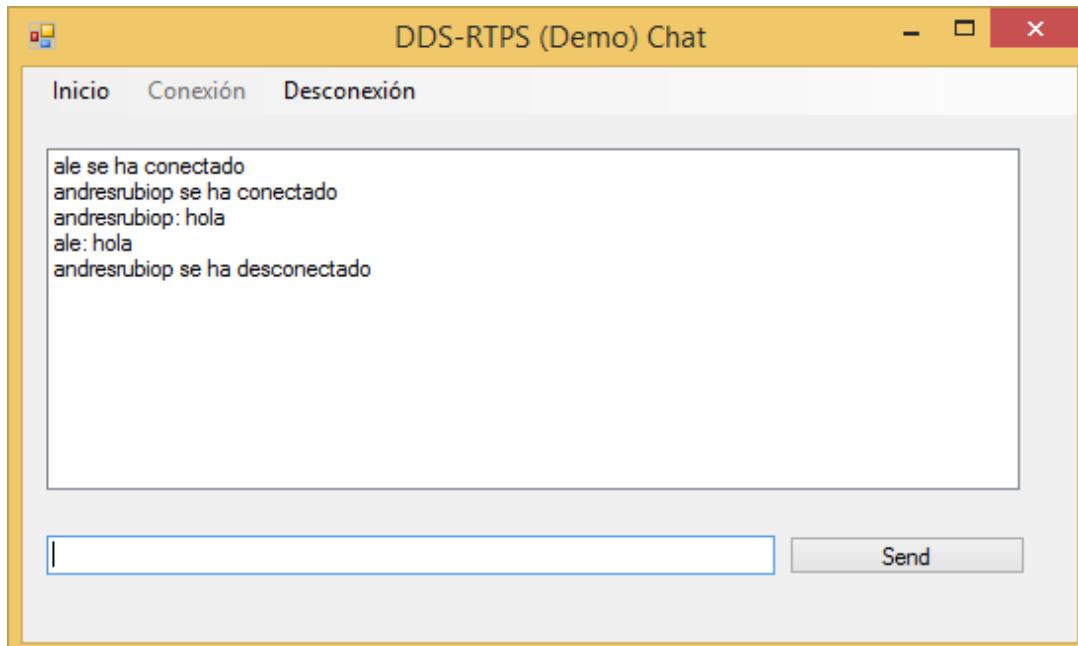


Figura 4-4. Desconexión del servicio de Chat.

5) Salida de la aplicación

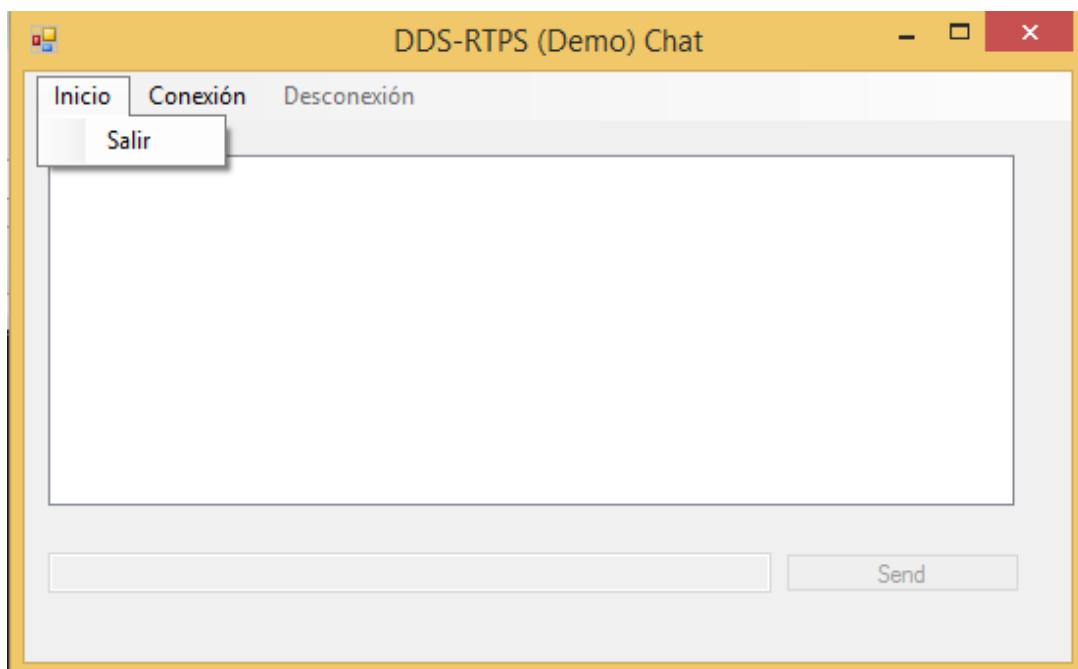


Figura 4-5. Salida de la aplicación.

4.3.2.2. Captura de paquetes.

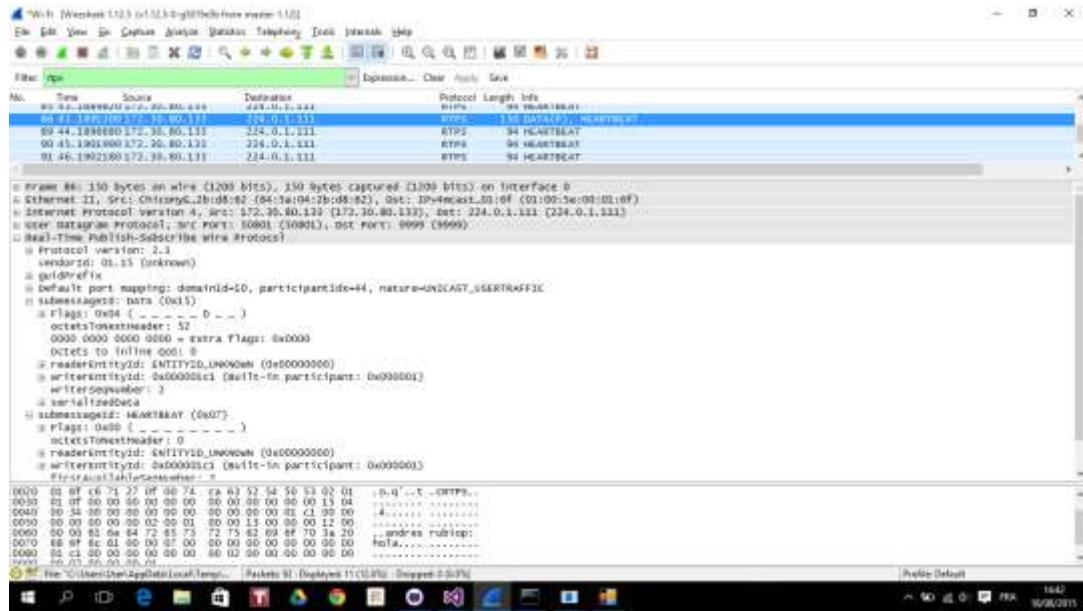


Figura 4-6. Paquete RTPS dentro del Chat.

4.4. PRUEBA APLICACIÓN CORBA

A continuación, se muestra los requerimientos necesarios para el uso de un chat con la tecnología CORBA y se presenta además la interfaz de aplicación.

4.4.1. REQUERIMIENTOS PARA EL USO DE LA APLICACIÓN

Para usar la aplicación se debe tener en cuenta los siguientes requerimientos:

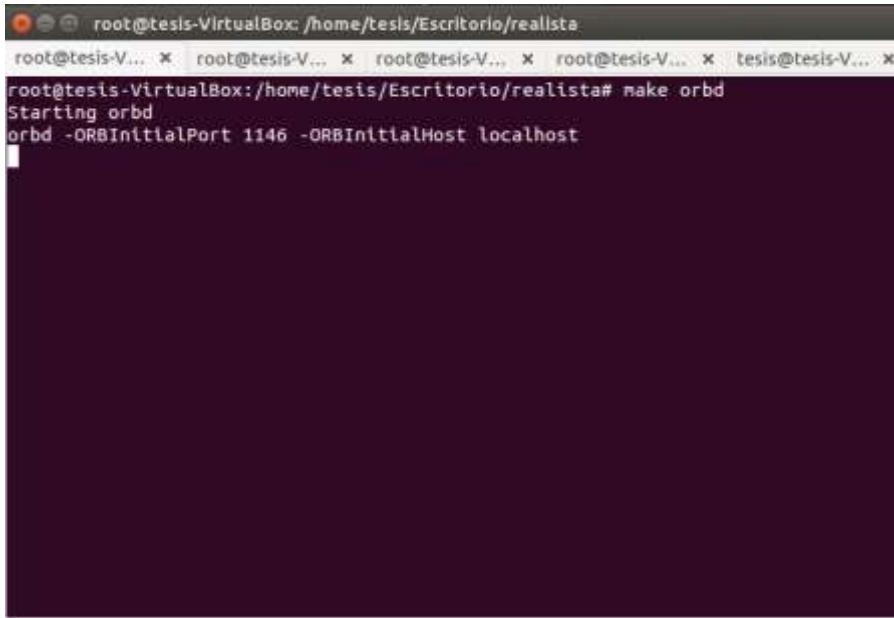
- Sistema Operativo Ubuntu 14.04 o superior.
- Instalar el comando *make*. Usando el comando *apt-get install build-essential*, dentro del terminal.
- Instalar la máquina virtual de Java llamada *Javac*. Usando el comando *apt-get install openjdk-7-jdk*, dentro del terminal.

4.4.2. APLICACIÓN CHAT CON TECNOLOGÍA CORBA

Se muestra la interfaz de usuario y las capturas de paquetes.

