Real-Time Publish-Subscribe (RTPS)

Wire Protocol Specification

Protocol Version: 1.0

Draft Document Version: 1.17

Real-Time Innovations

Product Version: **RTPS Wire Protocol Specification Version 1.0**

Published: **February 2002**

Draft Disclaimer
This draft contains a subset of the full RTPS protocol. The contents of this draft are subject to change without notice and are provided "as is" without warranty of any kind. The full and complete specification is scheduled for release to IDA Group members by the end of 2001. In addition, the protocol will be released as an Internet Engineering Task Force (IETF) Informational Request for Comment (RFC). If you would like a copy of the complete specification when it is released, please send an e-mail to *info@rti.com* and specify in the subject and body that you would like a copy of the RTPS Protocol Specification and provide a means for us to contact you. We will notify you when it is available.

Contact Information
Real-Time Innovations, Inc.
155A Moffett Park Drive
Sunnyvale, CA 94089

Phone:	408-734-4200
Fax:	408-734-5009
Email:	support@rti.com
Web Site:	http://www.rti.com

Contents

Figures

Tables

Chapter 1

Basic Concepts

1.1 Introduction

With the explosion of the Internet, the TCP/UDP/IP protocol suite has become the underlying framework upon which all Internet-based communications are built. Their success attests to the generality and power of these protocols. However, these transport-level protocols are too low level to be used directly by any but the simplest applications. Consequently, higher-level protocols such as HTTP, FTP, DHCP, DCE, RTP, DCOM, and CORBA have emerged. Each of these protocols fills a niche, providing well-tuned functionality for specific purposes or application domains.

In network communications, as in many fields of engineering, it is a fact that "one size does not fit all." Engineering design is about making the right set of trade-offs, and these trade-offs must balance conflicting requirements such as generality, ease of use, richness of features, performance, memory size and usage, scalability, determinism, and robustness. These trade-offs must be made in light of the types of information flow (e.g. periodic, one-to-many, request-reply, events), and the constraints imposed by the application and execution platforms.

The Real-Time Publish-Subscribe (RTPS) Wire Protocol provides two main communication models: the publish-subscribe protocol, which transfers data from publishers to subscribers; and the Composite State Transfer (CST) protocol, which transfers state.

The RTPS protocol is designed to run over an unreliable transport such as UDP/IP. The broad goals for the RTPS protocol design are:

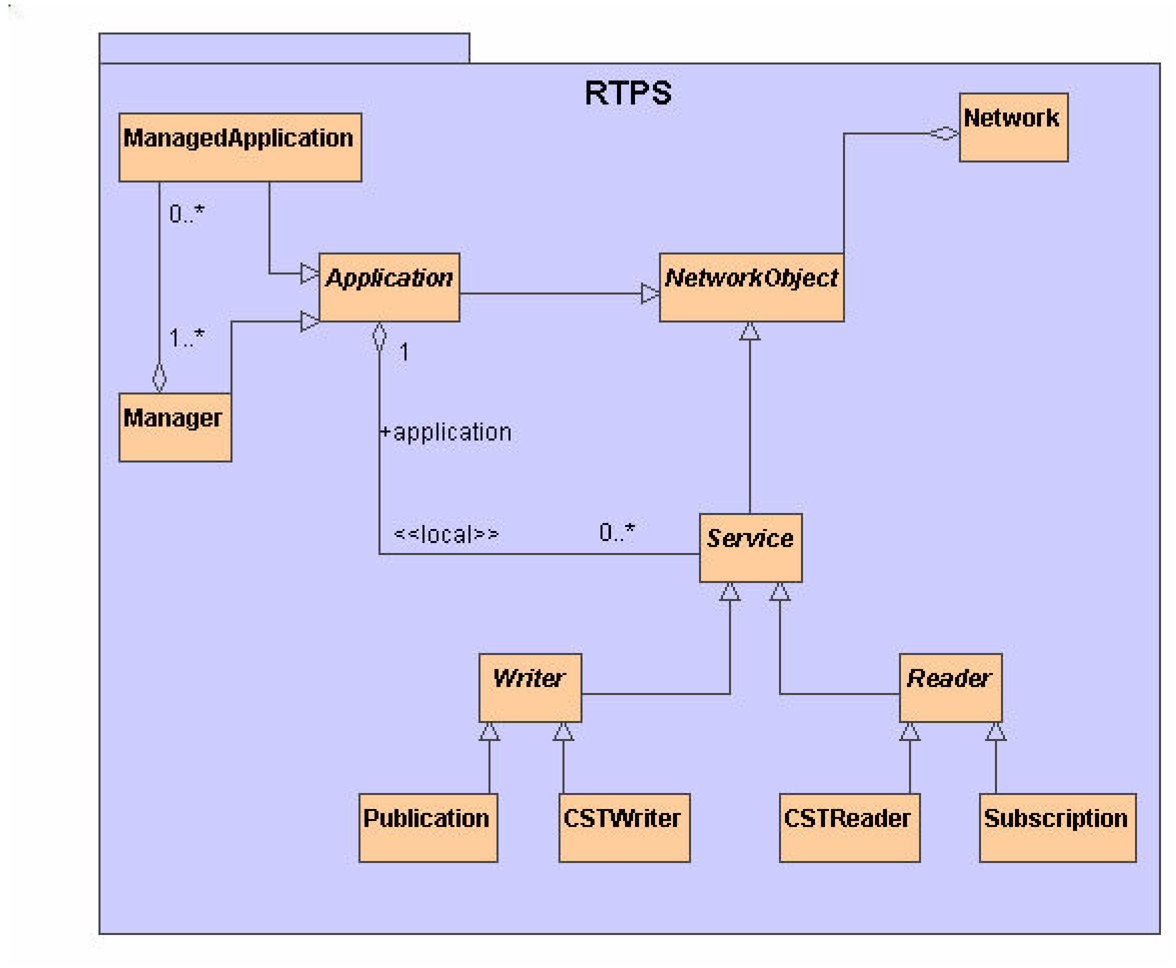
- ❑ Plug and play connectivity so that new applications and services are automatically discovered and applications can join and leave the network at any time without the need for reconfiguration.
- ❑ Performance and quality-of-service properties to enable best-effort and reliable publish-subscribe communications for real-time applications over standard IP networks.
- ❑ Configurability to allow balancing the requirements for reliability and timeliness for each data delivery.
- ❑ Modularity to allow simple devices to implement a subset and still participate in the network.
- ❑ Scalability to enable systems to potentially scale to very large networks.
- ❑ Extensibility to allow the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- ❑ Fault tolerance to allow the creation of networks without single points of failure.
- ❑ Type-safety to prevent application programming errors from compromising the operation of remote nodes.

This specification defines the message formats, interpretation, and usage scenarios that underlie all messages exchanged by applications that use the RTPS protocol.

1.2 The RTPS Object Model

Figure 1.1 shows the object model that underlies the RTPS Protocol.

Figure 1.1 **Object Model**



The RTPS Protocol runs in a **Network** of **Applications**. An **Application** contains local **Services** through which the application sends or receives information using the RTPS Protocols. The **Services** are either **Readers** or **Writers**. **Writers** provide locally available data (a composite state or a stream of issues) on the network. **Readers** obtain this information from the network.

There are two broad classes of **Writers**: **Publications** and **CSTWriters**. A **Publication** is a **Writer** that provides issues to one or more instances of a **Subscription** using the publish-subscribe protocol and semantics.

The presence of a **Publication** in an **Application** indicates that the **Application** is willing to publish issues to matching subscriptions. The attributes of the **Publication** service object describe the contents (the *topic*), the type of the issues, and the quality of the stream of issues that is published on the **Network**.

There are two broad classes of **Readers**: **Subscriptions** and **CSTReaders**. A **Subscription** is a **Reader** that receives issues from one or more instances of **Publication**, using the publish-subscribe service.

The presence of a **Subscription** indicates that the **Application** wants to receive issues from **Publications** for a specific *topic* on the **Network**. The **Subscription** has attributes that identify the contents (the *topic*) of the data, the *type* of the issues and the quality with which it wants to receive the stream of issues.

The **CSTWriter** and **CSTReader** are the equivalent of the **Publication** and **Subscription**, respectively, but are used solely for the state-synchronization protocol and are provided so that applications have a means to exchange state information about each other.

Every **Reader** (**CSTReader** or **Subscription**) and **Writer** (**CSTWriter** or **Publication**) is part of an **Application**. The **Application** and its **Readers** and **Writers** are local, which is indicated in Figure 1.1 by the keyword "local" on the relationship between an **Application** and its **Services**.

There are two kinds of **Applications**: **Managers** and **ManagedApplications**. A **Manager** is a special **Application** that helps applications automatically discover each other on the **Network**. A **ManagedApplication** is an **Application** that is managed by one or more **Managers**. Every **ManagedApplication** is managed by at least one **Manager**.

The protocol provides two types of functionality:

- ❑ **Data Distribution:** The RTPS protocol specifies the message formats and communication protocols that support the publish-subscribe protocol (to send issues from **Publications** to **Subscriptions**) and the Composite State Transfer (CST) protocol (to transfer state from a **CSTWriter** to a **CSTReader**) at various service levels.
- ❑ **Administration:** The RTPS protocol defines a specific use of the CST protocol that enables **Applications** to obtain information about the existence and attributes of all the other **Applications** and **Services** on the **Network**. This *metatraffic* enables every **Application** to obtain a complete picture of all **Applications**, **Managers**, **Readers** and **Writers** in the **Network**. This information allows every **Application** to send the data to the right locations and to interpret incoming packets.

1.3 The Basic RTPS Transport Interface

RTPS is designed to run on an unreliable transport mechanism, such as UDP/IP. The protocols implement reliability in the transfer of issues and state.

RTPS takes advantage of the multicast capabilities of the transport mechanism, where one message from a sender can reach multiple receivers.

RTPS is designed to promote determinism of the underlying communication mechanism. The protocol also provides an open trade-off between determinism and reliability.

1.3.1 The Basic Logical Messages

The RTPS protocol uses five logical messages:

- ❑ **ISSUE**: Contains the Application's **UserData**. **ISSUE**s are sent by **Publications** to one or more **Subscriptions.**
- ❑ **VAR**: Contains information about the attributes of a **NetworkObject**, which is part of a composite state. **VAR**s are sent by **CSTWriters** to **CSTReaders**.
- ❑ **HEARTBEAT**: Describes the information that is available in a **Writer**. **HEARTBEAT**s are sent by a **Writer** (**Publication** or **CSTWriter**) to one or more **Readers** (**Subscription** or **CSTReader**).
- ❑ **GAP**: Describes the information that is no longer relevant to **Readers.**
- ❑ **ACK**: Provides information on the state of a **Reader** to a **Writer**.

Each of these logical messages are sent between specific **Readers** and **Writers** as follows:

- ❑ **Publication** to **Subscription(s)**: **ISSUE**s and **HEARTBEAT**s
- ❑ **Subscription** to a **Publication**: **ACK**s
- ❑ **CSTWriter** to a **CSTReader**: **VAR**s, **GAP**s and **HEARTBEAT**s
- ❑ **CSTReader** to a **CSTWriter**: **ACK**s

Readers and **Writers** are both senders and receivers of RTPS **Messages**. In the protocol, the logical messages **ISSUE**, **VAR**, **HEARTBEAT**, **GAP** and **ACK** can be combined into a single message in several ways to make efficient use of the underlying communication mechanism. Chapter 3 explains the format and construction of a **Message**.

1.3.2 Underlying Wire Representation

RTPS uses the CDR (Common Data Representation) as defined by the Object Management Group (OMG) to represent all basic data and structures. Appendix A describes CDR and the specific choices that RTPS made in its usage of CDR.

1.4 Notational Conventions

1.4.1 Name Space

All the definitions in this document are part of the "RTPS" name-space. To facilitate reading and understanding, the name-space prefix has been left out of the definitions and classes in this document. For example, an implementation of RTPS will typically provide the service **RTPSPublication** or **RTPS::Publication**; however, in this document we will use the more simple **Publication**.

1.4.2 Representation of Structures

The following chapters often define structures, such as:

```
typedef struct {
    octet[3] instanceId;
    octet    appKind;
} AppId;
```

These definitions use the OMG IDL (Interface Definition Language). When these structures are sent on the wire, they are encoded using the corresponding CDR representation. Appendix A shows what standards describe this notation.

1.4.3 Representation of Bits and Bytes

This document often uses the following notation to represent an octet or byte:

```
+-+-+-+-+-+-+-+-+
|7|6|5|4|3|2|1|0|
+-+-+-+-+-+-+-+-+
```

In this notation, the leftmost bit (bit 7) is the most significant bit ("MSB") and the rightmost bit (bit 0) is the least significant bit ("LSB").

Streams of bytes are ordered per lines of 4 bytes each as follows:

```
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  first byte   |               |               |    4th byte   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                       -----------stream------------->>>>
```

In such representation, the byte that comes first in the stream is on the left. The bit on the extreme left is the MSB of the first byte; the bit on the extreme right is the LSB of the 4th byte.

Chapter 2

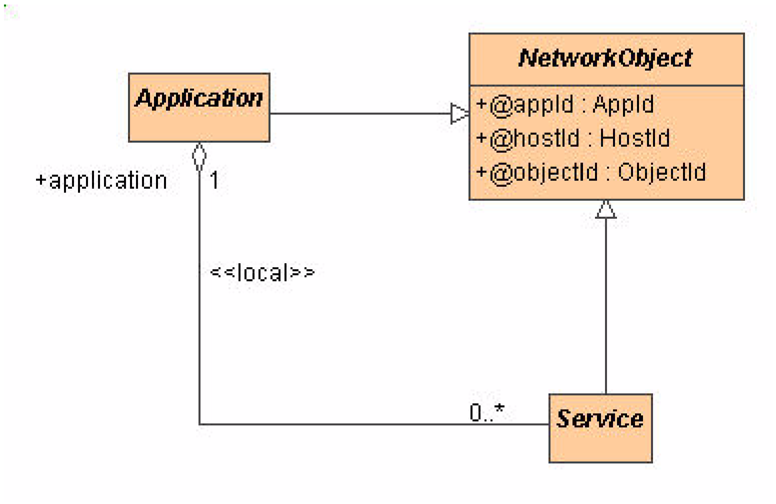
Structure Definitions

This chapter defines the Globally Unique ID (GUID) used to reference objects in a **Network** and the basic structures used in the protocol (to represent bitmaps, sequence numbers, etc.) These structures will be used in the following chapters where the RTPS **Message** is defined.

2.1 Referring to Objects: the GUID

The GUID (Globally Unique Id) is a unique reference to an **Application** or a **Service** on the **Network**.

The GUID is built as a 12-octet triplet: <**HostId** hostId, **AppId** appId, **ObjectId** objectId>. The GUID should be a globally unique reference to one specific **NetworkObject** within the **Network**.



The **HostId** and **AppId** are defined as follows:

```
typedef octet[4] HostId;

typedef struct {
    octet[3] instanceId;
    octet    appKind;
} AppId;
```

where *appKind* is one of the following:

```
0x01  ManagedApplication
0x02  Manager
```

An implementation based on this version (1.0) of the protocol will consider anything other than the above two to be an unknown class.

The unknown *hostId* and *appId* are defined as follows:

```
#define HOSTID_UNKNOWN { 0x00, 0x00, 0x00, 0x00 }
#define APPID_UNKNOWN { 0x00, 0x00, 0x00, 0x00 }
```

2.1.1 The GUIDs of Applications

Every **Application** on the **Network** has GUID <hostId, appId, OID_APP>, where the constant OID_APP is defined as follows.

```
#define OID_APP {0x00,0x00,0x01,0xc1}
```

The implementation is free to pick the *hostId* and *appId*, as long as the last octet of the *appId* identifies the *appKind* and as long as every **Application** on the **Network** has a unique GUID.

2.1.2 The GUIDs of the Services within an Application

The **Services** that are local to the **Application** with GUID <hostId, appId, OID_APP> have GUID <hostId, appId, *objectId*>. The *objectId* is the unique identification of the **NetworkObject** relative to the **Application**. The *objectId* also encapsulates what kind of **NetworkObject** this is, whether the object is a user-object or a meta-object and whether the *instanceId* is freely chosen by the middleware or is a *reserved* instanceId, which has special meaning to the protocol. One example of a reserved (protocol defined) objectId is OID_APP, which is used in the GUID of **Applications**.

The **ObjectId** structure is defined as follows:

```
typedef struct {
    octet[3] instanceId;
    octet    objKind;
} ObjectId;

#define OBJECTID_UNKNOWN { 0x0, 0x0, 0x0, 0x0 }
```

For *objKind*, the two most significant bits indicate whether the object is meta-level or user-level (M-bit) and whether its *instanceId* is chosen or reserved (R-bit), respectively.

```
ObjectId:
0...2...........8...............16.............24...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                instanceId                    |M|R|           |
+---------------+---------------+---------------+---------------+
```

M=1 The **NetworkObject** is a meta-object: it can be reached through the meta-ports of the **Application** to which it belongs (see Chapter 4).

R=1 The instanceId is reserved; it has a special meaning to the protocol. Chapter 5 lists all reserved instanceId's.

The last six bits of the objectId define the class to which the object belongs (**Application**, **Publication**, **Subscription**, **CSTWriter**, or **CSTReader**). Table 2.1 provides an overview. The meaning of the message IDs is fixed in this major version (1). New *objKinds* may be added in higher minor versions as the RTPS object-model is extended with new classes.

Table 2.1 **objKind octet of an objectId**

Class of Object	Normal User-object	Reserved User-object	Normal Meta-object	Reserved Meta-object
unknown	0x00	0x40	0x80	0xc0
Application	0x01	0x41	0x81	0xc1
CSTWriter	0x02	0x42	0x82	0xc2
Publication	0x03	0x43	0x83	0xc3
Subscription	0x04	0x44	0x84	0xc4
CSTReader	0x07	0x47	0x87	0xc7

2.2 Building Blocks of RTPS Messages

This section describes the basic structures that are used inside RTPS **Messages**.

2.2.1 VendorId

This structure identifies the vendor of the middleware implementing the RTPS protocol and allows this vendor to add specific extensions to the protocol. The vendor ID does not refer to the vendor of the device or product that contains RTPS middleware.

```
typedef struct {
    octet major;
    octet minor;
} VendorId;
```

The currently assigned vendor IDs are listed in Table 2.2.

Table 2.2 **Vendor IDs**

Major	Minor	Name
0x00	0x00	VENDOR_ID_UNKNOWN
0x01	0x01	Real-Time Innovations, Inc., CA, USA

2.2.2 ProtocolVersion

The following structure describes the protocol version.

```
typedef struct {
    octet major;
    octet minor;
} ProtocolVersion;
```

Implementations following this version of the document implement protocol version 1.0 (major = 1, minor = 0).

```
#define PROTOCOL_VERSION_1_0 { 0x1, 0x0 }
```

2.2.3 SequenceNumber

A sequence number, N, is a 64-bit signed integer, that can take values in the range:
-2^63 <= N <= 2^63-1.

On the wire, it is represented using two 32-bit integers as follows:

```
typedef struct {
    long high;
    unsigned long low;
} SequenceNumber;
```

Using this structure, the sequence number is: *high* * 2^32 + *low*.

The sequence number, 0, and negative sequence numbers are used to indicate special cases:

```
#define SEQUENCE_NUMBER_NONE     0
#define SEQUENCE_NUMBER_UNKNOWN -1
```

2.2.4 Bitmap

Bitmaps are used as parts of several messages to provide binary information about individual sequence numbers within a range. The representation of the **Bitmap** includes the length of the **Bitmap** in bits and the first **SequenceNumber** to which the **Bitmap** applies.

```
Bitmap:
0...2...........8...............16.............24...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                   SequenceNumber bitmapBase                   +
|                                                               |
+---------------+---------------+---------------+---------------+
|                   long      numBits                           |
+---------------+---------------+---------------+---------------+
|                   long      bitmap[0]                         |
+---------------+---------------+---------------+---------------+
|                   long      bitmap[1]                         |
+---------------+---------------+---------------+---------------+
|                              ...                              |
+---------------+---------------+---------------+---------------+
|                   long      bitmap[M-1]  M = (numBits+31)/32  |
+---------------+---------------+---------------+---------------+
```

Given a **Bitmap,** *bitmap*, the boolean value of the bit pertaining to SequenceNumber *N*, where *bitmapBase* <= *N* < *bitmapBase+numBits* is:

```
bit(N) = bitmap[deltaN/32]  &  (1 << (31 - deltaN%32) )
```

where

```
deltaN = N - bitmapBase
```

The bitmap does not indicate anything about sequence numbers outside of the range [bitmapBase, bitmapBase+numBits-1].

