



Product:	CDP Redundancy
Product version:	V2.3
Document ID:	PM-CDP Redundancy
Doc revision:	A
Written/Aprr.:	OH/ SL
Date:	28.10.2008

Industrial Control Design AS



CDP Redundancy V2.3

Programmers Manual

The content of this document is confidential information not to be published without the consent of Industrial Control Design AS.

Industrial Control Design AS, www.icd.no, support@icd.no, forum.icd.no

Contents

1. INTRODUCTION.....	3
1.1. About.....	3
1.2. Terminology.....	3
<hr/>	
2. INSTALLATION.....	4
2.1. Prerequisites.....	4
2.2. Quick setup of a project utilizing the RedundancyLib library.....	4
<hr/>	
3. PREPARE COMPONENTS.....	9
3.1. Overview.....	9
3.1.1. Must prepare components for redundancy.....	9
3.1.2. Custom serialization.....	9
3.2. Serialization basics.....	9
3.3. CDPUDPInputStream / CDPUDPOutputStream.....	11
3.3.1. Packet Format.....	13
3.3.2. Limitations.....	13

1. Introduction

1.1. About

This document describes how to use the CDP redundancy functionality, with the RDManager system component, to create redundant controller applications. The CDP Redundancy is a solution where one (active) controller is running the application code, while one to three (standby) controllers are in standby, being updated by the running controller. If the running controller fails, one of the standby controllers take over as the active controller. This takeover happens within a time-interval of at least one process period, but may take more depending on how the redundancy is set up. This means that the redundant application must be tolerant to lost 'samples' or 'runs', like any other CDP application.

1.2. Terminology

RD

The redundancy and standby- functionality on CDP.

RDManager

RD functionality manager component model.

Companion controllers

Two or more controllers that contain an instance of the same virtual application.

Virtual application

Logical application space for components running in more than one physical controller.

IOServer

A component that handles Input/Output to something, typically (but not limited to) another external hardware device.