4.4.2.1. Interfaz de Usuario

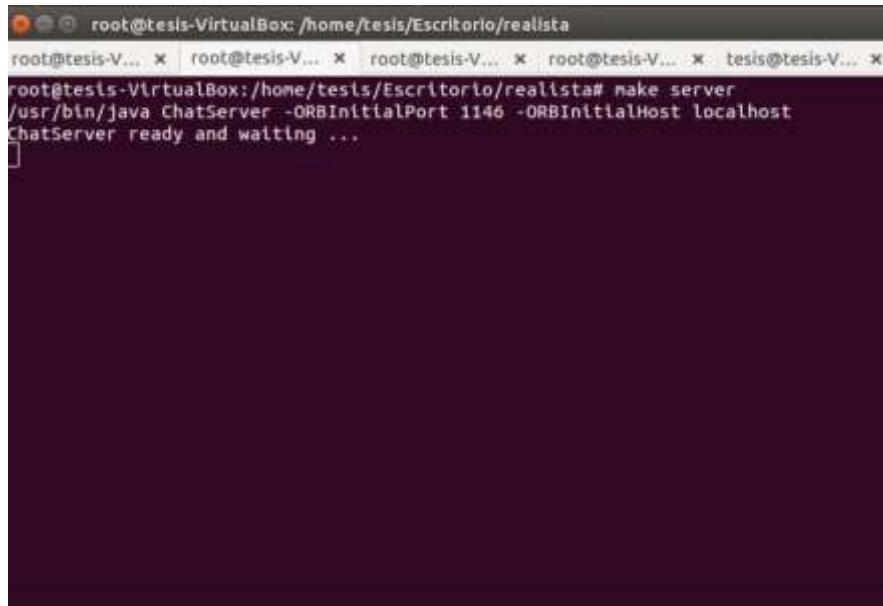
- 1) Inicialización del objeto CORBA.



```
root@tesis-VirtualBox: /home/tesis/Escritorio/realista
root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ tesis@tesis-V... ✘
root@tesis-VirtualBox:/home/tesis/Escritorio/realista# make orbd
Starting orbd
orbd -ORBInitialPort 1146 -ORBInitialHost localhost
```

Figura 4-7. Chat CORBA 1

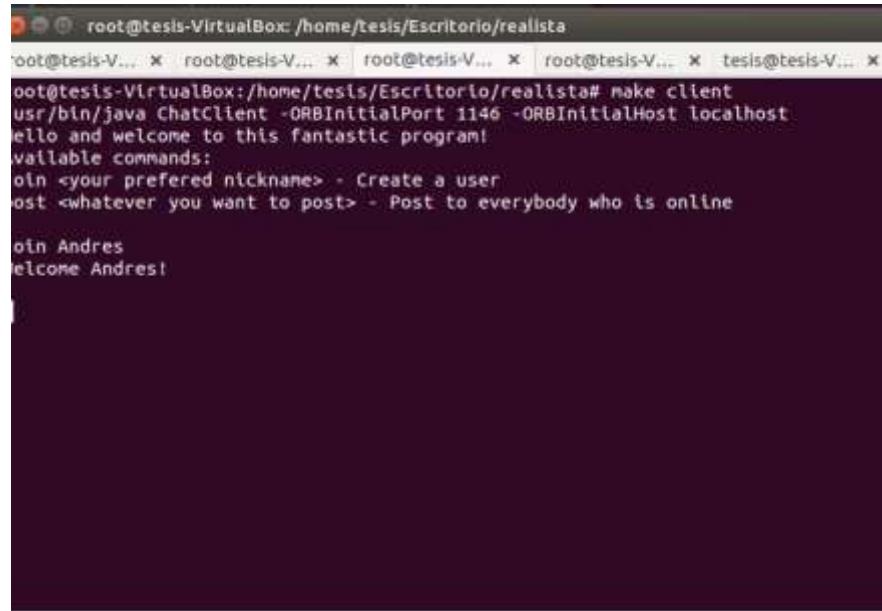
2) Inicialización del servidor.



```
root@tesis-VirtualBox: /home/tesis/Escritorio/realista
root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ tesis@tesis-V... ✘
root@tesis-VirtualBox:/home/tesis/Escritorio/realista# make server
/usr/bin/java ChatServer -ORBInitialPort 1146 -ORBInitialHost localhost
ChatServer ready and waiting ...
```

Figura 4-8. Chat CORBA 2

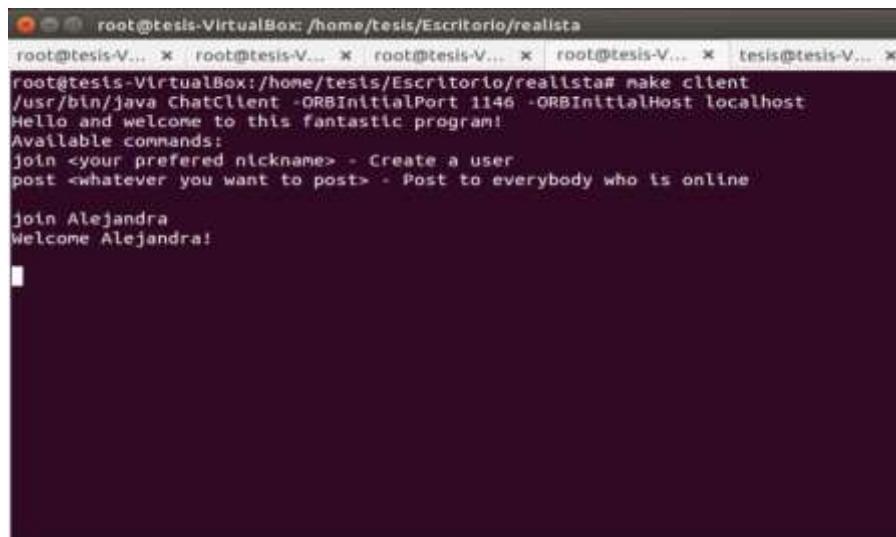
3) Inicialización de clientes.



```
root@tesis-VirtualBox:/home/tesis/Escritorio/realista
root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘
root@tesis-VirtualBox:/home/tesis/Escritorio/realista# make client
/usr/bin/java ChatClient -ORBInitialPort 1146 -ORBInitialHost localhost
Hello and welcome to this fantastic program!
Available commands:
join <your preferred nickname> - Create a user
post <whatever you want to post> - Post to everybody who is online

join Andres
Welcome Andres!
```

Figura 4-9. Chat CORBA 3

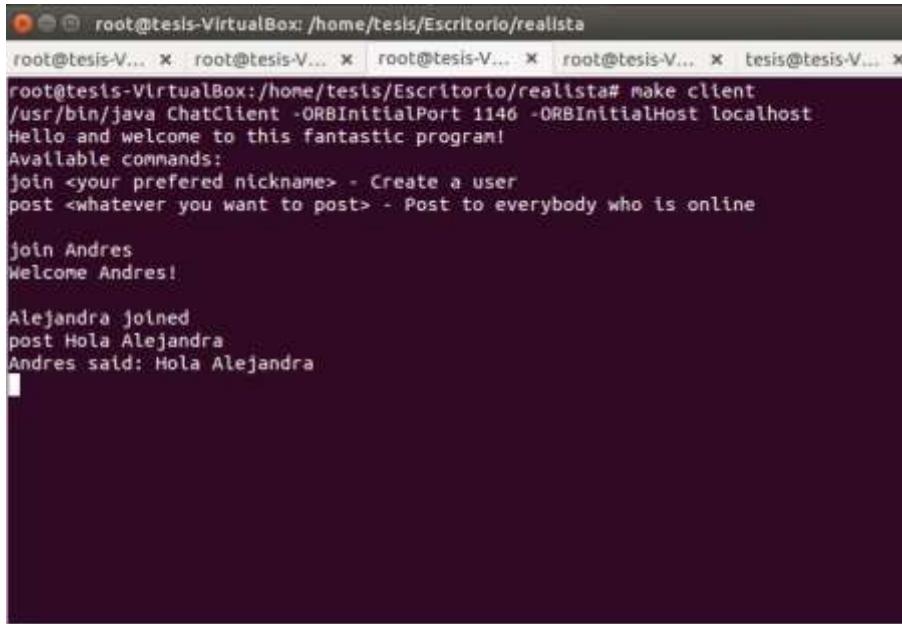


```
root@tesis-VirtualBox:/home/tesis/Escritorio/realista
root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘
root@tesis-VirtualBox:/home/tesis/Escritorio/realista# make client
/usr/bin/java ChatClient -ORBInitialPort 1146 -ORBInitialHost localhost
Hello and welcome to this fantastic program!
Available commands:
join <your preferred nickname> - Create a user
post <whatever you want to post> - Post to everybody who is online

join Alejandra
Welcome Alejandra!
```

Figura 4-10. Chat CORBA 4

- 4) Intercambio de mensajes entre el cliente 1 y el cliente 2.



```

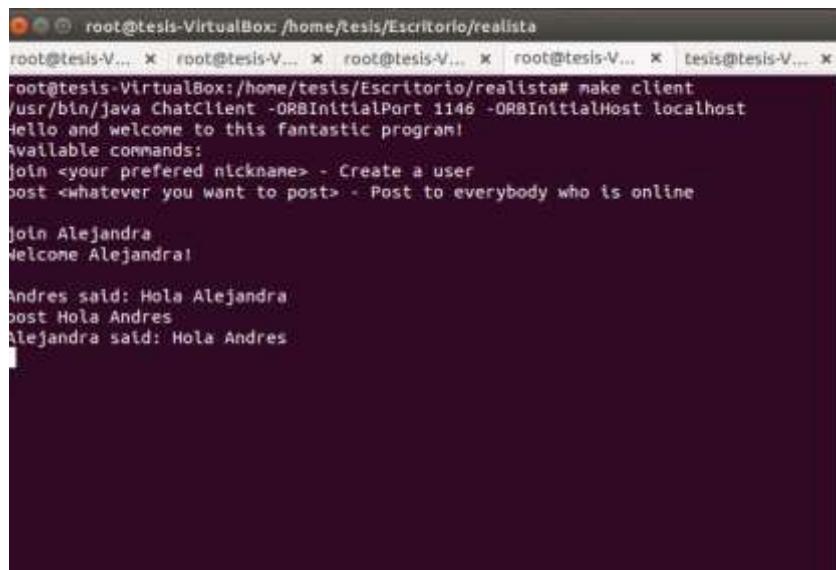
root@tesis-VirtualBox: /home/tesis/Escritorio/realista
root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ tesis@tesis-V... ✘
root@tesis-VirtualBox:/home/tesis/Escritorio/realista# make client
/usr/bin/java ChatClient -ORBInitialPort 1146 -ORBInitialHost localhost
Hello and welcome to this fantastic program!
Available commands:
join <your prefered nickname> - Create a user
post <whatever you want to post> - Post to everybody who is online

join Andres
Welcome Andres!

Alejandra joined
post Hola Alejandra.
Andres said: Hola Alejandra

```

Figura 4-11. Chat CORBA 5



```

root@tesis-VirtualBox: /home/tesis/Escritorio/realista
root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ root@tesis-V... ✘ tesis@tesis-V... ✘
root@tesis-VirtualBox:/home/tesis/Escritorio/realista# make client
/usr/bin/java ChatClient -ORBInitialPort 1146 -ORBInitialHost localhost
Hello and welcome to this fantastic program!
Available commands:
join <your prefered nickname> - Create a user
post <whatever you want to post> - Post to everybody who is online

join Alejandra
Welcome Alejandra!

Andres said: Hola Alejandra
post Hola Andres
Alejandra said: Hola Andres

```

Figura 4-12. Chat CORBA 6

4.4.2.2. Captura de paquetes.

- 1) Paquete de envío del cliente 1.

