

D7.4_rep - Communication Components

Jan Krakora
CTU

Pavel Pisa
CTU

Frantisek Vacek
CTU

Zdenek Sebek
CTU

Petr Smolik
CTU

Zdenek Hanzalek
CTU

D7.4_rep - Communication Components

by Jan Krakora, Pavel Pisa, Frantisek Vacek, Zdenek Sebek, Petr Smolik, and Zdenek Hanzalek

Published February 2004

Copyright © 2004 Ocera

You can (in fact you must!) use, modify, copy and distribute this document, of course free of charge, and think about the appropriate license we will use for the documentation.

Table of Contents

Preface	i
1. OCERA Real-Time Ethernet	1
1.1. ORTE Summary	1
1.1.1. Summary	1
1.2. ORTE Description	2
1.2.1. Introduction	2
1.2.2. The Publish-Subscribe Architecture	2
1.2.3. The Real-Time Publish-Subscribe Model	5
1.3. ORTE API	7
1.3.1. Data types	8
1.3.2. Functions	31
1.3.3. Macros	58
1.4. ORTE Implementation Issues	68
1.5. ORTE Examples	72
1.5.1. BestEffort Communication	73
1.5.2. Reliable communication	75
1.5.3. Serialization/Deserialization	76
1.6. ORTE Tests	77
1.7. ORTE Usage Information	77
1.7.1. Installation and Setup	77
1.7.2. The ORTE Manager	80
1.7.3. Simple Utilities	83

List of Figures

1-1. Publish-Subscribe Architecture	3
1-2. Generic Publish-Subscribe Architecture	4
1-3. Publication Arbitration	6
1-4. Subscription Issue Separation	6
1-5. ORTE Architecture	69
1-6. ORTE Internal Attributes	71
1-7. RTPS Communication among Network Objects	71
1-8. Periodic Snapshots of a BestEffort Publisher	72
1-9. Position of Managers in RTPS communication	80

Preface

This document presents elaborated version of communication component design report. It is composed of three chapters. The first and second chapters deal with design of communication stacks - RT Ethernet or CAN/CANopen respectively including analysis tools. The third chapter presents Basic features of verification methodology of distributed systems based on standard verification tools.

Chapter 1. OCERA Real-Time Ethernet

The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. RTPS is new application layer protocol targeted to real-time communication area, which is build on the top of standard UDP stack. Since there are many TCP/IP stack implementations under many operating systems and RTPS protocol does not have any other special HW/SW requirements, it should be easily ported to many HW/SW target platforms. Because it uses only UDP protocol, it retains control of timing and reliability.

1.1. ORTE Summary

1.1.1. Summary

Name of the component

OCERA Real-Time Ethernet (ORTE)

Author

Petr Smolik

ORTE Internet resources

<http://www.ocera.org> OCERA project home page

<http://sourceforge.net/projects/ocera> OCERA SourceForge project page. The OCERA CVS relative path to ORTE driver sources is

`ocera/components/comm/eth/orte`
(<http://cvs.sourceforge.net/viewcvs.py/ocera/ocera/components/comm/eth/orte/>).

<http://www.rti.com> Real-Time Innovation home page

Reviewer

not validated

Layer

High-level

Version

orte-0.2.3

Status

Beta

Dependencies

Ethernet adapter with a UDP stack.

Multi-threaded operating system OS

Memory allocator (functions malloc, free)

Supported OS

- Unix - Linux, FreeBSD, Solaris, MacOS
- Windows
- RTAI with RTNet

Release date

September 2004

1.2. ORTE Description

1.2.1. Introduction

The Ocera Real-Time Ethernet (ORTE) is open source implementation of RTPS communication protocol. This protocol is being to submit to IETF as an informational RFC and has been adopted by the IDA group.

1.2.2. The Publish-Subscribe Architecture

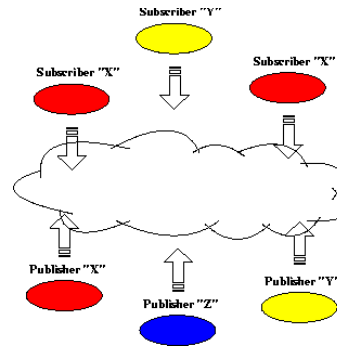
The publish-subscribe architecture is designed to simplify one-to-many data-distribution requirements. In this model, an application “publishes” data and “subscribes” to data. Publishers and subscribers are decoupled from each other too. That is:

- Publishers simply send data anonymously, they do not need any knowledge of the number or network location of subscribers.

- Subscribers simply receive data anonymously, they do not need any knowledge of the number or network location of the publisher.

An application can be a publisher, subscriber, or both a publisher and a subscriber.

Figure 1-1. Publish-Subscribe Architecture



Publish-subscribe supports anonymous, event-driven transfer between many nodes. The developer simply writes the application to send or receive the data.

Publish-subscribe architectures are best-suited to distributed applications with complex data flows. The primary advantages of publish-subscribe to applications developers are:

- Publish-subscribe applications are modular and scalable. The data flow is easy to manage regardless of the number of publishers and subscribers.
- The application subscribes to the data by name rather than to a specific publisher or publisher location. It can thus accommodate configuration changes without disrupting the data flow.
- Redundant publishers and subscribers can be supported, allowing programs to be replicated (e.g. multiple control stations) and moved transparently.
- Publish-subscribe is much more efficient, especially over client-server, with bandwidth utilization.

Publish-subscribe architectures are not good at sporadic request/response traffic, such as file transfers. However, this architecture offers practical advantages for applications with repetitive, time-critical data flows.

1.2.2.1. The Publish-Subscribe Model

Publish-subscribe (PS) data distribution is gaining popularity in many distributed applications, such as financial communications, command and control systems. PS popularity can be attributed to the dramatically reduced system development, deployment and maintenance effort and the performance advantages for applications with one-to-many and many-to-many data flows.

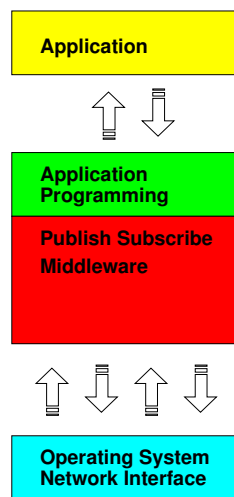
Several main features characterize all publish-subscribe architectures:

Distinct declaration and delivery. Communications occur in three simple steps:

- Publisher declares intent to publish a publication.
- Subscriber declares interest in a publication.
- Publisher sends a publication issue.

The publish-subscribe services are typically made available to applications through middleware that sits on top of the operating system's network interface and presents an application programming interface.

Figure 1-2. Generic Publish-Subscribe Architecture



Publish-subscribe is typically implemented through middleware that sits on top of the operating system's network interface. The middleware presents a publishsubscribe API so that applications make just a few simple calls to send and receive publications. The middleware performs the many and complex network functions that physically distribute the data.

The middleware handles three basic programming chores:

- Maintain the database that maps publishers to subscribers resulting in logical data channels for each publication between publishers and subscribers.
- Serialize (also called marshal) and deserialize (demarshal) the data on its way to and from the network to reconcile publisher and subscriber platform differences.
- Deliver the data when it is published.

1.2.2.2. Publish-Subscribe in Real Time

Publish-subscribe offers some clear advantages for real-time applications:

- Because it is very efficient in both bandwidth and latency for periodic data exchange, PS offers the best transport for distributing data quickly.
- Because it provides many-to-many connectivity, PS is ideal for applications in which publishers and subscribers are added and removed dynamically.

Real-time applications require more functionality than what is provided by desktop and Internet publish-subscribe semantics. For instance, real-time applications often require:

- **Delivery timing control:** Real-time subscribers are concerned with timing; for example, when the data is delivered and how long it remains valid.
- **Reliability control:** Reliable delivery conflicts with deterministic timing. Each subscriber typically requires the ability to specify its own reliability characteristics.
- **Request-reply semantics:** Complex real-time applications often have one-time requests for actions or data. These do not fit well into the PS semantics.
- **Flexible delivery bandwidth:** Typical real-time applications include both real-time and non-realtime subscribers. Each subscriber's bandwidth requirements - even for the same publication - can be different.
- **Fault tolerance:** Real-time applications often require "hot standby" publishers and/or subscribers.
- **Thread priority awareness:** Real-time communications often must work without affecting publisher or subscriber threads.
- **Robustness:** The communications layer should not introduce any single-node points-of-failure to the application.
- **Efficiency:** Real-time systems require efficient data collection and delivery. Only minimal delays should be introduced into the critical data-transfer path.

1.2.3. The Real-Time Publish-Subscribe Model

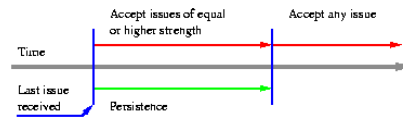
The Real-Time Publish-Subscribe (RTPS) communications model was developed to address these limitations of PS. RTPS adds publication and subscription timing parameters and properties so the developer can control the different types of data flows and achieve their application's performance and reliability goals.

1.2.3.1. Publication Parameters

Each publication is characterized by four parameters: topic, type, strength and persistence. The topic is the label that identifies each data flow. The type describes the data format. The strength indicates a

publisher's weight relative to other publishers of the same topic. The persistence indicates how long each publication issue is valid. Next figure illustrates how a subscriber arbitrates among publications using the strength and persistence properties.