A valid bitmap must satisfy the following conditions:

- bitmapBase >= 1
- 0 <= numBits <= 256
- there are M=(numBits+31)/32 longs containing the pertinent bits

This document uses the following notation for a specific bitmap:

```
bitmapBase/numBits:bitmap
```

In the bitmap, the bit corresponding to sequence number *bitmapBase* is on the left. The ending "0" bits can be represented as one "0".

For example, in bitmap “1234/12:00110”, bitmapBase=1234 and numBits=12. The bits apply as follows to the sequence numbers:

Table 2.3 **Example of bitmap: meaning of “1234/12:00110”**

Sequence	Bit
1234	0
1235	0
1236	1
1237	1
1238-1245	0

2.2.5 NtpTime

Timestamps follow the NTP standard and are represented on the wire as a pair of integers containing the high- and low-order 32 bits:

```
typedef struct {
    long seconds;           // time in seconds
    unsigned long fraction; // time in seconds / 2^32
NtpTime;
```

Time is expressed in seconds using the following formula:

```
seconds + (fraction / 2^(32))
```

The RTPS protocol does not require a concept of absolute time.

2.2.6 IPAddress

An IP address is a 4-byte unsigned number:

```
typedef unsigned long IPAddress
```

An IP address of zero is an invalid IP address:

```
#define IPADDRESS_INVALID 0
```

The mapping between the dot-notation "a.b.c.d" of an IP address and its representation as an unsigned long is as follows:

```
IPAddress ipAddress = ( ( ( a * 256 + b ) * 256 ) + c ) * 256 + d
```

For example, IP address "127.0.0.1" corresponds to the unsigned long number 2130706433 or 0x7F000001.

2.2.7 Port

A port number is a 4-byte unsigned number:

```
typedef unsigned long Port
```

The port number zero is an invalid port-number:

```
#define PORT_INVALID 0
```

If a port number represents an IPv4 UDP port, only the range of unsigned short numbers from 0x1 to 0x0000ffff is valid.

Chapter 3

RTPS Message Format

3.1 Overall Structure of RTPS Messages

The overall structure of a **Message** includes a leading **Header** followed by a variable number of **Submessages**. Each **Submessage** starts aligned on a 32-bit boundary with respect to the start of the **Message**.

```
Message:
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Header                                                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Submessage                                                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.................................................................
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Submessage                                                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A **Message** has a well-known length. This length is not sent explicitly by the RTPS protocol but is part of the underlying transport with which **Messages** are sent. In the case of UDP/IP, the length of the **Message** is the length of the UDP payload.

3.2 Submessage Structure

The general structure of each **Submessage** in a **Message** is as follows:

```
Submessage:
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| submessageId  |      flags  |E|   ushort octetsToNextHeader   |
+---------------+---------------+---------------+---------------+
|                                                               |
~                      contents of submessage                   ~
|                                                               |
+---------------+---------------+---------------+---------------+
```

This general structure cannot change in this major version (1) of the protocol. Sections 3.2.1 through 3.2.3 describe the meaning of the three fields of the submessage header: *submessageId, flags* and *octetsToNextHeader*.

3.2.1 submessageId in the Submessage Header

This octet identifies the kind of **Submessage**. Submessages with IDs 0x00 to 0x7f (inclusive) are protocol-specific. They are defined as part of the RTPS protocol. Version 1.0 defines the following submessages:

```
enum SubmessageId {
PAD         = 0x01,
VAR         = 0x02,
ISSUE       = 0x03,
ACK         = 0x06,
HEARTBEAT  = 0x07,
GAP        = 0x08,
INFO_TS    = 0x09,
INFO_SRC   = 0x0c,
INFO_REPLY = 0x0d,
INFO_DST   = 0x0e
};
```

The meaning of the submessage IDs cannot be modified in this major version (1). Additional submessages can be added in higher minor versions. Submessages with ID's 0x80 to 0xff (inclusive) are vendor-specific; they will not be defined by the protocol. Their interpretation is dependent on the *vendorId* that is current when the submessage is encountered. Section 3.3 describes how the current *vendorId* is determined. The current list of *vendorId*'s is provided in Section 2.2.1.

3.2.2 Flags in the Submessage Header

The least-significant bit (LSB) of the flags is always present in all **Submessages** and represents the endianness used to encode the information in the **Submessage**. E=0 means big-endian, E=1 means little-endian.

Other bits in the flag have interpretations that depend on the type of **Submessage**.

In the following descriptions of the **Submessages**, the character '**X**' is used to indicate a flag that is unused in version 1.0 of the protocol. RTPS implementations of version 1.0 should set these to zero when sending and ignore these when receiving. Higher minor versions of the protocol can use these flags.

3.2.3 octetsToNextHeader in the Submessage Header

The final two octets of the **Submessage** header contain the number of octets from the first octet of the contents of the submessage until the first octet of the header of the next **Submessage**. The representation of this field is a CDR unsigned short (ushort). If the **Submessage** is the last one in the **Message**, the *octetsToNextHeader* field contains the number of octets remaining in the **Message**.

3.3 How to Interpret a Message

The interpretation and meaning of a **Submessage** within a **Message** may depend on the previous **Submessages** within that same **Message**. Therefore the receiver of a **Message** must maintain state from previously deserialized **Submessages** in the same **Message**.

sourceVersion The major and minor version with which the following submessages need to be interpreted.

sourceVendorId The vendor identification with which the following vendor-specific extensions need to be interpreted.

sourceHostId, sourceAppId The originator's host and application identifiers. The following submessages need to be identified as if they are coming from this host and application.

destHostId, destAppId The destination's host and application identifiers. The following submessages need to be identified as if they are meant for this host and application.

unicastReplyIPAddress, unicastReplyPort An explicit IP address and port that provides an additional direct way for the receiver to reply directly to the originator over unicast.

multicastReplyIPAddress, multicastReplyPort An explicit IP address and port that provides an additional direct way for the receiver to reach the originator (and potentially many others) over multicast.

haveTimestamp, timestamp The timestamp applying to all the following submessages.

3.3.1 Rules Followed By A Message Receiver

The following algorithm outlines the rules that a receiver of any **Message** must follow:

1. If a 4-byte **Submessage** header cannot be read, the rest of the **Message** is considered invalid.
2. The last two bytes of a **Submessage** header, the *octetsToNextHeader* field, contains the number of octets to the next **Submessage**. If this field is invalid, the rest of the **Message** is invalid.
3. The first byte of a **Submessage** header is the *submessageId*. A **Submessage** with an unknown ID must be ignored and parsing must continue with the next **Submessage**. Concretely: an implementation of RTPS 1.0 must ignore any **Submessages** with IDs that are outside of the **SubmessageId** list used by version 1.0. IDs in the vendor-specific range coming from a *vendorId* that is unknown must be ignored and parsing must continue with the next **Submessage**.
4. The second byte of a **Submessage** header contains flags; unknown flags should be skipped. An implementation of RTPS 1.0 should skip all flags that are marked as "**X**" (unused) in the protocol.
5. A valid *octetsToNextHeader* field must *always* be used to find the next **Submessage**, even for **Submessages** with unknown IDs.
6. A known but invalid **Submessage** invalidates the rest of the **Message**. Sections 3.5 through 3.14 each describe known **Submessage** and when it should be considered invalid.

Reception of a valid header and/or submessage has two effects:

- It can change the state of the receiver; this state influences how the following **Submessages** in the **Message** are interpreted. Sections 3.5 through 3.14 show how the state changes for each **Submessage**. In this version of the protocol, only the **Header** and the **Submessages** **INFO_SRC**, **INFO_REPLY** and **INFO_TS** change the state of the receiver.
- The **Submessage**, interpreted within the **Message**, has a logical interpretation: it encodes one of the five basic RTPS messages: **ACK**, **GAP**, **HEARTBEAT**, **ISSUE** or **VAR**.

Sections 3.4 through 3.14 describe the detailed behavior of the **Header** and every **Submessage**.

3.4 Header

This is the **Header** found at the beginning of every **Message**.

3.4.1 Format

```
0...2..........7..............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      'R'      |      'T'      |      'P'      |      'S'      |
+---------------+---------------+---------------+---------------+
| ProtocolVersion version       | VendorId vendorId             |
+---------------+---------------+---------------+---------------+
| HostId hostId                                                 |
+---------------+---------------+---------------+---------------+
| AppId appId                                                   |
+---------------+---------------+---------------+---------------+
```

3.4.2 Validity

A **Header** is *invalid* when any of the following are true:

- The **Message** has less than the required number of octets to contain a full **Header**.
- Its first four octets are not '**R**' '**T**' '**P**' '**S**'.

- o The major protocol version is larger than the major protocol version supported by the implementation.

3.4.3 Change in State of the Receiver

```
sourceHostId   = Header.hostId
sourceAppId    = Header.appId
sourceVersion  = Header.version
sourceVendorId = Header.vendorId
haveTimestamp  = false
```

3.4.4 Logical Interpretation

None

3.5 ACK

This submessage is used to communicate the state of a **Reader** to a **Writer**.

3.5.1 Submessage Format

```
0...2...........7..............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       ACK     |X|X|X|X|X|X|F|E|     octetsToNextHeader        |
+---------------+---------------+---------------+---------------+
| ObjectId readerObjectId                                       |
+---------------+---------------+---------------+---------------+
| ObjectId writerObjectId                                       |
+---------------+---------------+---------------+---------------+
|                                                               |
~ Bitmap bitmap                                                 ~
|                                                               |
+---------------+---------------+---------------+---------------+
```

3.5.2 Validity

This submessage is *invalid* when any of the following is true:

- o *octetsToNextHeader* is too small.
- o *bitmap* is invalid.

3.5.3 Change in State of the Receiver

None

3.5.4 Logical Interpretation

Table 3.1 **Interpretation of ACK Submessage**

Field	Value
FINAL-bit	ACK.F
readerGUID	<sourceHostId, sourceAppId, ACK.readerObjectId>
writerGUID	<destHostId, destAppId, ACK.writerObjectId>
replyIPAddressPortList	{ unicastReplyIPAddress : unicastReplyPort, multicastReplyIPAddress : multicastReplyPort }
bitmap	ACK.bitmap

FINAL-bit ACK.F : When the F-bit is set, the application sending the ACK does not expect a response to the ACK.

readerGUID <sourceHostId, sourceAppId, ACK.readerObjectId> : The GUID of the **Reader** that acknowledges receipt of certain sequence numbers and/or requests to receive certain sequence numbers.

writerGUID <destHostId, destAppId, ACK.writerObjectId> : The GUID of the **Writer** that the reader has received these sequence numbers from and/or wants to receive these sequence numbers from.

replyIPAddressPortList { unicastReplyIPAddress : unicastReplyPort, multicastReplyIPAddress : multicastReplyPort } : This is an additional list of addresses that the receiving application can use to respond to this ACK.

bitmap ACK.bitmap : A "0" in this bitmap means that the corresponding sequence-number is missing. A "1" in the bitmap conveys no information, that is, the corresponding sequence number may or may not be missing. By sending an ACK, the readerGUID object acknowledges receipt of all messages up to and including the sequence number (bitmap.bitmapBase -1).

3.6 GAP

This submessage is sent from a **CSTWriter** to a **CSTReader** to indicate that a range of sequence numbers is no longer relevant. The set may be a contiguous range of sequence numbers or a specific set of sequence numbers.

3.6.1 Submessage Format

```
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      GAP      |X|X|X|X|X|X|X|E|   octetsToNextHeader          |
+---------------+---------------+---------------+---------------+
| ObjectId readerObjectId                                       |
+---------------+---------------+---------------+---------------+
| ObjectId writerObjectId                                       |
+---------------+---------------+---------------+---------------+
|                                                               |
+ SequenceNumber firstSeqNumber                                 +
|                                                               |
+---------------+---------------+---------------+---------------+
|                                                               |
~ Bitmap bitmap                                                 ~
|                                                               |
+---------------+---------------+---------------+---------------+
```

3.6.2 Validity

This submessage is *invalid* when any of the following are true:

- o *octetsToNextHeader* is too small.
- o *bitmap* is invalid.
- o *firstSeqNumber* is 0 or negative.

3.6.3 Change in State of the Receiver

None

3.6.4 Logical Interpretation

Table 3.2 **Interpretation of GAP Submessage**

Field	Value
readerGUID	<destHostId, destAppId, GAP.readerObjectId>
writerGUID	<sourceHostId, sourceAppId, GAP.writerObjectId>
ACKIPAddressPortList	{ unicastReplyIPAddress : unicastReplyPort }
gapList	{ GAP.firstSeqNumber, GAP.firstSeqNumber+1, ..., GAP.bitmap.bitmapBase-1 } *and* all sequence numbers that have a corresponding bit set to 1 in the bitmap

readerGUID <destHostId, destAppId, GAP.readerObjectId> : The GUID of the **CSTReader** for which the *gapList* is meant. The GAP.readerObjectId can be **OBJECTID_UNKNOWN**, in which case the **GAP** applies to all **Readers** within the **Application** <destHostId, destAppId>.

writerGUID <sourceHostId, sourceAppId, GAP.writerObjectId> : The GUID of the **CSTWriter** to which the *gapList* applies.

ACKIPAddressPortList { unicastReplyIPAddress : unicastReplyPort } : If the **CSTReader** that receives this submessage needs to reply with an **ACK** submessage, then this **ACK** can be sent to one of the explicit destinations in this list.

gapList The list of sequence numbers that are no longer available in the *writerObject*. This list is the union of:

- o All the sequence numbers in the range from *GAP.firstSeqNumber* up to *GAP.bitmap.bitmapBase* - 1. This list is empty if the *firstSeqNumber* is greater than or equal to the *bitmapBase* of the *bitmap*. *GAP.firstSeqNumber* should always be greater than or equal to 1.

  and
- o The sequence numbers that have the corresponding bit in the *bitmap* set to 1.

3.6.5 Example

A **GAP** with:

- o *firstSeqNumber* = 12
- o *bitmap* = 17/5:0011101

means that the *gapList* = {12, 13, 14, 15, 16, 19, 20, 22}.

3.7 HEARTBEAT

This message is sent from a **Writer** to a **Reader** to communicate the sequence numbers of data that the **Writer** has available.

3.7.1 Submessage Format

```
0...2..........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| HEARTBEAT     |X|X|X|X|X|X|F|E|   octetsToNextHeader          |
+---------------+---------------+---------------+---------------+
| ObjectId readerObjectId                                       |
+---------------+---------------+---------------+---------------+
| ObjectId writerObjectId                                       |
+---------------+---------------+---------------+---------------+
|                                                               |
+ SequenceNumber firstSeqNumber                                 +
|                                                               |
+---------------+---------------+---------------+---------------+
|                                                               |
+ SequenceNumber lastSeqNumber                                  +
|                                                               |
+---------------+---------------+---------------+---------------+
```

3.7.2 Validity

This submessage is *invalid* when any of the following are true:

- o *octetsToNextHeader* is too small.
- o *firstSeqNumber* is less than 0.
- o *lastSeqNumber* is less than 0.
- o *lastSeqNumber* is strictly less than *firstSeqNumber*.

3.7.3 Change in State of the Receiver

None

3.7.4 Logical Interpretation

Table 3.3 **Interpretation of HEARTBEAT Submessage**

Field	Value
FINAL-bit	HEARTBEAT.F
readerGUID	<destHostId, destAppId, HEARTBEAT.readerObjectId>
writerGUID	<sourceHostId, sourceAppId, HEARTBEAT.writerObjectId>
ACKIPAddressPortList	{ unicastReplyIPAddress : unicastReplyIPPort }
firstSeqNumber	HEARTBEAT.firstSeqNumber
lastSeqNumber	HEARTBEAT.lastSeqNumber

FINAL-bit HEARTBEAT.F : When the F-bit is set, the application sending the **HEARTBEAT** does not require a response.

readerGUID <destHostId, destAppId, HEARTBEAT.readerObjectId> : The **Reader** to which the heartbeat applies. The HEARTBEAT.readerObjectId can be **OBJECTID_UNKNOWN**, in which case the **HEARTBEAT** applies to all **Readers** of that *writerGUID* within the **Application** <destHostId, destAppId>.

writerGUID <sourceHostId, sourceAppId, HEARTBEAT.writerObjectId> : The **Writer** to which the **HEARTBEAT** applies.

ACKIPAddressPortList { unicastReplyIPAddress : unicastReplyIPPort } : An additional list of destinations where responses (**ACKs**) to this submessage can be sent.

firstSeqNumber HEARTBEAT.firstSeqNumber : The first sequence number, *firstSeqNumber*, that is still available and meaningful in the *writerObject*. This field must be greater than or equal to zero. If it is equal to **SEQUENCE_NUMBER_NONE**, the **Writer** has no data available

lastSeqNumber HEARTBEAT.lastSeqNumber : The last sequence number, *lastSeqNumber*, that is available in the **Writer**. This field must be greater than or equal to *firstSeqNumber*. If *firstSeqNumber* is **SEQUENCE_NUMBER_NONE**, *lastSeqNumber* must also be **SEQUENCE_NUMBER_NONE**.

3.8 INFO_DST

This submessage modifies the logical destination of the submessages that follow it.

3.8.1 Submessage Format

```
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|INFO_DST    |X|X|X|X|X|X|X|E|octetsToNextHeader   |
+---------------+---------------+---------------+---------------+
|              HostId hostId                               |
+---------------+---------------+---------------+---------------+
|               AppId appId                                |
+---------------+---------------+---------------+---------------+
```

3.8.2 Validity

This submessage is *invalid* when:

- *octetsToNextHeader* is too small.

3.8.3 Change In State Of The Interpreter

```
if(INFO_DST.hostId != HOSTID_UNKNOWN) {
    destHostId = INFO_DST.hostId
} else {
    destHostId = hostId of application receiving the message
}

if(INFO_DST.appId != APPID_UNKNOWN) {
    destAppId = INFO_DST.appId
} else {
    destAppId = appId of application receiving the message
}
```

In other words, an INFO_DST with a HOSTID_UNKNOWN means that any host may interpret the following submessages as if they were meant for it. Similarly, an INFO_DST with a APPID_UNKNOWN means that any application may interpret the following submessages as if they were meant for it.

3.8.4 Logical Interpretation

None; this only affects the interpretation of the submessages that follow it.

3.9 INFO_REPLY

This submessage contains explicit information on where to send a reply to the submessages that follow it within the same message.

3.9.1 Submessage Format

```
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|INFO_REPLY     |X|X|X|X|X|X|M|E|      octetsToNextHeader       |
+---------------+---------------+---------------+---------------+
| IPAddress unicastReplyIPAddress                               |
+---------------+---------------+---------------+---------------+
| Port unicastReplyPort                                         |
+---------------+---------------+---------------+---------------+
| IPAddress multicastReplyIPAddress [ only if M==1 ]            |
+---------------+---------------+---------------+---------------+
| Port multicastReplyPort [ only if M==1 ]                      |
+---------------+---------------+---------------+---------------+
```

3.9.2 Validity

This submessage is *invalid* when the following is true:

- *octetsToNextHeader* is too small.

3.9.3 Change in State of the Receiver

```
if ( INFO_REPLY.unicastReplyIPAddress != IPADDRESS_INVALID) {
    unicastReplyIPAddress  = INFO_REPLY.unicastReplyIPAddress;
}
unicastReplyPort  = INFO_REPLY.replyPort
if ( M==1 ) {
    multicastReplyIPAddress  = INFO_REPLY.multicastReplyIPAddress
    multicastReplyPort       = INFO_REPLY.multicastReplyPort
} else {
    multicastReplyIPAddress  = IPADDRESS_INVALID
    multicastReplyPort       = PORT_INVALID
}
```

3.9.4 Logical Interpretation

None, this only affects the interpretation of the submessages that follow it.

3.10 INFO_SRC

This submessage modifies the logical source of the submessages that follow it.

3.10.1 Submessage Format