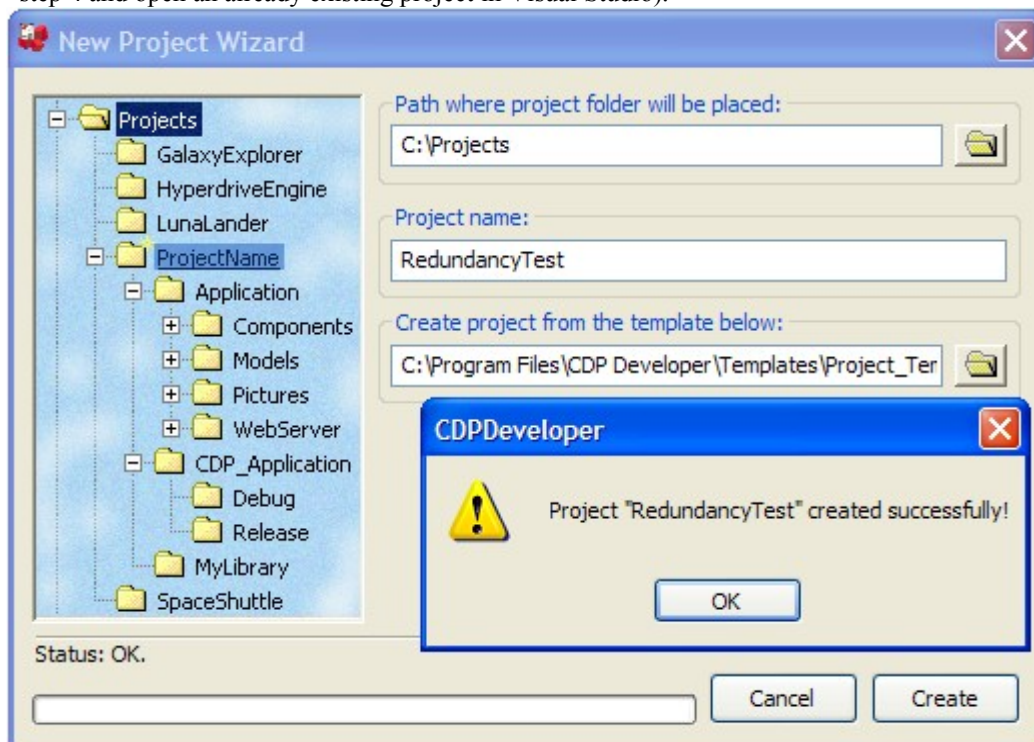
2. Installation

2.1. Prerequisites

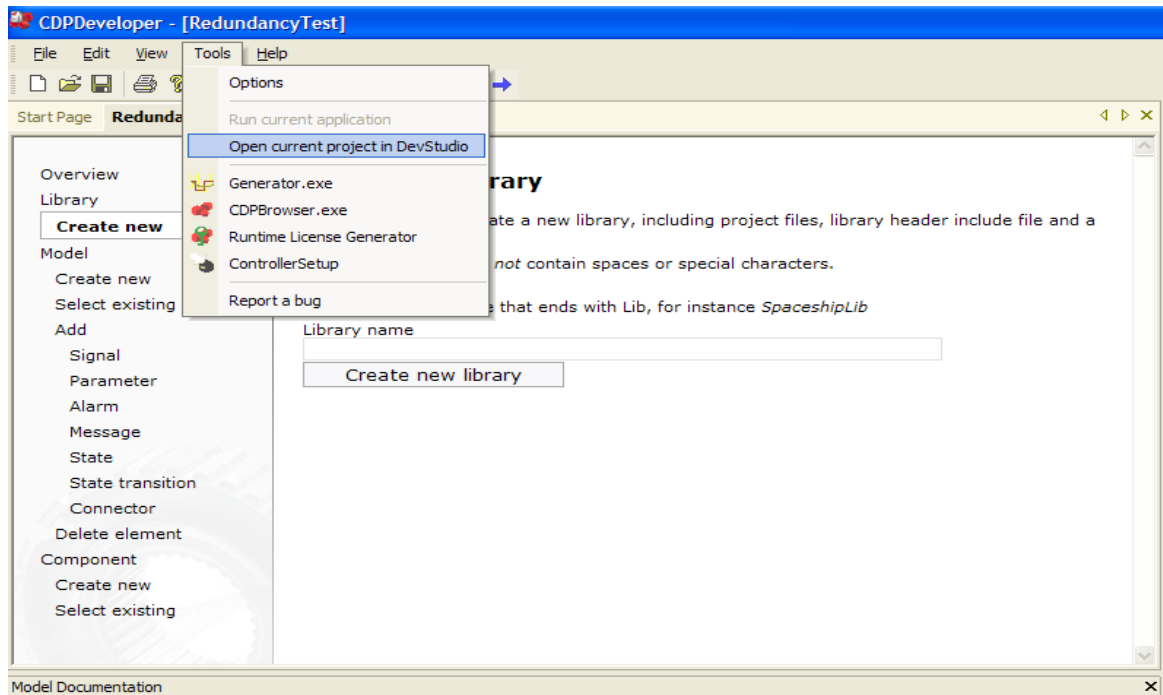
- A valid CDP license
- Familiar with CDP
- CDP version: 2.3.1.0
- OS: “Windows” or “RTOS version 5.11” or “Linux (based on gcc-4.1.1 and glibc-2.3.6 or glibc-2.6.1)”.

2.2. Quick setup of a project utilizing the RedundancyLib library

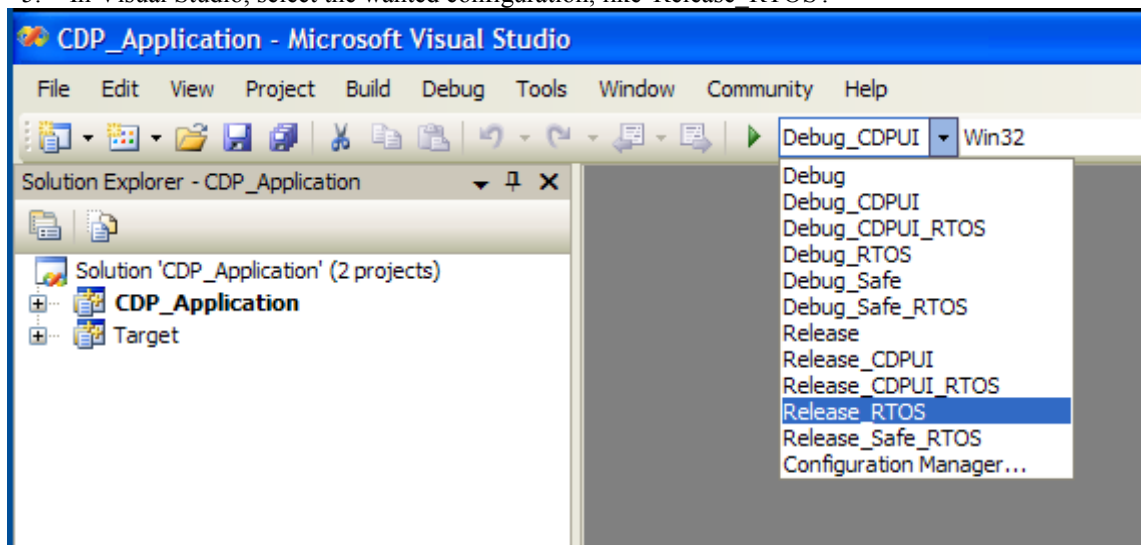
1. Start CDP Developer
2. Make a new project by selecting File->'New Project', type in project name and click Create (or skip to step 4 and open an already existing project in Visual Studio).