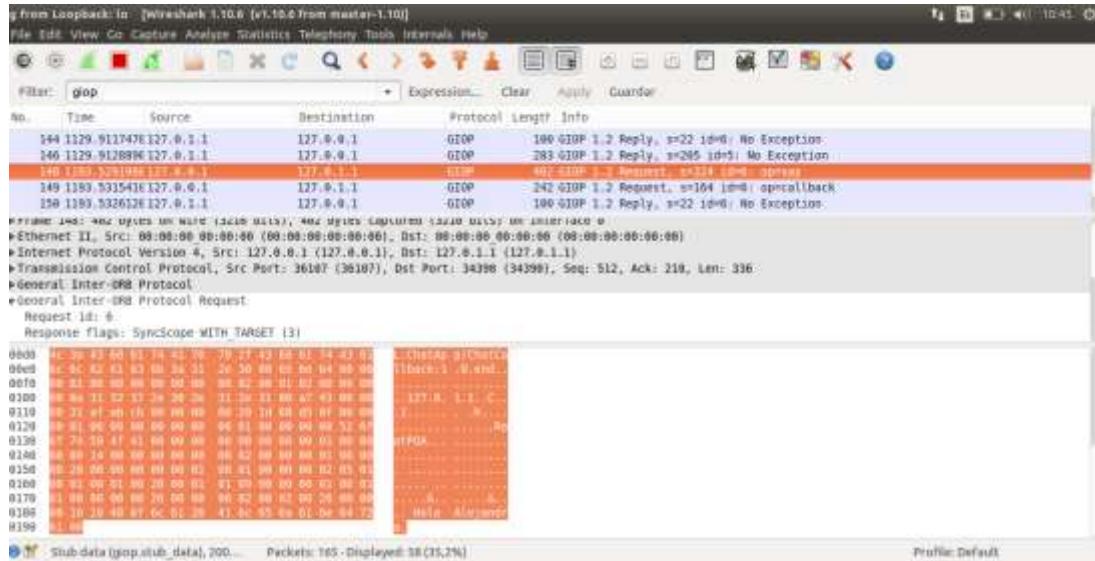


Figura 4-13. Captura de Paquetes CORBA 1

2) Paquete de Recepción del cliente 1 hacia el cliente 2.

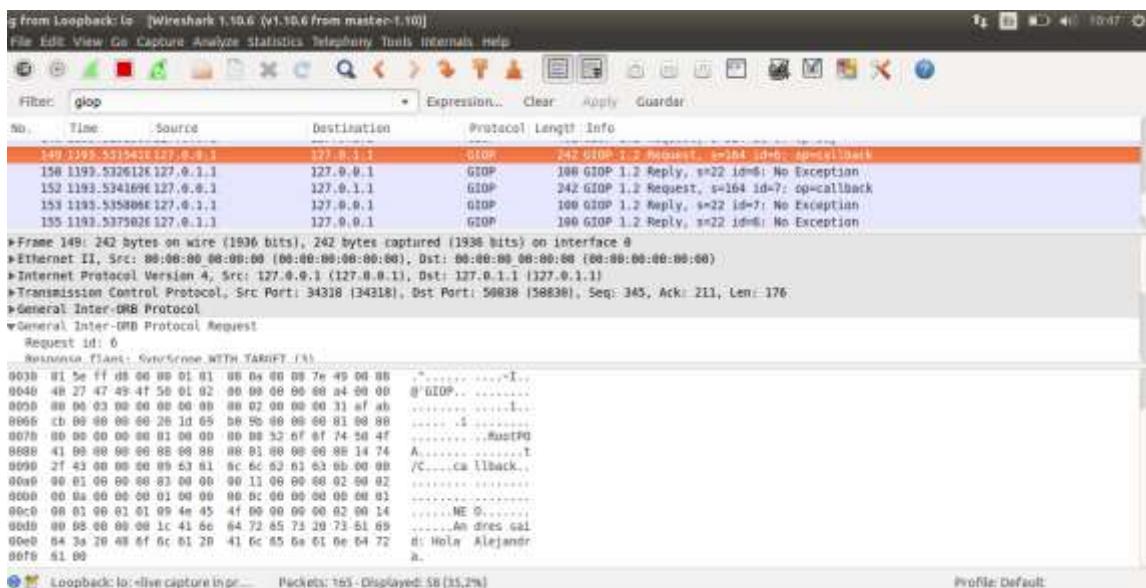


Figura 4-14. Captura de Paquetes CORBA 2

3) Paquete de envío del cliente 2.

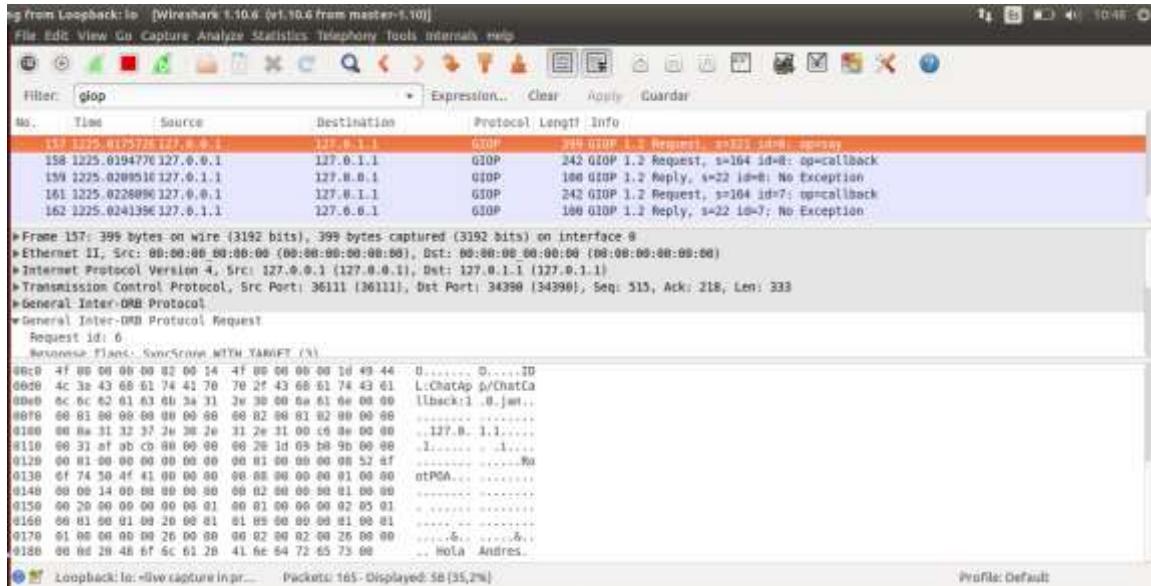


Figura 4-15. Captura de Paquetes CORBA 3

4) Paquete de recepción del cliente 2 hacia el cliente 1.

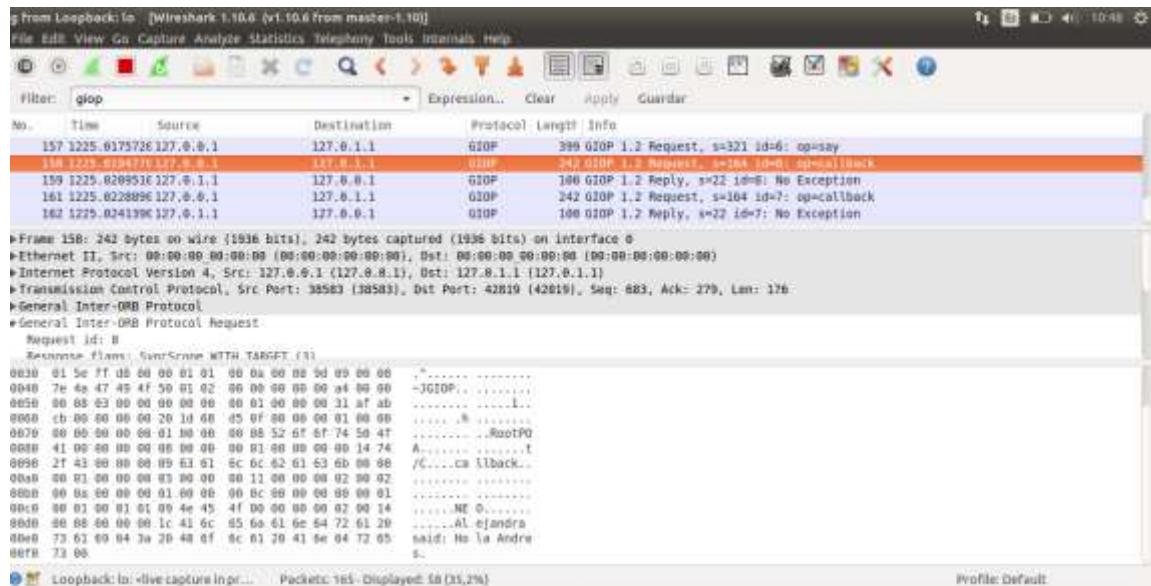


Figura 4-16. Captura de Paquetes CORBA 4

4.5. MANUAL DE USUARIO DE LA APLICACIÓN Y USO DE LAS LIBRERÍAS DDS-RTPS

4.5.1. MANUAL PARA EL USO DE LAS LIBRERÍAS DDS-RTPS.

A continuación se presenta el manual de usuario para la implementación del DDS – RTPS.

4.5.1.1. Requerimientos

Para utilizar la implementación DDS – RTPS se debe cumplir con varios requisitos necesarios para su uso:

- Sistema Operativo Windows 8, 8.1, 10 o superior.
- Microsoft Visual Studio Ultimate 2013 (se requiere ultimate para el caso de necesitar pruebas unitarias).
- Descargar las librerías Common.Login.Core.dll, Common.Login.dll, DDS.dll, Doopec.dll, DoopecSerializer.dll, log4net.dll, Mina.NET.dll, RTPS.dll, SerializerDebug.dll, de los anexos.

4.5.1.2. Proceso De Creación De Un Proyecto En Lenguaje C#

- 1) Crear un proyecto en Visual Studio, seleccionando Archivo – Nuevo – Proyecto.

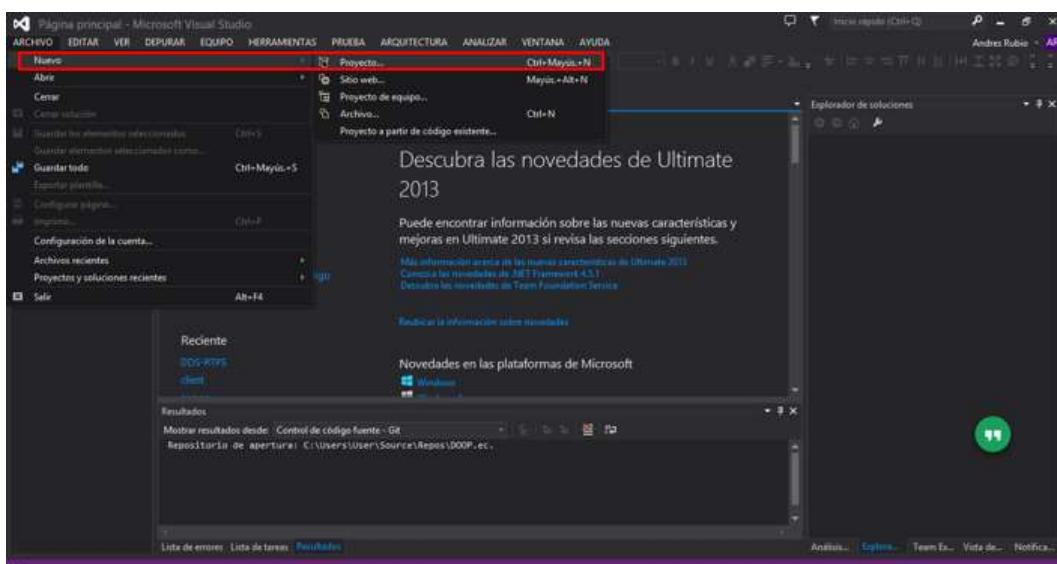


Figura 4-17. Creación de un proyecto en Visual Studio.