```
0...2..........7..............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| INFO_SRC      |X|X|X|X|X|X|X|E|   octetsToNextHeader          |
+---------------+---------------+---------------+---------------+
| IPAddress appIPAddress                                        |
+---------------+---------------+---------------+---------------+
| ProtocolVersion version       | VendorId vendorId             |
+---------------+---------------+---------------+---------------+
| HostId hostId                                                 |
+---------------+---------------+---------------+---------------+
| AppId appId                                                   |
+---------------+---------------+---------------+---------------+
```

3.10.2 Validity

This submessage is *invalid* when the following is true:

o *octetsToNextHeader* is too small.

3.10.3 Change in State of the Receiver

```
sourceHostId             = INFO_SRC.hostId
sourceAppId              = INFO_SRC.appId
sourceVersion            = INFO_SRC.version
sourceVendorId           = INFO_SRC.vendorId
unicastReplyIPAddress    = INFO_SRC.appIPAddress
unicastReplyPort         = PORT_INVALID
multicastReplyIPAddress  = IPADDRESS_INVALID
multicastReplyPort       = PORT_INVALID
haveTimestamp            = false
```

3.10.4 Logical Interpretation

None, this only affects the interpretation of the submessages that follow it.

3.11 INFO_TS

This submessage is used to send a timestamp which applies to the submessages that follow within the same message.

3.11.1 Submessage Format

```
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| INFO_TS       |X|X|X|X|X|X|I|E|   octetsToNextHeader          |
+---------------+---------------+---------------+---------------+
|                                                               |
+ NtpTime ntpTimestamp [only if I==0]                           +
|                                                               |
+---------------+---------------+---------------+---------------+
```

3.11.2 Validity

This submessage is *invalid* when the following is true:

- o *octetsToNextHeader* is too small.

3.11.3 Change in State of the Receiver

```
if (INFO_TS.I==0) {
    haveTimestamp   =  true
    timestamp       = INFO_TS.ntpTimestamp
} else {
    haveTimestamp   =  false
}
```

3.11.4 Logical Interpretation

None, this only affects the interpretation of the submessages that follow it.

3.12 ISSUE

This submessage is used to send issues from a **Publication** to a **Subscription**.

3.12.1 Submessage Format

```
0...2..........7..............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ISSUE         |X|X|X|X|X|X|P|E|      octetsToNextHeader       |
+---------------+---------------+---------------+---------------+
| ObjectId readerObjectId                                       |
+---------------+---------------+---------------+---------------+
| ObjectId writerObjectId                                       |
+---------------+---------------+---------------+---------------+
|                                                               |
+ SequenceNumber issueSeqNumber                                 +
|                                                               |
+---------------+---------------+---------------+---------------+
|                                                               |
+ ParameterSequence parameters  [only if P==1]                  +
|                                                               |
+---------------+---------------+---------------+---------------+
|                                                               |
~ UserData issueData                                            ~
|                                                               |
+---------------+---------------+---------------+---------------+
```

3.12.2 Validity

This submessage is *invalid* when any of the following are true:

- *octetsToNextHeader* is too small.
- *issueSeqNumber* is either not strictly positive (1,2,...) or is not **SEQUENCE_NUMBER_UNKNOWN**.
- the parameter sequence is invalid.

3.12.3 Change in State of the Receiver

None

3.12.4 Logical Interpretation

Table 3.4 **Interpretation of ISSUE Submessage**

Field	Value
subscriptionGUID	<destHostid, destAppId, ISSUE.readerObjectId>
publicationGUID	<sourceHostId, sourceAppId, ISSUE.writerObjectId>
issueSeqNumber	ISSUE.issueSeqNumber
(parameters)	ISSUE.parameters (iff ISSUE.P==1)
ACKIPAddressPortList	{ unicastReplyIPAddress : unicastReplyPort }
(timestamp)	timestamp (present iff haveTimestamp == true)
issueData	ISSUE.issueData

subscriptionGUID <destHostid, destAppId, ISSUE.readerObjectId> : The **Subscription** for which the ISSUE is meant. The ISSUE.readerObjectId can be **OBJECTID_UNKNOWN**, in which case the **ISSUE** applies to all **Subscriptions** within the **Application** <destHostId, destAppId>.

publicationGUID <sourceHostId, sourceAppId, ISSUE.writerObjectId> : The **Publication** object that originated this issue.

issueSeqNumber ISSUE.issueSeqNumber : The sequence number of this issue; this should either be a strictly positive number (1,2,3,...) or the special sequence-number **SEQUENCENUMBER_UNKNOWN**. The latter may be used by a simple publication that does not number consecutive issues.

parameters (optional) ISSUE.parameters : This is present iff P == 1. These parameters will allow future extensions of the protocol. An implementation of RTPS 1.0 can ignore the contents of this **ParameterSequence**.

ACKIPAddressPortList { unicastReplyIPAddress : unicastReplyPort } : The destinations to which the **Publication** can send an **ACK** message in response to this **ISSUE**.

timestamp (optional) Timestamp of this issue. This is present iff Timestamp == true.

issueData ISSUE.issueData : The actual user data in this issue.

3.13 PAD

This submessage has no meaning.

3.13.1 Submessage Format

```
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   PAD         |X|X|X|X|X|X|X|E|     octetsToNextHeader        |
+---------------+---------------+---------------+---------------+
```

3.13.2 Validity

This submessage is always valid.

3.13.3 Change in State of the Receiver

None

3.13.4 Logical Submessage Generated On Reception

None; the receiver skips the **PAD** using *octetsToNextHeader.*

3.14 VAR

This submessage is used to communicate information about a **NetworkObject** (which is part of the Composite State). It is sent from a **CSTWriter** to a **CSTReader**.

3.14.1 Submessage Format

```
0...2...........7...............15.............23...............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   VAR         |X|X|X|X|H|A|P|E|     octetsToNextHeader        |
+---------------+---------------+---------------+---------------+
| ObjectId readerObjectId                                  |
+---------------+---------------+---------------+---------------+
| ObjectId writerObjectId                                       |
+---------------+---------------+---------------+---------------+
| HostId hostId (iff H==1)                                      |
+---------------+---------------+---------------+---------------+
| AppId appId (iff H==1)                                        |
+---------------+---------------+---------------+---------------+
| ObjectId objectId                                             |
+---------------+---------------+---------------+---------------+
|                                                               |
+ SequenceNumber writerSeqNumber                                +
|                                                               |
+---------------+---------------+---------------+---------------+
|                                                               |
~ ParameterSequence parameters [only if P==1]                   ~
|                                                               |
+---------------+---------------+---------------+---------------+
```

3.14.2 Validity

This submessage is *invalid* when any of the following are true:

- *octetsToNextHeader* is too small.
- *writerSeqNumber* is not strictly positive (1, 2, ...) or is **SEQUENCENUMBER_UNKNOWN**.
- the parameter sequence is invalid.

3.14.3 Change in State of the Receiver

None

3.14.4 Logical interpretation

Table 3.5 **Interpretation of VAR Submessage**

Field	Value
readerGUID	<destHostId, destAppId, VAR.readerObjectId>
writerGUID	<sourceHostId, sourceAppId, VAR.writerObjectId>
objectGUID	<VAR.hostId, VAR.appId, VAR.objectId> (iff H == 1) <sourceHostId, sourceAppId, VAR.objectId> (iff H == 0)
writerSeqNumber	VAR.writerSeqNumber
(timestamp)	current.timestamp if curent.haveTimestamp == true
(parameters)	VAR.parameters and VAR.P
ALIVE-bit	VAR.A

ACKIPAddressPortList	{
	unicastReplyIPAddress : unicastReplyIPPort,
	writer->IPAddressPortList()
	}

readerGUID <destHostId, destAppId, VAR.readerObjectId> : The **Reader** to which the heartbeat applies. The VAR.readerObjectId can be **OBJECTID_UNKNOWN**, in which case the **VAR** applies to all **Readers** of that *writerGUID* within the **Application** <destHostId, destAppId>.

writerGUID <sourceHostId, sourceAppId, VAR.writerObjectId> : The **CSTWriter** that sent the information.

objectGUID <VAR.hostId, VAR.appId, VAR.objectId> (iff H == 1) or <sourceHostId, sourceAppId, VAR.objectId> (iff H == 0) : The object this information (contained in the parameters) is about.

writerSeqNumber VAR.writerSeqNumber : Incremented each time a change in the Composite State provided by the **CSTWriter** occurs. This should be a strictly positive number (1, 2, ...). Or, the special sequence number, **SEQUENCE_NUMBER_UNKNOWN**, may be sent to indicate that the sender does not keep track of the sequence number.

timestamp (optional) current.timestamp : This is present iff curent.haveTimestamp == true. Timestamp of the new parameters sent with this submessage.

parameters (optional) VAR.parameters : This is present iff VAR.P == 1. Contains information about the object.

ALIVE-bit VAR.A : See Chapter 7.

ACKIPAddressPortList { unicastReplyIPAddress : unicastReplyIPPort, writer->IPAddressPortList() } : Where to sent **ACKs** in reply to this submessage.

3.15 Versioning and Extensibility

An implementation based on this version (1.0) of the protocol should be able to process RTPS messages not only with the same major version (1) but possibly higher minor versions.

3.15.1 Allowed Extensions Within This Major Version

Within this major version, future minor versions of the protocol can augment the protocol in the following ways:

- o Additional submessages with other *submessageIds* can be introduced and used anywhere in an RTPS message. Therefore, a 1.0 implementation should skip over unknown submessages (using the *octetsToNextHeader* field in the submessage header).
- o Additional fields can be added to the end of a submessage that was already defined in the current minor version. Therefore, a 1.0 implementation should skip over possible additional fields in a submessage using the *octetsToNextHeader* field.
- o Additional object-kinds and built-in objects with new IDs can be added; these should be ignored by the 1.0 implementation.
- o Additional parameters with new IDs can be added; these should be ignored by the 1.0 implementation.

All such changes require an increase of the minor version number.

3.15.2 What Cannot Change Within This Major Version

The following items cannot be changed within the same major version:

- o A submessage cannot be deleted.
- o A submessage cannot be modified except as described in Section 3.15.1.
- o The meaning of the *submessageIds* (described in Section 3.2.1) cannot be modified.

All such changes require an increase of the major version number.

Chapter 4

RTPS and UDP/IPv4

This chapter describes the mapping of RTPS on UDP/IP v4.

4.1 Concepts

4.1.1 RTPS Messages and the UDP Payload

When RTPS is used over UDP/IP, a **Message** is the contents (payload) of exactly one UDP/IP Datagram.

4.1.2 UDP/IP Destinations

A UDP/IP destination consists of an **IPAddress** and a **Port**. This document uses notation such as "12.44.123.92:1024" or "225.0.1.2:6701" to refer to such a destination. The IP address can be a unicast or multicast address.

4.1.3 Note On Relative Addresses

The RTPS protocol often sends IP addresses to a sender of **Messages**, so that the sender knows where to send future **Messages**. These destinations are always interpreted locally by the sender of UDP datagrams. Certain IP addresses, such as "127.0.0.1" have only relative meaning (i.e. they do not refer to a unique host).

4.2 RTPS Packet Addressing

The following sections describe how a sending application can construct a list of **IPAddress:Port** pairs that it can use to send **Messages** to remote **Services**. Every **Service** has a method, **IPAddressPortList()**, that represents this list. This **IPAddressPortList** is gathered by combining four sources:

- ❑ The well-known ports of the **Network**.
- ❑ The attributes of the **Application** in which the **Service** exists, as well as whether the **Application** is a **Manager** or a **ManagedApplication**.
- ❑ Whether the **Service** is user-level or meta-level (M-bit in the GUID).
- ❑ Additional attributes of the **Service** itself.

The sender's implementation is free to send the information to any valid destination(s) in this list and is encouraged to make good choices, depending on its network interfaces, resources or optimization concerns.

4.2.1 Well-known Ports

At the **Network** level, RTPS uses the following three well-known ports:

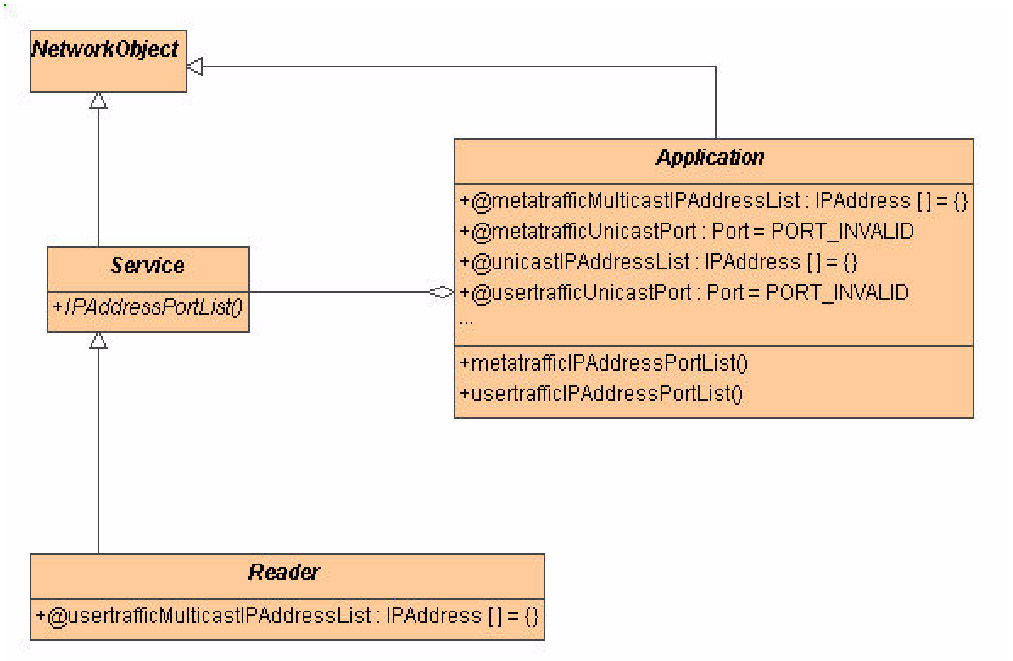
```
wellknownManagerPort =   portBaseNumber + 10 * portGroupNumber

wellknownUsertrafficMulticastPort =
                         1 + portBaseNumber + 10 * portGroupNumber
wellknownMetatrafficMulticastPort =
                         2 + portBaseNumber + 10 * portGroupNumber
```

Within a **Network**, all applications need to use the same ***portBaseNumber***. Applications that want to communicate with each other use the same ***portGroupNumber***; applications that need to be isolated from each other use a different ***portGroupNumber***.

Each application needs to be configured with the correct ***portBaseNumber*** and ***portGroupNumber***.

Except for the rules stated above, RTPS does not define which ***portBaseNumber*** and ***portGroupNumber*** are used nor how the **Applications** participating in a **Network** obtain this information.

4.2.2 Relevant Attributes of an Application


NetworkObject
Application
+@metatrafficMulticastIPAddressList : IPAddress [] = {}
+@metatrafficUnicastPort : Port = PORT_INVALID
+@unicastIPAddressList : IPAddress [] = {}
+@usertrafficUnicastPort : Port = PORT_INVALID
...
+metatrafficIPAddressPortList()
+usertrafficIPAddressPortList()
Service
+IPAddressPortList()
Reader
+@usertrafficMulticastIPAddressList : IPAddress [] = {}

The relevant attributes of an **Application** are:

unicastIPAddressList These are the unicast IP addresses of the **Application**; they are the unicast IP addresses of the host on which the **Application** runs (there can be multiple addresses on a multi-NIC host). Depending on the network topology, a sending application might only be able to address that application on a subset of these IP addresses.

metatrafficMulticastIPAddressList For the purposes of meta-traffic, an **Application** can also accept **Messages** on this set of multicast addresses.

usertrafficUnicastPort and **metatrafficUnicastPort** Every **Application** has exactly two application-dependent ports where it receives unicast user-traffic and unicast meta-traffic, respectively. A datagram sent to one of the application's unicast IP addresses and to one of these ports should only be received by one **Application**.

These attributes define two lists of UDP destinations. The first list, represented by the method **usertrafficAddressPortList()**, is used for user data; the second list, **metatrafficAddressPortList()**, is used for the RTPS metatraffic. These lists are defined as follows:

```
Application::metatrafficIPAddressPortList() =
{
unicastIPAddressList[] : metatrafficUnicastPort,
metatrafficMulticastIPAddressList[] : wellknownMetatrafficMulticastPort
}

Application::usertrafficIPAddressPortList() =
{
unicastIPAddressList[] : usertrafficUnicastPort
}
```

RTPS messages sent to the multicast destinations can be received by multiple applications on multiple hosts.

4.2.3 Manager

For the special case of a **Manager**, these lists are defined as follows:

```
Manager::metatrafficIPAddressPortList() =
{
unicastIPAddressList[]              : wellknownManagerPort,
metatrafficMulticastIPAddressList[] : wellknownManagerPort
}
```

A manager receives all data on one well-known port, the ***wellknownManagerPort***.

```
Manager::usertrafficIPAddressPortList() = NULL
```

A **Manager** does not handle user data, only meta-data.

4.2.4 Definition of the IPAddressPortList()

A distinction needs to be made between a **Reader** and a **Writer**.

A **Writer** that is a meta-object is addressed through the metatraffic ports of the **Application** to which it belongs; if the **Writer** is a user-object, it is addressed through its **Application**'s user-data ports:

```
iff user-object
Writer::IPAddressPortList() = Application()->usertrafficIPAddressPortList()

iff meta-object
Writer::IPAddressPortList() = Application()->metatrafficIPAddressPortList()
```

Note that the *GUID* of the object immediately shows whether the object is a meta-object or a user-object.

A **Reader** (such as a **Subscription**) has an additional attribute: *usertrafficMulticastIPAddressList*.

The *IPAddressPortList* of a **Reader** is defined as follows:

```
iff user-object
Reader::IPAddressPortList() =
    {
       Application()->usertrafficIPAddressPortList(),
       usertrafficMulticastIPAddressList[] : wellknownUsertrafficMulticastPort
    }

iff meta-object
Reader::IPAddressPortList() = Application()->metatrafficIPAddressPortList()
```

A user-level **Reader** can be addressed by unicast over the destination in the *usertrafficIPAddressPortList* of the **Application** to which it belongs or by sending UDP multicast to the additional multicast addresses the **Reader** provides at the *wellknownUsertrafficMulticastPort*.

A meta-**Reader** is addressed through the *metatrafficIPAddressPortList* of the application to which it belongs.

4.3 Possible Destinations for Specific Submessages

This section lists the UDP/IP destinations to which the basic **Submessages** (**ACK**, **HEARTBEAT**, **GAP**, **ISSUE** and **VAR**) can be sent.

4.3.1 Possible Destinations of an ACK

An **ACK** is usually sent to one of the known ports of the **Writer** (this could be a **Publication** or a **CSTWriter**) for which the **ACK** is meant (these ports are defined in Section 4.2.4 as *writer->IPAddressPortList()*).

An **ACK** can also be sent in response to a **VAR**, **HEARTBEAT**, **GAP** or **ISSUE**. The logical interpretation of these submessages explicitly contains an *ACKIPAddressPortList*, which contains possible additional destinations where such an **ACK** can be sent.

4.3.2 Possible Destinations of a GAP

A **GAP** is normally sent to a **CSTReader** which can be addressed through the *reader->IPAddressPortList()*, defined in Section 4.2.4.

A **GAP** can also be sent in response to an **ACK**, in which case the **GAP** can be sent to one of the destinations in the logical *replyIPAddressPortList* of the **ACK**.

4.3.3 Possible Destinations of a HEARTBEAT

A **HEARTBEAT** is sent to a **Reader**, *reader*, (either a **CSTReader** or a **Subscription**); which can be addressed on *reader->IPAddressPortList()*, defined in Section 4.2.4.

A **HEARTBEAT** can also be sent in response to an **ACK**, in which case the **HEARTBEAT** can be sent to one of the destinations in the logical *replyIPAddressPortList* of the **ACK**.

4.3.4 Possible Destinations of an ISSUE

To address a **Subscription**, *sub*, (a subclass of a **Reader**), this submessage needs to be sent to one of the destinations in *sub->IPAddressPortList()*.

An **ISSUE** can also be sent in response to an **ACK**, in which case the **ISSUE** can also be sent to one of the destinations in the logical *replyIPAddressPortList* of the **ACK**.

4.3.5 Possible Destinations of a VAR

To address a **Reader**, *reader*, the **VAR** is sent to one of the address/ports in *reader->IPAddressPortList()*.

A **VAR** can also be sent in response to an **ACK**, in which case the **VAR** can also be sent to one of the destinations in the logical *replyIPAddressPortList* of the **ACK**.

Chapter 5

Attributes of Objects and Metatraffic

5.1 Concept

Figure 5.1 shows an overview of all the attributes of the **NetworkObjects**. Some of the attributes are *frozen*, indicated by the symbol "@" in front of them. The value of a *frozen* attribute cannot change during the life of the object. All attributes of a **NetworkObject** (except for its GUID) have default values.The protocol uses the CST protocol to convey information about the creation, destruction and attributes of **Objects** (**Applications** and their **Services**) on the **Network**.

On the wire, the attributes of the objects are encoded in the **ParameterSequence** that is part of the **VAR** submessage (see Section 3.14). The information in the **ParameterSequence** applies to the object with GUID *objectGUID*. This GUID immediately encodes the class of the object and, therefore, the relevant attributes of the object and their default values.

When the parameter sequence does not contain information about an attribute that is part of the class, the receiving application may assume that attribute has the default value.

The semantics of these classes and their attributes cannot be changed in this major version (1) of the protocol. Higher minor versions can extend this model in two ways:

- ❑ New classes may be added.
- ❑ New attributes may be added to the existing classes.

Table 5.1 shows the attributes of a **ManagedApplication**. The convention followed is that a preceding "@" denotes that the attribute is *frozen* and thus cannot be changed. A trailing "[ ]" denotes an array that indicates that the attribute can be repeated. a **Manager** submessage has the contents described in Table 5.1 and another attribute described in Table 5.2. The description of the types are included in Section 6.1.

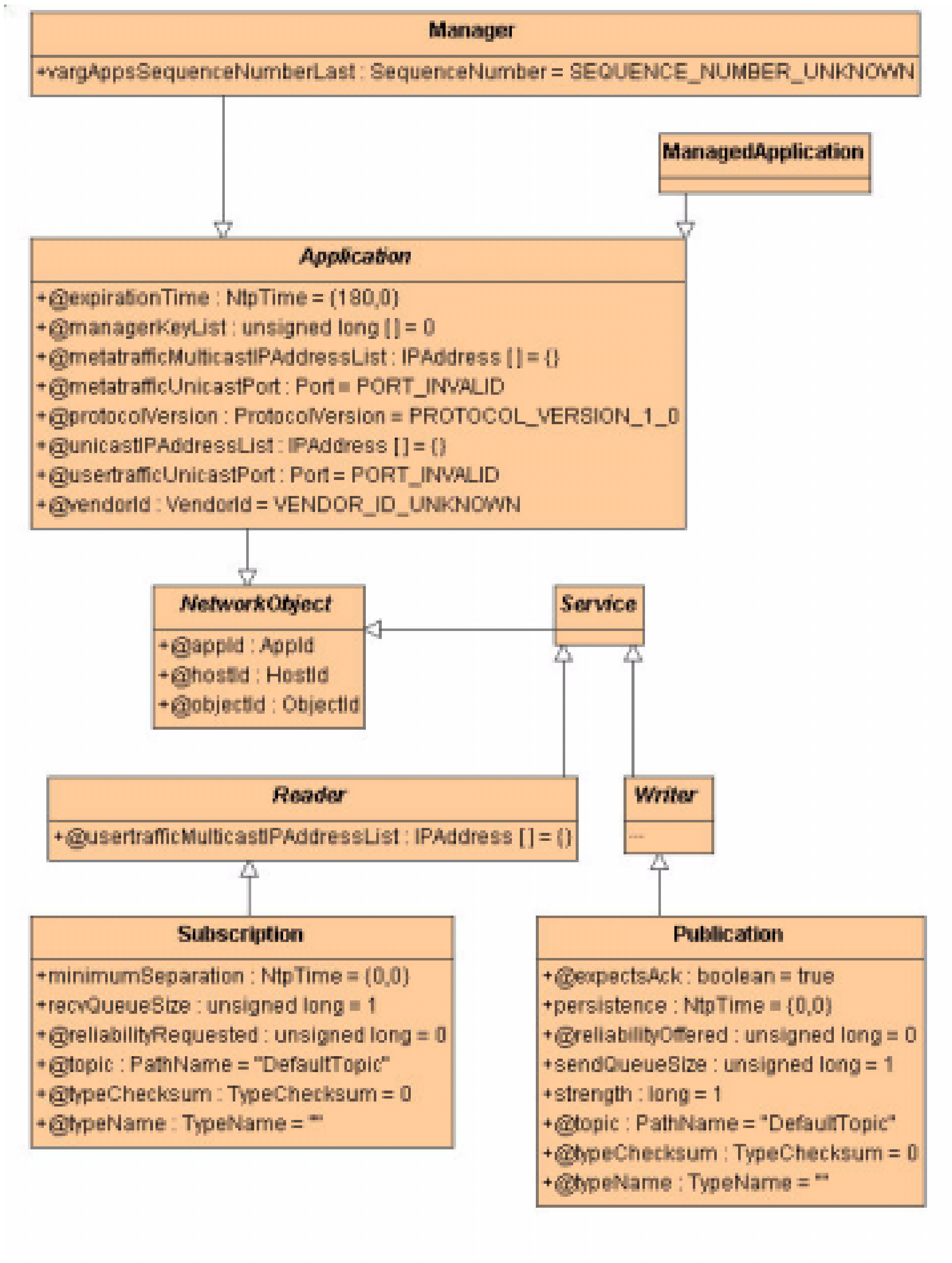
Table 5.1 **ManagedApplication Attributes**

Attributes	Type	Default
unicastIPAddressList [ ]	IPAddress	{ }
@protocolVersion	ProtocolVersion	PROTOCOL_VERSION_1_0
@vendorId	VendorId	VENDOR_ID_UNKNOWN
@expirationTime	NtpTime	{180, 0}
@managerKeyList	unsigned long	0
@metatrafficMulticastIPAddressList [ ]	IPAddress	{ }
@metatrafficUnicastPort	Port	PORT_INVALID
@usertrafficUnicastPort	Port	PORT_INVALID

Table 5.2 **Manager Submessage attributes (in addition to Table 5.1)**

Attributes	Type	Default
vargAppsSequenceNumberLast	SequenceNumber	SEQUENCE_NUMBER_UNKNOWN

Figure 5.1 **Object Attributes**


Manager
+vargAppsSequenceNumberLast : SequenceNumber = SEQUENCE_NUMBER_UNKNOWN
ManagedApplication
Application
+@expirationTime : NtpTime = {180,0}
+@managerKeyList : unsigned long [] = 0
+@metatrafficMulticastIPAddressList : IPAddress [] = {}
+@metatrafficUnicastPort : Port = PORT_INVALID
+@protocolVersion : ProtocolVersion = PROTOCOL_VERSION_1_0
+@unicastIPAddressList : IPAddress [] = {}
+@usertrafficUnicastPort : Port = PORT_INVALID
+@vendorId : VendorId = VENDOR_ID_UNKNOWN
NetworkObject
+@appId : AppId
+@hostId : HostId
+@objectId : ObjectId
Service
Reader
+@usertrafficMulticastIPAddressList : IPAddress [] = {}
Writer
Subscription
+minimumSeparation : NtpTime = {0,0}
+recvQueueSize : unsigned long = 1
+@reliabilityRequested : unsigned long = 0
+@topic : PathName = "DefaultTopic"
+@typeChecksum : TypeChecksum = 0
+@typeName : TypeName = ""
Publication
+@expectsAck : boolean = true
+persistence : NtpTime = {0,0}
+@reliabilityOffered : unsigned long = 0
+sendQueueSize : unsigned long = 1
+strength : long = 1
+@topic : PathName = "DefaultTopic"
+@typeChecksum : TypeChecksum = 0
+@typeName : TypeName = ""

The next two tables represent the **Publication** and **Subscription** attributes, respectively.

Table 5.3 **Publication attributes**

Attributes	Type	Default
@topic	PathName	“DefaultTopic”
@typeName	TypeName	“”
@typeChecksum	TypeChecksum	0
strength	long	1
persistence	NtpTime	{0, 0}

Table 5.3 **Publication attributes**

Attributes	Type	Default
@expectsAck	boolean	true
sendQueueSize	unsigned long	1
@reliabilityOffered	unsigned long	0

5.2 Wire Format of the ParameterSequence

A **ParameterSequence** is a sequence of **Parameters**, terminated with a sentinel. Each **Parameter** starts aligned on a 4-byte boundary with respect to the start of the **ParameterSequence**. The representation of each parameter starts with a **ParameterId** (identifying the parameter), followed by a **ParameterLength** (the number of octets from the first octet of the value to the ID of the next parameter), followed by the value of the parameter itself.

When an attribute is a list (indicated by the "**[ ]**" after the type-name in the object model), the elements of the array are represented in the parameter sequence by listing the individual elements with the same (repeated) parameter ID.

```
ParameterSequence
....2..........8...............16.............24...............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ParameterId id_1             | ParameterLength length_1      |
+---------------+---------------+---------------+---------------+
|                                                               |
~                             value_1                           ~
|                                                               |
+---------------+---------------+---------------+---------------+
| ParameterId id_2             | ParameterLength length_2      |
+---------------+---------------+---------------+---------------+
|                                                               |
~                             value_2                           ~
|                                                               |
~                              .....                            ~
|                                                               |
+---------------+---------------+---------------+---------------+
|       PID_SENTINEL           |             ignored           |
+---------------+---------------+---------------+---------------+
```

ParameterId and **ParameterLength** are unsigned shorts:

```
typedef unsigned short ParameterId;
typedef unsigned short ParameterLength;
```

The parameter length is the number of octets following the length of the parameter to reach the ID of the next parameter (or the ID of the sentinel). Because every **ParameterId** starts on a 4-byte boundary, the **ParameterLength** is always a multiple of four.

5.3 ParameterID Definitions

Table 5.4 **ParameterID Values**

ID	Name	Used For Fields
0x0000	PID_PAD	-
0x0001	PID_SENTINEL	-
0x0002	PID_EXPIRATION_TIME	Application::expirationTime : NtpTime
0x0003	PID_PERSISTENCE	Publication::persistence : NtpTime
0x0004	PID_MINIMUM_SEPARATION	Subscription::minimumSeparation : NtpTime
0x0005	PID_TOPIC	Publication::topic : PathName, Subscription::topic : PathName
0x0006	PID_STRENGTH	Publication::strength : long
0x0007	PID_TYPE_NAME	Publication::typeName : TypeName, Subscription::typeName : TypeName
0x0008	PID_TYPE_CHECKSUM	Publication::typeChecksum : TypeChecksum, Subscription::typeChecksum : TypeChecksum
0x0009	RTPS_PID_TYPE2_NAME	
0x000a	RTPS_PID_TYPE2_CHECKSUM	
0x000b	PID_METATRAFFIC_ MULTICAST_IPADDRESS	Application::metatrafficMulticastIPAddressList: IPAddress[]
0x000c	PID_APP_IPADDRESS	Application::unicastIPAddressList : IPAddress[]
0x000d	PID_METATRAFFIC_ UNICAST_PORT	Application::metatrafficUnicastPort : Port
0x000e	PID_USERDATA_ UNICAST_PORT	Application::userdataUnicastPort :Port
0x0010	PID_EXPECTS_ACK	Publication::expectsAck : boolean
0x0011	PID_USERDATA_ MULTICAST_IPADDRESS	Reader::userdataMulticastIPAddressList : IPAddress[]
0x0012	PID_MANAGER_KEY	Application::managerKeyList : unsigned long []
0x0013	PID_SEND_QUEUE_SIZE	Publication::sendQueueSize : unsigned long
0x0015	PID_PROTOCOL_VERSION	Application::protocolVersion : ProtocolVersion
0x0016	PID_VENDOR_ID	Application::vendorId : VendorId
0x0017	PID_VARGAPPS_SEQUENCE_ NUMBER_LAST	Manager::vargAppsSequenceNumberLast : SequenceNumber
0x0018	PID_RECV_QUEUE_SIZE	Subscription::recvQueueSize : unsigned long
0x0019	PID_RELIABILITY_ OFFERED	Publication::reliabilityOffered : unsigned long
0x001a	PID_RELIABILITY_ REQUESTED	Subscription::reliabilityRequested : unsigned long

Future minor versions of the protocol can add new parameters up to a maximum parameter ID of 0x7fff. The range 0x8000 to 0xffff is reserved for vendor-specific options and will not be used by any future versions of the protocol.

5.4 Reserved Objects

5.4.1 Description

To ensure the automatic discovery of **Applications** and **Services** in a **Network**, every **Manager** and every **ManagedApplication** contains a number of special built-in **NetworkObjects**, which have reserved objectId's.

These special objects fall into these categories:

- ❏ The **Application** itself is a **NetworkObject** with a special GUID (the instance of the **Application** is called *applicationSelf*). In addition, every **Application** has a **CSTWriter** (*writerApplicationSelf*) that disseminates the attributes of the local **Application** on the **Network**.
- ❏ Several objects are dedicated to the discovery of **Managers** and **ManagedApplications** on the **Network**. Every **ManagedApplication** has the **CSTReaders** *readerApplications* and *readerManagers*, through which the existence and attributes of the remote **ManagedApplications** and remote **Managers**, respectively, are obtained. Every **Manager** has the corresponding **CSTWriters** *writeApplications* and *writeManagers*.
- ❏ As seen in Figure 5.2, every **ManagedApplication** has, among others, two instances of a **CSTReader** (*readerPublications* and *readerSubscriptions*) and two instances of a **CSTWriter** (*writerPublications* and *writerSubscriptions*). Through the **CSTReaders**, the **ManagedApplication** can receive information about the existence and attributes of all the remote **Publications** and **Subscriptions** in the **Network**. Through the **CSTWriters**, the **ManagedApplication** can send out information about its local **Publications** and **Subscriptions**.

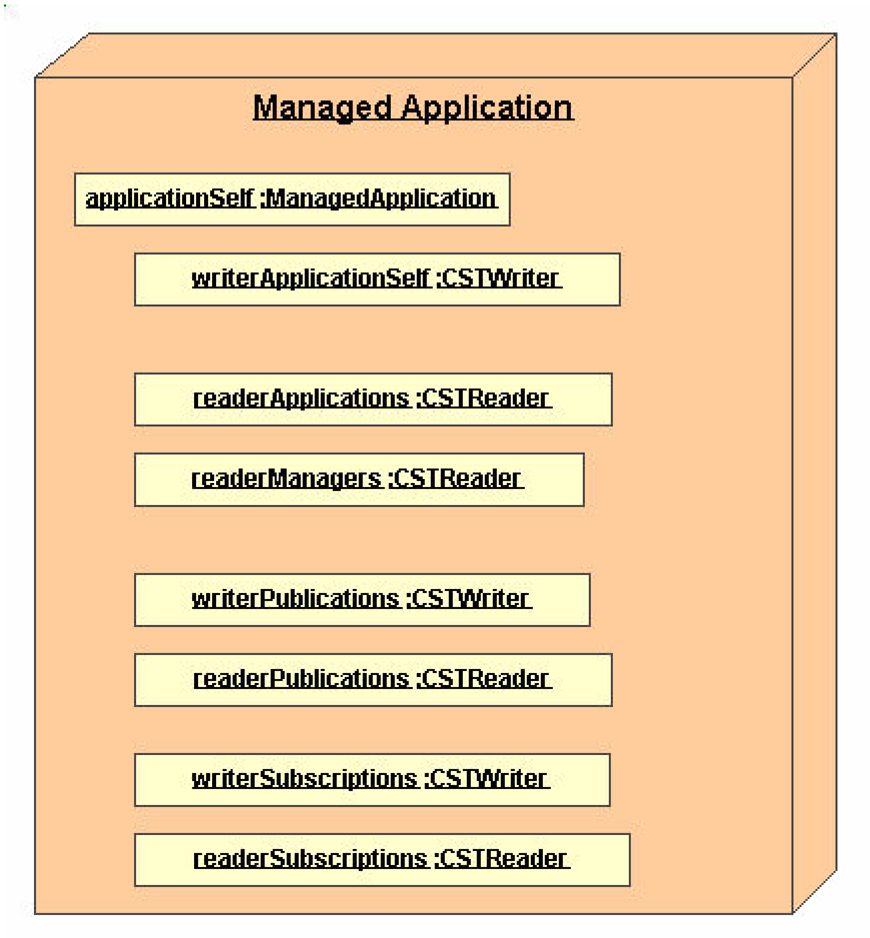
Future versions of the protocol may add additional special objects and expand the list of reserved objectId's within the same major version number.

Chapter 8 describes in detail what Messages are exchanged between these special objects.

5.4.2 Overview: Special Objects in a ManagedApplication

Every **ManagedApplication** contains the following special objects seen in Figure 5.2.

Figure 5.2 **Special Objects of a ManagedApplication**


Managed Application
applicationSelf :ManagedApplication
writerApplicationSelf :CSTWriter
readerApplications :CSTReader
readerManagers :CSTReader
writerPublications :CSTWriter
readerPublications :CSTReader
writerSubscriptions :CSTWriter
readerSubscriptions :CSTReader

applicationSelf :ManagedApplication The attributes of the **ManagedApplication** itself.

writerApplicationSelf :CSTWriter A **Writer** that makes the attributes of the application itself available.

readerApplications :CSTReader The **Reader** through which the application receives the attributes of other **Applications** on the **Network**.

readerManagers :CSTReader The **Reader** through which the application receives the attributes of **Managers** on the **Network**.

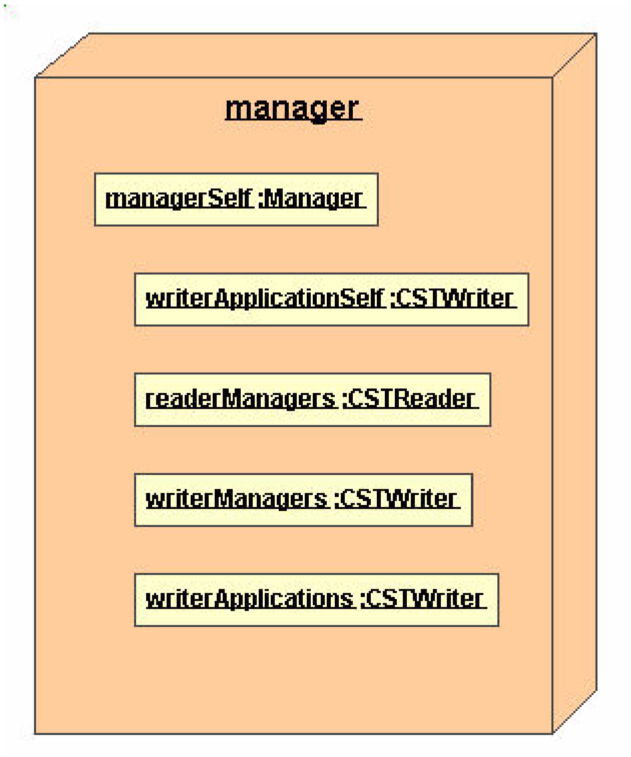
readerPublications :CSTReader The **Reader** through which the application receives information about remote **Publications** that exist on the **Network**.

writerPublications :CSTWriter The **Writer** that makes the attributes of the local **Publications** (contained in the local application) available on the **Network**.

readerSubscriptions:CSTReader The **Reader** through which the application receives information about remote **Subscriptions** that exist on the **Network**.

writerSubscriptions :CSTWriter The **Writer** that makes the attributes of the local **Subscriptions** (contained in the local application) available on the **Network**.

Figure 5.3 **Special Objects of a Manager**


manager
managerSelf :Manager
writerApplicationSelf :CSTWriter
readerManagers :CSTReader
writerManagers :CSTWriter
writerApplications :CSTWriter

5.4.3 Overview: Special Objects in a Manager

Every **Manager** contains the following special objects seen in Figure 5.3.

managerSelf :Manager The attributes of the **Manager** itself.

writerApplicationSelf :CSTWriter A **Writer** that makes the attributes of the application itself available.

readerManagers :CSTReader The **Reader** through which the **Manager** discovers the other **Managers** on the **Network**.

writerManagers :CSTWriter The **Writer** through which a **Manager** provides information on all the other **Managers** in the **Network** to its managees.

writerApplications :CSTWriter The **Writer** through which a **Manager** provides information on all its managees.

5.4.4 Reserved ObjectIds

Table 5.5 lists the current reserved *objectIds*. All these objects are also meta-objects; so the M-bit and R-bit are set in the objectId. The meaning of these objects cannot change in this major version (1) of the protocol but future minor versions may add additional reserved *objectId*'s.

Table 5.5 **Predefined instanceIds**

Predefined instanceId	objectId of this Built-in Object	Description
applicationSelf	(OID_APP) = {00,00,01,c1}	The Application (ManagedApplication or Manager) itself.
writerApplicationSelf	(OID_WRITE_APPSELF) = {00,00,08,c2}	The CSTWriter which makes the attributes of the local Application available on the Network. Every Application has one of these.
writerApplications	(OID_WRITE_APP) = {00,00,01,c2}	Every Manager has this CSTWriter, to make the attributes of the ManagedApplications that are its managees available on the Network.
readerApplications	(OID_READ_APP) = {00,00,01,c7}	Every Manager has such a CSTReader, through which it reads the managees from Managers.
writerManagers	(OID_WRITE_MGR) = {00,00,07,c2}	Every Manager has this CSTWriter containing the other Managers.
readerManagers	(OID_READ_MGR) = {00,00,07,c7}	CSTReader through which an Application obtains information about the attributes of the Managers on the Network.
writerPublications	(OID_WRITE_PUBL) = {00,00,03,c2}	Every ManagedApplication makes its local Publications available through this CSTWriter.
readerPublications	(OID_READ_PUBL) = {00,00,03,c7}	This CSTReader reads the attributes of remote Publications. It is present in every ManagedApplication.
writerSubscriptions	(OID_WRITE_SUBS) = {00,00,04,c2}	Every ManagedApplication makes its local Subscriptions available through this CSTWriter.
readerSubscriptions	(OID_READ_SUBS) = {00,00,04,c7}	This CSTReader reads the attributes of remote Subscriptions. It is present in every ManagedApplication.

5.5 Examples

5.5.1 Examples of GUIDs

Table 5.6 shows some examples of GUIDs and their interpretations.

Table 5.6 **Interpretation of Sample GUIDs**

<hostId,	appId,	objectId>	Interpretation
{aa,bb,cc,dd}	{11,22,33,01}	{00,00,07,03}	A user-level object of class Publication.
	{11,22,33,02}	{00,00,07,c2}	A meta-CSTWriter that resides on a Manager; the object has a special instanceId: it is the CSTWriter of all Managers for which the Manager keeps information.
	{11,22,33,01}	{00,00,17,c2}	This is a special instanceId; the object is a meta-level CSTWriter, however, version 1.0 does not define this special instanceId (a higher-level minor-version might define it). An implementation of version 1.0 should classify this GUID as UNKNOWN.
	{11,22,33,01}	{ee,ee,ee,02}	A user-level CSTWriter in an Application.
	{11,22,33,01}	{dd,dd,dd,82}	A meta-level CSTWriter in an Application.
	{11,22,33,01}	{00,00,01,c1}	A special meta-object of kind Application: the special instanceId "000001c1" is defined to refer to the application itself, <{aa,bb,cc,dd},{11,22,33,01}>.
	{11,22,33,02}	{00,00,01,c1}	The same objectId as the previous example; the only difference is that the receiver knows from the appId that it is dealing with a special application, a Manager.
	{11,22,33,17}	{00,00,01,c1}	Should be classified as UNKNOWN, because the kind of application ("17") is unknown.
	{11,22,33,01}	{00,00,01,40}	Should be classified as UNKNOWN because the kind of objectId is unknown.
{00,00,00,00}	{11,22,33,01}	{00,00,01,c1}	Should be classified as UNKNOWN because the hostId is unknown.

5.5.2 Examples of ParameterSequences

Suppose an application receives a **VAR** submessage for an object with GUID <{11,22,33,44},{55,66,77,02},{00,00,01,c1}>. This GUID indicates this is a **Manager** (the kind of the appId is 0x02).

Suppose the parameter list in the **VAR** submessage contains a parameter sequence with the contents listed in Table 5.7. This means that the application knows that the remote **Manager** object with GUID

Table 5.7 **Example VAR Submessage**

Parameter ID	Value
PID_EXPIRATION_TIME	{10,0}
PID_APP_IPADDRESS	206.197.67.102
PID_APP_IPADDRESS	206.167.12.12
PID_METATRAFFIC_UNICAST_PORT	1051
PID_USERDATA_UNICAST_PORT	1052
PID_TOPIC	"abc"
0x00a0	123456
0x9001	abcdef

<{11,22,33,44},{55,66,77,02},{00,00,01,c1}> has the attributes listed in Table 5.8.

Note that the application uses default values for those attributes for which it has not explicitly received information.

Table 5.8 **Example Manager Attributes**

Attribute	Contents
expirationTime	{10,0}
managerKey	0
metatrafficMulticastIPAddressList	{}
metatrafficUnicastPort	1051
usertrafficUnicastPort	1052
protocolVersion	PROTOCOL_VERSION_1_0
unicastIPAddressList	{ 206.197.67.102, 206.167.12.12}
vendorId	VENDORID_UNKNOWN
vargAppsSequenceNumberLast	SEQUENCE_NUMBER_UNKNOWN

The receiving application ignores the last three parameters in the parameter sequence of Table 5.7:

❑ The parameter **PID_TOPIC** is a known parameter; but in version 1.0 of the protocol, it does not change a known attribute of a **Manager**; this parameter should be ignored. This is not an error.

❑ The parameter with ID 0x00a0 is an unknown parameter that might have been added in a higher minor version of the protocol; this parameter should be ignored. This is not an error.

❑ The parameter with ID 0x9001 is a vendor-specific parameter: if the application does not know about this vendor-specific extension, this parameter should be ignored. This is not an error.

Chapter 6

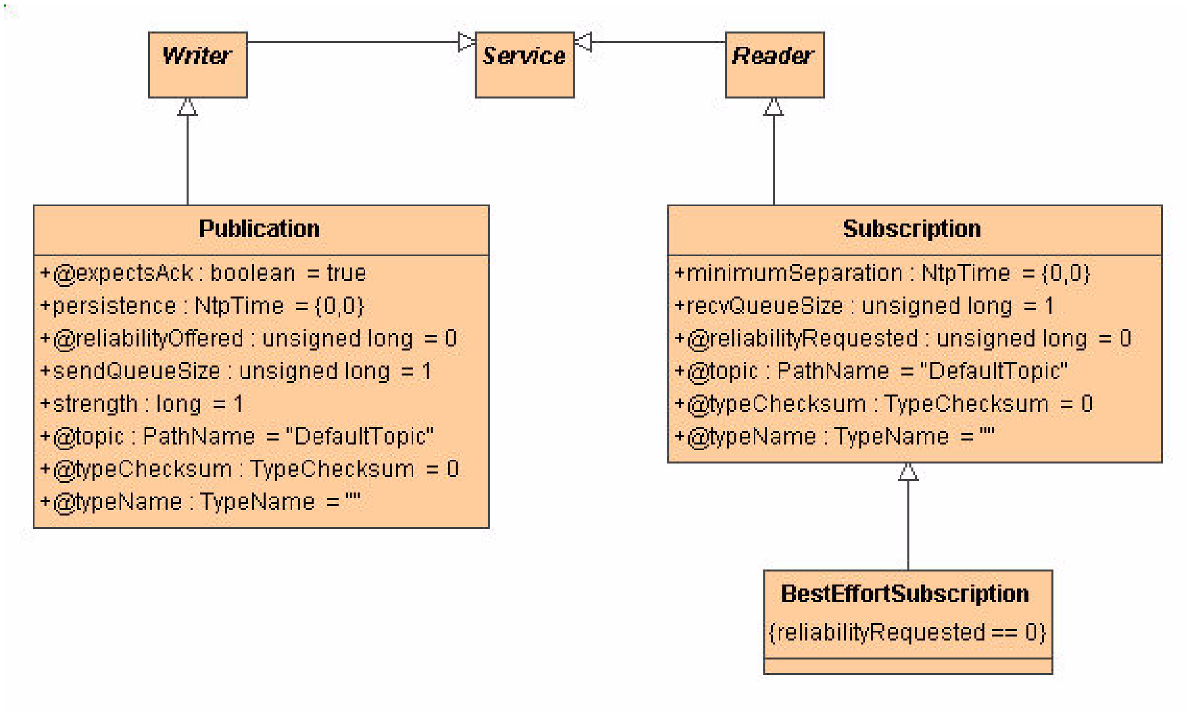
Publish-Subscribe Protocol

This chapter describes the Publish-Subscribe Protocol, which sends issues containing **UserData** from **Publications** to **Subscriptions**. The chapter separately describes the protocols for the case of best-effort publish-subscribe and reliable publish-subscribe and shows the representation of **UserData** and the related type-checking.

6.1 Publication and Subscription Objects

6.1.1 Object Model

The following figure shows the relevant aspects of the RTPS object model. This section only describes the simple case of best-effort **Subscriptions** (the **Subscription** attribute *reliabilityRequested* is 0)


Writer
Service
Reader
Publication
+@expectsAck : boolean = true
+persistence : NtpTime = {0,0}
+@reliabilityOffered : unsigned long = 0
+sendQueueSize : unsigned long = 1
+strength : long = 1
+@topic : PathName = "DefaultTopic"
+@typeChecksum : TypeChecksum = 0
+@typeName : TypeName = ""
Subscription
+minimumSeparation : NtpTime = {0,0}
+recvQueueSize : unsigned long = 1
+@reliabilityRequested : unsigned long = 0
+@topic : PathName = "DefaultTopic"
+@typeChecksum : TypeChecksum = 0
+@typeName : TypeName = ""
BestEffortSubscription
{reliabilityRequested == 0}

6.1.1.1 Topic And Type Properties

Every **Publication** and **Subscription** has the following three properties:

topic The name of the information in the **Network** that is published or subscribed to.

typeName The name of the type of this data.

typeChecksum A checksum that identifies the CDR-representation of the data.

The types and meaning of these attributes is described in detail in Section 6.2.

A **Publication** and **Subscription** "match" when the following conditions are satisfied:

- o They have the same value for the attribute *topic*.
- o They have the same value for the attribute *typeName* or this string is the empty string for one of the two objects.
- o They have the same value for the attribute *typeChecksum* or this number is 0 for one of the two objects.

6.1.1.2 Subscription Properties: minimumSeparation

The *minimumSeparation* is the minimum time between two consecutive issues received by the **Subscription**. It defines the maximum rate at which the **Subscription** is prepared to receive issues. **Publications** sending to this **Subscription** should try to send issues so that they are spaced at least this far apart.

6.1.1.3 Publication Properties: strength, persistence

The *strength* is the precedence of the issue sent by the **Publication**; the *persistence* indicates how long the issue is valid. *Strength* and *persistence* allow the receiver to arbitrate if issues are received from several matching publications.

6.1.1.4 Reliability

Publications can offer multiple reliability policies ranging from best-efforts to strict (blocking) reliability. Subscription can request multiple policies of desired reliability and specify the relative precedence of each policy. Publications will automatically select among the highest precedence requested policy that is offered by the publication.

The reliability policies offered by the publication are part of the publication declaration and are listed with using the parameter PID_PUBL_RELIABILITY_OFFERED. The reliability policies requested by the subscription are part of the subscription declaration and are listed with using the parameter PID_SUBS_RELIABILITY_REQUESTED.

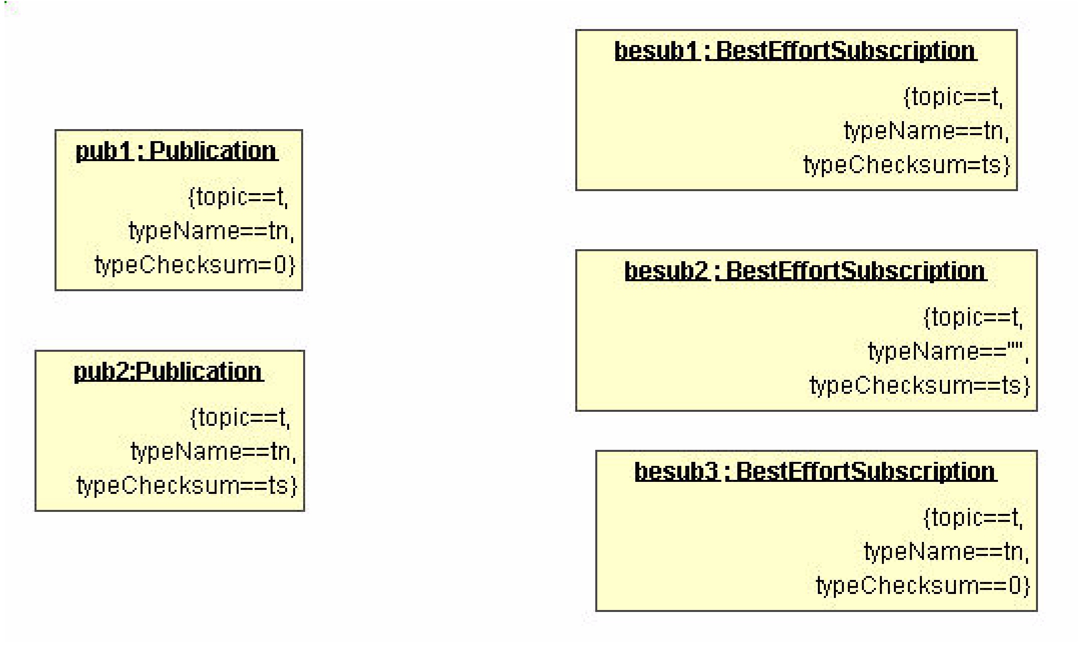
The relative order of each PID_SUBS_RELIABILITY_REQUESTED in the subscription declaration indicates relative precedence. The policies are ordered in decreasing order of precedence, that is, starting with the highest precedence requested policy.

Version 1.0 of the RTPS protocol defines two reliability policies: best-efforts and strict.

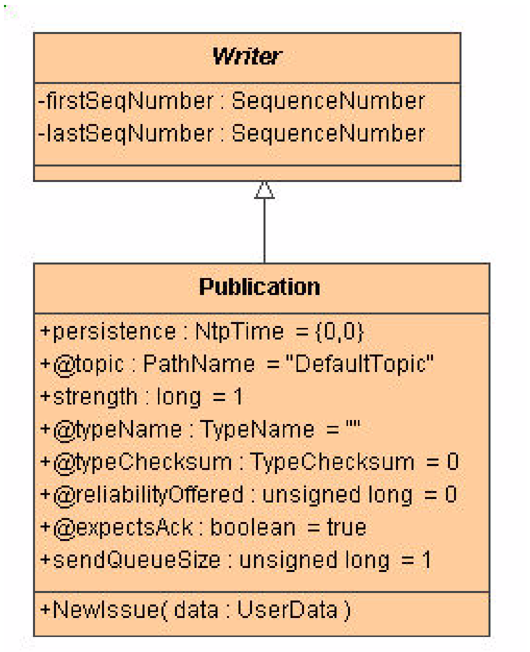
Value	Name
0	PID_VALUE_RELIABILITY_BEST_EFFORTS
1	PID_VALUE_RELIABILITY_STRICT

6.1.1.5 Deployment

The following figure shows a possible deployed system of **Publications** and **Subscriptions**: for the following description, only matching objects matter. In RTPS, there can be multiple matching **Publications** and **Subscriptions** on the **Network**.


besub1 : BestEffortSubscription
{topic==t, typeName==tn, typeChecksum=ts}
pub1 : Publication
{topic==t, typeName==tn, typeChecksum=0}
besub2 : BestEffortSubscription
{topic==t, typeName=="", typeChecksum==ts}
pub2:Publication
{topic==t, typeName==tn, typeChecksum==ts}
besub3 : BestEffortSubscription
{topic==t, typeName==tn, typeChecksum==0}

6.1.2 Publication Behavior Towards Best-Effort Subscriptions


Writer
-firstSeqNumber : SequenceNumber
-lastSeqNumber : SequenceNumber
Publication
+persistence : NtpTime = {0,0}
+@topic : PathName = "DefaultTopic"
+strength : long = 1
+@typeName : TypeName = ""
+@typeChecksum : TypeChecksum = 0
+@reliabilityOffered : unsigned long = 0
+@expectsAck : boolean = true
+sendQueueSize : unsigned long = 1
+NewIssue(data : UserData)

The **Publication** is given user data by the application (represented by the method **NewIssue()**, which gives the **UserData** to the **Publication**). The **Publication** maintains a queue called the **sendQueue** with space for **sendQueueSize** issues. Every time a new issue is given to the **Publication**, it places it in the **sendQueue** and increases the *lastModificationSeqNumber*.

The **Publication** sends this **UserData** to all the *matching* **Subscriptions** on the **Network** using the **ISSUE** submessage.

6.1.2.1 Contents of the Publication Message

A **Publication** puts the information from Table 6.1 in the **ISSUE** submessage.

Table 6.1 **ISSUE generated by an RTPSPublication Publication**

Field in ISSUE Submessage	Contents
subscriptionGUID	< HOSTID_UNKNOWN, APPID_UNKNOWN, OBJECTID_UNKNOWN> (the issues sent by a best-effort publication will be usable by all interested subscriptions)
publicationGUID	< pub->hostId, pub->appId, pub->objectId >
issueSeqNumber	pub->lastModificationSequenceNumber
(parameters)	NONE
(timestamp)	optional timestamp of the issue
issueData	user data

6.1.2.2 When to Send an Issue

The publication needs to try to minimize latency while also trying to respect the *minimumSeparation* of the subscriptions.

6.1.2.3 Best-Effort Subscriptions

A best-effort subscription is a completely passive element that receives **Messages** containing **ISSUE**s from matching publications; it does not send messages itself.

6.1.3 Publication Behavior Towards Strict-Reliable Subscriptions

The **Publication** is given user data by the application (represented by the method **NewIssue()**, which gives the **UserData** to the **Publication**).

The **Publication** maintains a queue called the **sendQueue** with space for **sendQueueSize** issues. Every time a new issue is given to the **Publication**, it attempts to place it in the **sendQueue**. The attempt will succeed if either the queue has space available, or else there are some issues that can be removed from the queue. Otherwise the attempt will fail.

If the attempt succeeds, the *lastModificationSeqNumber* is increased, and the issue is associated with that sequence number.

If the attempt fails the **Publication** will block until it is possible to remove at least one issue from the queue.

The **Publication** keeps track of all the *matching* strict-reliable **Subscriptions** on the **Network. T**he **Publication** keeps track of the issues (identified by the associated sequenceNumber) that have been acknowledged by each strict-reliable **Subscription**.

Issues can only be removed from the **sendQueue** if they have been acknowledged by all **Active** strict-reliable **Subscriptions** on the **Network.**

A strict-reliable **Subscriptions** is **Active** if and only if the Publication receives timely ACK messages from it in response to the **HEARTBEAT** messages it sends. The actual timing of **HEARBEAT** messages sent and the elapsed time required to declare a **Subscription** non-**Active** is middleware dependent. It is expected that the implementation will allow the application developer to tune the behavior to the specific timing and reliability requirements of the application.

The **Publication** sends this **UserData** to all the *matching* **Subscriptions** on the **Network** using the **ISSUE** submessage.

The **Publication** sends **HEARTBEAT** messages to all *matching* strict-reliable **Subscriptions** on the **Network.**

HEARTBEAT messages sent to strict-reliable **Subscriptions** that have not acknowledged all issues in the sendQueue must have the FINAL-bit unset.

HEARTBEAT messages sent to strict-reliable **Subscriptions** that have acknowledged all issues in the sendQueue can have the FINAL-bit set or unset. The decision is middleware specific.

6.1.3.1 When to Send an Issue

The publication needs to try to minimize latency while also trying to respect the *minimumSeparation* of the subscriptions.

6.1.3.2 When to Send a HEARTBEAT

The timing of **HEARTBEAT** messages is middleware dependent. However, the publication must continue sending **HEARTBEAT** messages to all **Active** strict-reliable subscriptions that have not acknowledged all issues up to and including the one with sequence number *lastModificationSeqNumber*.

6.1.3.3 Strict-Reliable Subscriptions

Strict-reliable **Subscriptions** receives **Messages** containing **ISSUE**s and **HEARTBEAT**s from matching publications and send back **ACK Messages** in response.

6.1.3.4 When to Send an ACK

Strict-reliable **Subscriptions** should only send **ACK Messages** in response to **HEARTBEAT**s.

If the **HEARTBEAT** does not have the FINAL-bit set, then the subscription must send an **ACK Message** back.

If the **HEARTBEAT** does has the FINAL-bit set, then the subscription should only send an **ACK Message** back if it has not received all issues up to **HEARTBEAT**'s lastSeqNumber.

The strict-reliable **Subscriptions** can choose to send the **ACK Messages** back immediately in response to the **HEARTBEAT**s or else it can schedule the response for a certain time in the future. It can also coalesce related responses so there need not be a one-to-one correspondence between a HEARTBEAT and an ACK response. These decisions and the timing specifics are middleware dependent.

6.1.3.5 Contents of the ACK Message

A **Subscription** puts the information from Table 6.2 in the **ACK** submessage. In this table HEARTBEAT stands for the heartbeat message that triggered the ACK as a response.

6.1.3.6 Contents of the HEARTBEAT Message

A **Publication** puts the information from Table 6.3 in the **HEARTBEAT** submessage sent to strict-reliable subscriptions.

6.2 Representation of User Data

UserData is sent in the **ISSUE** submessage from a **Publication** to one or more **Subscriptions**.

The *topic* of that data is an attribute of the **Publication** and **Subscription**. The type of this *topic* attribute is **PathName**.

Table 6.2 **ACK Sent By a Subscription in Response to a HEARTBEAT Sent By a Matching Publication**

Field in ISSUE Submessage	Contents
readerGUID	< sub->hostId, sub->appId, sub->objectId >
writerGUID	< pub->hostId, pub->appId, pub->objectId >
Bitmap	The specifics of the bitmap are middleware-dependent. However, it must meet the following constraints: 1 Bitmap.bitmapBase>= HEARTBEAT.firstSeqNum. 2. The Subscriber has received all issues up-to and including Bitmap.bitmapBase-1 3. Bits are only set to "0" if the Subscription is missing the corresponding sequence numbers.

Table 6.3 **HEARTBEAT Sent By a Publication to a Matching Strict-Reliable Subscription**

Field in ISSUE Submessage	Contents
readerGUID	This can take several forms to indicate whether the message is directed to a specific subscription or to all subscriptions. The distinction is based on whether the objectId portion is OBJECTID_UNKNOWN. If the objectId=OBJECTID_UNKNOWN then the reader GUID is: < HOSTID_UNKNOWN, APPID_UNKNOWN, OBJECTID_UNKNOWN> This indicates the heartbeat applies to all subscriptions. If the objectId!=OBJECTID_UNKNOWN then the readerGUID is: < sub->hostId, sub->appId, sub->objectId > This indicates that the heartbeat applies to one specific subscription.
writerGUID	< pub->hostId, pub->appId, pub->objectId >
firstSeqNumber	The first sequence number available to the Subscription. This sequence number must be greater or equal to (lastSeqNumber-sendQueueSize). It may not be exactly this because either not enough issues have been published to fill the sendQueue, or else some middleware-specific option causes certain issues to expire their validity.
lastSeqNumber	pub->lastModificationSequenceNumber

To ensure type-consistency between the **Publication** and **Subscription**, both have additional attributes *typeName* (of type **TypeName**) and *typeChecksum* (of type **TypeChecksum**).

The following sections describe the encapsulation of user data in CDR format in the **ISSUE**, and the meaning of the **TypeName** and **TypeChecksum** structures and the **PathName** structure that is used in the *topic*.

6.2.1 Format of Data in UserData

User data is represented on the wire in CDR format, as specified in Appendix A[ ]. The endianness used in the representation of the user data is defined by the endianness of the submessage: the E-bit present in every submessage (see Section 3.2.2). For purposes of alignment when encoding/decoding user data elements that need 8-byte alignment, the CDR stream will be reset at the start of the **UserData** block.

The RTPS protocol assumes that the sender and receiver of **UserData** know the format of the type, so that they can serialize and deserialize the data in the correct CDR format. RTPS does not define how the sender and receiver get this type information but does define optional mechanisms to check whether the types are consistent.

6.2.2 TypeName

TypeName is a string composed of up to **TYPENAME_LEN_MAX** characters.