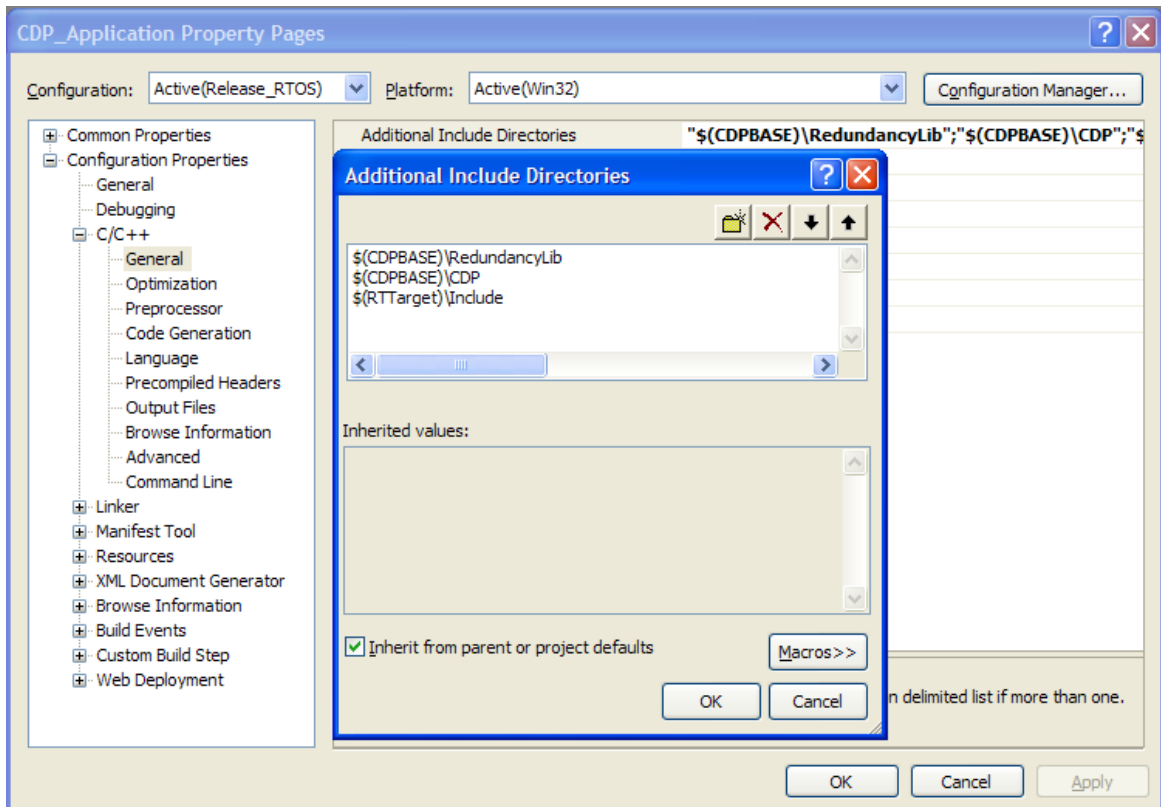
3. Choose Tools->'Open current project in DevStudio':



4. If Visual Studio asks to convert the project, accept this by selecting 'Next'/'Finish'/'Close' until done. Close the conversion report.
5. In Visual Studio, select the wanted configuration, like 'Release_RTOS'.