Figure 1-3. Publication Arbitration



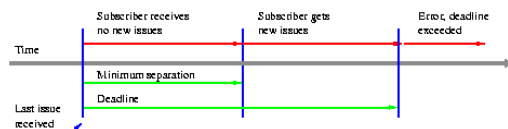
Fault tolerant applications use redundant publishers sending publications with the same topic to ensure continuous operation. Subscribers arbitrate among the publications on an issue-by-issue basis based on the strength and persistence of each issue.

When there are multiple publishers sending the same publication, the subscriber accepts the issue if its strength is greater than the last-received issue or if the last issue's persistence has expired. Typically, a publisher that sends issues with a period of length T will set its persistence to some time T_p where $T_p > T$. Thus, while the strongest publisher is functional, its issues will take precedence over publication issues of lesser strength. Should the strongest publisher stop sending issues (willingly or due to a failure), other publisher(s) sending issues for the same publication will take over after T_p elapses. This mechanism establishes an inherently robust, quasi-stateless communications channel between the then-strongest publisher of a publication and all its subscribers.

1.2.3.2. Subscription Parameters

Subscriptions are identified by four parameters: topic, type, minimum separation and deadline. The topic is the label that identifies the data flow, and type describes the data format (same as the publication properties). Minimum separation defines a period during which no new issues are accepted for that subscription. The deadline specifies how long the subscriber is willing to wait for the next issue. Next figure illustrates the use of these parameters.

Figure 1-4. Subscription Issue Separation



Once the subscriber has received an issue, it will not receive another issue for at least the minimum separation time. If a new issue does not arrive by the deadline, the application is notified.

The minimum separation protects a slow subscriber against publishers that are publishing too fast. The deadline provides a guaranteed wait time that can be used to take appropriate action in case of communication delays.

1.2.3.3. Reliability and Time-Determinism

Publish-subscribe can support a variety of message delivery reliability models, not all of which are suitable to real-time applications. The RTPS reliability model recognizes that the optimal balance between time determinism and data-delivery reliability varies between real-time applications, and often among different subscriptions within the same application. For example, signal subscribers will want only the most up-to-date issues and will not care about missed issues. Command subscribers, on the other hand, must get every issue in sequence. Therefore, RTPS provides a mechanism for the application to customize the determinism versus reliability trade-off on a per subscription basis.

The RTPS determinism vs. reliability model is subscriber-driven. Publishers simply send publication issues. However, to provide message delivery reliability, publishers must be prepared to resend missed issues to subscriptions that require reliable delivery.

The RTPS reliability model uses publication buffers publisher and subscriber and retries to ensure that subscribers who need each issue receive them in the proper sequence. In addition, the publisher applies sequence number to each publication issue.

The publisher uses the publication buffer to store history of the most recently sent issues. The subscriber uses its publication buffer to cache the most recently received issues. The subscriber acknowledges issues received in order and sends a request for the missing issue when the most recent issue's sequence number is out of order. The publisher responds by sending the missed update again.

Publishers remove an issue from their history buffers in two cases: the issue has been acknowledged by all reliable subscribers or the publisher overflows the history buffer space. Flow control can be implemented by setting high and low watermarks for the buffer. These publication-specific parameters let the publisher balance the subscribers' need for issues against its need to maintain a set publication rate.

1.3. ORTE API

1.3.1. Data types

enum SubscriptionMode

Name

enum SubscriptionMode — mode of subscription

Synopsis

```
enum SubscriptionMode {  
    PULLED,  
    IMMEDIATE  
};
```

Constants

PULLED

polled

IMMEDIATE

using callback function

Description

Specifies whether user application will poll for data or whether a callback function will be called by ORTE middleware when new data will be available.

enum SubscriptionType

Name

enum SubscriptionType — type of subscription

Synopsis

```
enum SubscriptionType {  
    BEST_EFFORTS,  
    STRICT_RELIABLE  
};
```

Constants

BEST_EFFORTS

best effort subscription

STRICT_RELIABLE

strict reliable subscription.

Description

Specifies which mode will be used for this subscription.

enum ORTERecvStatus

Name

enum ORTERecvStatus — status of a subscription

Synopsis

```
enum ORTERecvStatus {  
    NEW_DATA,  
    DEADLINE  
};
```

Constants

NEW_DATA

new data has arrived

DEADLINE

deadline has occurred

Description

Specifies which event has occurred in the subscription object.

enum ORTESendStatus

Name

enum ORTESendStatus — status of a publication

Synopsis

```
enum ORTESendStatus {  
    NEED_DATA,  
    CQL  
};
```

Constants

NEED_DATA

need new data (set when callback function specified for publication is being called)

CQL

transmit queue has been filled up to critical level.

Description

Specifies which event has occurred in the publication object. Critical level of transmit queue is specified as one of publication properties (ORTEPublProp.criticalQueueLevel).

struct ORTEIFProp

Name

struct ORTEIFProp — interface flags

Synopsis

```
struct ORTEIFProp {  
    int32_t ifFlags;  
    IPAddress ipAddress;  
};
```

Members

ifFlags

flags

ipAddress

IP address

Description

Flags for network interface.

struct ORTEMulticastProp

Name

struct ORTEMulticastProp — properties for ORTE multicast (not supported yet)

Synopsis

```
struct ORTEMulticastProp {  
    Boolean enabled;  
    unsigned char ttl;  
    Boolean loopBackEnabled;  
    IPAddress ipAddress;  
};
```



```
};
```

Members

enabled

ORTE_TRUE if multicast enabled otherwise ORTE_FALSE

ttl

time-to-live (TTL) for sent datagrams

loopBackEnabled

ORTE_TRUE if data should be received by sender itself otherwise ORTE_FALSE

ipAddress

desired multicast IP address

Description

Properties for ORTE multicast subsystem which is not fully supported yet. Multicast IP address is assigned by the ORTE middleware itself.

struct ORTEGetMaxSizeParam

Name

struct ORTEGetMaxSizeParam — parameters for function ORTETypeGetMaxSize

Synopsis

```
struct ORTEGetMaxSizeParam {
    CDR_Endianness host_endian;
    CDR_Endianness data_endian;
    CORBA_octet * data;
    unsigned int max_size;
    int recv_size;
    int csize;
};
```

Members

host_endian

data_endian

data

max_size

recv_size

csize

Description

It used to determine maximal size of internal buffer for incoming data

struct ORTETypeRegister

Name

struct ORTETypeRegister — registered data type

Synopsis

```
struct ORTETypeRegister {  
    const char          * typeName;  
    ORTETypeSerialize serialize;  
    ORTETypeDeserialize deserialize;  
    ORTETypeGetMaxSize getMaxSize;  
    unsigned int        maxSize;  
};
```

Members

typeName

name of data type

serialize

pointer to serialization function

deserialize

pointer to deserialization function

getMaxSize

pointer to function given maximal data length

maxSize

maximal size of ser./deser. data

Description

Contains description of registered data type. See *ORTETypeRegisterAdd* function for details.

struct ORTEDomainBaseProp

Name

struct ORTEDomainBaseProp — base properties of a domain

Synopsis

```
struct ORTEDomainBaseProp {
    unsigned int    registrationMgrRetries;
    NtpTime registrationMgrPeriod;
    unsigned int    registrationAppRetries;
    NtpTime registrationAppPeriod;
    NtpTime expirationTime;
    NtpTime refreshPeriod;
    NtpTime purgeTime;
    NtpTime repeatAnnounceTime;
    NtpTime repeatActiveQueryTime;
    NtpTime delayResponseTimeACKMin;
    NtpTime delayResponseTimeACKMax;
```

```

unsigned int      HBMaxRetries;
unsigned int      ACKMaxRetries;
NtpTime maxBlockTime;
};

```

Members

registrationMgrRetries

a manager which want to start communication have to register to other manager. This parametr is used for specify maximal repetition retries of registration process when it fail.

registrationMgrPeriod

an application which want to start communication have to register to a manager. This parametr is used for specify maximal repetition retries of registration process when it fail.

registrationAppRetries

same like registrationMgrRetries parameter, but is used for an application

registrationAppPeriod

repetition time for registration process

expirationTime

specifies how long is this application taken as alive in other applications/managers (default 180s)

refreshPeriod

how often an application refresh itself to its manager or manager to other managers (default 60s)

purgeTime

how often the local database should be cleaned from invalid (expired) objects (default 60s)

repeatAnnounceTime

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.

repeatActiveQueryTime

???

delayResponseTimeACKMin

minimum time the CSTWriter waits before responding to an incoming message.

delayResponseTimeACKMax

maximum time the CSTWriter waits before responding to an incoming message.