- 2) Escoger un proyecto Aplicación de Consola o de Aplicación de Windows Forms (Se ha escogido aplicación de consola). Escribiendo el nombre que se desee y seleccionar Aceptar.

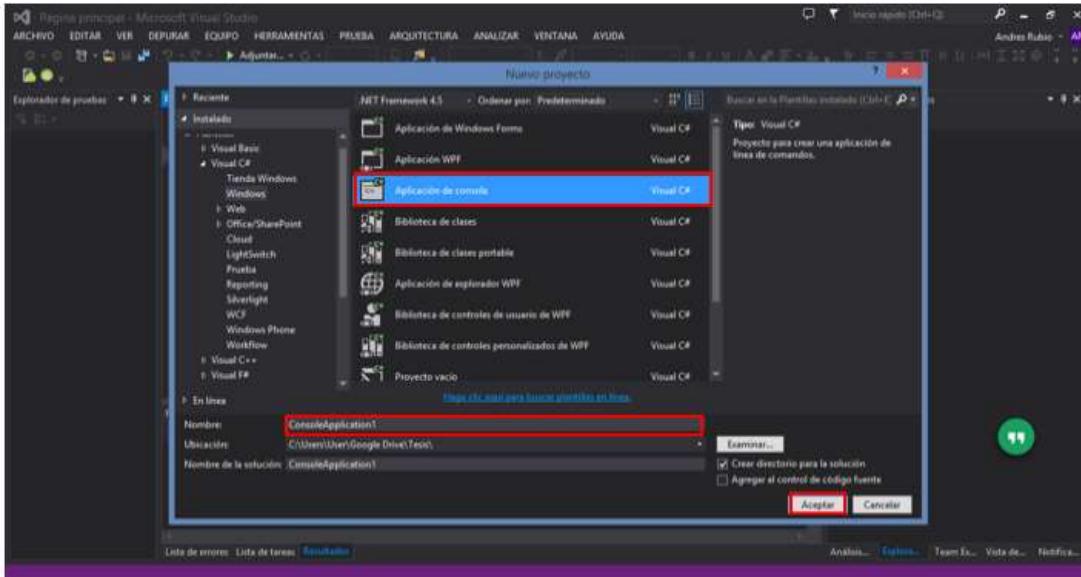


Figura 4-18. Creación proyecto Aplicación Consola.

- 3) Una vez creada la solución se procede a hacer clic derecho en References y Agregar Referencia.

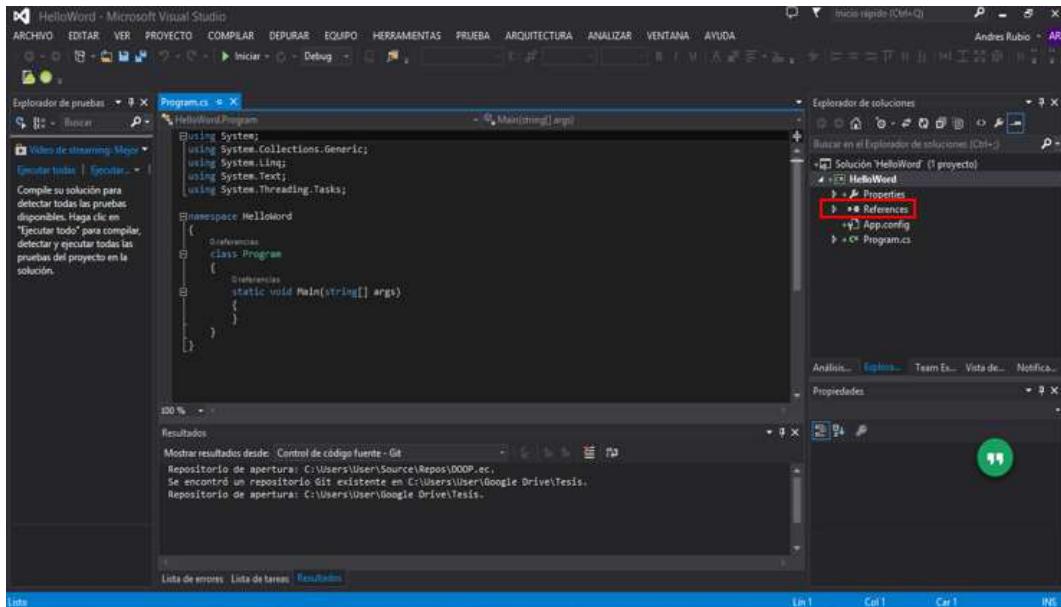


Figura 4-19. Agregación de referencias

4) Una vez en el Administrador de Referencias, diríjase a Examinar.

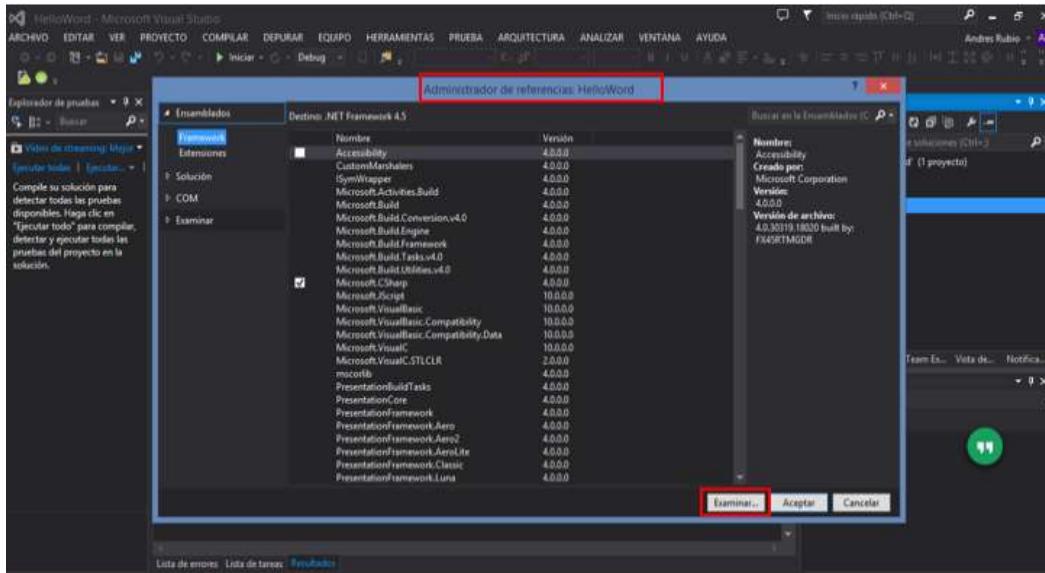


Figura 4-20. Administrador de Referencias de Visual Studio.

5) Dentro del botón Examinar, buscar la carpeta donde se encuentran las librerías dll y agregarlas.

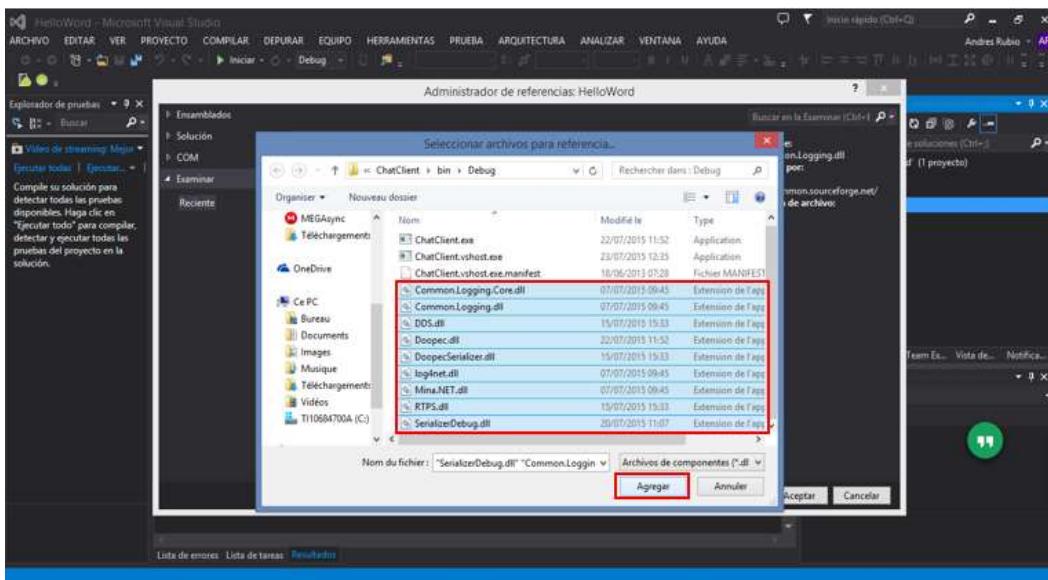


Figura 4-21. Librerías .dll de la implementación DDS – RTPS.

- 6) Una vez agregadas las librerías, marcarlas con un visto a todas y aceptar.

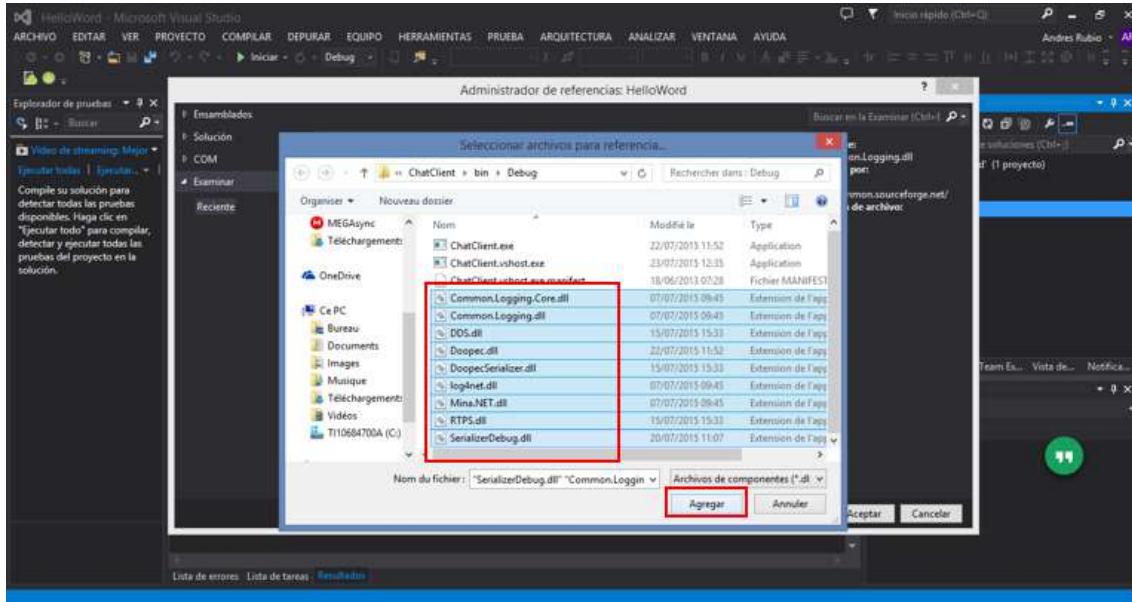


Figura 4-22. Librerías .dll

- 7) Se verifica en la pestaña de References que se encuentren todas las librerías agregadas.