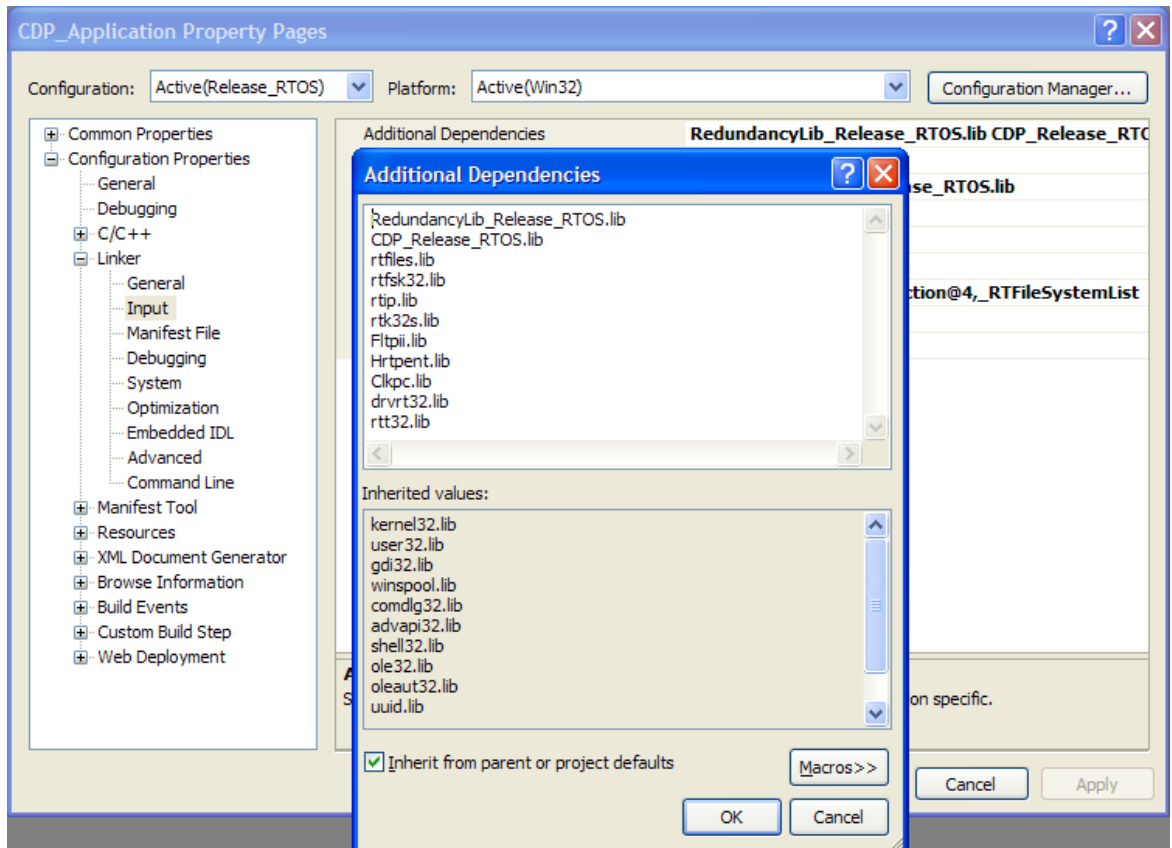


6. Select 'CDP_Application' from the 'Solution Explorer', right-click and select Properties.
7. In C++/Additional Include Directories, make sure it says:
"\$\$(CDPBASE)\RedundancyLib";"\$(CDPBASE)\CDP";"\$(RTTarget)\Include"