HBMMaxRetries

how many times a HB message is retransmitted if no response has been received until timeout

ACKMaxRetries

how many times an ACK message is retransmitted if no response has been received until timeout

maxBlockTime

timeout for send functions if sending queue is full (default 30s)

struct ORTEDomainWireProp

Name

struct ORTEDomainWireProp — wire properties of a message

Synopsis

```
struct ORTEDomainWireProp {
    unsigned int      metaBytesPerPacket;
    unsigned int      metaBytesPerFastPacket;
    unsigned int      metabitsPerACKBitmap;
};
```

Members

metaBytesPerPacket

maximum number of bytes in single frame (default 1500B)

metaBytesPerFastPacket

maximum number of bytes in single frame if transmitting queue has reached *criticalQueueLevel* level (see *ORTEPublProp* struct)

metabitsPerACKBitmap

not supported yet (default 32)

struct ORTEPublProp

Name

struct ORTEPublProp — properties of a publication

Synopsis

```
struct ORTEPublProp {
    PathName topic;
    TypeName typeName;
    TypeChecksum typeChecksum;
    Boolean expectsAck;
    NtpTime persistence;
    uint32_t reliabilityOffered;
    uint32_t sendQueueSize;
    int32_t strength;
    uint32_t criticalQueueLevel;
    NtpTime HBNormalRate;
    NtpTime HBCQLRate;
    unsigned int          HBMaxRetries;
    NtpTime maxBlockTime;
};
```

Members

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

expectsAck

indicates whether publication expects to receive ACKs to its messages

persistence

indicates how long the issue is valid

reliabilityOffered

reliability policy as offered by the publication

<code>sendQueueSize</code>	size of transmitting queue
<code>strength</code>	precedence of the issue sent by the publication
<code>criticalQueueLevel</code>	treshold for transmitting queue content length indicating the queue can became full immediately
<code>HBNormalRate</code>	how often send HBs to subscription objects
<code>HBCQLRate</code>	how often send HBs to subscription objects if transmittiong queue has reached <i>criticalQueueLevel</i>
<code>HBMaxRetries</code>	how many times retransmit HBs if no replay from target object has not been received
<code>maxBlockTime</code>	unsupported

struct ORTESubsProp

Name

`struct ORTESubsProp` — properties of a subscription

Synopsis

```
struct ORTESubsProp {
    PathName topic;
    TypeName typeName;
    TypeChecksum typeChecksum;
    NtpTime minimumSeparation;
    uint32_t recvQueueSize;
    uint32_t reliabilityRequested;
    //additional parametersNtpTime deadline;
    uint32_t mode;
    IPAddress multicast;
};
```

Members

`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

`minimumSeparation`

minimum time between two consecutive issues received by the subscription

`recvQueueSize`

size of receiving queue

`reliabilityRequested`

reliability policy requested by the subscription

`deadline`

deadline for subscription, a callback function (see *ORTESubscriptionCreate*) will be called if no data were received within this period of time

`mode`

mode of subscription (strict reliable/best effort), see *SubscriptionType* enum for values

`multicast`

registered multicast IP address(read only)

struct ORTEAppInfo

Name

`struct ORTEAppInfo` —

Synopsis

```
struct ORTEAppInfo {
    HostId hostId;
    AppId appId;
```



```

IPAddress * unicastIPAddressList;
unsigned char    unicastIPAddressCount;
IPAddress * metatrafficMulticastIPAddressList;
unsigned char    metatrafficMulticastIPAddressCount;
Port metatrafficUnicastPort;
Port userdataUnicastPort;
VendorId vendorId;
ProtocolVersion protocolVersion;
};

```

Members

hostId

hostId of application

appId

appId of application

unicastIPAddressList

unicast IP addresses of the host on which the application runs (there can be multiple addresses on a multi-NIC host)

unicastIPAddressCount

number of IPAddresses in *unicastIPAddressList*

metatrafficMulticastIPAddressList

for the purposes of meta-traffic, an application can also accept Messages on this set of multicast addresses

metatrafficMulticastIPAddressCount

number of IPAddresses in *metatrafficMulticastIPAddressList*

metatrafficUnicastPort

UDP port used for metatraffic communication

userdataUnicastPort

UDP port used for metatraffic communication

vendorId

identifies the vendor of the middleware implementing the RTPS protocol and allows this vendor to add specific extensions to the protocol

protocolVersion

describes the protocol version

struct ORTEPubInfo

Name

struct ORTEPubInfo — information about publication

Synopsis

```
struct ORTEPubInfo {  
    const char      * topic;  
    const char      * type;  
    ObjectId objectId;  
};
```

Members

topic

the name of the information in the Network that is published or subscribed to

type

the name of the type of this data

objectId

object providing this publication

struct ORTESubInfo

Name

struct ORTESubInfo — information about subscription

Synopsis

```
struct ORTESubInfo {  
    const char      * topic;  
    const char      * type;
```

```

    ObjectId objectId;
};

```

Members

topic

the name of the information in the Network that is published or subscribed to

type

the name of the type of this data

objectId

object with this subscription

struct ORTEPublStatus

Name

struct ORTEPublStatus — status of a publication

Synopsis

```

struct ORTEPublStatus {
    unsigned int    strict;
    unsigned int    bestEffort;
    unsigned int    issues;
};

```

Members

strict

count of unreliable subscription (strict) connected on responsible subscription

bestEffort

count of reliable subscription (best effort) connected on responsible subscription

issues

number of messages in transmitting queue

struct ORTESubsStatus

Name

struct ORTESubsStatus — status of a subscription

Synopsis

```
struct ORTESubsStatus {
    unsigned int    strict;
    unsigned int    bestEffort;
    unsigned int    issues;
};
```

Members

strict

count of unreliable publications (strict) connected to responsible subscription

bestEffort

count of reliable publications (best effort) connected to responsible subscription

issues

number of messages in receiving queue

struct ORTERecvInfo

Name

struct ORTERecvInfo — description of received data

Synopsis

```
struct ORTERecvInfo {
    ORTERecvStatus status;
    const char      * topic;
    const char      * type;
    GUID_RTPS senderGUID;
    NtpTime localTimeReceived;
    NtpTime remoteTimePublished;
    SequenceNumber sn;
};
```

Members

status

status of this event

topic

the name of the information

type

the name of the type of this data

senderGUID

GUID of object who sent this information

localTimeReceived

local timestamp when data were received

remoteTimePublished

remote timestamp when data were published

sn

sequential number of data

struct ORTESendInfo

Name

struct ORTESendInfo — description of sending data

Synopsis

```
struct ORTESendInfo {
    ORTESendStatus status;
    const char      * topic;
    const char      * type;
    GUID_RTPS senderGUID;
    SequenceNumber sn;
};
```

Members

status

status of this event

topic

the name of the information

type

the name of the type of this information

senderGUID

GUID of object who sent this information

sn

sequential number of information

struct ORTEPublicationSendParam

Name

struct ORTEPublicationSendParam — description of sending data

Synopsis

```
struct ORTEPublicationSendParam {
    void * instance;
    int data_endian;
};
```

Members

instance

pointer to new data instance

data_endian

endianing of sending data (BIG | LITTLE)

struct ORTEDomainAppEvents

Name

struct ORTEDomainAppEvents — Domain event handlers of an application

Synopsis

```
struct ORTEDomainAppEvents {
    ORTEOnRegFail onRegFail;
    void * onRegFailParam;
    ORTEOnMgrNew onMgrNew;
    void * onMgrNewParam;
    ORTEOnMgrDelete onMgrDelete;
    void * onMgrDeleteParam;
    ORTEOnAppRemoteNew onAppRemoteNew;
    void * onAppRemoteNewParam;
    ORTEOnAppDelete onAppDelete;
    void * onAppDeleteParam;
    ORTEOnPubRemote onPubRemoteNew;
    void * onPubRemoteNewParam;
    ORTEOnPubRemote onPubRemoteChanged;
    void * onPubRemoteChangedParam;
    ORTEOnPubDelete onPubDelete;
    void * onPubDeleteParam;
    ORTEOnSubRemote onSubRemoteNew;
    void * onSubRemoteNewParam;
    ORTEOnSubRemote onSubRemoteChanged;
    void * onSubRemoteChangedParam;
    ORTEOnSubDelete onSubDelete;
    void * onSubDeleteParam;
};
```

Members

`onRegFail`

registration protocol has been failed

`onRegFailParam`

user parameters for *onRegFail* handler

`onMgrNew`

new manager has been created

`onMgrNewParam`

user parameters for *onMgrNew* handler

`onMgrDelete`

manager has been deleted

`onMgrDeleteParam`

user parameters for *onMgrDelete* handler

`onAppRemoteNew`

new remote application has been registered

`onAppRemoteNewParam`

user parameters for *onAppRemoteNew* handler

`onAppDelete`

an application has been removed

`onAppDeleteParam`

user parameters for *onAppDelete* handler

`onPubRemoteNew`

new remote publication has been registered

`onPubRemoteNewParam`

user parameters for *onPubRemoteNew* handler

`onPubRemoteChanged`

a remote publication's parameters has been changed

`onPubRemoteChangedParam`

user parameters for *onPubRemoteChanged* handler

`onPubDelete`

a publication has been deleted

`onPubDeleteParam`

user parameters for *onPubDelete* handler

`onSubRemoteNew`

a new remote subscription has been registered

`onSubRemoteNewParam`

user parameters for *onSubRemoteNew* handler

`onSubRemoteChanged`

a remote subscription's parameters has been changed

`onSubRemoteChangedParam`

user parameters for *onSubRemoteChanged* handler

`onSubDelete`

a publication has been deleted

`onSubDeleteParam`

user parameters for *onSubDelete* handler

Description

Prototypes of events handler functions can be found in file `typedefs_api.h`.