```
#define TYPENAME_LEN_MAX 63
typedef string<TYPENAME_LEN_MAX> TypeName;
```

The RTPS protocol does *not* define the relationship between this type-name and the CDR type of the issues. The contents of the type-name can be used freely by the application level. The RTPS mechanism only checks that the *typeName* of **Publication** and **Subscription** are equal. The middleware should not allow communication if the strings are not equal in length and contents.

The default **TypeName** is the empty string (""). The empty string means that the type-name is unknown and that type-checking should not be done.

6.2.3 TypeChecksum

The *typeChecksum* is used to verify that the format of the user data is consistent. It is a 4-byte unsigned number:

```
typedef unsigned long TypeChecksum;
```

In contrast to the **TypeName**, the RTPS protocol defines the relationship between the CDR type of the data and the number in the checksum. The default checksum is the number 0, which means that the checksum has not been generated and cannot be used to check type-safety. If both the sender and receiver declare the checksum to be something other than 0, the RTPS mechanism should only allow communication if the numbers are equal. Future versions will expand on how this field is generated.

6.2.4 PathName

The **PathName** is a string with a maximum length of 255 characters:

```
#define PATHNAME_LEN_MAX 255
typedef string<PATH_LEN_MAX> PathName;
```

This is the type of the field *topic* in a **Publication** and **Subscription**.

Chapter 7

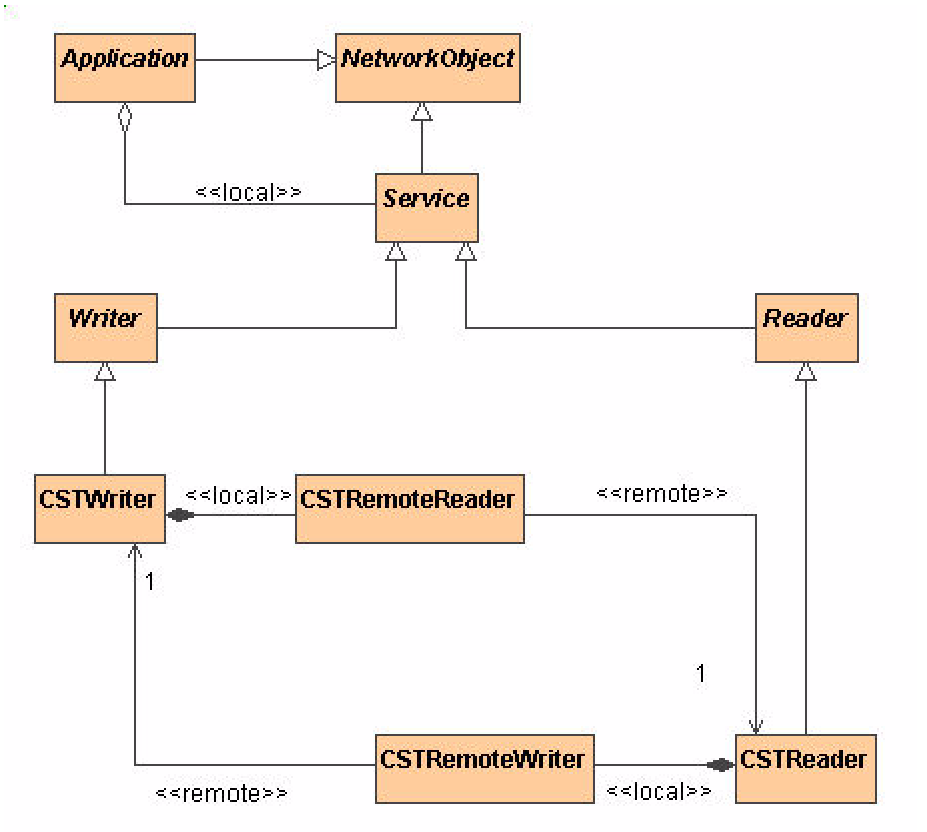
CST Protocol

The Composite State Transfer (CST) protocol transfers Composite State from **CSTWriters** to **CSTReaders**.

7.1 Object Model

Figure 7.1 shows the relevant aspects of the RTPS object model.

Figure 7.1 **CST Protocol Object Model**



The classes **CSTWriter** and **CSTReader** and their base-classes are part of the RTPS object model described in earlier chapters. To facilitate the description of the CST protocol, two classes are added: **CSTRemoteReader** and **CSTRemoteWriter**.

A **CSTWriter** locally instantiates a **CSTRemoteReader** for each remote **CSTReader** that it transfers information to. Because the CST protocol allows one **CSTWriter** to transfer data concurrently to many **CSTReaders**, the **CSTWriter** can have several local **CSTRemoteReaders**.

The complementary class on the reader's side is the **CSTRemoteWriter**. A **CSTReader** has a local **CSTRemoteWriter** for each remote **CSTWriter** it receives data from.

The **CSTRemoteWriter** and **CSTRemoteReader** are *not* **NetworkObjects**; they do not have a GUID and are therefore not remotely accessible.

7.2 Structure of the Composite State (CS)

The goal of the CST protocol is to transfer *Composite State* (**CS**) from **CSTWriters** to the interested **CSTReaders**. This **CS** is composed of the attributes of a set of **NetworkObjects**.

The initial **CS** is an empty set. This **CS** can change dynamically through the following three kinds of **CSChanges**:

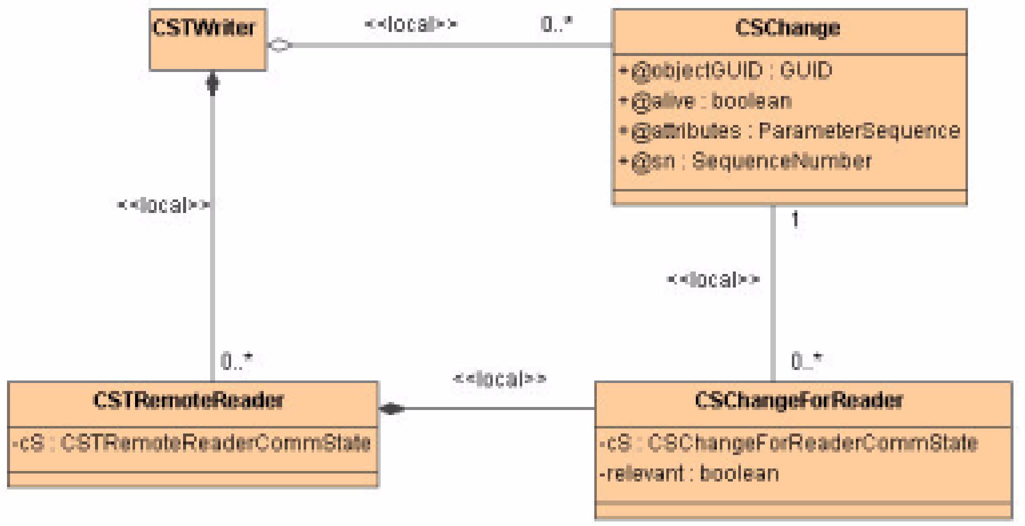
- ❑ A new **NetworkObject** (with a new unique GUID) is added to the **CS** of the **CSTWriter**.
- ❑ A **NetworkObject** is removed from the **CS** of the **CSTWriter**.
- ❑ One or more attributes of a **NetworkObject** within the **CS** change.

The goal of the CST protocol is to allow the **CSTReaders** to reconstruct the **CS** in the **CSTWriter**: the full set of **NetworkObjects** in the **CS** and their attributes. The CST protocol is aimed at transferring the *current* **CS** and avoids transferring the entire history of **CSChanges** that led to the *current* **CS**.

7.3 CSTWriter

7.3.1 Overview

The following sections describe the behavior of the **CSTWriter**, the **CSTChangeForReader** and the **CSTRemoteReader**.



7.3.2 CSTWriter Behavior

The current **CS** of the **CSTWriter** is represented by a sequence of **CSChanges**. The CSChanges are sequentially ordered by their SequenceNumber.

Every change in the **CS** of the **CSTWriter** creates a new CSChange with a new SequenceNumber. The *objectGUID* of the new **CSChange** is the GUID of the **NetworkObject** that the change in the **CS** applies to. The *attributes* of that **NetworkObject** are represented as a **ParameterSequence in the CSChange**. The *alive* boolean is set to FALSE iff the **CSChange** represents the removal of the **NetworkObject** from the set of objects in the **CS**.

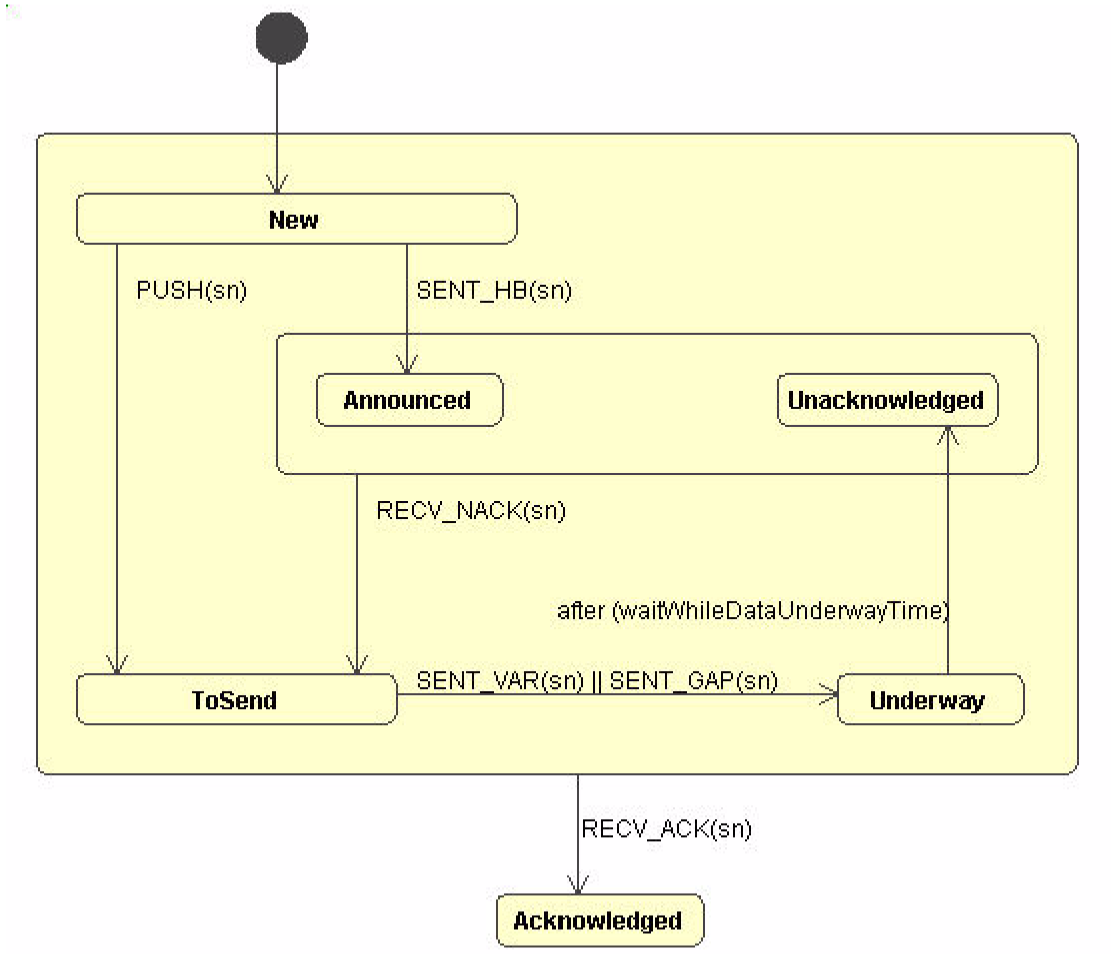
7.3.3 CSChangeForReader Behavior

The **CSTChangeForReader** keeps track of the communication state (attribute *cS*) and relevance (attribute *relevant*) of each **CSChange** with respect to a specific remote **CSTReader**.

This *relevant* boolean is set to TRUE when the **CSChangeForReader** is created; it can be set to FALSE when the **CSChange** has become irrelevant for the remote Reader because of later **CSChanges**. This can happen, for example, when an attribute of a **NetworkObject** changes several times: in that case a later **CSChange** can make a previous **CSChange** irrelevant because a **Reader** is only interested in the latest attributes of the **NetworkObject**. It is the responsibility of the **CSTRemoteReader** to use this argument correctly so that the **CSTReader** can reconstruct the correct **CS** from the *relevant* **CSChanges** it receives.

Figure 7.2 shows the Finite State Machine representing the state of the attribute *cS* of the **CSChange-ForReader**.

Figure 7.2 **State of Attribute cS for CSChangeForReader**



The states have the following meanings:

<**New**> a **CSChange** with **SequenceNumber** *sn* is available in the **CSTWriter** but this has not been announced or sent to the **CSTReader** yet.

<**Announced**> the existence of this SequenceNumber has been announced.

<**ToSend**> it is time to send either a **VAR** or **GAP** with this *sn* to the **CSTReader**.

<**Underway**> the **CSChange** has been sent but the Writer will ignore new requests for this CSChange.

<**Unacknowledged**> the CSChange should have been received by the CSTReader, but this has not been acknowledged by the CSTReader. As the message could have been lost, the **CSTReader** can request the **CSChange** to be sent again.

<**Acknowledged**> the **CSTWriter** *knows* that the **CSTReader** has received the **CSChange** with **SequenceNumber** *sn*.

The following describes the main events in this Finite State Machine. Note that this FSM just keeps track of the state of the **CSChangeForReader**; it does not imply any specific actions:

SENT_HB(sn) : The **CSTWriter** has sent a **HEARTBEAT** with firstSeqNumber <= sn <= lastSeqNumber; which means that the **CSChange** has been announced to the **CSTReader**.

RECV_NACK(sn) : The **CSTWriter** has received an **ACK** where *sn* is inside the bitmap of the **ACK** and has a bitvalue of 0.

SENT_VAR(sn) : The **CSTWriter** has sent a **VAR** for *sn*. The CSTReader will use the received VAR to adjust its local copy of the CS.

SENT_GAP(sn) : The **CSTWriter** has sent a **GAP** where *sn* is in the **GAP**'s *gapList*, which means that the *sn* is irrelevant to the **CSTReader**.

RECV_ACK(sn) : The **CSTWriter** has received an **ACK** with bitmap.bitmapBase > *sn*. This means the **CSChange** with **SequenceNumber** *sn* has been received by the **CSTReader**.

PUSH(sn) : A **CSTWriter** can push out **CSChanges** that have not been requested *explicitly* by the reader, by moving them directly from the state <**New**> to the state <**ToSend**>.

7.3.4 CSTRemoteReader Behavior

Each **CSTRemoteReader** has a communication state *cS*, which represents the current behavior of the **CSTWriter** with respect to one remote **CSTReader**. The behavior of the **CSTReader** is partly influenced by the attribute *fullAcknowledge*.

The following is an overview of the most important abbreviations used in Figure 7.3 to represent events:

RECV_ACKf : an **ACK** was received from the **CSTReader** with FINAL-bit==FALSE.

SENT_HB : a HB was sent to the CSTReader

^VAR : this is an action: send a VAR submessage

^HB : this is an action: send a HB submessage

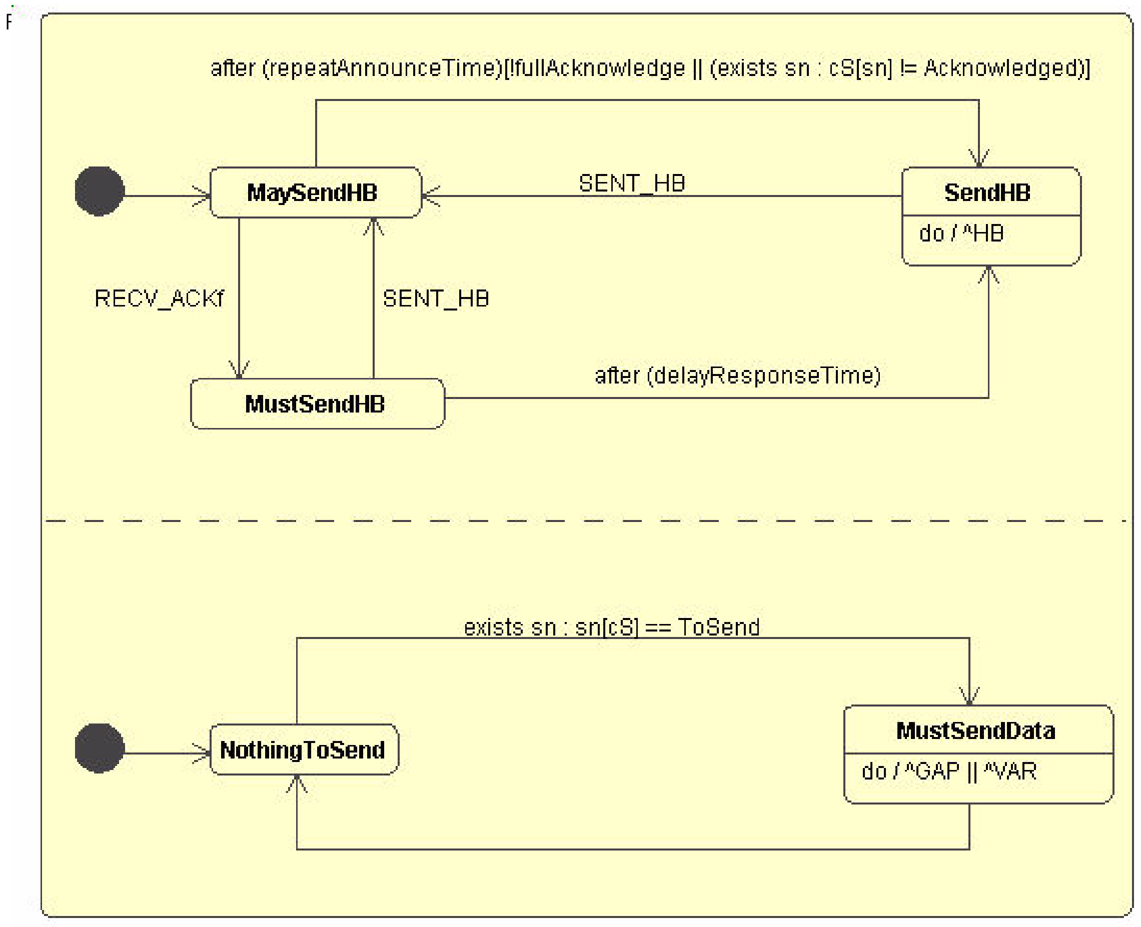
^GAP : this is an action: send a GAP submessage

The overall behavior of the CSTRemoteReader is modelled by two concurrent FSMs.

The bottom FSM deals with sending data: GAPs or VARs. Whenever there are CSChanges in state <**ToSend**>, the CSTRemoteReader is in state <**MustSendData**>. In this state, the **CSTWriter** will send **VARs** for *relevant* **CSChanges** and will include the *irrelevant* **CSChanges** in the *gapList* of a **GAP**. Section 7.5.3 and Section 7.5.4 show the contents of the **VAR** and **GAP**.

The most efficient **CSTWriter** will send the **VAR**s consecutively and in order (lowest sequence-numbers first) to facilitate the reconstruction of the **CS** by the **CSTReader**, but this is not a requirement. Likewise, the **CSTReader** will deal more efficiently with the **CSTWriter** that sends a **GAP** before **VAR** if there is a gap in the sequence numbers of the **VAR**, since the **CSTReader** then knows that sequence number is irrelevant. A possible sequence of submessages might be: GAP(1->100) VAR(101) VAR(102) GAP(103,105) VAR(104) VAR(106).

The top FSM shows the heartbeating behavior of a CSTWriter. In case an ACK without FINAL-bit is received, the CSTWriter must send a heartbeat within the delayResponseTime. In addition, a CSTWriter must regularly announce itself by sending a heartbeat. In case the CST protocol is in "fullAcknowledge" mode, the heartbeating only is necessary when there are unacknowledged CSChanges.


after (repeatAnnounceTime)[!fullAcknowledge || (exists sn : cS[sn] != Acknowledged)]
MaySendHB
SENT_HB
SendHB
do / ^HB
RECV_ACKf
SENT_HB
after (delayResponseTime)
MustSendHB
exists sn : sn[cS] == ToSend
NothingToSend
MustSendData
do / ^GAP || ^VAR

7.3.5 Timing Parameters on the CSTWriter side

The behavior is determined by the following timing parameters:

CSTWriter::waitWhileDataUnderwayTime: The CSTWriter is allowed to ignore NACKs for data that it considers to be underway to the CSTReader. The size of this window is the "waitWhileDataUnderwayTime". The window could be the **CSTWriter**'s estimate of the time it takes a message (a **VAR** or **GAP**) to be sent by the **CSTWriter** to the **CSTReader**, plus the time it takes for the **CSTReader** to process the message and immediately send a response (an **ACK**) to the **CSTWriter**, plus the time it takes the **CSTWriter** to receive and process this **ACK**. A larger *waitWhileDataUnderwayTime* will cause the CST protocol to slow down and be less aggressive; a lower time might cause data to be sent unnecessarily. *waitWhileDataUnderwayTime* can be 0 seconds.

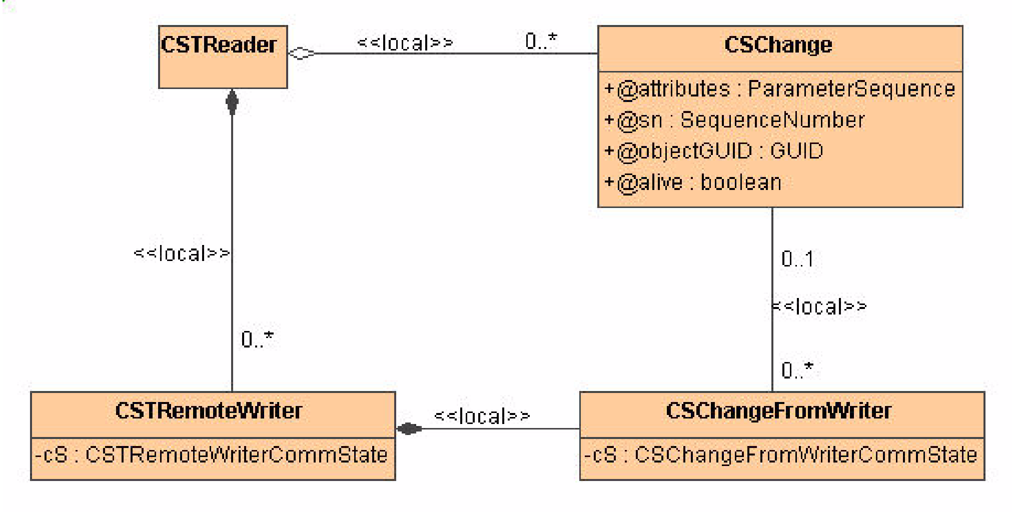
CSTWriter::repeatAnnouncePeriod: This is the period with which the **CSTWriter** will announce its existence and/or the availability of new **CSChanges** to the **CSTReader**. This period determines how quickly the protocol recovers when an announcement of data is lost. **CSTWriter::repeatAnnouncePeriod** cannot be 0 nor INFINITY for the protocol to function correctly.

CSTWriter::responseDelayTime: This is the time the **CSTWriter** waits before responding to an incoming message. Higher values allow the **CSTWriter** to combine more information in one **Message** or to service many concurrent **CSTReaders** more efficiently. **CSTWriter::delayResponseTime** can be 0 seconds.

7.4 CSTReader

7.4.1 Overview

The following sections describe the behavior of the **CSTWriter**, the **CSTChangeForReader** and the **CSTRemoteReader**. The **CSTReader** receives **CSChangeFromWriters** from the **CSTWriter**. In case a **VAR** was received for the **CSChangeFromWriters**, the **CSTReader** will store the contents of the **VAR** in an associated **CSChange**. The **CSTReader** should be able to reconstruct the current **CS** of a specific **CSTWriter** by interpreting all consecutive **CSChanges**.

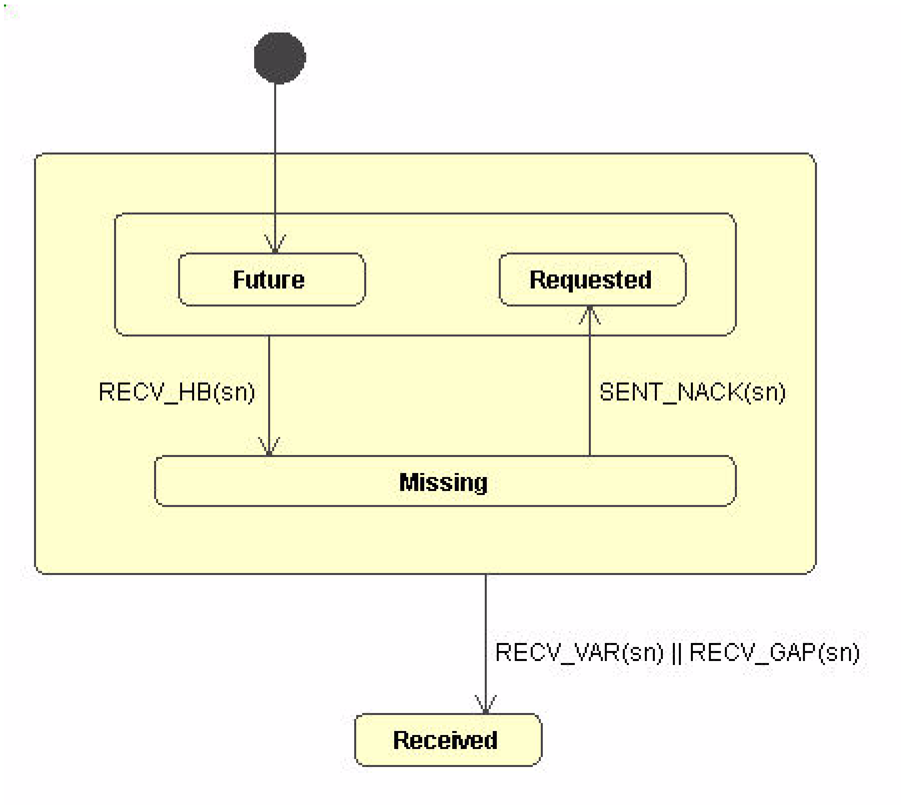


In the current version of the protocol, the **CSTReader** should reconstruct the **CS** for each **CSTRemoteWriter**. Future versions of the protocol will specify the correct interpretation in the case that several **CSTRemoteWriters** provide information on the *same* **NetworkObject**.

7.4.2 CSTReader Behavior

As in the case of the **CSTWriter**, the **CSTReader** maintains a state **CSTRemoteWriterCommState** *cS* per **CSTRemoteWriter**, as well as a state **CSChangeFromWriterCommState** *cS* for most **CSChanges** (since there may be a **CSChangeFromWriter** that has no corresponding **CSChange**, for example, a **GAP** message).

7.4.3 CSChangeFromWriter Behavior



Here is the meaning of the abbreviated events in this FSM:

RECV_HB(sn) : the **CSTReader** received a **HEARTBEAT** with firstSeqNumber <= sn <= lastSeqNumber

SENT_NACK(sn) : the **CSTReader** sent an **ACK** with sn inside the bitmap-range and with bit-value 0

RECV_GAP(sn) : the **CSTReader** received a **GAP** with sn in the gapList

RECV_VAR(sn) : the **CSTReader** received a **VAR** for sequenceNumber sn

The four states have the following meaning:

<**Future**> : A CSChange with SequenceNumber *sn* may not be used yet by the CSTWriter

<**Missing**>: The *sn* is available in the **CSTWriter** and is needed to reconstruct the **CS**.

<**Requested**>: The *sn* was requested from the **CSTWriter**, a response might be pending or underway

<**Received**> : The *sn* was received: as a **VAR** if the *sn* is relevant to reconstruct the **CS** or as a **GAP** if the sn is irrelevant.

7.4.4 CSTRemoteWriter Behavior

The abbreviations used for events are as follows:

RECV_HBf : received a **HEARTBEAT** from the **CSTWriter** with FINAL-bit==FALSE

The abbreviations used for the actions are as follows:

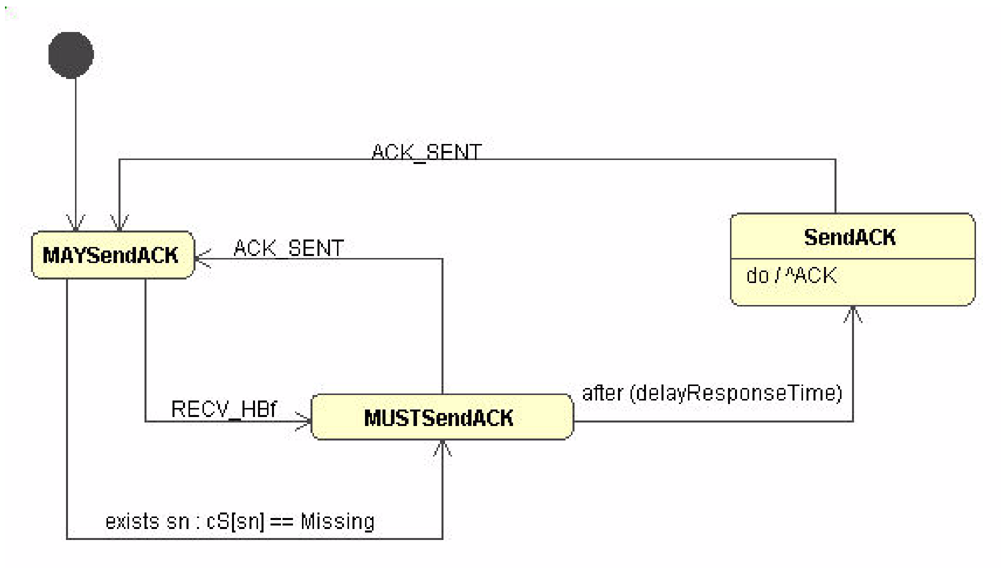
^ACK : send an **ACK** to the **CSTWriter**

In these **ACK**s, the ACK.bitmap.bitmapBase always is the *lowest* sequenceNumber whose corresponding **CSChangeFromReader** is not in state <**Received**>. This can be 0 (SEQUENCE_NUMBER_NONE). The **CSTReader** can choose the length of the bitmap as this will determine how much CSChanges move to <ToSend> state on the CSTWriter side and how much information the CTSReader will receive from the **CSTWriter**. The bitmap can only contain "0"'s when the corresponding **CSChangeFromReaders** are in state <**Missing**> (or <**Future**>). See Section 7.5.1 for a description of the further fields in these **ACK**s.

The CSTRemoteWriter must send an ACK in two cases:

1. First, when a **HEARTBEAT** with the FINAL-bit==FALSE ("RECV_HBf") is received, the **CSTReader** *must* respond with an ACK that has the FINAL-bit==TRUE. The CSTReader can delay its response.

2. Second, when the CSTReader has evidence of Missing data, it needs to request the data by sending the appropriate ACK.



7.4.5 Timing Parameters on the CSTReader side

The timing parameters of the **CSTReader** are:

CSTReader::responseDelayTime: how long the **CSTReader** waits before sending a response to a HEARTBEAT to the **CSTWriter**.

7.5 Overview of Messages used by CST

This section gives an overview of the contents of the **Messages** that a **CSTReader** and **CSTWriter** exchange and the contents of the various fields of the **Messages**.

The submessages may need to be preceded by other messages that modify the context (see Section 3.3).

7.5.1 ACKs—Sent from a CSTReader to a CSTWriter