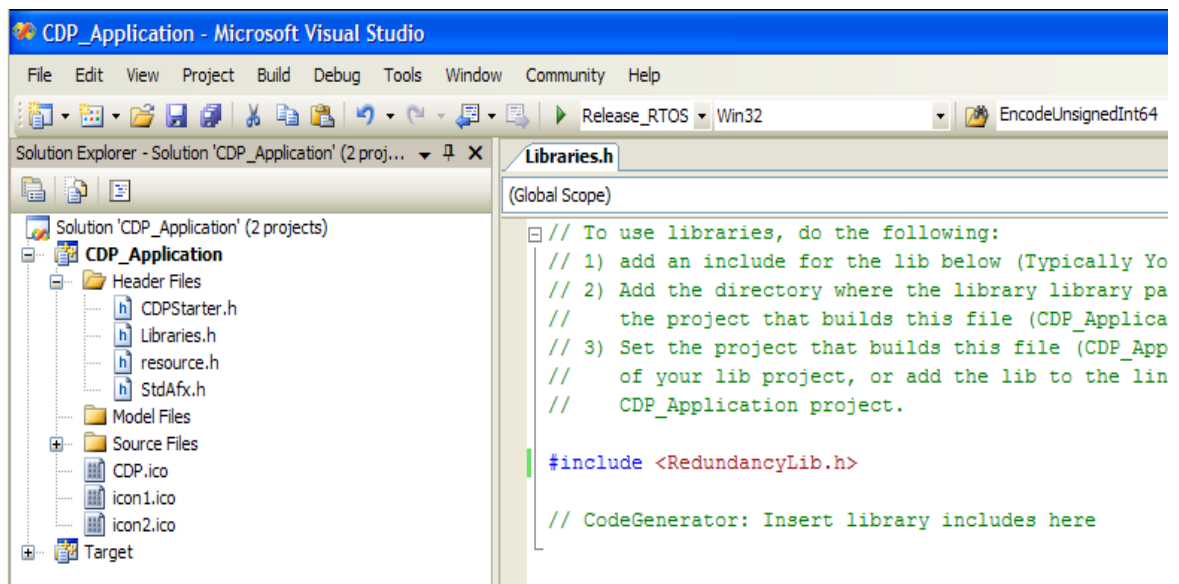


The compiler will look in the directories specified in 'Additional Include Directories' for files that you #include in your .cpp and .h files. If you get an error 'Can not open include file...', then it is most likely caused by a missing include directory, or that the file you #include does not exist.

8. In Linker->Input, make sure you list RedundancyLib_Release_RTOS.lib CDP_Release_RTOS.lib in addition to all the on-time rtos32 libraries you need. This will ensure that the linker finds all the functions that are referenced in the code.



9. In the 'Solution Explorer', inside the 'CDP_Application' project, locate the Header file 'Libraries.h', and add the line
`#include <RedundancyLib.h>`



10. If there is no 'Target' project in your 'Solution Explorer', select 'Solution'->'Add'->'Existing Project'. Choose the 'Target.vcproj' from the same folder in which your 'CDP_Application.vcproj' is at.
11. Right-Click Solution, select 'Configuration Manager...'. Make sure that all configurations have 'Release_RTOS' selected in 'Configuration':

3. Prepare components

3.1. Overview

3.1.1. Must prepare components for redundancy

Serialization must be implemented on all components running redundant.

If the component does not have any custom members, such as stl lists or int or arrays, but only CDPObject-derived members like Signal<T>, then it is not necessary to extend the serialization as this is already handled in the CDPComponent base class. Note that timers will not be Serialized by the CDP base, you will have to do this yourself.

3.1.2. Custom serialization

Custom serialization is made by overriding methods SerializeIn(...) and SerializeOut(...), adding copying in and out for all custom members that need to be synchronized.

3.2. Serialization basics

The serialization of objects is handled by the InputStream / OutputStream classes.

These classes implement all the basic operators needed for serialization, and contain a pure virtual function to do the actual transport of the serialized data. These functions (WriteStreamData() and ReadStreamData()) must be overridden by derived classes. CDPUDPInputStream and CDPUDPOutputStream are such derived classes; they implement a transport layer on the UDP Multicast protocol.

The InputStream class implements the following operators:

```
virtual InputStream& operator>>(bool &bData);
virtual InputStream& operator>>(char &chData);
virtual InputStream& operator>>(unsigned char &chData);
virtual InputStream& operator>>(short &nData);
virtual InputStream& operator>>(unsigned short &nData);
virtual InputStream& operator>>(int &nData);
virtual InputStream& operator>>(unsigned int &nData);
virtual InputStream& operator>>(long &nData);
virtual InputStream& operator>>(unsigned long &nData);
virtual InputStream& operator>>(float &fData);
virtual InputStream& operator>>(double &dData);
virtual InputStream& operator>>(int64_t &uiData);
virtual InputStream& operator>>(uint64_t &uiData);
virtual InputStream& operator>>(std::string &strData);
```

The output stream implements the reverse operators:

```
virtual OutputStream& operator<<(bool & bData);
virtual OutputStream& operator<<(char & chData);
virtual OutputStream& operator<<(unsigned char & chData);
virtual OutputStream& operator<<(short & nData);
virtual OutputStream& operator<<(unsigned short & nData);
virtual OutputStream& operator<<(int & nData);
virtual OutputStream& operator<<(unsigned int & nData);
```

```

virtual OutputStream& operator<<(long & nData);
virtual OutputStream& operator<<(unsigned long & nData);
virtual OutputStream& operator<<(float & fData);
virtual OutputStream& operator<<(double & dData);
virtual OutputStream& operator<<(int64_t & uiData);
virtual OutputStream& operator<<(uint64_t & uiData);
virtual OutputStream& operator<<(std::string & strData);