struct ORTETasksProp

Name

`struct ORTETasksProp` — ORTE task properties, not supported

Synopsis

```
struct ORTETasksProp {
    Boolean realTimeEnabled;
    int smtStackSize;
    int smtPriority;
```

```

    int rmtStackSize;
    int rmtPriority;
};

```

Members

realTimeEnabled

not supported

smtStackSize

not supported

smtPriority

not supported

rmtStackSize

not supported

rmtPriority

not supported

struct ORTEDomainProp

Name

struct ORTEDomainProp — domain properties

Synopsis

```

struct ORTEDomainProp {
    ORTETasksProp tasksProp;
    ORTEIFProp * IFProp;
    //interface propertiesunsigned char          IFCount;
    //count of interfacesortedomainBaseProp      baseProp;
    ORTEDomainWireProp wireProp;
    ORTEMulticastProp multicast;
    //multicast properiesORTEPublProp            publPropDefault;
    //default properties for a Publ/SubORTESubsProp      subsPropDefault;
    char * mgrs;
    //managerschar * keys;
    //keysIPAddress appLocalManager;
};

```

```

//applicationschar * version;
//string product versionint
int sendBuffSize;
recvBuffSize;
};

```

Members

tasksProp

task properties

IFProp

properties of network interfaces

IFCount

number of network interfaces

baseProp

base properties (see *ORTEDomainBaseProp* for details)

wireProp

wire properties (see *ORTEDomainWireProp* for details)

multicast

multicast properties (see *ORTEMulticastProp* for details)

publPropDefault

default properties of publications (see *ORTEPublProp* for details)

subsPropDefault

default properties of subscriptions (see *ORTESubsProp* for details)

mgrs

list of known managers

keys

access keys for managers

appLocalManager

IP address of local manager (default localhost)

version

string product version

`recvBuffSize`

receiving queue length

`sendBuffSize`

transmitting queue length

1.3.2. Functions

IPAddressToString

Name

`IPAddressToString` — converts IP address `IPAddress` to its string representation

Synopsis

```
char* IPAddressToString (IPAddress ipAddress, char * buff);
```

Arguments

ipAddress

source IP address

buff

output buffer

StringToIPAddress

Name

`StringToIPAddress` — converts IP address from string into `IPAddress`

Synopsis

```
IPAddress StringToIPAddress (const char * string);
```

Arguments

string

source string

NtpTimeToStringMs

Name

`NtpTimeToStringMs` — converts `NtpTime` to its text representation in milliseconds

Synopsis

```
char * NtpTimeToStringMs (NtpTime time, char * buff);
```

Arguments

time

time given in `NtpTime` structure

buff

output buffer

NtpTimeToStringUs

Name

`NtpTimeToStringUs` — converts `NtpTime` to its text representation in microseconds

Synopsis

```
char * NtpTimeToStringUs (NtpTime time, char * buff);
```

Arguments

time

time given in `NtpTime` structure

buff

output buffer

ORTEDomainStart

Name

`ORTEDomainStart` — start specific threads

Synopsis

```
void ORTEDomainStart (ORTEDomain * d, Boolean recvUnicastMetatrafficThread,  
Boolean recvMulticastMetatrafficThread, Boolean recvUnicastUserdataThread,  
Boolean recvMulticastUserdataThread, Boolean sendThread);
```

Arguments

d

domain object handle

recvUnicastMetatrafficThread

specifies whether *recvUnicastMetatrafficThread* should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

recvMulticastMetatrafficThread

specifies whether *recvMulticastMetatrafficThread* should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

recvUnicastUserdataThread

specifies whether *recvUnicastUserdataThread* should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

recvMulticastUserdataThread

specifies whether *recvMulticastUserdataThread* should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

sendThread

specifies whether *sendThread* should be started (ORTE_TRUE) or remain suspended (ORTE_FALSE).

Description

Functions *ORTEDomainAppCreate* and *ORTEDomainMgrCreate* provide facility to create an object with its threads suspended. Use function *ORTEDomainStart* to resume those suspended threads.

ORTEDomainPropDefaultGet

Name

ORTEDomainPropDefaultGet — returns default properties of a domain

Synopsis

```
Boolean ORTEDomainPropDefaultGet (ORTEDomainProp * prop);
```

Arguments

prop

pointer to struct ORTEDomainProp

Description

Structure ORTEDomainProp referenced by *prop* will be filled by its default values. Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainInitEvents

Name

ORTEDomainInitEvents — initializes list of events

Synopsis

```
Boolean ORTEDomainInitEvents (ORTEDomainAppEvents * events);
```

Arguments

events

pointer to struct ORTEDomainAppEvents

Description

Initializes structure pointed by *events*. Every member is set to NULL. Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainAppCreate

Name

`ORTEDomainAppCreate` — creates an application object within given domain

Synopsis

```
ORTEDomain * ORTEDomainAppCreate (int domain, ORTEDomainProp * prop,
ORTEDomainAppEvents * events, Boolean suspended);
```

Arguments

domain

given domain

prop

properties of application

events

events associated with application or NULL

suspended

specifies whether threads of this application should be started as well (`ORTE_FALSE`) or stay suspended (`ORTE_TRUE`). See *ORTEDomainStart* for details how to resume suspended threads

Description

Creates new Application object and sets its properties and events. Return handle to created object or NULL in case of any error.

ORTEDomainAppDestroy

Name

`ORTEDomainAppDestroy` — destroy Application object

Synopsis

```
Boolean ORTEDomainAppDestroy (ORTEDomain * d);
```

Arguments

d

domain

Description

Destroys all application objects in specified domain. Returns ORTE_TRUE or ORTE_FALSE in case of any error.

ORTEDomainAppSubscriptionPatternAdd

Name

ORTEDomainAppSubscriptionPatternAdd — create pattern-based subscription

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternAdd (ORTEDomain * d, const char *
topic, const char * type, ORTESubscriptionPatternCallBack
subscriptionCallBack, void * param);
```

Arguments

d

domain object

topic

pattern for topic

type

pattern for type

subscriptionCallback

pointer to callback function which will be called whenever any data are received through this subscription

param

user params for callback function

Description

This function is intended to be used in application interested in more published data from possibly more remote applications, which should be received through single subscription. These different publications are specified by pattern given to *topic* and *type* parameters.

For example suppose there are publications of topics like *temperatureEngine1*, *temperatureEngine2*, *temperatureEngine1Backup* and *temperatureEngine2Backup* somewhere on our network. We can subscribe to each of Engine1 temperatures by creating single subscription with pattern for topic set to “temperatureEngine1*”. Or, if we are interested only in values from backup measurements, we can use pattern “*Backup”.

Syntax for patterns is the same as syntax for *fnmatch* function, which is employed for pattern recognition.

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainAppSubscriptionPatternRemove

Name

ORTEDomainAppSubscriptionPatternRemove — remove subscription pattern

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternRemove (ORTEDomain * d, const char *
topic, const char * type);
```

Arguments

d

domain handle

topic

pattern to be removed

type

pattern to be removed

Description

Removes subscriptions created by *ORTEDomainAppSubscriptionPatternAdd*. Patterns for *type* and *topic* must be exactly the same strings as when *ORTEDomainAppSubscriptionPatternAdd* function was called.

Returns ORTE_TRUE if successful or ORTE_FALSE if none matching record was found

ORTEDomainAppSubscriptionPatternDestroy

Name

ORTEDomainAppSubscriptionPatternDestroy — destroys all subscription patterns

Synopsis

```
Boolean ORTEDomainAppSubscriptionPatternDestroy (ORTEDomain * d);
```

Arguments

d

domain handle

Description

Destroys all subscription patterns which were specified previously by *ORTEDomainAppSubscriptionPatternAdd* function.

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEDomainMgrCreate

Name

ORTEDomainMgrCreate — create manager object in given domain

Synopsis

```
ORTEDomain * ORTEDomainMgrCreate (int domain, ORTEDomainProp * prop,
ORTEDomainAppEvents * events, Boolean suspended);
```

Arguments

domain

given domain

prop

desired manager's properties

events

manager's event handlers or NULL

suspended

specifies whether threads of this manager should be started as well (ORTE_FALSE) or stay suspended (ORTE_TRUE). See *ORTEDomainStart* for details how to resume suspended threads

Description

Creates new manager object and sets its properties and events. Return handle to created object or NULL in case of any error.

ORTEDomainMgrDestroy

Name

ORTEDomainMgrDestroy — destroy manager object

Synopsis

```
Boolean ORTEDomainMgrDestroy (ORTEDomain * d);
```

Arguments

d

manager object to be destroyed

Description

Returns ORTE_TRUE if successful or ORTE_FALSE in case of any error.