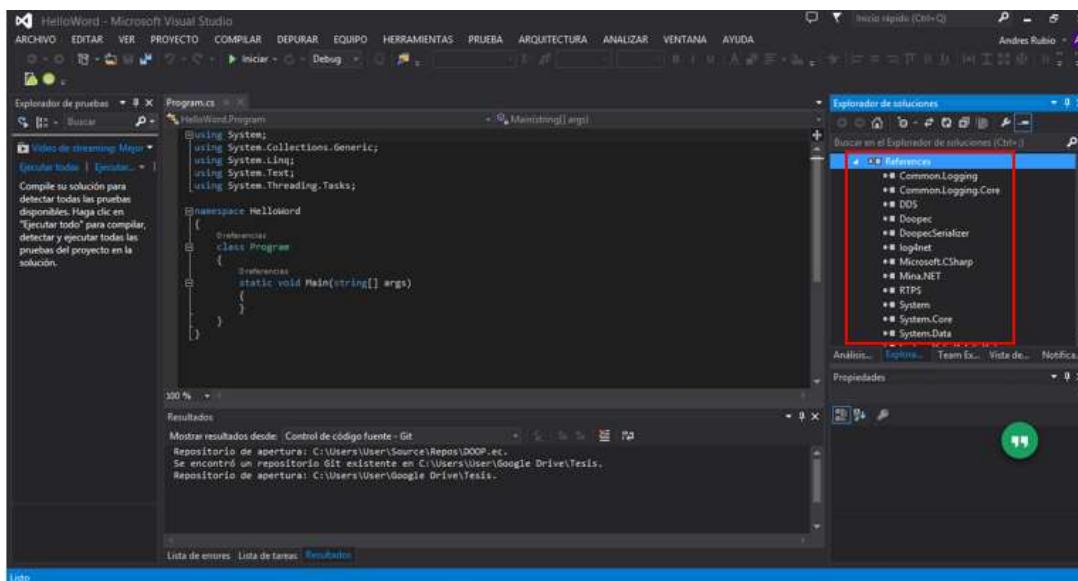


Figura 4-23. Pestaña de Referencias del proyecto creado.

- 8) Agregar las referencias a la clase principal del proyecto, tal como se muestra en el gráfico.

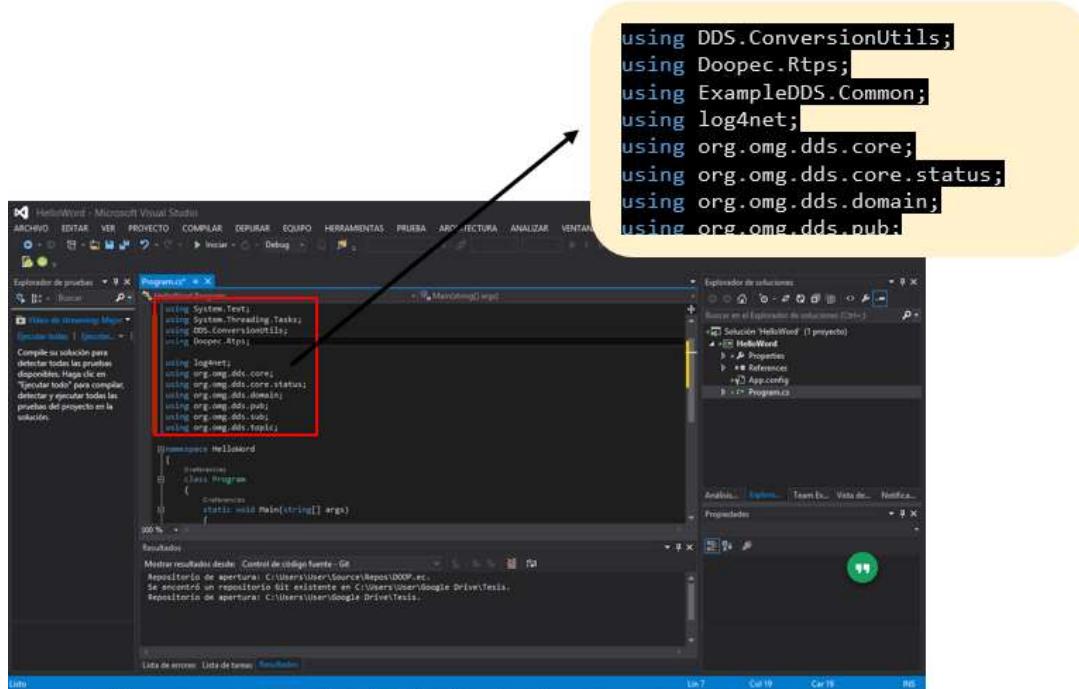


Figura 4-24. Agregación de Referencias en una clase.

4.5.2. MANUAL DE USUARIO DE LA APLICACIÓN CHAT RTPS.

A continuación se presenta el manual de usuario para la aplicación Chat con la tecnología DDS – RTPS.

4.5.2.1. Requerimientos

Para utilizar la implementación DDS – RTPS se debe cumplir con varios requisitos necesarios para su uso:

- Sistema Operativo Windows 8, 8.1, 10 o superior.
- Microsoft Visual Studio Ultimate 2013 (se requiere ultimate para el caso de necesitar pruebas unitarias).
- Verificar las librerías Common.Login.Core.dll, Common.Login.dll, DDS.dll, Doopec.dll, DoopecSerializer.dll, log4net.dll, Mina.NET.dll, RTPS.dll, SerializerDebug.dll que se encuentren en los anexos.

4.5.2.2. Utilización de la aplicación Chat

- 1) Ejecutar la aplicación

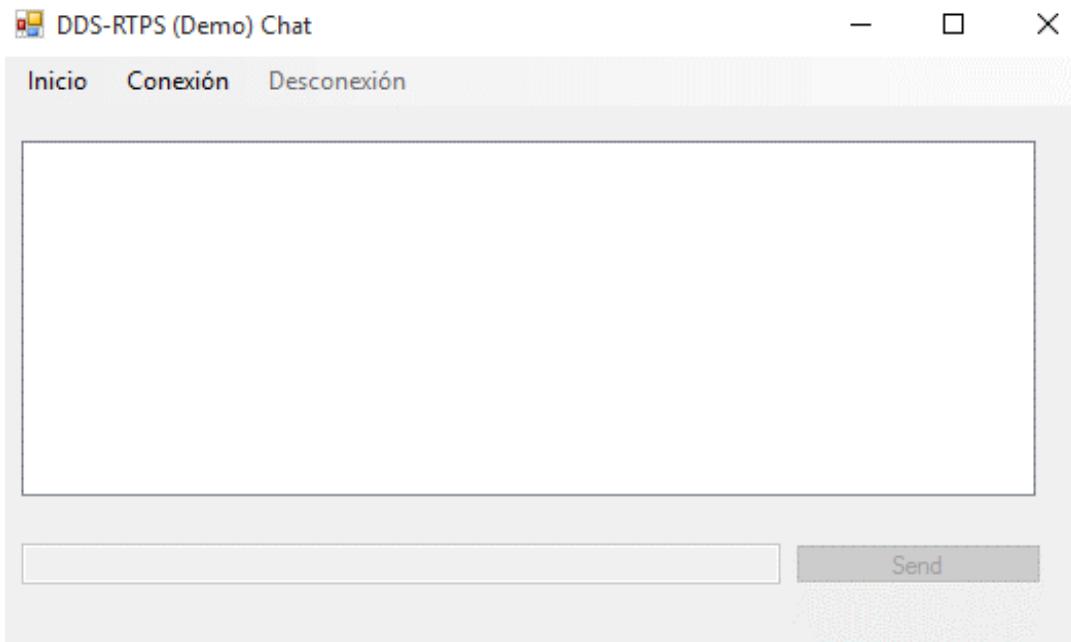


Figura 4-25. Utilización de la aplicación Chat 1

2) Conectarse al servicio de chat, seleccionando en el menú conexión.

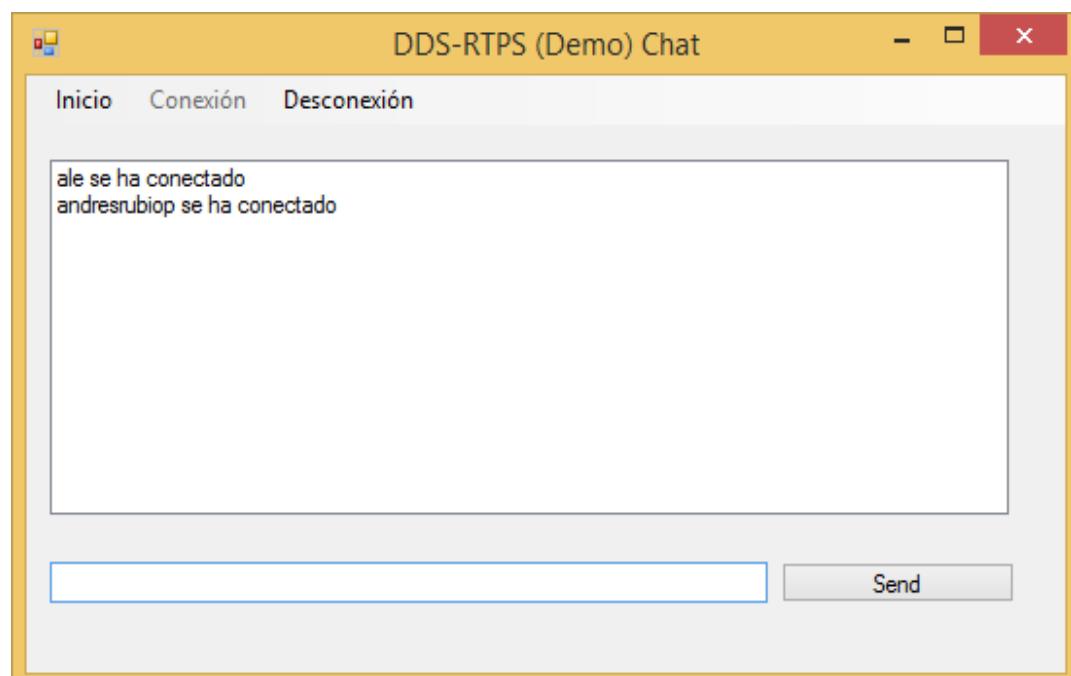


Figura 4-26. Utilización de la aplicación Chat 2

3) Ingrese su nombre de Usuario.