The only logical **SubMessage** that a **CSTReader** *reader* sends to a **CSTWriter** *writer* are **ACKs**. As shown in the table above, in this version of the protocol, the *replyIPAddressPortList* must be set explicitly to all the destinations of the *reader*.

Field	Value in the CST protocol
FINAL-bit	see description of the behavior of the CSTRemoteWriter
readerGUID	reader.GUID
writerGUID	writer.GUID
replyIPAddressPortList	**required: must explicitly contain *all* destinations of the reader (reader.IPAddressPortList())**
bitmap	see description of the behavior of the CSTRemoteWriter

7.5.2 HEARTBEATs—Sent from a CSTWriter to a CSTReader

The **HEARTBEATs** sent by the **CSTWriter** *writer* to the **CSTReader** *reader* always have the contents listed in this table.

Field	Value in the CST protocol
FINAL-bit	see description of the behavior of the CSTRemoteReader
readerGUID	reader.GUID
writerGUID	writer.GUID
ACKIPAddressPortList	optional destinations of the writer
firstSeqNumber	writer.firstSeqNumber
lastSeqNumber	writer.lastSeqNumber

7.5.3 GAPs—Sent from a CSTWriter to a CSTReader

The contents of the **GAPs** sent by the **CSTWriter** *writer* to the **CSTReader** *reader* is shown in the table. The contents of the *gapList* is described in the detailed description of the behavior of the **CSTRemoteReader**.

Field	Value in the CST protocol
readerGUID	reader.GUID
writerGUID	writer.GUID
ACKIPAddressPortList	optional; additional destinations of the writer
gapList	see description of the behavior of the CSTRemoteReader

7.5.4 VARs—Sent from a CSTWriter to a CSTReader

A **VAR** encodes the contents of a specific **CSChange** *cSChange* and is sent from the **CSTWriter** *writer* to the **CSTReader** *reader.*

Field	Value in the CST protocol
readerGUID	reader.GUID
writerGUID	writer.GUID
objectGUID	cSChange.GUID
writerSeqNumber	cSChange.sn
(timestamp)	optional timestamp
(parameters)	cSChange.attributes (iff cSChange.alive==TRUE)

Field	Value in the CST protocol
ALIVE-bit	csChange.alive
ACKIPAddressPortList	optional; additional destinations of the writer

Chapter 8

Discovery with the CST Protocol

RTPS defines mechanisms that allow every **Application** to automatically discover other relevant **Applications** and their **Services** in the **Network**. These mechanisms use the CST Protocol that is described in the previous chapter.

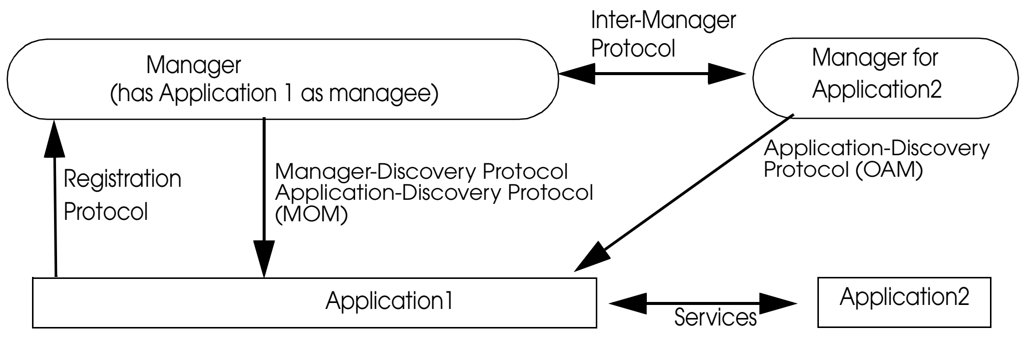
8.1 Overview

The **Manager** that manages a **ManagedApplication** is called the **Application's** MOM (My Own Manager). The other **Managers** in the **Network** are the **Application's** OAMs (Other Applications' Manager).

Figure 8.1 provides an overview of the protocols used for the discovery:

- ❑ The **Inter-Manager Protocol** allows **Managers** to discover each other in the **Network**. This protocol is described in Section 8.3.
- ❑ The **Manager-Discovery Protocol** allows every **ManagedApplication** to discover other **Managers** in the **Network**: the **ManagedApplication** receives this information from its MOM. This protocol is described in Section 8.5.
- ❑ The **Registration Protocol** allows **Managers** to find their managees and obtain their managees' state. This protocol is described in Section 8.4.
- ❑ The **Application-Discovery Protocol** allows every **ManagedApplication** to discover other **ManagedApplications** on the **Network**. This protocol is described in Section 8.6.
- ❑ The **Services-Discovery Protocol** allows every **ManagedApplication** to find out about the **Services** (the **Publications** and **Subscriptions**) in the other **ManagedApplications** on the **Network**. This protocol is described in Section 8.7.

Figure 8.1 **Relationship between Applications and Managers**


Inter-Manager Protocol
Manager (has Application 1 as managee)
Manager for Application2
Registration Protocol
Manager-Discovery Protocol Application-Discovery Protocol (MOM)
Application-Discovery Protocol (OAM)
Application1
Services
Application2

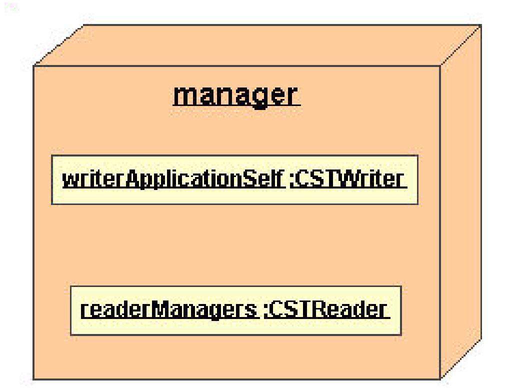
The discovery protocol uses reserved objects described in Section 5.4

8.2 Managers Keep Track of Their Managees

Every **Manager** keeps track of its managees and their attributes. To provide this information on the **Network**, every **Manager** has a special **CSTWriter** *writerApplications*.

The Composite State that the **CSTWriter** *writerApplications* provides are the attributes of all the **ManagedApplications** that the **Manager** manages (its managees).

8.3 Inter-Manager Protocol



Every **Manager** has a special **CSTWriter** *writerApplicationSelf* through which the **Manager** makes its own state available on the **Network**. The **CS** of the *writerApplicationSelf* contains the attributes of only one **NetworkObject**: the **Manager** itself.

The attribute *vargAppsSequenceNumberLast* of the **Manager** is equal to the *lastModificationSeqNumber* of the **CSTWriter** *writerApplications*. Whenever the **Manager** accepts a new **ManagedApplication** as its managee, whenever the **Manager** loses a **ManagedApplication** as a managee or whenever an attribute of a managee changes, the **CS** of the *writerApplications* changes and the **Manager**'s *vargAppsSequenceNumberLast* is updated.

Formally: for every **Manager** manager : manager.vargAppsSequenceNumberLast = manager.writerApplications.lastModificationSeqNumber.

Every **Manager** has the special **CSTReader** *readerManagers* through which the **Manager** obtains information on the state of all other **Managers** on the **Network**.

The communication between the Manager::writerApplicationSelf and Manager::readerManagers uses the CST Protocol that was described in the previous section, with a specific configuration.

The Manager::writerApplicationSelf needs to be configured with the destinations (IP-addresses) of the Manager::readerManagers on the **Network**. This configuration is necessary to bootstrap the plug-and-play mechanism of RTPS. In case multicast is used, one single multicast address is sufficient: this is the multicast-address the **Managers** will then use to discover each other on the **Network**.

To support the automatic dynamic discovery and aging of **Managers**, the Manager::writerApplicationSelf *must* announce its presence repeatedly: the value of the repeatAnnouncePeriod timing-parameter of the **Manager's** writerApplicationSelf must be small relative to the expirationTime of the **Manager**.

Similarly, the readerManagers **CSTReader** will only consider the remote **Manager** alive within the expirationTime of the Manager. If no **Message** is received from the **Manager's** *writerApplicationSelf* during the expirationTime, the remote **Manager** must be considered dead; the **CSTReader** should behave as if it received a **CSChange** with the ALIVE-bit set to FALSE.

Because the CST Protocol for the inter-management traffic relies on repetitive messages, the fullAcknowledge attribute of the CSTReader and CSTWriter must be FALSE.

Here is a summary of the inter-manager protocol:

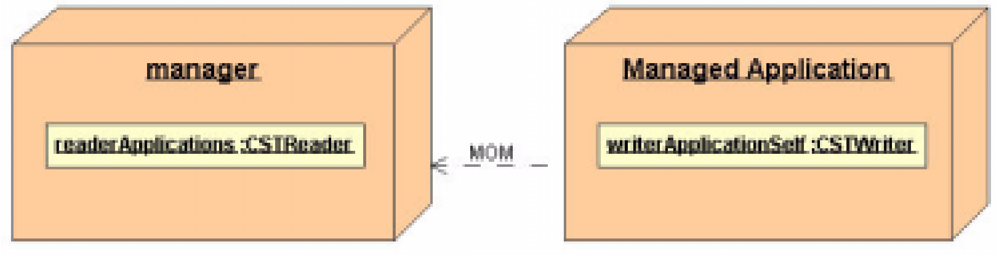
Initial Condition: New **Managers** know how to reach other *potential* managers on the **Network**.

Protocol: CST Protocol between Manager::writerApplicationSelf and Manager::readerManagers with repetition (repeatAnnouncePeriod of the writerApplicationSelf must be sufficiently high) and no acknowledgements (fullAcknowledge == FALSE).

Final Condition: Every **Manager** has the state of all other **Managers** on the **Network**. Repeated keep-alive HEARTBEATING is needed.

8.4 The Registration Protocol

The **registration protocol** enables managees to discover their **Managers** in the **Network**.


manager
readerApplications :CSTReader
MOM
Managed Application
writerApplicationSelf :CSTWriter

Initial Condition: The **ManagedApplication** is configured with a way to contact the *readerApplications* of its *potential* **Managers** (this configuration can be one single multicast address that will be used for the discovery of managers by applications). In addition, the **ManagedApplication** and **Manager** are configured with a *managerKeyList* which makes it possible for **Applications** and **Managers** to decide which **Managers** will manage which **Applications**.

Final Condition: Every **Manager** knows all its **Managees** and their attributes.

Protocol: CST Protocol (with sufficient repeatAnnouncePeriod and fullAcknowledge==FALSE) between the **ManagedApplication**'s *writerApplicationSelf* and the **Manager**'s *readerApplications*.

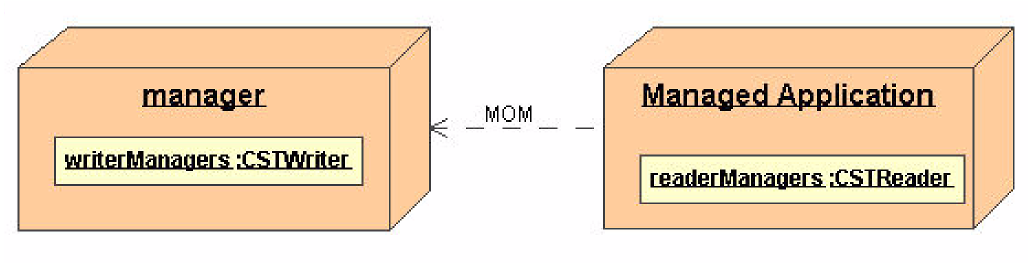
The **ManagedApplication** has a special **CSTWriter** *writerApplicationSelf*. The Composite State of the ManagedApplication::writerApplicationSelf contains only one **NetworkObject**: the application itself. As is the case for the *writerApplicationSelf* of the **Manager**, the *writerApplicationSelf* of the **ManagedApplication** must be configured to announce its presence repeatedly (the repeatAnnouncePeriod of that writer must be smaller than expirationTime of the **ManagedApplication**) and does not request nor expect acknowledgements (fullAcknowledge==FALSE).

A **Manager** that discovers a new **ManagedApplication** through its readerApplications must decide whether it must manage this **ManagedApplication** (become its MOM) or not (stay an OAM). For this purpose, the attribute *managerKeyList* of the **Application** is used: if one of the **ManagedApplication's** keys (in the attribute *managerKeyList*) is equal to one of the **Manager's** keys, the **Manager** accepts the **Application** as a managee and becomes its MOM. If none of the keys are equal, the managed application is ignored: the **Manager** will not manage this **Application** and stay an OAM for the **Application**. The *managerKey* 0x7F000001 (IP loopback) has a special meaning: the **Manager** will accept the **ManagedApplication** with key 0x7F000001 as a managee when that **ManagedApplication** runs on the same host as the **Manager**.

The application state in the **Manager** is only temporary. This approach is completely similar to the repeatAnnouncePeriod mechanism of **Managers** described in Section 8.3. The duration of the lease is based on the value of the **ManagedApplication**'s expirationTime. The repeatAnnouncePeriod of the *writerApplicationSelf* must be small enough so that the **Manager** receives regular messages from the **ManagedApplication**. If the **Manager** has not received a **Message** from the **ManagedApplication** during the expirationTime of that **ManagedApplication**, it considers the **ManagedApplication** dead and behaves as if a CSChange has been received declaring the **Application** dead.

8.5 The Manager-Discovery Protocol

With the Manager-Discovery protocol, a **Manager** will send the state of all **Managers** in the **Network** to all its managees.



Initial Condition: Every **Manager** has obtained the state of other **Managers** (using the inter-manager protocol) and knows its managees.

Protocol: CST Protocol between Manager::writerManagers and ManagedApplication::readerManagers.

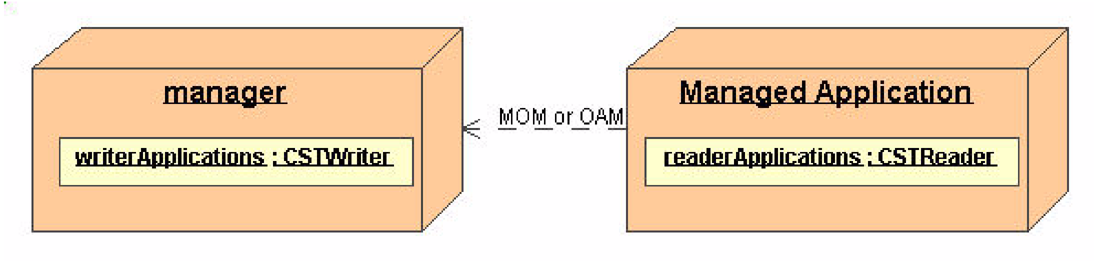
Final Condition: Every managee of every **Manager** has the state of all **Managers** on the **Network**.

8.6 The Application Discovery Protocol

Initial Condition: The **Managers** have discovered their managees and the **ManagedApplications** know all **Managers** in the **Network** (they got this information from their MOMs).

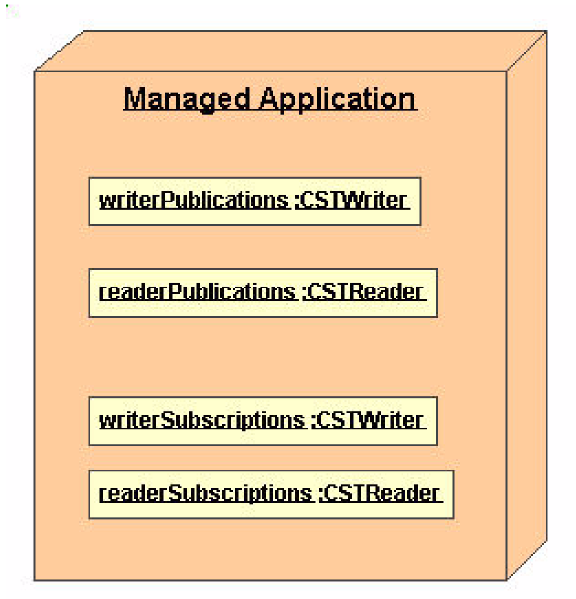
Protocol: The CST Protocol is used between the writerApplications of the **Managers** and the readerApplications of the **ManagedApplications**.

Final Condition: The **ManagedApplications** have discovered the other **ManagedApplications** in the **Network**.


manager
writerApplications : CSTWriter
MOM or OAM
Managed Application
readerApplications : CSTReader

8.7 Services Discovery Protocol

This section describes how the **ManagedApplications** transfer information to each other about their local **Services**.


Managed Application
writerPublications :CSTWriter
readerPublications :CSTReader
writerSubscriptions :CSTWriter
readerSubscriptions :CSTReader

As mentioned previously, every **ManagedApplication** has two special **CSTWriters**, *writerPublications* and *writerSubscriptions*, and two special **CSTReaders**, *readerPublications* and *readerSubscriptions*.

The Composite State that the **CSTWriters** make available on the **Network** are the attributes of all the local **Publication** and **Subscriptions**. The **CSTWriter** *writerPublications*/*Subscriptions* needs to instantiate a local **CSTRemoteReader** for each remote **ManagedApplication** on the **Network**.

Similarly, the **CSTReaders** *writerPublication/Subscription* need to instantiate a **CSTRemoteWriter** for each remote **ManagedApplication** on the **Network**.

Once **ManagedApplications** have discover each other, they use the standard CST protocol through these special **CSTReaders** and **CSTWriter** to transfer the attributes of all **Publications** and **Subscriptions** in the **Network**.

Because all **CSTRemoteReaders** and **CSTRemoteWriters** for Service-discovery are known (as a result of Application-Discovery), the CST Protocol must support the acknowledgement of received issues (fullAcknowledge==TRUE) and repeated heartbeating should be turned off (repeatAnnouncePeriod==INFINITE).

Initial Condition: The **ManagedApplications** have discovered each other on the **Network**

Protocol: CST Protocol from *writerPublications* to *readerPublications* and from *writerSubscriptions* to *readerSubscriptions* (repeatAnnouncePeriod==INFINITE and fullAcknowledge==TRUE)

Final Condition: The **ManagedApplications** know about each others **Services**.

Appendix A

CDR for RTPS

The following is a summary of the CDR data format and the OMG IDL syntax to the extent that they are used by the RTPS protocol and its description in this document.

The authoritative source of the CDR specification and OMG IDL is the CORBA protocol (available through the Object Management Group). In the CORBA V2.3.1 spec, the relevant sections are 15.3 (General Inter-ORB Protocol—CDR Transfer Syntax) and 3.10 (OMG IDL Syntax and Semantics—Type Declaration). Unless mentioned explicitly, CDR for RTPS follows the CDR standard for GIOP version 1.1.

RTPS makes some additional restrictions on CDR and makes concrete choices where CDR for GIOP 1.1 is not fully defined. Notable are the implementation of the wide characters and strings (wchar and wstring) and the definition of the RTPSIdentifier, which only allows certain characters.

A.1 Primitive Types

A.1.1 Semantics

OMG IDL-name	size	meaning
octet	1	8 uninterpreted bits
boolean	1	TRUE or FALSE
unsigned short	2	integer N, 0 <= N < 2^16
short	2	integer N, -2^15 <= N < 2^15
unsigned long	4	integer N, 0 <= N < 2^32
long	4	integer N, -2^31 <= N < 2^31
unsigned long long	8	integer N, 0 <= N < 2^64
long long	8	integer N, -2^63 <= N < 2^63
float	4	IEEE single-precision fp number
double	8	IEEE double-precision fp number
char	1	a character following ISO8859-1
wchar	2	a wide-character following UNICODE

Remarks:

- ❏ CDR defines some additional primitive types, such as "long double"; these are currently disallowed by RTPS.
- ❏ CDR leaves the width of the wchar open; RTPS gives it a fixed length of two bytes.

A.1.2 Encoding

CDR has both a big-endian ("BE") and a little-endian ("LE") encoding. The sender is allowed to choose the encoding. The receiver needs to know which encoding has been used by the sender to unpack the data correctly. This endianness-bit is transmitted as part of the RTPS protocol.

A.1.3 octet