```

All CDPObjects and CDPObject-derived classes contain these two virtual functions:

```

virtual OutputStream& SerializeOut(OutputStream& outstream);
virtual InputStream& SerializeIn(InputStream& instream);

```

SerializeOut(...) is used to transport data from this object to another object.
 SerializeIn(...) is used to transport data from another object to this object.

The following CDPObjects are serialized automatically from a CDPCOMPONENT: Signals, Alarms, Parameters, local connectors, parent connector, and remote connectors. States are also serialized.

Other objects must be serialized manually. Example:

Assume you have a CDPCOMPONENT, named MyComponent, which has a CDPTimerMs and 5 double types.

To serialize these objects, you must:

1. Add the serialization functions to the CDPCOMPONENT derived class header file:

```

class MyComponent: public CDPCOMPONENT
{
  ....
  ....
public:
virtual OutputStream& SerializeOut(OutputStream& outstream);
virtual InputStream& SerializeIn(InputStream& instream);
  ....
  ....
  CDPTimerMs    m_Timer;
  double        m_d1,m_d2,m_d3,m_d4,m_d5;
};

```

2. Implement the Serialization functions in the CDPCOMPONENT derived class cpp file:

```

OutputStream& MyComponent::SerializeOut(OutputStream& outstream)
{
  // always call base first
  CDPCOMPONENT::SerializeOut(outstream);

  // Serialize Timer:
  m_Timer.SerializeOut(outstream);

  // Serialize doubles:
  outstream << m_d1;
  outstream << m_d2;
  outstream << m_d3;
  outstream << m_d4;
  outstream << m_d5;

  // Always return stream.
  return outstream;
}

```

```

InputStream& MyComponent::SerializeIn(InputStream& instream)
{
  // always call base first
  CDPCOMPONENT::SerializeIn(instream);

  // Serialize Timer:

```

```

m_Timer.SerializeIn(instream);

// Serialize doubles:
instream >> m_d1;
instream >> m_d2;
instream >> m_d3;
instream >> m_d4;
instream >> m_d5;

// Always return stream.
return instream;
}

```

If you have an array that must be serialized, or some other type that is not directly supported by the streaming operators, it is advised that you serialize the size first, and then serialize <size> elements. Note that CDP automatically handles network conversion of data types >byte, so you do not need to worry about endian-ness incompatibility.

You should also note that the order of the objects is the same regardless of serialization direction (in or out), see example above.

There are two types of serialization:

Fast-sync: This is serialization that happens at a specified interval (SynchFs). Only non-persistent data is synchronized to minimize transport time and amount of data.

Slow-sync: This is serialization that happens when a persistent value has changed. The slow-sync actually happens after an xml-file for a redundant component has been write modified. This happens regardless of redundancy state. If you upload a new xml-file belonging to a 'Standby' (or 'Active') component, this file will be automatically distributed to the other companion(s).

To distinguish between slow-sync and fast-sync, there is a method in the InputStream and OutputStream objects that can be called to determine whether it is performing fast or slow synchronization:

```

if(instream.ShouldDoPersistentDataOnly())
{
    // do all persistent values
}
else
{
    // do non-persistent values...
}

```

3.3. CDPUDPInputStream / CDPUDPOutputStream

Note: The following information is technical and is not required to understand to use the redundancy solution. However, there are some limitations you should be aware of, see Limitations in chapter 4.3.2

These two classes contain the actual transport layer for the synchronization. The SerializeIn and SerializeOut functions will be allocated a certain time to run to avoid blocking. The implementation is like this:

The SerializeOut function will build all serialized data into one buffer. This buffer will be transmitted as one or more UDP packets (depending on size). Each UDP packet is 1032 bytes (1024 bytes data + 8 bytes header). After all packets are sent, the remaining time will be used for waiting for NACK (negative acknowledge) on any of the packets, and resending the last packet. If one or more nack(s) is received within the allocated time, the nack'ed-packets will be re-sent.

The SerializeIn function will wait for data. If packet == expected packet, receive packet and increase expected packet. If packet > expected packet, receive packet and send NACK for packets [expected packet, got packet>, increase expected packet. If packet < expected packet and a resend, receive packet.