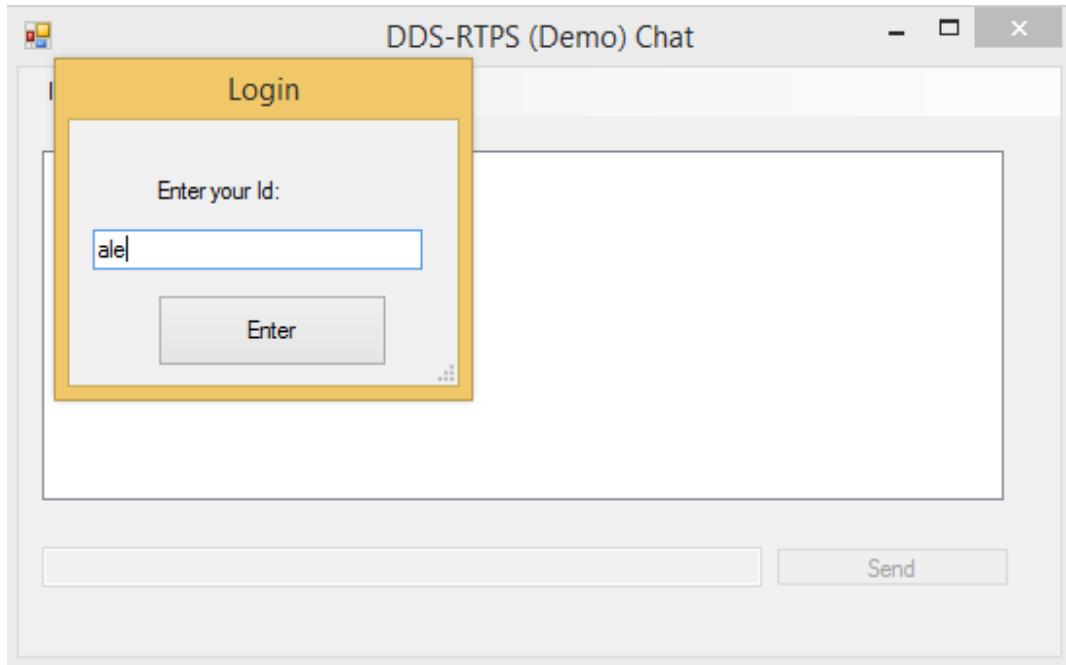


Figura 4-27. Utilización de la aplicación Chat 3

4) Intercambio de mensajes, ingresando su mensaje en el cuadro dialogo, y para enviar el mensaje presione enter o el botón send.

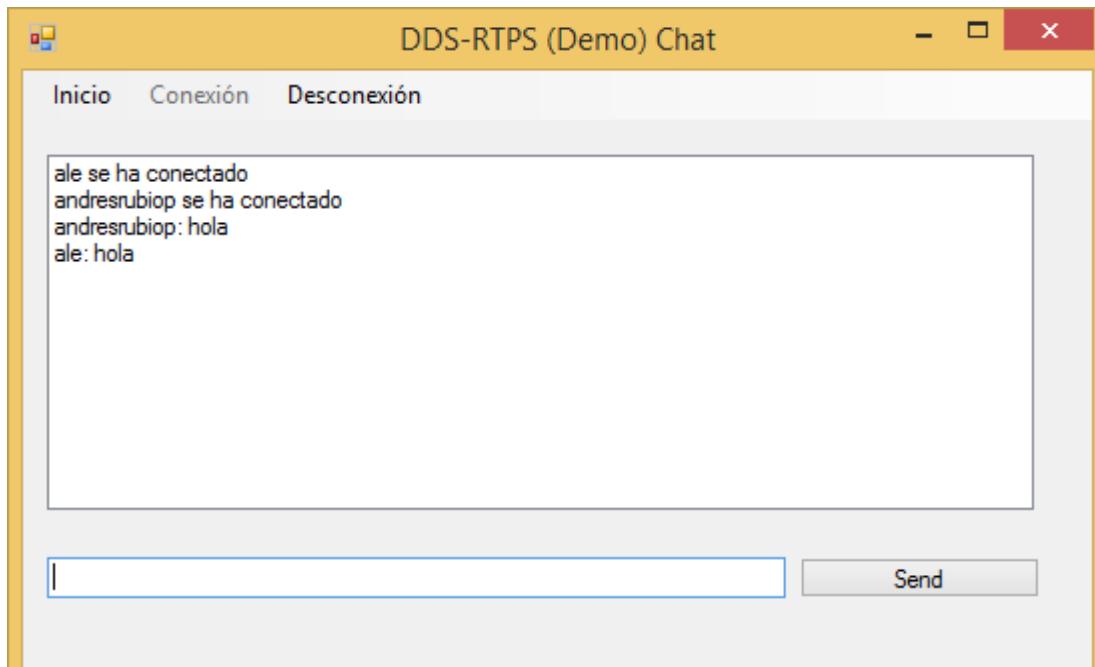


Figura 4-28. Utilización de la aplicación Chat 4

- 5) Desconexión del servicio de chat, seleccione del menú la opción Desconexión.

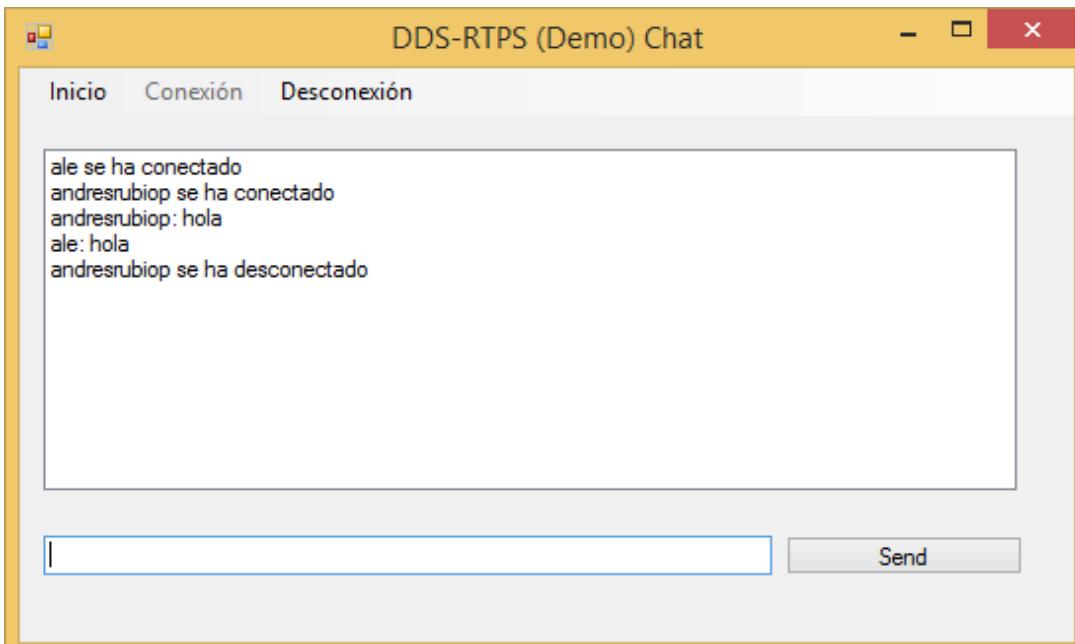


Figura 4-29. Utilización de la aplicación Chat 5

- 6) Salida de la aplicación, seleccione en el botón inicio del menú Salir.

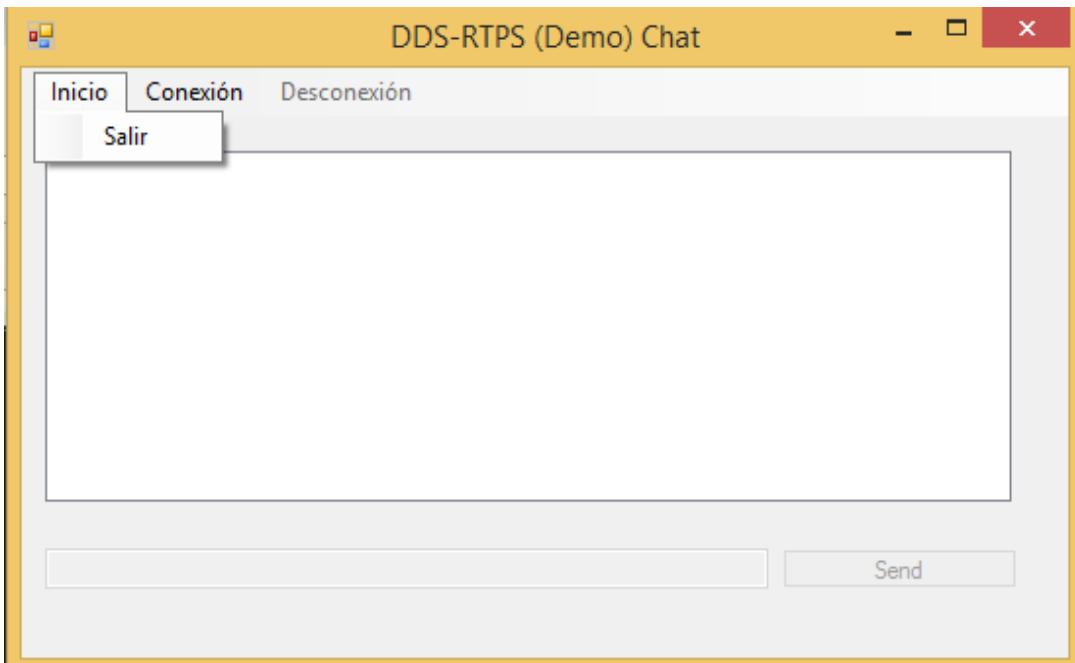


Figura 4-30. Utilización de la aplicación Chat 6

4.6. COMPARACIÓN DE APLICACIONES

A continuación, se muestra la comparación de la tecnología DDS-RTPS y CORBA-RT.

4.6.1. CAPTURAS DE PAQUETES DDS-RTPS Y CORBA-RT

En las siguientes imágenes se muestra un diagrama de flujo generado por la herramienta Wireshark para las dos tecnologías.

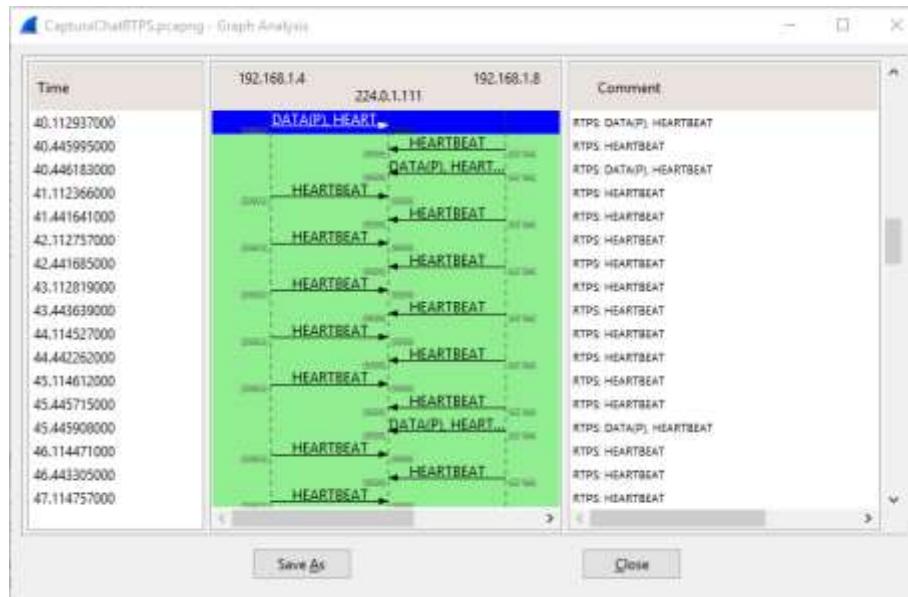


Figura 4-31. Flujo de datos DDS-RTPS

Para transmitir un mensaje de datos en RTPS es: 13.77ms y se requiere de 2 paquetes los cuales son HEARTBEAT y el DATA.