```
BE/LE
0...2..........7
+-+-+-+-+-+-+-+-+
|7|6|5|4|3|2|1|0|
+-+-+-+-+-+-+-+-+
```

A.1.4 boolean

```
TRUE BE/LE
0...2..........7
+-+-+-+-+-+-+-+-+
|0|0|0|0|0|0|0|1|
+-+-+-+-+-+-+-+-+

FALSE BE/LE
0...2..........7
+-+-+-+-+-+-+-+-+
|0|0|0|0|0|0|0|0|
+-+-+-+-+-+-+-+-+
```

A.1.5 unsigned short

```
BE
0...2..........7..............15
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|MSB            |            LSB|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
LE
0...2..........7..............15
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            LSB|MSB            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.1.6 short

A short has the same encoding as an unsigned short, but uses 2's complement representation.

A.1.7 unsigned long

```
BE
0...2..........7..............15.............23.............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|MSB            |MSB    X       |MSB     Y      |            LSB|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

LE
0...2..........7..............15............23..............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            LSB|MSB    Y       |MSB     X      |MSB            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.1.8 long

A long has the same encoding as an unsigned long, but uses 2's complement representation.

A.1.9 unsigned long long

```
BE
0...2...........7...............15.............23..............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|MSB            |MSB    A       |MSB    B       |MSB    C       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|MSB    D       |MSB    E       |MSB    F       |            LSB|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

LE
0...2...........7...............15.............23..............31
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            LSB|MSB    F       |MSB     E      |MSB     D      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|MSB    C       |MSB    B       |MSB     A      |MSB            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.1.10 long long

A long long has the same encoding as an unsigned long long, but uses 2's complement representation.

A.1.11 float

```
BE
....2...........8...............16.............24..............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|      E1     |E|     F1      |      F2       |       F3      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

LE
....2...........8...............16.............24..............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       F3      |       F2      |E|     F1      |S|      E1     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.1.12 double

```
BE
....2...........8...............16.............24..............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|      E1     |   E2  |  F1   |      F2       |      F3       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       F4      |       F5      |      F6       |      F7       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

LE
....2...........8...............16.............24..............32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       F7      |      F6       |      F5       |      F4       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       F3      |       F2      |  E2   |   F1  |S|      E1     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A.1.13 char

A character has the same encoding as an octet.

A.1.14 wchar

A wide-character occupies two octets and follows UNICODE encoding.

A.2 Constructed Types

A.2.1 Alignment

In CDR, only the primitive types listed in Section A.1 have alignment constraints. The primitive types need to be aligned on their length. For example, a long must start on a 4-byte boundary. The boundaries are counted from the start of the CDR stream.

A.2.2 Identifiers

An identifier is a sequence of ASCII alphabetic, numeric and underscore characters. The first character must be an ASCII alphabetic character.

A.2.3 List of constructed types

RTPS supports the following subset of CDR constructed types:

struct	structure
array	fixed size array (the length is part of the type)
sequence	variable size array (the maximum length is part of the type)
enum	enumeration
string	string of 1-byte characters
wstring	string of wide characters

Note: there are some additional constructed types in CDR, such as unions and fixed-point decimal types; these are currently not supported in RTPS.

A.2.4 Struct

A structure has a name (an identifier) and an ordered sequence of elements. Each element has a name (an identifier) and a type. In OMG IDL, a structure is defined by the keyword "struct", followed by an identifier and a sequence of the elements of the structure. An example of the definition of a structure named "myStructure" in OMG IDL is:

```
struct myStructure {
    long long l;
    unsigned short s;
    myType t;
}
```

In CDR, the components of such a structure are encoded in the order of their declaration in the structure. The only alignment requirements are at the level of the primitive types.

A.2.5 Enumeration

An enumeration has a name (an identifier) and an ordered set of case-keywords which also are identifiers. In OMG IDL, an enumeration is defined by the keyword "enum", followed by an identifier and a list of identifiers in the enumeration. For example:

```
enum myEnumeration { case1, case2, case3 }
```

In CDR, enumerations are encoded as unsigned longs, where the identifiers in the enumeration are numbered from left to right, starting with 0.

A.2.6 Sequence

A sequence is a variable number of elements of the same type. Optionally, the type can specify the maximum number of elements in the sequence. OMG IDL uses the keyword "sequence". The syntax for an unbounded sequence of floats is:

```
sequence<float>
```

The syntax for a sequence of unsigned long longs with a maximum length is:

```
sequence<unsigned long long, MAX_NUMBER_OF_ELEMENTS>
```

In CDR, sequences are encoded as the number of elements (as an unsigned long) followed by each of the elements in the sequence.

A.2.7 Array

Arrays have a fixed and well-known number of elements of the same type. In OMG IDL, an array is defined using the symbols "[" and "]", following the C/C++ style. An example is:

```
float[17]
```

In CDR, arrays are encoded by encoding each of its elements from low to high index. In multi-dimensional arrays, the index of the last dimension varies most quickly.

A.2.8 String

A string is an optionally bounded sequence of characters. In OMG IDL, a string of unbounded length is identified by the keyword "string"; a bounded string is specified as follows:

```
string<MAX_LENGTH>
```

MAX_LENGTH is the maximum number of actual characters in the string (not including a possible terminating zero). For example: the string "Hello" can be stored in a variable of type string<5>.

On the wire, strings are encoded as an unsigned long (indicating the number of octets that follow to encode the string), followed by each of the characters in the string and a terminating zero. For example, the string "Hello" is encoded as the unsigned long 6 followed by the octets 'H', 'e', 'l', 'l', 'o', 0.

A.2.9 Wstring

A wide-string is a string of wide-characters. In OMG IDL, unbounded and bounded strings are specified, respectively, as follows:

```
wstring
wstring<MAX_LENGTH>
```

In CDR (GIOP 1.1), a wide-string is encoded as an unsigned long indicating the length of the string on octets or unsigned integers (determined by the transfer syntax for wchar), followed by the individual wide characters. Both the string length and contents include a terminating NULL.