ORTEPublicationCreate

Name

ORTEPublicationCreate — creates new publication

Synopsis

```
ORTEPublication * ORTEPublicationCreate (ORTEDomain * d, const char * topic,
const char * typeName, void * instance, NtpTime * persistence, int strength,
ORTESendCallBack sendCallBack, void * sendCallBackParam, NtpTime *
sendCallBackDelay);
```

Arguments

d

pointer to application object

topic

name of topic

typeName

data type description

instance

output buffer where application stores data for publication

persistence

persistence of publication

strength

strength of publication

sendCallBack

pointer to callback function

sendCallBackParam

user parameters for callback function

sendCallBackDelay

periode for timer which issues callback function

Description

Creates new publication object with specified parameters. The *sendCallBack* function is called periodically with *sendCallBackDelay* periode. In strict reliable mode the *sendCallBack* function will be called only if there is enough room in transmitting queue in order to prevent any data

loss. The *sendCallback* function should prepare data to be published by this publication and place them into *instance* buffer.

Returns handle to publication object.

ORTEPublicationDestroy

Name

ORTEPublicationDestroy — removes a publication

Synopsis

```
int ORTEPublicationDestroy (ORTEPublication * cstWriter);
```

Arguments

cstWriter

handle to publication to be removed

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *cstWriter* is not valid *cstWriter* handle.

ORTEPublicationPropertiesGet

Name

ORTEPublicationPropertiesGet — read properties of a publication

Synopsis

```
ORTEPublicationPropertiesGet (ORTEPublication * cstWriter, ORTEPublProp *
```

```
pp);
```

Arguments

cstWriter

pointer to *cstWriter* object which provides this publication

pp

pointer to ORTEPublProp structure where values of publication's properties will be stored

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *cstWriter* is not valid *cstWriter* handle.

ORTEPublicationPropertiesSet

Name

ORTEPublicationPropertiesSet — set properties of a publication

Synopsis

```
int ORTEPublicationPropertiesSet (ORTEPublication * cstWriter, ORTEPublProp *
```

```
pp);
```

Arguments

cstWriter

pointer to *cstWriter* object which provides this publication

pp

pointer to ORTEPublProp structure containing values of publication's properties

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *cstWriter* is not valid publication handle.

ORTEPublicationGetStatus

Name

ORTEPublicationGetStatus — removes a publication

Synopsis

```
int ORTEPublicationGetStatus (ORTEPublication * cstWriter, ORTEPublStatus *
status);
```

Arguments

cstWriter

pointer to cstWriter object which provides this publication

status

pointer to ORTEPublStatus structure

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *happ* is not valid publication handle.

ORTEPublicationSend

Name

ORTEPublicationSend — force publication object to issue new data

Synopsis

```
int ORTEPublicationSend (ORTEPublication * cstWriter);
```

Arguments

cstWriter

publication object

Description

Returns ORTE_OK if successful.

ORTEPublicationSendEx

Name

ORTEPublicationSendEx — force publication object to issue new data with additional parameters

Synopsis

```
int ORTEPublicationSendEx (ORTEPublication * cstWriter,  
ORTEPublicationSendParam * psp);
```

Arguments

cstWriter

publication object

psp

publication parameters

Description

Returns ORTE_OK if successful.

ORTEPublicationGetInstance

Name

ORTEPublicationGetInstance — return pointer to an instance

Synopsis

```
void * ORTEPublicationGetInstance (ORTEPublication * cstWriter);
```

Arguments

cstWriter

publication object

Description

Returns handle

ORTESubscriptionCreate

Name

ORTESubscriptionCreate — adds a new subscription

Synopsis

```
ORTESubscription * ORTESubscriptionCreate (ORTEDomain * d, SubscriptionMode
mode, SubscriptionType sType, const char * topic, const char * typeName, void
* instance, NtpTime * deadline, NtpTime * minimumSeparation, ORTERecvCallBack
recvCallBack, void * recvCallBackParam, IPAddress multicastIPAddress);
```

Arguments

d

pointer to ORTEDomain object where this subscription will be created

mode

see enum SubscriptionMode

sType

see enum SubscriptionType

topic

name of topic

typeName

name of data type

instance

pointer to output buffer

deadline

deadline

minimumSeparation

minimum time interval between two publications sent by Publisher as requested by Subscriber
(strict - minumSep musi byt 0)

recvCallback

callback function called when new Subscription has been received or if any change of subscription's status occurs

recvCallbackParam

user parameters for *recvCallback*

multicastIPAddress

in case multicast subscription specify multicast IP address or use `IPADDRESS_INVALID` to unicast communication

Description

Returns handle to Subscription object.

ORTESubscriptionDestroy

Name

`ORTESubscriptionDestroy` — removes a subscription

Synopsis

```
int ORTESubscriptionDestroy (ORTESubscription * cstReader);
```

Arguments

cstReader

handle to subscription to be removed

Description

Returns `ORTE_OK` if successful or `ORTE_BAD_HANDLE` if *cstReader* is not valid subscription handle.

ORTESubscriptionPropertiesGet

Name

ORTESubscriptionPropertiesGet — get properties of a subscription

Synopsis

```
int ORTESubscriptionPropertiesGet (ORTESubscription * cstReader, ORTESubsProp  
* sp);
```

Arguments

cstReader

handle to publication

sp

pointer to ORTESubsProp structure where properties of subscription will be stored

ORTESubscriptionPropertiesSet

Name

ORTESubscriptionPropertiesSet — set properties of a subscription

Synopsis

```
int ORTESubscriptionPropertiesSet (ORTESubscription * cstReader, ORTESubsProp  
* sp);
```

Arguments

cstReader

handle to publication

sp

pointer to ORTESubsProp structure containing desired properties of the subscription

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *cstReader* is not valid subscription handle.

ORTESubscriptionWaitForPublications

Name

ORTESubscriptionWaitForPublications — waits for given number of publications

Synopsis

```
int ORTESubscriptionWaitForPublications (ORTESubscription * cstReader,
NtpTime wait, unsigned int retries, unsigned int noPublications);
```

Arguments

cstReader

handle to subscription

wait

time how long to wait

retries

number of retries if specified number of publications was not reached

noPublications

desired number of publications

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *cstReader* is not valid subscription handle or ORTE_TIMEOUT if number of retries has been exhausted..

ORTESubscriptionGetStatus

Name

ORTESubscriptionGetStatus — get status of a subscription

Synopsis

```
int ORTESubscriptionGetStatus (ORTESubscription * cstReader, ORTESubsStatus *
status);
```

Arguments

cstReader

handle to subscription

status

pointer to ORTESubsStatus structure

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *cstReader* is not valid subscription handle.

ORTESubscriptionPull

Name

ORTESubscriptionPull — read data from receiving buffer

Synopsis

```
int ORTESubscriptionPull (ORTESubscription * cstReader);
```

Arguments

cstReader

handle to subscription

Description

Returns ORTE_OK if successful or ORTE_BAD_HANDLE if *cstReader* is not valid subscription handle.

ORTESubscriptionGetInstance

Name

ORTESubscriptionGetInstance — return pointer to an instance

Synopsis

```
void * ORTESubscriptionGetInstance (ORTESubscription * cstReader);
```

Arguments

cstReader

publication object

Description

Returns handle

ORTETypeRegisterAdd

Name

ORTETypeRegisterAdd — register new data type

Synopsis

```
int ORTETypeRegisterAdd (ORTEDomain * d, const char * typeName,  
ORTETypeSerialize ts, ORTETypeDeserialize ds, ORTETypeGetMaxSize gms,  
unsigned int ms);
```

Arguments

d

domain object handle

typeName

name of data type

ts

pointer to serialization function. If NULL data will be copied without any processing.

ds

deserialization function. If NULL data will be copied without any processing.

gms

pointer to a function given maximum length of data (in bytes)

ms

default maximal size

Description

Each data type has to be registered. Main purpose of this process is to define serialization and deserialization functions for given data type. The same data type can be registered several times, previous registrations of the same type will be overwritten.

Examples of serialization and deserialization functions can be found if contrib/shape/ortedemo_types.c file.

Returns ORTE_OK if new data type has been succesfully registered.

ORTETypeRegisterDestroyAll

Name

ORTETypeRegisterDestroyAll — destroy all registered data types

Synopsis

```
int ORTETypeRegisterDestroyAll (ORTEDomain * d);
```

Arguments

d

domain object handle

Description

Destroys all data types which were previously registered by function *ORTETypeRegisterAdd*.

Return ORTE_OK if all data types has been succesfully destroyed.

ORTEVerbositySetOptions

Name

ORTEVerbositySetOptions — set verbosity options

Synopsis

```
void ORTEVerbositySetOptions (const char * options);
```

Arguments

options

verbosity options

Description

There are 10 levels of verbosity ranging from 1 (lowest) to 10 (highest). It is possible to specify certain level of verbosity for each module of ORTE library. List of all supported modules can be found in *linorte/usedSections.txt* file. Every module has been aassigned with a number as can be seen in *usedSections.txt* file.

For instance

options = "ALL,7" Verbosity will be set to level 7 for all modules.

options = "51,7:32,5" Modules 51 (RTPSCSTWrite.c) will use verbosity level 7 and module 32 (ORTEPublicationTimer.c) will use verbosity level 5.

Maximum number of modules and verbosity levels can be changed in order to save some memory space if small memory footprint is necessary. These values are defined as macros `MAX_DEBUG_SECTIONS` and `MAX_DEBUG_LEVEL` in file `include/defines.h`.

Return `ORTE_OK` if desired verbosity levels were successfully set.