Figura 4-32. Flujo de datos CORBA-RT

Para transmitir un mensaje de datos en CORBA-RT es: 3.25ms y se requiere de 5 paquetes REQUEST y REPLY.

Tabla 4-146. Comparación de las tecnologías DDS-RTPS con CORBA-RT.

CARACTERÍSTICAS	DDS-RTPS	CORBA-RT
	Envío de 1 dato	
Tiempo de transmisión de un mensaje de datos	13,77ms	3,25ms
Número de paquetes para la transmisión	2	5
Paquetes necesarios para transmitir a dos suscriptores	2	10
Tiempo de transmisión de un mensaje a dos suscriptores	13,77ms	6,50ms

La tecnología CORBA-RT permite tener transmisiones de datos de tipo *unicast* sumamente rápidas, pero a medida que la tecnología requiere más escalabilidad el tiempo de transmisión de datos se muestra afectado; en cambio, la tecnología DDS-RTPS no se ve afectada en gran manera cuando se requiere escalabilidad, si embargo en redes pequeñas esta no llega a superar a CORBA-RT.

CAPÍTULO 5

CONCLUSIONES Y RECOMENDACIONES

5.1. CONCLUSIONES

- El modelo de abstracción de datos dentro del Middleware DDS, en el cual se tiene al Publicador y al Suscriptor, centraliza la obtención de datos, es decir, se independiza el origen; facilitando la difusión asincrónica de la información la cual es un requisito común en varias aplicaciones distribuidas.
- La gestión de recursos del procesador propuesto por el Middleware DDS-RTPS, no incluye planificación en los procesadores, pero define varios parámetros de tiempo los cuales sirven para construir el hilo o los hilos en el procesador, que son necesarios para la escucha, o estructura de espera y establecimiento, o pedido de la disponibilidad de datos.
- La gestión de recursos de red presentes en el middleware DDS-RTPS, puede provocar un incremento en los tiempos de respuesta de las aplicaciones, aunque esta sobrecarga depende de casi exclusivamente de cada aplicación, este efecto es más significativo dentro de DDS-RTPS ya que define un conjunto de entidades que consumen recursos del procesador y de la red.
- El estándar DDS fue diseñado explícitamente para construir sistemas distribuidos en tiempo real, añadiendo un conjunto de parámetros de calidad de servicio para configurar propiedades no funcionales y también permitiendo la reconfiguración dinámica del sistema, es decir, modificar parámetros en tiempo de ejecución.
- El uso de pruebas unitarias permiten comprobar que los componentes de la aplicación trabajen de la manera esperada, además permite mejorar el código para que tenga un mejor funcionamiento, se las pueden realizar independientemente del lenguaje de programación o de la plataforma de desarrollo utilizada.
- Dentro del comportamiento de la interacción entre las entidades DDS y sus correspondientes entidades RTPS, en lo concerniente a la

escritura de datos el *DataWriter* DDS es el encargado de añadir y remover cambios del tipo *CacheChange* desde y hacia el *HistoryCache* del *Writer RTPS* asociado, es decir el *Writer RTPS* no está en control cuando un cambio es removido desde *HistoryCache*.

- La implementación sin estado está optimizada para la escalabilidad, esta mantiene virtualmente un estado sumamente simple en las entidades remotas y por lo tanto esta puede escalar de manera adecuada en sistemas grandes. La implementación sin estado es ideal para las comunicaciones con mejor esfuerzo, ya que al trabajar sin estado se requiere menos uso de memoria y por lo tanto la comunicación más rápida.
- La implementación con estado mantiene un total estado en las entidades remotas, esto minimiza el uso de ancho de banda, pero requiere una mayor capacidad de la memoria, y la escalabilidad es reducida, por lo tanto garantiza una comunicación confiable.
- Dentro del descubrimiento existen dos fases, la primera concerniente al protocolo SPDP el cual se encarga de descubrir y anunciar de los participantes y la segunda al protocolo SEDP el cual se encarga de descubrir y anunciar de los servicios que publica el participante.
- El *DataWriter* es la cara del Publicador, el cual representa a los objetos responsables de la emisión de datos, lo usan los participantes para comunicar el valor y los cambios de los datos; una vez que la nueva información ha sido comunicada al Publicador, es responsabilidad de este determinar cuando es apropiado emitir el correspondiente mensaje, es decir, lo realiza de acuerdo a su calidad de servicio asociada al correspondiente *DataWriter* o a su estado interno.
- Para acceder a los datos recibidos, el participante debe utilizar un tipo *DataReader* asociado al suscriptor, este recibe los datos publicados y los hace disponibles al participante. Un Suscriptor debe recibir y despachar datos de diferentes tipos especificados y asociar a un objeto *DataWriter*, el cual representa a una publicación, con el objeto

DataReader, que representa la suscripción, es hecha por la entidad *Topic*.

- El Topic tiene el propósito de asociar un nombre único en el dominio, es decir, el conjunto de aplicaciones que se comunican entre sí.
- La semántica proporcionada por las operaciones *read* y *take* permite usar el DDS como una caché distribuida o como un sistema de cola, o ambos. Esta es una poderosa combinación que raramente se encuentra en la misma plataforma Middleware. Esta es una de las razones porque DDS es usado en una variedad de sistemas, algunas veces como una caché distribuida de alto rendimiento, otras como tecnología de mensajería de alto rendimiento, y sin embargo, otras veces como una combinación de las dos.
- El uso del lenguaje de programación C#, a pesar de no ser un lenguaje común para programar middleware de comunicación, este nos permite tener una interacción más directa con las herramientas que comúnmente son requeridas al momento de desarrollar aplicaciones dirigidas a los usuarios. Además permite la creación simple de aplicaciones multi tarea y permite utilizar fácilmente las sobrecarga en métodos.
- La tecnología DDS-RTPS permite desarrollar aplicaciones de comunicación que no se ven afectadas de una manera significativa al momento en que la red de datos deba crecer, es decir que a comparación de otras tecnologías como CORBA-RT al tener un mayor número de usuarios la eficiencia de la comunicación es mínimamente afectada.

5.2. RECOMENDACIONES

- Para implementaciones críticas que utilicen el Middleware DDS-RTPS: como transmisión de flujos de video y audio; se recomienda el uso de calidad de servicio que proporciona el Middleware y el uso adecuado de los diferentes tipos de escritores y lectores, para obtener la mejor calidad en transmisión.
- Se sugiere contar con versiones recientes de software, ya que permite el uso de nuevas funcionalidades a las aplicaciones y obtener productos de calidad. Particularmente se recomienda el uso de Visual Studio 2013 y 2015 ya que a la fecha es una herramienta que cuenta con una colección completa de servicios que permiten crear una gran variedad de aplicaciones.
- Se recomienda revisar las publicaciones de la OMG con respecto a futuras actualizaciones o al estado actual de DDS y RTPS, con el propósito de tener un panorama mas amplio al momento de realizar actualizaciones dentro Middleware.
- Una mejora al Middleware que se podría incluir, es la interoperabilidad con otros sistemas que trabajen con DDS-RTPS como OpenDDS, realizando pruebas y mejorando la compatibilidad con los perfiles de calidad de servicio.
- Se sugiere que este proyecto de titulación pueda ser la base para el análisis del comportamiento de la arquitectura de comunicaciones del tipo Publicador- Suscriptor.
- Se sugiere que al implementar un API primeramente se realice el *stack* de pruebas unitarias ya que estas permitirán tener un mayor control en la implementación.

REFERENCIAS

- [1] R. Klefstad, D. C. Schmidt y C. O'Ryan, «Towards highly configurable real-time object request brokers,» de *In Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2002.
- [2] A. Gokhale, K. Balasubramanian, A. S. Krishna, J. Balasubramanian, G. Edwards, G. Deng, E. Tukay, J. Parsons y D. C. Schmidt, «Model driven middleware: A new paradigm for developing distributed real-time and embedded systems.,» *Science of Computer Programming*, vol. 73, pp. 39-58, 2008.
- [3] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlantzas y K. Saikoski, «The design and implementation of open ORB 2.,» *In IEEE Distributed Systems Online*, vol. 2, 2001.
- [4] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli y U. Scholz, «MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Software Engineering for Self-Adaptive Systems,» *Lecture Notes in Computer Science*, vol. 5525, pp. 164-182, 2009.
- [5] H. Pérez y J. J. Gutiérrez, «A survey on standards for real-time distribution middleware,» *ACM Computing Surveys*, vol. 46, nº 49, p. 39, Marzo 2014.
- [6] R. I. Davis y A. Burns, «A survey of hard real-time scheduling for multiprocessor systems,» 2011.
- [7] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst y M. G. Harbour, «Influence of different system abstractions on the performance analysis of distributed real-time systems.,» de *In Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT'07)*, New York, 2007.

- [8] C. L. Liu y J. W. Layland, «Scheduling algorithms for multiprogramming in a hard-real-time environments,» de *Journal of the ACM*, 1973.
- [9] L. Sha, R. Rjkumar y J. P. Lehoczky, «Priority inheritance protocols: An approach to real-time synchronization,» de *IEEE Transactions on Computers* 39, 1990.
- [10] OMG, «Corba Core Specification. v3.2.,» 2011.
- [11] OMG, «Realtime Corba Specification. v1.2.,» 2005.
- [12] M. Amoretti, S. Caselli y M. Reggiani, «Designing distributed, component-based systems for industrial robotic applications,» In *Industrial Robotics: Programming, Simulation and Applications*, 2006.
- [13] ISO/IEC, «Ada 2012 Reference Manual. Language and Standard Libraries—International Standard,» *ISO/IEC*, vol. 8652, 2012.
- [14] ISO/IEC, S. T. Taft, R. A. Duff, R. Brukardt, E. Ploedereder y P. Leroy, «Ada 2005 Reference Manual. Language and Standard Libraries—International Standard ISO/IEC 8652 (E) with Technical Corrigendum 1 and Amendment 1,» *Lecture Notes in Computer Science*, vol. 4348, 2006.
- [15] T. Vergnaud, J. Hugues, F. Kordon y L. Pautet, «PolyORB: A schizophrenic middleware to build versatile reliable distributed applications,» *Lecture Notes in Computer Science*, vol. 3063, pp. 106-119, 4 Mayo 2004.
- [16] Y. Kermarrec, «CORBA vs. Ada 95 DSA: A programmer's view,» vol. XIX, pp. 39-46, 1999.
- [17] P. Basanta-Val, M. García-Valls y I. Estévez-Ayres, «An architecture for distributed real-time Java based on RMI and RTSJ.,» de *In Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, 2010.
- [18] Sun Microsystems, «JSR-50: Distributed Real-Time Specification,» 2000.
- [19] G. Bollella y J. Gosling, «The real-time specification for Java,» *IEEE Computer*, 2000.
- [20] Sun Microsystems, «Java Remote Method Invocation (RMI),» 2004.