The reasoning to use negative acknowledgment is as follows: This allows us to have one sender and many listeners without needing a lot of bandwidth, and it's a simpler implementation than the 'ack' approach. Due to the nature of UDP, packets can (and probably will) be lost. If we were to send an acknowledgment for each packet, the sender would need to know how many listeners it had, and it would need to have a way of identifying these listeners. It would be a more complex protocol, less deterministic, it would be slower (due to resend/packet loss) and require more bandwidth than the nack-approach.

Example showing the 'Nack' approach (sending 5 packets):

```

Sender:                                     Receiver X:
packet 1   --->                             packet 1
                                                (timed wait for a packet)
packet 2   -X->                             no receive
                                                (timed wait for a packet)
packet 3   -X->                             no receive
                                                (timed wait for a packet)
packet 4   -X->                             no receive
                                                (timed wait for a packet)
packet 5   --->                             packet 5
done sending.
wait for nacks:
NO        <-X-                             resend packet (2,3,4)
                                                (timed wait for a packet)
packet 5   --->                             (ignore, already got it)
OK        <---                             resend packet (2,3,4)
                                                (timed wait for a packet)
packet 2   --->                             packet 2
OK        <---                             resend packet (3,4)
                                                (timed wait for a packet)
packet 3   -X->                             packet 3
OK        <---                             resend packet (3,4)
                                                (timed wait for a packet)
packet 4   --->                             packet 4
OK        <---                             resend packet (3)
                                                (timed wait for a packet)
packet 3   --->                             packet 3
                                                (got all packets), done.
packet 5   --->                             (not listening)
(wait4nack) packet 5 --->                     (not listening)
(wait4nack) packet 5 --->                     (not listening)
(time is up, done).
```

The last packet is re-sent at most 10 times if no 'nacks' are received. This is to ensure that all listeners get the last packet and to ensure that they have a possibility to send 'nack's for packets they are missing. If there is very little time left after sending the initial batch of packets (i.e. less than 5ms), the last packet is at most re-sent only three times.

3.3.1. Packet Format

The packet format for send-packets:

Type	Description
unsigned char	Version number (0x01)
unsigned char	sequence number (0-255)
unsigned short	Total number of packets in this sequence
unsigned short	Packet number in this sequence
unsigned short	packet length
array	packet length of data

The packet format for 'nack' packets:

Type	Description
unsigned char	Version number (0x01)
unsigned char	sequence number (0-255)
unsigned short	Total number of packets to nack (logical OR with 0x8000)
unsigned short	Packet number 1 to request
unsigned short	Packet number n-1 to request
unsigned short	Packet number n to request

3.3.2. Limitations

Since the packets sent out may also be received by the sender, a way of distinguishing between send-packets and 'nack' packets must exist. It has been chosen that the 'total number' should have its high bit set if it is a 'nack' packet. Since the total number of packets to ack is logically or'ed with 0x8000 (msb is set), this effectively limits the number of packets to send to 32767 packets. Each packet is 1024 bytes of actual data, so this yields a maximum of 1024*32767 bytes of data, which is slightly less than 32 MB of data for each SynchFs. However, unless the SynchFs is far below '1', this number is purely theoretical on today's systems. Note that the sender and receiver will require a lot of memory to handle this amount of data.

As the transport protocol is UDP, there is no guarantee that all packets arrive at the receivers. Depending on the send-frequency and amount of data to send, less time might be allocated to the 'wait4nack' loop. A low send-frequency (SynchFs) and low amount of data to send increases the chances of successful transmission. The RDManager SerializeCounter can be monitored on all RDManagers to see if they follow the active RDManager:



For the slow synchronization, specify the maximum time allowed to perform a slow-sync. If this time is set too low, the slowsynch might never succeed. This timeout-value should be set to at least 5 times the time needed to send the largest component XML file.

Assuming 100Mb/s network speed, and CPU able to cope, the practical network speed using our protocol is around 8MBytes/s. If the network-load is high, slower transfer rates are plausible due to more collisions and packet-loss in switches. Assuming your largest component file is 500 KB would compute to a theoretical slow-sync speed setting of around 0.07 seconds. Since we never want the slow-sync to fail you should add a good margin to this number; so round up to the nearest 0.1 seconds to be sure. Setting this value too high will cause delays to the slowsynch of the other redundant components.