ORTEVerbositySetLogFile

Name

`ORTEVerbositySetLogFile` — set log file

Synopsis

```
void ORTEVerbositySetLogFile (const char * logfile);
```

Arguments

logfile

log file name

Description

Sets output file where debug messages will be written to. By default these messages are written to stdout.

ORTEInit

Name

`ORTEInit` — initialization of ORTE layer (musi se zavolat)

Synopsis

```
void ORTEInit ( void );
```

Arguments

void

no arguments

ORTESleepMs

Name

ORTESleepMs — suspends calling thread for given time

Synopsis

```
void ORTESleepMs (unsigned int ms);
```

Arguments

ms

milliseconds to sleep

1.3.3. Macros

SeqNumberCmp

Name

`SeqNumberCmp` — comparison of two sequence numbers

Synopsis

```
SeqNumberCmp ( sn1, sn2 );
```

Arguments

sn1

source sequential number 1

sn2

source sequential number 2

Return

1 if $sn1 > sn2$ -1 if $sn1 < sn2$ 0 if $sn1 = sn2$

SeqNumberInc

Name

`SeqNumberInc` — incrementation of a sequence number

Synopsis

```
SeqNumberInc ( res, sn );
```


Arguments

res

result

sn

sequential number to be incremented

Description

$res = sn + 1$

SeqNumberAdd

Name

SeqNumberAdd — addition of two sequential numbers

Synopsis

```
SeqNumberAdd ( res, sn1, sn2 );
```

Arguments

res

result

sn1

source sequential number 1

sn2

source sequential number 2

Description

$res = sn1 + sn2$

SeqNumberDec

Name

SeqNumberDec — decrementation of a sequence number

Synopsis

`SeqNumberDec (res, sn);`

Arguments

res

result

sn

sequential number to be decremented

Description

$res = sn - 1$

SeqNumberSub

Name

SeqNumberSub — substraction of two sequential numbers

Synopsis

```
SeqNumberSub ( res, sn1, sn2);
```

Arguments

res

result

sn1

source sequential number 1

sn2

source sequential number 2

Description

$res = sn1 - sn2$

NtpTimeCmp

Name

NtpTimeCmp — comparison of two NtpTimes

Synopsis

```
NtpTimeCmp ( time1, time2);
```

Arguments

time1

source time 1

time2

source time 2

Return value

1 if time 1 > time 2 -1 if time 1 < time 2 0 if time 1 = time 2

NtpTimeAdd

Name

NtpTimeAdd — addition of two NtpTimes

Synopsis

```
NtpTimeAdd ( res, time1, time2 );
```

Arguments

res

result

time1

source time 1

time2

source time 2

Description

$res = time1 + time2$

NtpTimeSub

Name

NtpTimeSub — subtraction of two NtpTimes

Synopsis

```
NtpTimeSub ( res, time1, time2);
```

Arguments

res

result

time1

source time 1

time2

source time 2

Description

$res = time1 - time2$

NtpTimeAssembFromMs

Name

NtpTimeAssembFromMs — converts seconds and milliseconds to NtpTime

Synopsis

```
NtpTimeAssembFromMs ( time, s, msec);
```

Arguments

time

time given in NtpTime structure

s

seconds portion of given time

msec

milliseconds portion of given time

NtpTimeDisAssembToMs

Name

NtpTimeDisAssembToMs — converts NtpTime to seconds and milliseconds

Synopsis

```
NtpTimeDisAssembToMs ( s, msec, time );
```

Arguments

s

seconds portion of given time

msec

milliseconds portion of given time

time

time given in NtpTime structure

NtpTimeAssembFromUs

Name

NtpTimeAssembFromUs — converts seconds and useconds to NtpTime

Synopsis

```
NtpTimeAssembFromUs ( time, s, usec );
```

Arguments

time

time given in NtpTime structure

s

seconds portion of given time

usec

microseconds portion of given time

NtpTimeDisAssembToUs

Name

NtpTimeDisAssembToUs — converts NtpTime to seconds and useconds

Synopsis

```
NtpTimeDisAssembToUs ( s, usec, time );
```

Arguments

s

seconds portion of given time

usec

microseconds portion of given time

time

time given in NtpTime structure

Domain2Port

Name

Domain2Port — converts Domain value to IP Port value

Synopsis

```
Domain2Port ( d, p );
```

Arguments

d

domain

p

port

Domain2PortMulticastUserdata

Name

Domain2PortMulticastUserdata — converts Domain value to userdata IP Port value

Synopsis

```
Domain2PortMulticastUserdata ( d, p );
```

Arguments

d
domain

p
port

Domain2PortMulticastMetatraffic

Name

Domain2PortMulticastMetatraffic — converts Domain value to metatraffic IP Port value

Synopsis

```
Domain2PortMulticastMetatraffic ( d, p );
```

Arguments

d
domain

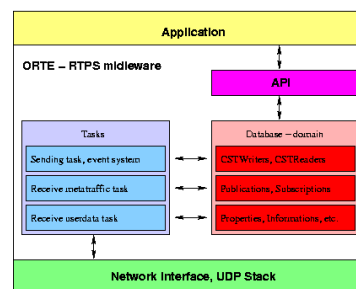
p
port

1.4. ORTE Implementation Issues

ORTE is network middleware for distributed, real-time application development that uses the real-time, publish-subscribe model. The middleware is available for a variety of platforms including RTAI, RTLinux, Windows, and a several versions of Unix. The compilation system is mainly based on autoconf.

ORTE is middleware composed of a database, and tasks. On the top of ORTE architecture is application interface (API). By using API users should write self application. The tasks perform all of the message addressing serialization/deserialization, and transporting. The ORTE components are shown in Figure 1-5

Figure 1-5. ORTE Architecture



The RTPS protocol defines two kinds of Applications:

- **Manager:** The manager is a special Application that helps applications automatically discover each other on the Network.
- **ManagedApplication:** A ManagedApplication is an Application that is managed by one or more Managers. Every ManagedApplication is managed by at least one Manager.

The manager is mostly designed like separate application. In RTPS architecture is able to create application which contains manager and managedapplication, but for easy managing is better split both. The ORTE contains a separate instance of manager located in directory `orte/manager`.

The manager is composed from five kinds of objects:

- **WriterApplicationSelf:** through which the Manager provides information about its own parameters to Managers on other nodes.
- **ReaderManagers:** CSTReader through which the Manager obtains information on the state of all other Managers on the Network.
- **ReaderApplications:** CSTReader which is used for the registration of local and remote managedApplications.

- **WriterManagers:** CSTWriter through which the Manager will send the state of all Managers in the Network to all its managees.
- **WriterApplications:** CSTWriter through which the Manager will send information about its managees to other Managers in the Network.

A Manager that discovers a new ManagedApplication through its readerApplications must decide whether it must manage this ManagedApplication or not. 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. If none of the keys are equal, the managed application is ignored. At the end of this process all Managers have discovered their managees and the ManagedApplications know all Managers in the Network.

The managedApplication is composed from seven kinds of objects:

- **WriterApplicationSelf:** a CSTWriter through which the ManagedApplication registers itself with the local Manager.
- **ReaderApplications:** a CSTReader through which the ManagedApplication receives information about another ManagedApplications in the network.
- **ReaderManagers:** a CSTReader through which the ManagedApplication receives information about Managers.
- **WriterPublications:** CSTWriter through which the Manager will send the state of all Managers in the Network to all its managees.
- **ReaderPublications:** a Reader through which the Publication receives information about Subscriptions.
- **WriterSubscriptions:** a Writer that provides information about Subscription to Publications.
- **ReaderSubscriptions:** a Reader that receives issues from one or more instances of Publication, using the publish-subscribe service.

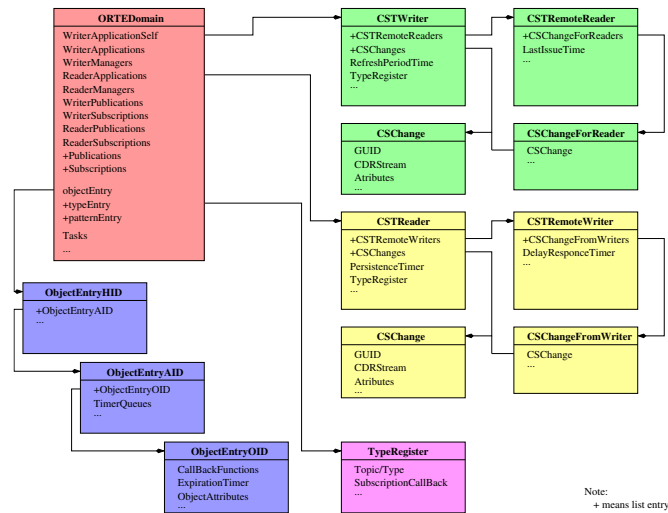
The ManagedApplication has a special CSTWriter writerApplicationSelf. The Composite State (CS) of the ManagedApplication's writerApplicationSelf object contains only one NetworkObject - the application itself. The writerApplicationSelf of the ManagedApplication must be configured to announce its presence repeatedly and does not request nor expect acknowledgments.

The ManagedApplications now use the CST Protocol between the writerApplications of the Managers and the readerApplications of the ManagedApplications in order to discover other ManagedApplications in the Network. Every ManagedApplication has two special CSTWriters, writerPublications and writerSubscriptions, and two special CSTReaders, readerPublications and readerSubscriptions.