- [21] Sun Microsystems, «Distributed Real-Time Specification (Early draft),» 2012. [En línea]. Available: <http://jcp.org/en/egc/download/drtsj.pdf?id=50&fileId=5028>. [Último acceso: 06 Marzo 2015].
- [22] D. Tejera, A. Alonso y M. A. de Miguel, «RMI-HRT: Remote method invocation—hard real time.,» *In Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'07)*, pp. 113-120, 2007.
- [23] OMG, «Data Distribution Service for Real-Time Systems. v1.2.,» 2007.
- [24] OMG, «The Real-Time Publish-Subscribe Wire Protocol. DDS interoperability wire protocol specification. v2.1.,» 2009. [En línea]. Available: <http://www.omg.org/spec/DDS/I/2.1/>. [Último acceso: 6 Marzo 2015].
- [25] OMG, «Extensible and Dynamic Topic Types for DDS. v1.0.,» 2012. [En línea]. Available: <http://www.omg.org/spec/DDS-XTypes/1.0/>.
- [26] H. Pérez y J. J. Gutiérrez, «On the schedulability of a data-centric real-time distribution middleware.,» *Computer Standards and Interfaces* 34., pp. 203-211, 2012.
- [27] D. C. Schimidt, A. Corsaro y H. V. Hag, «Addressing the challenges of tactical information management in net-centric systems with DDS.,» de *Journal of Defense Software Engineering*, 2008, pp. 24-29.
- [28] M. Ryll y S. Ratchev, «Application of the data distribution service for flexible manufacturing automation.,» *International Journal of Aerospace and Mechanical Engineering* , pp. 193-200, 2008.
- [29] M. Gillen, J. Loyall, K. Z. Haigh, R. Walsh, C. Partridge, G. Lauer y T. Strayer, «Information dissemination in disadvantaged wireless communications using a data dissemination service and content data network.,» de *In Proceedings of the SPIE Conference on Defense Transformation and Net-Centric Systems*, 2012.
- [30] H. Pérez, J. J. Gutiérrez, D. Sangorrín y M. Harbour, «Real-time distribution middleware from the Ada perspective. In Proceedings of the 13th Ada-

Europe International Conference on Reliable Software Technologies,» de *Lecture Notes in Computer Science*, 2008.

- [31] «Wikipedia,» 09 Marzo 2015. [En línea]. Available: http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling. [Último acceso: 09 Marzo 2015].
- [32] J. L. Campos, J. J. Gutiérrez y M. G. Harbour, «Interchangeable scheduling policies in real-time middleware for distribution,» *Lecture Notes in Computer Science*, vol. 4006, pp. 227-240, 2006.
- [33] «WIKIPEDIA,» 2 Mayo 2014. [En línea]. Available: http://it.wikipedia.org/wiki/Priority_ceiling_protocol. [Último acceso: 9 Marzo 2015].
- [34] IEEE, «The Institute of Electrical and Electronics Engineers STD 802.1Q. 2006. Virtual bridged local area networks. Annex G.,» 2006. [En línea]. Available: <http://www.ieee802.org/1/pages/802.1Q.html>.
- [35] M. Aldea, G. Bernat, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. González Harbour, G. Guidi, J. J. Gutiérrez, T. Lennvall, G. Lipari, J. M. Martínez, J. L. Medina, J. C. P. Gutiérrez y M. Trimarchi, «FSF: A real-time scheduling architecture framework.,» *n Proceedings of the IEEE Real Time Technology and Applications Symposium.*, pp. 113-124, 2006.
- [36] FRESCOR, «Framework for Real-Time Embedded Systems Based on COntRacts. Project Web page. Retrieved September 2013,» 2006. [En línea]. Available: <http://www.frescor.org>. [Último acceso: 10 Marzo 2015].
- [37] A. Corsaro, «Advanced DDS Tutorial,» [En línea]. Available: <http://www.prismTech.com/dds-community>.
- [38] G. Pardo-Castellote, «OMG Data-Distribution Service: Architectural Overview,» Real-Time Innovations, Inc..
- [39] Twin Oaks Computing, Inc., «Interoperable DDS Strategies,» Diciembre 2011. [En línea]. Available: <http://www.twinoakscomputing.com>. [Último acceso: 17 Marzo 2015].

- [40] WIKIPEDIA, «Polling,» 8 Marzo 2013. [En línea]. Available: <http://es.wikipedia.org/wiki/Polling>. [Último acceso: 11 Marzo 2015].
- [41] OMG, «The Real-time Publish-Suscribe Protocol (RTPS) DDS Interoperability Wire Protocol Specification Version 2.2,» 2014.
- [42] MSDN, «Futures,» [En línea]. Available: <https://msdn.microsoft.com/en-us/library/ff963556.aspx>.
- [43] MSDN, [En línea]. Available: <https://msdn.microsoft.com/es-es/library/bb397687.aspx>.
- [44] J. Bard y V. J. Kovarik, «Software Defined Radio: The Software Communications Architecture,» 2007.
- [45] C. Grelck, J. Julju y F. Penczek, «Distributed S-Net: Cluster and grid computing without the hassle,» de *In Proceedings of the 12th IEEE /ACM International Symposium on Cluster, Cloud and Grid Computing(CCGrid)*, 2012.
- [46] L. Neumeyer, B. Robbins, A. Nair y A. Kesari, «S4: Distributed stream computing platform,» de *In Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2010.
- [47] E. Freeman, S. Hupfer y K. Arnold, «JavaSpaces: Principles, Patterns, and Practice,» 1999.
- [48] Sun Microsystems, «JavaTM Message Service Specification. v1.1.,» 2002.
- [49] K. H. Kim, «Object-oriented real-time distributed programming and support middleware.,» *In Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 10-20, 2000.

ANEXOS

Los anexos se incluyen en el DVD adjunto al presente documento.

ANEXO A: GLOSARIO

A

API

Application Programming Interface · 14, 23, 30, 32, 33, 34, 72, 73, 100, 101, 102, 103, 104, 105, 107, 109, 252, 342

C

CDR

Common Data Representation · 16, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 262

D

DDS

Data Distributed System · 1, 14, 16, 18, 19, 20, 21, 22, 23, 24, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 43, 55, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 88, 100, 103, 104, 105, 109, 110, 112, 115, 126, 127, 128, 129, 136, 137, 138, 139, 141, 142, 143, 145, 146, 147, 149, 150, 151, 152, 156, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 176, 177, 181, 189, 191, 252, 253, 255, 257, 259, 299, 320, 328, 329, 331, 333, 337, 338, 339, 340, 341, 342, 351, 354, 355

DDSI

DDS Interoperability Wire Protocol · 14, 16, 20, 36, 38

DOM

Distribution based on objects · 3, 5, 12

DR

Data Reader · 14, 15, 19, 37

DW

Data Writer · 14, 15, 37, 38

E

EDF

Earliest deadline first · 18

F

FPS

Fixed Priorities Schudeling · 18

G

GUID

Globally Unique Identifier · 43, 59, 77, 208, 209, 210, 248, 249

I

IDL

Lenguaje de definición de interfaces · 7, 14, 70, 71

IP

Internet Protocol · 7, 8, 16, 20, 22, 28, 30, 40, 79, 116, 117, 134, 177, 178

M

Middleware MD

Corresponde a los Middleware de tipo *Model-Driven* · 1

MOM

Distribution based on messages · 3

O

OMG

Object Management Group · 1, 5, 8, 14, 33, 100, 342

OSI

Open System Interconnection · 8

P

PDP

Participant Discovery Protocol · 97, 98, 99

PIM

Plataform Independent Model · 29, 76

Q

QoS

Quality of Service · 15, 23, 24, 25, 26, 30, 31, 32, 33, 34, 35, 36, 38, 45, 47, 49, 69, 78, 86, 88, 119, 121, 129, 131, 132, 133, 136, 139, 140, 143, 147, 148, 152, 157, 159, 161, 164, 165, 166, 167, 181, 183, 187, 189, 190

R

RPC

Remote Procedure Calls · 3, 10

RTP

Real-Time Transport Protocol · 16

RTPS

Real-Time Publish-Subscribe Protocol · i, 1, 24, 28, 29, 30, 33, 38, 40, 41, 43, 45, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 62, 64, 67, 68, 69, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 95, 97, 98, 99, 100, 101, 102, 103, 104, 105, 107, 109, 110, 111, 112, 113, 114, 115, 116, 119, 120, 121, 122, 124, 125, 126, 127, 129, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 173, 174, 176, 177, 178, 179, 181, 189, 191, 193, 226, 227, 228, 229, 230, 231, 237, 238, 241, 242, 244, 245, 247, 252, 320, 323, 328, 329, 331, 333, 337, 338, 339, 340, 341, 342, 351, 352, 354, 355

U

UDP

User Datagram Protocol · 20, 28, 40, 79, 112, 116, 117, 118, 119, 137, 139, 140, 141, 144, 147, 148, 149, 151, 152, 153, 154, 155, 157, 158, 160, 161, 163, 164, 166, 168, 169, 170, 171, 172, 173, 174, 226, 228, 230, 237, 238, 241, 245

ANEXO B: MIDDLEWARE

ANEXO B.1: CÓDIGO FUENTE DEL MIDDLEWARE

ANEXO B.2: MANUAL DE USO DE LA LIBRERÍAS DDS-RTPS

ANEXO C: APLICACIÓN DE ESCRITORIO

ANEXO C.1: CÓDIGO FUENTE DEL CHAT RTPS

ANEXO C.2: MANUAL DE USUARIO DEL CHAT RTPS

ANEXO D: APLICACIÓN CORBA

ANEXO D.1: CÓDIGO FUENTE CHAT CORBA

ANEXO E: ESTÁNDAR

**ANEXO E.1: THE REAL-TIME PUBLISH-SUSCRIBE PROTOCOL (RTPS) DDS
INTEROPERABILITY WIRE PROTOCOL SPECIFICATION.**

**ANEXO E.2: DATA DISTRIBUTION SERVICE FOR REAL-TIME SYSTEM
SPECIFICATION (DDS).**

ANEXO F: MANUAL DE EJEMPLOS

ANEXO F.1: MANUAL DE USUARIO PARA LA CREACIÓN DE EJEMPLOS CON LAS LIBRERÍAS DDS-RTPS.