Once ManagedApplications have discovered 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.

The ORTE stores all data in local database per application. There isn't central store where are data saved. If an application comes into communication, then will be created local mirror of all applications parameters. Parts of internal structures are shown in Figure 1-6.

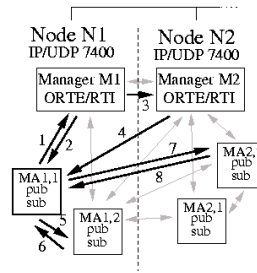
Figure 1-6. ORTE Internal Attributes



Following example shows communication between two nodes (N1, N2). There are applications running on each node - MA1.2 on node N1 and MA2.1, MA2.2 on node N2. Each node has its own manager (M1, M2). The example shows, what's happen when a new application comes into communication (MA1.1).

1. MA1.1 introduces itself to local manager M1
2. M1 sends back list of remote managers Mx and other local applications MA1.x
3. MA1.1 is introduced to all Mx by M1
4. All remote MAs are reported now to M1.1
5. MA1.1 is queried for self services (publishers and subscribers) from others MAX.
6. MA1.1 asks for services to others MAX.
7. All MAs know information about others.

The corresponding publishers and subscribers with matching Topic and Type are connected and starts their data communication.

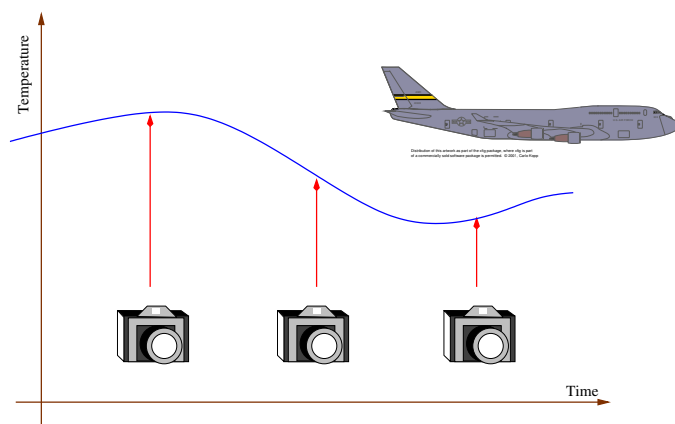
Figure 1-7. RTPS Communication among Network Objects

1.5. ORTE Examples

This chapter expects that you are familiar with RTPS communication architecture described in Section 1.2.

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.

- **BestEffort:** This reliability policy is suitable for data that are sent with a period. There are no message resending when a message is lost. On the other hand, this policy offers maximal predictable behaviour. For instance, consider a publication which sends data from a sensor (pressure, temperature, ...).

Figure 1-8. Periodic Snapshots of a BestEffort Publisher

- **StrictReliable:** The ORTE supports the reliable delivery of issues. This kind of communication is used where a publication want to be sure that all data will be delivered to subscriptions. For instance, consider a publication which send commands.

Command data flow requires that each instruction in the sequence is delivered reliably once and only once. Commands are often not time critical.

1.5.1. BestEffort Communication

Before creating a Publication or Subscription is necessary to create a domain by using function *ORTEDomainAppCreate*. The code should looks like:

```
int main(int argc, char *argv[])
{
    ORTEDomain *d = NULL;
    ORTEBoolean suspended= ORTE_FALSE;

    ORTEInit();

    d = ORTEDomainAppCreate(ORTE_DEFAULT_DOMAIN, NULL, NULL, suspended);
    if (!d)
    {
        printf("ORTEDomainAppCreate failed\n");
        return -1;
    }
}
```

The *ORTEDomainAppCreate* allocates and initializes resources that are needed for communication. The parameter *suspended* says if *ORTEDomain* takes suspend communicating threads. In positive case you have to start threads manually by using *ORTEDomainStart*.

Next step in creation of a application is registration serialization and deserialization routines for the specific type. You can't specify this functions, but the incoming data will be only copied to output buffer.

```
ORTETypeRegisterAdd(d, "HelloMsg", NULL, NULL, 64);
```

To create a publication in specific domain use the function *ORTEPublicationCreate*.

```
char instance2send[64];
NtpTime persistence, delay;

NTPTIME_BUILD(persistence, 3); /* this issue is valid for 3 seconds */
NTPTIME_DELAY(delay, 1);      /* a callback function will be called every 1 second */
p = ORTEPublicationCreate( d,
```

```

"Example HelloMsg",
"HelloMsg",
&instance2Send,
&persistence,
1,
sendCallBack,
NULL,
&delay);

```

The callback function will be then called when a new issue from publisher has to be sent. It's the case when you specify callback routine in *ORTEPublicationCreate*. When there isn't a routine you have to send data manually by call function *ORTEPublicationSend*. This option is useful for sending periodic data.

```

void sendCallBack(const ORTESendInfo *info, void *vinstance, void *sendCallBackParam)
{
    char *instance = (char *) vinstance;
    switch (info->status)
    {
        case NEED_DATA:
            printf("Sending publication, count %d\n", counter);
            sprintf(instance, "Hello world (%d)", counter++);
            break;

        case CQL: //criticalQueueLevel has been reached
            break;
    }
}

```

Subscribing application needs to create a subscription with publication's Topic and TypeName. A callback function will be then called when a new issue from publisher will be received.

```

ORTESubscription *s;
NtpTime deadline, minimumSeparation;

NTPTIME_BUILD(deadline, 20);
NTPTIME_BUILD(minimumSeparation, 0);
p = ORTESubscriptionCreate( d,
                            IMMEDIATE,
                            BEST_EFFORTS,
                            "Example HelloMsg",
                            "HelloMsg",
                            &instance2Recv,
                            &deadline,
                            &minimumSeparation,
                            recvCallBack,
                            NULL);

```

The callback function is shown in the following example:

```
void recvCallBack(const ORTERecvInfo *info, void *vinstance, void *recvCallBackParam)
{
    char *instance = (char *) vinstance;
    switch (info->status)
    {
        case NEW_DATA:
            printf("%s\n", instance);
            break;

        case DEADLINE: //deadline occurred
            break;
    }
}
```

Similarly examples are located in ORTE subdirectory `orte/examples/hello`. There are demonstrating programs how to create an application which will publish some data and another application, which will subscribe to this publication.

1.5.2. Reliable communication

The reliable communication is used especially in situations where we need guarantee data delivery. The ORTE supports the inorder delivery of issues with built-in retry mechanism

The creation of reliable communication starts like besteffort communication. Difference is in creation a subscription. Third parameter is just only changed to `STRICT_RELIABLE`.

```
ORTESubscription *s;
NtpTime deadline, minimumSeparation;

NTPTIME_BUILD(deadline, 20);
NTPTIME_BUILD(minimumSeparation, 0);
p = ORTESubscriptionCreate( d,
                            IMMEDIATE,
                            STRICT_RELIABLE,
                            "Example HelloMsg",
                            "HelloMsg",
                            &instance2Recv,
                            &deadline,
                            &minimumSeparation,
                            recvCallBack,
                            NULL);
```


Note:

Strict reliable subscription must set `minimumSeparation` to zero! The middleware can't guarantee that the data will be delivered on first attempt (retry mechanism).

Sending of a data is blocking operation. It's strongly recommended to setup sending queue to higher value. Default value is 1.

```
ORTEPublProp *pp;

ORTEPublicationPropertiesGet(publisher, pp);
pp->sendQueueSize=10;
pp->criticalQueueLevel=8;
ORTEPublicationPropertiesSet(publisher, pp);
```

An example of reliable communication is in ORTE subdirectory `orte/examples/reliable`. There are located a `strictreliable` subscription and publication.

1.5.3. Serialization/Deserialization

Actually the ORTE doesn't support any automatic creation of serialization/deserialization routines. This routines have to be designed manually by the user. In next is shown, how should look both for the structure `BoxType`.

```
typedef struct BoxType {
    int32_t color;
    int32_t shape;
} BoxType;

void
BoxTypeSerialize(CDR_Codec *cdrCodec, void *instance) {
    BoxType *boxType=(BoxType*)instance;

    CDR_put_long(cdrCodec, boxType->color);
    CDR_put_long(cdrCodec, boxType->shape);
}

void
BoxTypeDeserialize(CDR_Codec *cdrCodec, void *instance) {
    BoxType *boxType=(BoxType*)instance;

    CDR_get_long(cdrCodec, &boxType->color);
    CDR_get_long(cdrCodec, &boxType->shape);
}
```

When we have written a serialization/deserialization routine we need to register this routines to middleware by function `ORTETypeRegisterAdd`

```
ORTETypeRegisterAdd(  
    domain,  
    "BoxType",  
    BoxTypeSerialize,  
    BoxTypeDeserialize,  
    sizeof(BoxType));
```

The registration must be called before creation a publication or subscription. Normally is `ORTETypeRegisterAdd` called immediately after creation of a domain (`ORTEDomainCreate`).

All of codes are part of the Shapedemo located in subdirectory `orte/contrib/shape`.

1.6. ORTE Tests

There were not any serious tests performed yet. Current version has been intensively tested against reference implementation of the protocol. Results of these test indicate that ORTE is fully interoperable with implementation provided by another vendor.

1.7. ORTE Usage Information

1.7.1. Installation and Setup

In this chapter is described basic steps how to makes installation and setup process of the ORTE. The process includes next steps:

1. Downloading the ORTE distribution
2. Compilation
3. Installing the ORTE library and utilities
4. Testing the installation

Note:

On windows systems we are recommend to use Mingw or Cygwin systems. The ORTE support also MSVC compilation, but this kind of installation is not described here.

1.7.1.1. Downloading

The ORTE component can be obtained from OCERA SourceForge web page (<http://www.sf.net/projects/ocera/>). Here is the component located also in self distribution branch as well as in OCERA distribution. Before developing any application check if there is a new file release.

The CVS version of ORTE repository can be checked out be anonymous (pserver) CVS with the following commands.

```
cvs -d:pserver:anonymous@cvs.ocera.sourceforge.net:/cvsroot/ocera login
cvs -z3 -d:pserver:anonymous@cvs.ocera.sourceforge.net:/cvsroot/ocera co ocera/components/c
```

Attention, there is developing version and can't be stable!

1.7.1.2. Compilation

Before the compilation process is necessary to prepare the source. Create a new directory for ORTE distribution. We will assume name of this directory `/orte` for Linux case. Copy or move downloaded ORTE sources to `/orte` (assume the name of sources `orte-0.2.3.tar.gz`). Untar and unzip this files by using next commands:

```
gunzip orte-0.2.3.tar.gz
tar xvf orte-0.2.3.tar
```

Now is the source prepared for compilation. Infrastructure of the ORTE is designed to support GNU make (needs version 3.81) as well as autoconf compilation. In next we will continue with description of autoconf compilation, which is more general. The compilation can follows with commands:

```
mkdir build
cd build
../configure
make
```

This is the case of outside autoconf compilation. In directory `build` are all changes made over ORTE project. The source can be easy move to original state be removing of directory `build`.

1.7.1.3. Installing

The result of compilation process are binary programs and ORTE library. For the next developing is necessary to install this result. It can be easy done be typing:

```
make install
```

All developing support is transferred into directories with direct access of design tools.

name	target path
ortemanager, orteping, ortespy	/usr/local/bin
library	/usr/local/lib
include	/usr/local/include

The installation prefix `/usr/local/` can be changed during configuration. Use command `../configure --help` for check more autoconf options.

1.7.1.4. Testing the Installation

To check of correct installation of ORTE open three shells.

1. In first shell type

```
ortemanager
```

2. In second shell type

```
orteping -s
```

This command will invoked creation of a subscription. You should see:

```
deadline occurred
deadline occurred
...
```

3. In third shell type

```
orteping -p
```

This command will invoked creation of a publication. You should see:

```
sent issue 1
sent issue 2
sent issue 3
sent issue 4
...
```

If the ORTE installation is properly, you will see incoming messages in second shell (**orteping -s**).

```
received fresh issue 1
received fresh issue 2
```

```
received fresh issue 3
received fresh issue 4
...
```

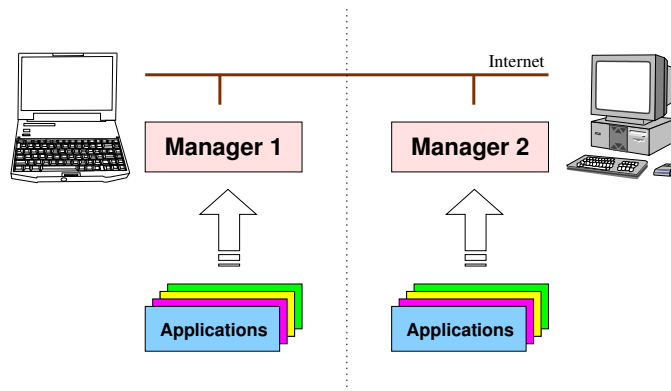
It's sign, that communication is working correctly.

1.7.2. The ORTE Manager

A manager is special application that helps applications automatically discover each other on the Network. Each time an object is created or destroyed, the manager propagate new information to the objects that are internally registered.

Every application precipitate in communication is managed by least one manager. The manager should be designed like separated application as well as part of designed application.

Figure 1-9. Position of Managers in RTPS communication



The ORTE provides one instance of a manager. Name of this utility is `ortemanager` and is located in directory `orte/ortemanager`. Normally is necessary to start `ortemanager` manually or use a script on UNIX systems. For Mandrake and Red-hat distribution is this script created in subdirectory `rc`. Windows users can install `ortemanager` like service by using option `/install_service`.

Note:

Don't forget to run a manager (ortemanager) on each RTPS participate node. During live of applications is necessary to be running this manager.

1.7.2.1. Example of Usage ortemanager

Each manager has to know where are other managers in the network. Their IP addresses are therefore specified as IPAddressX parameters of ortemanager. All managers participate in one kind of communication use the same domain number. The domain number is transferred to port number by equation defined in RTPS specification (normally domain 0 is transferred to 7400 port).

Let's want to distribute the RTPS communication of nodes with IP addresses 192.168.0.2 and 192.168.0.3. Private IP address is 192.168.0.1. The ortemanager can be execute with parameters:

```
ortemanager -p 192.168.0.2:192.168.0.3
```

To communicate in different domain use (parameter -d):

```
ortemanager -d 1 -p 192.168.0.2:192.168.0.3
```

Very nice feature of ortemanager is use event system to inform of creation/destruction objects (parameter -e).

```
ortemanager -e -p 192.168.0.2:192.168.0.3
```

Now, you can see messages:

```
[smolik@localhost smolik]$ortemanager -e -p 192.168.0.2:192.168.0.3
manager 0xc0a80001-0x123402 was accepted
application 0xc0a80002-0x800301 was accepted
application 0xc0a80002-0x800501 was accepted
application 0xc0a80002-0x800501 was deleted
manager 0xc0a80001-0x123402 was deleted
```

ortemanager

Name

ortemanager — the utility for discovery others applications and managers on the network

Synopsis

```
ortemanager [-d domain] [-p ip addresses] [-k ip addresses] [-R refresh] [-P purge] [-D ] [-E expiration] [-e ] [-v verbosity] [-l filename] [-V] [-h]
```

Description

Main purpose of the utility **ortemanager** is debug and test ORTE communication.

OPTIONS

-d --domain

The number of working ORTE domain. Default is 0.

-p --peers

The IP addresses parsipiates in RTPS communication. See Section 1.7.2 for example of usage.

-R --refresh

The refresh time in manager. Default 60 seconds.

-P --purge

The searching time in local database for finding expired application. Default 60 seconds.

-E --expiration

Expiration time in other applications.

-m --minimumSeparation

The minimum time between two issues.

-v --verbosity

Set verbosity level.

-l --logfile

All debug messages can be redirect into specific file.

-V --version

Print the version of **ortemanager**.

-h --help

Print usage screen.

1.7.3. Simple Utilities

The simple utilities can be found in the `orte/examples` subdirectory of the ORTE source subtree. These utilities are useful for testing and monitoring RTPS communication.

The utilities provided directly by ORTE are:

`orteping`

the utility for easy creating of publications and subscriptions.

`ortespy`

monitors issues produced by other application in specific domain.

orteping

Name

`orteping` — the utility for debugging and testing of ORTE communication

Synopsis

```
orteping [-d domain] [-p ] [-S strength] [-D delay] [-s ] [-R refresh] [-P purge] [-E
expiration] [-m minimumSeparation] [-v verbosity] [-q ] [-l filename] [-V] [-h]
```

Description

Main purpose of the utility **orteping** is debug and test ORTE communication.

OPTIONS

`-d --domain`

The number of working ORTE domain. Default is 0.

`-p --publisher`

Create a publisher with Topic - Ping and Type - PingData. The publisher will publish a issue with period by parameter delay.

`-s --strength`

Setups relative weight against other publishers. Default is 1.

`-D --delay`

The time between two issues. Default 1 second.

`-s --subscriber`

Create a subscriber with Topic - Ping and Type - PingData.

`-R --refresh`

The refresh time in manager. Default 60 seconds.

`-P --purge`

The searching time in local database for finding expired application. Default 60 seconds.

`-E --expiration`

Expiration time in other applications.

`-m --minimumSeparation`

The minimum time between two issues.

`-v --verbosity`

Set verbosity level.

`-q --quite`

Nothing messages will be printed on screen. It can be useful for testing maximal throughput.

`-l --logfile`

All debug messages can be redirect into specific file.

`-V --version`

Print the version of **orteping**.

`-h --help`

Print usage screen.

ortespy

Name

ortespy — the utility for monitoring of ORTE issues

Synopsis

orteping [-d *domain*] [-v *verbosity*] [-R *refresh*] [-P *purge*] [-e *expiration*] [-l *filename*] [-V] [-h]

Description

Main purpose of the utility **ortepy** is monitoring data traffic between publications and subscriptions.

OPTIONS

-d --domain

The number of working ORTE domain. Default is 0.

-v --verbosity

Set verbosity level.

-R --refresh

The refresh time in manager. Default 60 seconds.

-P --purge

Create publisher

-e --expiration

Expiration time in other applications.

-l --logfile

All debug messages can be redirect into specific file.

-V --version

Print the version of **orteping**.

-h --help

Print usage screen.